A Reinterpretation of The Deductive Tableaux System in Higher-Order Logic

Master's thesis:

Abdelwaheb Ayari

Technische Fakultät
der Universität des Saarlands
e-mail: abdu@mpi-sb.mpg.de

Supervisors:
Priv. Doz. Dr. David Basin
Prof. Dr. Harald Ganzinger

June 11, 1995

Declaration
This master's thesis is the result of my own work and includes nothing which is the outcome of work done in collaboration. I have used only the resources listed in the bibliography.

Acknowledgments
I want to thank my supervisor Dr. David Basin, for the interest he took in my work. His patient consideration of my ideas and the influence of his own research have been of the great value through the development of this work. I thank Tobias Nipkow and Larry Paulson for their advice on Isabelle.

I want to thank Alexander Bach, Stefan Friedrich, Georg Struth and members of the Max-Planck-Institut[1] for their helpful comments on an earlier draft of this work.

Last, but by no means least, I thank my family for their continued support.

[1]This work has been developed in collaboration with the Max-Planck-Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany.

Cum DEUS calculat et cogitationem exercet, fit mundus.

G.W. Leibniz (Dialog 1677)

Contents

Chapter 1

Introduction

Software development is one of the central tasks in computer science. Applications of software tools can be found everywhere in real life (industry, administration, etc). The fundamental question that one ask is : How sure are we that the developed tools do what we expect from them? For the answer we have to consider how computer programs are built. Roughly speaking, conventional software engineering emphasizes two phases: The production of programs and the verification and testing of them. So at the first stage programs are constructed in conventional programming languages and in the second stage one attempts to prove their correctness. The notion of correctness that we adopt here means, following Hoare's definition, that the programs satisfy their specification (*partial correctness*) and that they terminate (*total correctness*).

Many systems such as [4, 24] are developed to support the activities to be achieved at the second stage. The most propagate and well known verification system is the prover *NQTHM* developed by Boyer and Moore in 1979. This prover presents a formal method to solve problems of correctness. Its application reaches from proof of mathematical theorems to the verification of the complete development of a microprocessor [5].

In recent years another formal method appeared to develop programs in computer science. This method is called *Program Synthesis* and consists in deriving programs from their specifications. In contrast to program verification, in this approach program construction and verification will be done hand by hand. Thus the constructed programs are automatically correct. In literature we distinguish two main approaches of program synthesis. The first approach is called "Constructive Program Synthesis" and the second is called "Deductive Program Synthesis". The former is based on constructive logics or variants of Martin-Löf's type theory [40]. The central key idea in this approach is the curry-howard-isomorphism [28] known as "propositions-as-types principle". Following this principle; specifications can be seen as types and programs can be seen as members of types. Then devolving a program p that satisfies a specification *Spec* can be reduced to prove that the type *Spec* is inhabited. The program p can be then extracted from the proof and because of the constructivity of the logic program p is an executable computer program. Deductive program synthesis is based on classical methods. A specification is this approach is considered as theorem that has to be proved. Parallel to the deduction process a program is generated step by step. Manna and Waldinger have developed the better known frameworks, called *Deductive Tableaux System*, in this area.

In the following we give a short overview of their system. In chapter 2 we will give a formal description of it illustrated with an example in which we derive two programs *front* and *last* that compute all but the last character and the last character of a non-empty string. The specifications that can be proved in the tableaux system have the general form $\forall x.\ \exists z.\ Q(x, z)$, where Q is a first-order formula. Such specification states that for each input x there is an output z satisfying the formula $Q(x, z)$ called *input-output relation*. In a preprocessing step the given specification is skolemized to $Q(a, f(a))$, where a is turned to an eigenvariable and the symbol f is a skolem function which denotes the desired program. Then the term $f(a)$ occuring in the goal $Q(a, f(a))$ is replaced with a new variable, say z and the initial tableau is built

Assertions	*Goals*	*Outputs*
		$f(a)$
	$Q(a, z)$	z

This tableau can be understood as follows: Our goal is to prove the validity of $Q(a, z)$ using some assertions of the underlying theory (at this point we have no assertions). And the desired program has at this stage the form $f(a) = z$. As we have already mentioned z is a variable that represents the current body of the function f. The proof is then constructed by applying deduction rules. These rules are essentially based on the nonclausal resolution. Their applications introduce additional now rows to the initial tableau. During the proof variables, such as z, can be incrementally instantiated. The deduction process terminates if one of the truth values *true/false* appears in of the column *goals/assertions*. The final row has then the form

false		t

or

	true	t

and the desired program has the final form $f(a) = t$, where is a term may depends on a. In order to obtain executable programs Manna and Waldinger introduced the notion of *primitive expressions*. These are expressions (terms or formulas) that contain only function and predicate symbols, for which one already has developed executable programs. In Manna and Waldinger's approach one require that during the deduction-process the output entries should be primitive expressions. Thus the final programs also are primitive expressions.

Let us now turn our attention to the aim of this work. Although the proof techniques used in the tableaux framework of Manna and Waldinger such as *induction*, *nonclausal resolution* and *simplification* are traditional and already implemented in many systems, the deductive tableaux system, however, has only recently been implemented. The literature abounds with work based on this approach and in which program derivations are entirely hand performed. These derivations are done at a fine level of detail and thereby show the possibility of supporting the development in a mechanical system. Our first aim is to implement the tableaux framework. As tools for the implementation (interpretation) we have used *Isabelle*, a logical framework supporting proof construction by higher-order resolution. The calculus constituting the interpretation are inference rules derived in Isabelle's higher-order logic. Thereby the calculus is then formally verified. In chapter 3 we give an overview of the framework Isabelle and we describe the encoding of higher-oder logic. In chapter 5 we first motivate the translation of the tableaux framework in Isabelle by an example. In this example we give a one-to-one simulation of a tableau proof to an Isabelle proof and we show thereby that the transformation is simple, however, some attention is needed. In the remainder of this chapter we formally give the transformation and we prove that we construct all programs that can be constructed using the tableaux framework. In a further focus point of this work we develop a new approach for program development. Our approach can be seen as extension of Manna and Waldinger's approach in the sense that instead of working in fixed-set of well-founded relations that are maintained in a data base, one can now generate constraints specifying new, not existing in the data base, well-founded relations, that can be then instantiated from the user. For this purpose we encode in chapter 4 a theory for well-founded relations. We have implemented our approach in the framework Isabelle. The deciding advantage using such a framework is the use of metavariables in the proofs as well is in the proof rules. This provides to further advantages over the deductive tableaux system. First we develop programs in a top-down way. Second we divide program construction in two parts: The first part is that in which we generate programs and the second is that in which we prove their termination. Following our approach we have synthesized a family of sorting programs, such as *quicksort*, *mergesort*, etc. Furthermore the framework Isabelle provides the use of metavariables in proofs; this has been exploit in our approach to divide program construction in two parts: The first part is that where we generate programs and in the second part we prove their termination. In chapter 7 we give a conclusion of our work with a comparison with some related work. Note that sources needed to run the sorting programs derived in chapter 6 in Isabelle are displayed in appendix B[1]

[1]The sources also are available per WWW `http://www.mpi-sb.mpg.de/~abdu/`

Chapter 2

Tableaux

2.1 Introduction

In this chapter we present the deductive synthesis framework of Manna and Waldinger and explain the relationship between their framework and the semantic tableaux [52] introduced by Raymond Smullyan in 1968.

2.2 Deductive Tableaux

2.2.1 Description of Tableaux

Let us now describe the deductive tableau framework. The fundamental structure of the deductive system is a tableau, which consists of two-dimensional array. Each row contains a sentence either an **assertion** A or a **goal** G and an optional term t, **output entry**. Typical rows in tableau have the following form:

$Assertions$	$Goals$	$Outputs$
A		t_1
	G	t_2

We wish to prove a sentence of the form $\forall x.\ \exists z.\ Q(x,z)$ (we call $Q(a,t)$ the *input-output condition* for the *input* a and the *output* t) within a particularly theory *TH*, characterized by a list of axioms $A_1, \ldots, A_n$. Proving the above sentence in this theory is equivalent to show the validity of the formula

$$A_1 \ \wedge \ \ldots \ \wedge \ A_n \ \rightarrow \ \forall x.\ \exists z.\ Q(x,z)$$

called **the associated sentence** of the tableau. We treat $A_1, \ldots$ and A_n as assertions and $\forall x.\ \exists z.\ Q(x,z)$ as a goal. The distinction between assertions and goals reflects the usual classical distinction between hypothesis and theorems. Any goal can be moved into the assertion column by negating it (and vice versa). The tableaux system possesses of a *simplifier*, *skolemization* rules, *nonclausal resolution* rules and an *induction* rule. The application of a deduction rule consists of adding new rows to the tableau and at the same time some variables occuring in the output columns can be instantiated with program-constructors such as *if-then-else*. The deduction process terminates when either a *true* goal or a *false* assertion is derived. Then the associated output slot contains the desired program. The output (usually associated with the goal) is said to be **suitable** if it satisfies, under a given interpretation, the input-output condition whenever the corresponding instance of the goal is true.

2.2.2 Preliminaries

In this section we introduce some informal preliminaries such as the notion of interpretation, model, polarity , etc.[1]

2.2.2.1 Language

The *expressions* of our language consist of terms and formulas (called also sentences). Terms are built from constants a, b, c, ... and variables u, v, w, ... More complex terms can be constructed by repeated applications of functions$f$, g, ... to other terms. In addition we have a *term constructor*, called *if-then-else*, that builds the new term *if* F *then* s *else* t from a sentence F and terms s and t. Atomic sentences are constructed by application of predicate symbols $=$, p, q, r, ... to terms. Sentences are built from truth values *true* and *false*, atomic sentences. They can also be constructed by applications of the connectives $\neg$, $\wedge$, ... and the quantifiers $\forall$ and $\exists$ to other sentences.

Definition 2.1
Expressions that contain no free variables are called **closed expressions**. And if they contain no variables at all then they are called **ground expressions**. **Herbrand expressions** are those that are ground and contain neither term constructors, connectives, nor equality symbols. $\square$

Notation: In the sentence $A[P]$ (or, respectively, $A < P >$) the brackets $[\,]$ denote replacement (respectively the angle brackets $< >$ indicate that not necessarily all occurrences of the subsentence P of A will be replaced)

Polarity The notion **polarity** is introduced to indicate how the truth of subsentences contributes to the truth of a sentence. We assign a polarity, positive $(+)$ or negative $(-)$, to each occurrence of a subsentence P of a sentence A. Informally, an occurrence of a subsentence P is positive [negative] in A, if it is in the scope of an even [odd] number of negation (signs). In the tableaux the assertions have a negative polarity and the goals have a positive polarity.

Definition 2.2 (Polarity-assignment Rules)
Let π indicate the polarity $(+, -, \pm)$ of a sentence and $-\pi$ is its opposite $(-, +, \pm)$, (here the polarity $(\pm)$ means that the corresponding subsentence has

[1] Here we give informal definitions. [34, 35, 37, 39, 20] can be consulted for more details.

name	polarity of sentence	polarity of subsentences
negation	$\boxed{\neg A}^{\pi}$	$\neg\boxed{A}^{-\pi}$
conjunction	$\boxed{A \wedge B}^{\pi}$	$\boxed{A}^{\pi} \wedge \boxed{B}^{\pi}$
disjunction	$\boxed{A \vee B}^{\pi}$	$\boxed{A}^{\pi} \vee \boxed{B}^{\pi}$
implication	$\boxed{\textit{if } A \textit{ then } B}^{\pi}$	$\textit{if}\boxed{A}^{-\pi} \textit{ then}\boxed{B}^{\pi}$
equivalence	$\boxed{A \equiv B}^{\pi}$	$\boxed{A}^{\pm\pi} \equiv \boxed{B}^{\pm\pi}$
universal quantifier	$\boxed{A\forall x.A}^{\pi}$	$\forall x.\boxed{A}^{\pi}$
existens quantifier	$\boxed{\exists x.A}^{\pi}$	$\exists x.\boxed{A}^{\pi}$
conditional-terms	$\boxed{\textit{if } A \textit{ then } s \textit{ else } t}^{\pi}$	$\textit{if}\boxed{A}^{\pm\pi} \textit{ then } s \textit{ else } t$

Table 2.1: Polarity Rules

both polarities: positive and negative). The polarity of each occurrence of a subsentence of a sentence is defined in table 2.1 □

Property 2.1 (Polarity Proposition)
The truth of a sentence is directly related to the truth of its strictly positive subsentences but inversely related to the truth of its strictly negative subsentences. □

Force of Quantifiers The role of a quantifier plays in a sentence depends on the polarity of the subsentence in which it occurs. For example, the existential quantifier in the sentence $\neg\ (\exists\ x.\ P(x))$ plays the same role as the universal quantifier in the logically equivalent sentence $\forall\ x.\ \neg\ P(x)$. The notion of **force** of quantifiers is introduced to indicate if the universal quantifier plays the same role as the existential quantifier and vice versa. We will annotate the universal force with the superscript $(\ldots)^{\forall}$ and the existential force with the superscript $(\ldots)^{\exists}$. (Notice that there are subsentences that can have both forces universal and existential. In this case they will be annotated with $(\ldots)^{\forall\exists}$). The rules given the force of quantifiers are giving in following two definitions.

Definition 2.3 (Strict universal force)
The occurrence of an universal (or, respectively, existential) quantifier has *strict universal force* if it has strict positive (or, respectively, negative) polarity.

$[\forall x.A]^{+} \Longrightarrow (\forall x)^{\forall}.A$

$[\exists x.A]^{-} \Longrightarrow (\forall x)^{\forall}.A$ □

Definition 2.4 (Strict existential force)
The occurrence of an universal (or, respectively, existential) quantifier has *strict existential force* if it has strict negative (or, respectively, positive) polarity.

$[\exists x.A]^{+} \Longrightarrow \exists x.A^{\exists}$

$[\forall x.A]^{-} \Longrightarrow \exists x.A^{\exists}$ □

2.2.2.2 Semantic Notions

So far we have explained how sentence can be build. Now we come to define their meanings. Classically a sentence can be assigned one of the truth values true or false. To determine the truth value of a given sentence we first have to consider a "situation" or "case" that states the truth values of some "basic" sentences. This situation or case is called *interpretation*. Once we have an interpretation, we can determine the truth values of each sentences by applying certain rules.

Definition 2.5 (Interpretation and Truth)
We adopt the Herbrand notion (see [25]) and define an **interpretation** as a finite or infinite set of Herbrand atoms. (Informally, we consider an interpretation as a complete set of Herbrand atoms that must be true). The truth-value of a closed sentence with respect to an interpretation is determined by recursively applying the classical semantic rules, such as $F \wedge G$ is true under $\mathcal{I}$, if F and G are true under $\mathcal{I}$. □

Definition 2.6 (Models and Theories)
Let S be a finite or infinite set of closed sentences. An interpretation $\mathcal{I}$ is a **model** of S, if every sentence in S is true under $\mathcal{I}$. We define the consequence relation $\models$ as follows:

$S \models \varphi$ if and only if, for all model $\mathcal{I}$ for S, φ is true under $\mathcal{I}$.

The **theory** of S is defined as the deductive closure of S:

$$TH(S) = \{\varphi \mid \ S \models \varphi\}.$$

Finally a sentence φ is said **valid** in a theory TH if it is true in every model of TH. □

2.2.2.3 Specifications, Programs and Correctness

Specifications are arbitrary sentences of the above defined language. **Programs** are terms of the form $f(a_1, \ldots, a_n) = t(a_1, \ldots, a_n)$, where f is a function symbol, a_1, …, a_n are variables and $t(a_1, \ldots, a_n)$ is a primitive expression.

We say that a program $f(a) = t(a)$ is correct with respect to the specification $\forall x.\ \exists z.\ Q(x,z)$ in a theory TH, if the sentence $\forall x.\ Q(x, f(x))$ is valid in TH augmented by the axiom $\forall x.\ f(x) = t(x)$.

2.2.2.4 Primitive expressions

In the program synthesis we are interested in constructing programs that are executable, "computer programs". For this purpose Manna and Waldinger introduce the notion of **primitive expressions**. Intuitively speaking, an primitive expression is that we know how to compute with. For example constants, expressions that are defined in terms of functions for which we have developed programs, etc. Primitive expressions are defined inductively as follows

Definition 2.7 (primitive expression) We suppose we are given a set of constants, functions and predicates symbols, called the *primitive set*. An expression is primitive if it is quantifier-free and all its constants, functions and predicates belong to a given primitive set. □

Heretofore we have only defined some preliminary notions that we need to explain the tableau mechanism. In the following section we describe the deduction process in the tableaux system and we give some properties of tableaux.

2.2.3 Properties of Tableaux and The Deduction Process

Deduction rules are used to add new rows to the tableau. This is done in such a way as to preserve validity of the associated sentence of the tableau and preserve suitability of the output entries. We give some definitions, which are intended to formalize this, namely that the set of all possible computable terms ("primitive closed terms"), that satisfy the input-output condition, is preserved, while the deduction process.

Definition 2.8 (Suiting a Row)
A closed term t suits a tableau $\mathcal{T}$ under the interpretation $\mathcal{I}$, if it exists a row in $\mathcal{T}$ $\begin{array}{|c|c||c|}\hline A & & s \\ \hline\end{array}$ (or, respectively, $\begin{array}{|c|c||c|}\hline & G & s \\ \hline\end{array}$) and substitution θ such that:

(i) $A\theta$ is closed and false under the interpretation $\mathcal{I}$ (or, respectively, $G\theta$ is closed and true under the model $\mathcal{I}$)

(ii) $s\theta$ is closed and $s\theta = t$ is true under the interpretation $\mathcal{I}$.

□

Definition 2.9 (Satisfying a Tableau)
A closed term t **satisfies a tableau** $\mathcal{T}$ in the theory TH, if for every model $\mathcal{I}$ of TH t suits the tableau $\mathcal{T}$ under $\mathcal{I}$ □

Definition 2.10 (Primitively Similar)
Two tableaux $\mathcal{T}_1$ and $\mathcal{T}_2$ are **primitively** similar in the theory TH, if they are satisfied by the same primitive terms. □

Definition 2.11 (Soundness)
A rule for adding new rows to a tableau is **sound** in the theory TH, if the same primitive closed terms satisfy the tableau before and after applying the rule. □

Now, to attain our claim, that the rules whose application add new rows preserve the relation of primitively similarity we require that all deduction rules of the tableaux system are sound. This will be showed in the next section. In the following we give some basic properties of deduction tableaux.

Property 2.2 (Duality)
In a tableau assertions (or, respectively, goals) can be pushed in the goals (or, respectively, assertions) column by negating them. The obtained Tableau is primitively similar to the initial one. □

Property 2.3 (Instantiation)
Let θ be a substitution and G a goal (or, respectively, A an assertion) in the tableau. Then adding or deleting the new row $G\theta$ (or, respectively $A\theta$) produces primitively similar tableau. □

Property 2.4 (No-output-entry)
A row without output entry is equivalent to one whose output entry a new variable that doesn't occur in the tableau. This is because if some closed terms suit the row under an interpretation $\mathcal{I}$, then any closed term suits it, since the truth of the condition (i) of definition 2.8 does not depend on the term.

$$\begin{array}{|c|c||c|}\hline A & & \\ \hline\end{array} \leftrightarrow \begin{array}{|c|c||c|}\hline A & & u \\ \hline\end{array}$$

and

$$\begin{array}{|c|c||c|}\hline & G & \\ \hline\end{array} \leftrightarrow \begin{array}{|c|c||c|}\hline & G & u \\ \hline\end{array}$$

where u is an arbitrary term. □

2.2.3.1 Derivation Process

We begin with a specification $\forall x \exists z.\, Q(x,z)$ in the theory TH and we are given a set of primitive symbols. First we form the initial tableau as follows:

(Initial tableau)

Assertions	*Goals*	*Outputs*
		$f(a)$
	$Q(a,z)$	z
A_1		
$\vdots$		
A_n		

Here $A_1, \ldots, A_n$ are some axioms of the theory *TH* that will be needed later in the proof. The function symbol f is introduced as a name for the desired program and a is a new skolem constant introduced during the normalization of the input specification. We now extend the initial primitive set by f and a. In the output entry we introduce the function call $f(a)$. The free variable z occuring in the goal $Q(a, z)$ stands for $f(a)$. So if we proceed by applying deduction rules (see next section) we obtain new rows in which the variable z becomes incrementally instantiated. The process continues until we obtain one of the following rows:

(Final tableau)

false		$t(a)$

or

	true	$t(a)$

where $t(a)$ is ground primitive term depending on a. We obtain $f(a) = t(a)$.

2.2.4 Deduction Rules

Now we come to the kernel of the deductive tableaux system, namely its calculus. The rules constituting Manna and Waldinger's Calculus can be roughly divided into in the following tree groups

- The *nonclausal resolution* rule, which performs a case analysis on the truth of a subsentence. This rule applies a *simplifier* in order to rewrite the sentence obtained as result of the case analysis.
- The equality rule, which is used to prove of sentences in a theory with equality.
- The *skolemization* rules, which eliminate quantifiers from the assertions or goals.
- The *induction* rule, which allows reasoning about recursive defined objects of the underlying theory.

In the remainder of this section we will describe these deduction rules, giving some of their derived rules, at this occusion.

2.2.4.1 Simplification

Simplification allows to replace every occurrence of a subexpression of an assertion, goal and an output entry by a logically equivalent, "simpler" expression. Manna and Waldinger's simplifier contains a set of rewriting rules including the *true-false* simplification rules such as $f \wedge true \Rightarrow f$ as well as others from the corresponding background theory, such as $x + 0 \Rightarrow x$ in the theory of positive integer numbers. By applying simplification during a proof in the tableaux system no new rows will be generated. Furthermore their applications preserve the relation "primitively similar" of the tableaux, so they are sound. A complete listing of the logical simplification rules can be found in appendix A.3.

2.2.4.2 Nonclausal Resolution

The main deduction rule of the tableaux system is a *nonclausal resolution* rule. The version that we present here is different from the original one formulated by Robinson in [49] that requires sentences in conjunctive normal form. The rule proposed here corresponds to a case analysis and each application of it introduces a conditional term in the output entries.

Rule (AA-resolution[2])

Assertions	*Goals*	*Outputs*
$A_1[P]$		s
$A_2[Q]$		t
$A_1\theta[false] \ \vee \ A_2\theta[true]$		$if\ P\theta\ then\ t\theta\ else\ s\theta$

Thereby the following conditions will be satisfied.

Side-Conditions:

1. A_1 and A_2 has no variables in common.
2. P and Q are quantifier-free and unifiable.
3. θ is the most-general unifier of P and Q.

■

Meaning of the rule AA-resolution This rule allows us to add a new line to a tableau if it possible to unify two subsentences P and Q of two rows. The sentence of the new line is the substituted disjunction of the initial assertions, where the unifiable sub sentences are replaced respectively by *t*rue and *f*alse, respectively. The output of the new rows contains a conditional term, whose test-part is the sentence $P\theta$ and the then-part corresponds to the case of a true $P\theta$, that is $s\theta$, and the else-part corresponds the case of a false $P\theta$, that is $t\theta$. The application of the resolution rule usually leads to sentences that can be simplified using the simplifier presented in section 2.2.4.1.

[2] AA stands for assertion-assertion

Correctness of the rule AA-resolution We assume that the assertions A_1 and A_2 are two valid sentences in the theory TH and P and Q are quantifier-free and unifiable. Let θ be the most-general unifier of them and $\mathcal{I}$ a model for the theory TH, consequently, $A_1\theta$ and $A_2\theta$ are true under $\mathcal{I}$.
We have to consider two cases:

- $P\theta$ is true under $\mathcal{I}$
  $A_2\theta[true] = A_2\theta[P\theta] = A_2\theta$ and therefore is true under $\mathcal{I}$ and so
  $A_1\theta[false] \vee A_2\theta[true]$ is true under $\mathcal{I}$
- $P\theta$ is false under $\mathcal{I}$
  Analogous to above, we prove that $A_1\theta[false] \vee A_2\theta[true]$ is true under $\mathcal{I}$.

Polarity Strategy Polarity is a syntactic condition, that will allow us to avoid many fruitless applications of the resolution rule.

Definition 2.12 (Polarity Strategy)
An application of the nonclausal resolution is in **accordance with the polarity strategy**, if at least one negative occurrence of the unified subsentences P is replaced by *f*alse and at least one positive occurrence of the unified sentences Q is replaced by *t*rue. □

Remark

1. The system is still complete.
2. Violating the polarity strategy leads not necessaryily to deriving trivial rows however, the proofs will be longer.

□

Derived rules According to the duality property of the tableaux between assertions and goals we can derive analogous, dual versions of the AA-resolution rule such as GG-, AG- and GA-resolution rule. Other derived rules are the three splitting rules, *i*fsplit, *a*ndsplit and *o*rsplit. They are, of course, redundant, but they clarify the presentation of the proofs by decomposing an assertion or a goal in its logical components. A complete list of these rules is given in appendix A.1

2.2.4.3 The Equality rule

In order to prove validity of sentences in a theory with equality, we have to add equality rules. A naive way to do this is to give the following equality axioms:

- reflexivity
  $\forall\, x.\ x = x$
- symmetry
  $\forall\, x\, y.\ \mathit{if}\ x = y\ \mathit{then}\ y = x$

- transitivity
  $\forall\, x\, y\, z.\ if\ x = y\ and\ y = z\ then\ x = z$
- function-substitution
  $\forall\, x_1 \ldots x_n.\ if\ x_i = y\ then\ f(x_1, \ldots, x_i, \ldots, x_n) = f(x_1, \ldots, y, \ldots, x_n)$
- predicate-substitution
  $\forall\, x_1 \ldots x_n.\ if\ x_i = y\ then\ P(x_1, \ldots, x_i, \ldots, x_n) = P(x_1, \ldots, y, \ldots, x_n)$

The substitution axioms have to be included for each function symbol and for each predicate symbol. This approach is inefficient and not practical. A more efficient way to do this is to use the **paramodulation**.
The rule suggested by Manna and Waldinger here is a nonclausal version of the paramodulation rule (Robinson and Wos 1969 see [48]). This rule discards all the equality axioms, except for the reflexivity axiom.

Rule (AA-equality)

Assertions	*Goals*	*Outputs*
$A_1[r = l]$		s
$A_2 < l' >$		t
$A_1\theta[false] \vee A_2\theta < l'\theta >$		$if\ (r = l)\theta\ then\ t\theta\ else\ s\theta$

Thereby the following conditions will be satisfied.

Side-Conditions:

1. A_1 and A_2 has no variables in common and $r = l$ is a subsentence of A_1
2. r and l' are unifiable with most general unifier θ.

■

Meaning of the equality rule The rule can be applied if a term r, which occurs in the first assertion as a part of an equality $r = l$, and an other term l', which occurs in the second assertion of the tableau, are unifiable. The result is a substituted disjunction of the two assertions: in the first one the equality $r = l$ is replaced by *false* , and in the other one some of the occurrence of the term l' is substituted by l. Analogous rules can be formulated for the other combinations of assertions and goals.

Polarity strategy An application of the equality rule is in accordance with the polarity strategy if at least one negative occurrence of $r = l$ is replaced by *false*.

2.2.4.4 Quantifier Elimination Rules

We have seen that the application of each defined as well as derived *nonclausal resolution* rule requires that the considered subsentences are quantifier-free. To achieve this we must be able to drop quantifiers from assertions and goals of the tableau.

$\forall$-Elimination rules Remove a quantifier of strict universal force in the tableau.

Rule (Assertion-form)

Assertions	*Goals*	*Outputs*
$A[(\forall(\exists))^{\forall}\, x.\ P(x)]$		s
$A[P(f(y,z))]$		s

Side-Conditions:

1. A contains at least one occurrence of the subsentence $\forall(\exists)\, x.\, P(x)$[3], which is of strict universal force in the the tableau.
2. y is a vector formed from all free variables of A.
3. z all quantifiers of strict existential force surrounded the occurrence $\forall(\exists)\, x.\, P(x)$.
4. f is a new skolem function.

■

For the Goal-form we have the same way to generate the new row with the same restrictions.

$\exists$-Elimination rules Remove a quantifier of strict existential force in the tableau.

Rule (Assertion-form)

Assertions	*Goals*	*Outputs*
$A[(\forall(\exists))^{\exists}\, x.\, P(x)]$		s
$A[P(z)]$		s

Side-Conditions:

[3] $\forall(\exists)\, x.\, P(x)$ stands for $\forall\, x.\, P(x)$ or also $\exists\, x.\, P(x)$

1. A contains at least one occurrence of the subsentence $\forall(\exists)\, x.\, P(x)$, which is of strict existential force in the tableau.

2. The occurrence of $\forall(\exists)\, x.\, P(x)$ in A is not within the scope of any quantifier of universal force.

3. z is a new variable in the tableau.

■

The Goal-form is as above analogous to the Assertion-form.

2.2.4.5 Induction Rule

This rule is of special importance for the deductive system. It is intended to introduce recursive calls into the target programs.

Rule (induction rule)

Assertions	*Goals*	*Outputs*
	$Q(a, z)$	z
$if\ (x <_w a)\ then\ Q(x, f(x))$		z

■

Meaning of the Rule In attempting to prove the sentence $Q(a, f(a))$ for a arbitrary element a of some well-founded set, we inductively assume that the sentence holds for all x strictly smaller than a in the well-founded ordering $<_w$.

Recursion Formation We note that the induction hypothesis contains occurrences of the function symbol f, which denotes the function that we are trying to compute. If the induction hypothesis is used in the proof, it can happen that terms of the form $f(t)$ will be introduced in the output entries, therefore we obtain recursive calls in the final program.

Choice of the Well-found Relation A number of well-founded relations are given in advance. Typical ones are the **less-than** relation over natural numbers, **proper substring** relation, and the order defined in terms of the function **length** over lists.

2.2.5 Example

We give here an example from the theory of strings. Strings are built using the basic cons-operator "." and the empty string denoted by ϵ. So $c.s$ is the string s affixed with the character c. The first character of a string s is denoted by $hd(s)$. All but the first character of a string s is represented by $tl(s)$. With the symbol function $*$ we represent the concatenation operator over strings.

Additional consider the predicate *char* on non-empty strings which has the following meaning : *char*(s) is *true* if s consists of a single character. Assume now that a non empty string s is given and we want to construct two functions: *last* and *front*. The first returns the last character of the string s and the second returns all but the last character of s. As specification we have the following formula

$$\forall s.\ \exists s_1 s_2.\ \ \mathit{if}\,(s \neq \epsilon)\ \mathit{then}\ \ \mathit{char}(s_2)\ \wedge\ (s = s_1 * s_2)$$

2.2.5.1 Derivation of Front and Last

- **The initial Tableau** We start the tableau proof by normalizing the input specification. We bind the variable s to a new skolem constant a and the variables s_1 and s_2 becomes the terms *front*(a) and *last*(a), respectively, where *front* and *last* are both new skolem functions. In the output entries we introduce the function calls *front*(a) and *last*(a). In the goal we replace both these terms with variables z_1 and z_2. The tableau thus has the following form

label	*Assertions*	*Goals*	*Outputs*	
			front(a)	*last*(a)
1		$\mathit{if}\,(a \neq \epsilon)\ \mathit{then}$ $\mathit{char}(z_2)\ \wedge\ a = z_1 * z_2$	z_1	z_2

- **Induction** Now intuitively the functions that we expect to compute will be recursive on its unique argument a. Therefore we apply at this stage the induction rule to the goal. This means to introduce a new assertion, called induction hypothesis, into the tableau. Note that we propose the proper substring order as well-founded relation. We get the new row[4]

2	$\mathit{if}\,(x <_{string} a)\ \mathit{then}$ $\mathit{if}\,(x \neq \epsilon)\ \mathit{then}$ $\boxed{\mathit{char}(\mathit{last}(x))}^{-}\ \wedge\ x = \mathit{front}(x) * \mathit{last}(x)$			

  To clarify the proof we first decompose the goal using the rule *ifsplit* in an assertion and a goal.

3	$a \neq \epsilon$			
4		$\mathit{char}(z_2)\ \wedge\ a = z_1 * z_2$	z_1	z_2

[4]We use $\boxed{formula}^{+}$ (or, respectively, $\boxed{formula}^{-}$) to denote the polarity of the formula *formula*

- **Case analysis**
  Base case : We consider the first case in which a consists only of one character.

5	$\boxed{\epsilon * w}^{-} = w$			
6		$char(z_2) \wedge \boxed{a = z_2}^{+}$	ϵ	z_2
7	$\boxed{x = x}^{-}$			
8		$char(a)$	ϵ	a

The rows 5 and 7 are assertions and therefore have negative polarity (Furthermore these assertions are valid sentences in the theory of strings.) We proceed first by applying the *equality* rule to the assertions of the row 5 and the goal in row 4. So we unify the formulas $w * \epsilon$ and $z_1 * z_2$ and we obtain thereby the following unifier: $\{z_1 \leftarrow \epsilon,\ w \leftarrow z_2\}$. In the new goal we replace $w * \epsilon = w$ by *false*, because this assertion has a negative polarity (the application of the *equality* rule was in accordance with the polarity strategy). So we get $(char(u)\ \wedge\ a = z_2)\ \wedge\ \neg false$ which can be simplified to the goal in row 6. Similarly by applying the *AG-resolution* to the last two rows, with the most-general unifier $\{x \leftarrow a, z_2 \leftarrow a\}$, we obtain row 8.
Recursive case: In order to apply the induction hypothesis we decompose the string a in two strings in such a way that one of the two obtained strings will be a proper substring of a. For this purpose we introduce an axiom of concatenation of strings in row 9, then we apply the *equality* rule to this axiom and row 4. The result of the resolution is row 10.

9	*if* $char(u)$ *then* $\boxed{(u.y_1) * y_2}^{-} = u.(y_1 * y_2)$			
10		$char(u)\ \wedge\ char(y_2)\ \wedge$ $a = u.(y_1 * y_2)$	$u.y_1$	y_2

Now we apply twice the induction hypothesis, at first the *equality* rule to row 2 and row 10 to obtain row 11, secondly *AG-resolution* again to row 2 and row 11.

11		$(x <_{string} a) \wedge (x \neq \epsilon) \wedge$ $char(u)\ \wedge\ \boxed{char(last(x))}^{+}$ $a = u.x$	$u.front(x)$	$last(x)$
12		$(x <_{string} a) \wedge (x \neq \epsilon) \wedge$ $char(u)\ \wedge\ \boxed{a = u.x}^{+}$	$u.front(x)$	$last(x)$

Now we must instantiate the variables x and u , this is done by introducing the *decomposing property* of string in row 13, then we apply the *AG-resolution* to the rows 12 and 13, with the most-general unifier $\{w \leftarrow a, u \leftarrow hd(a), x \leftarrow tl(a)\}$, this leads to row 14.

13	$\mathit{if}\ (w \neq \epsilon)\ \mathit{then}$ $\boxed{w = hd(w).tl(w)}^{-}$			
14		$(tl(a) <_{string} a) \wedge (tl(a) \neq \epsilon)$ $\wedge$ $char(hd(a)) \ \wedge\ a \neq \epsilon$	$hd(a).front(tl(a))$	$last(tl(a))$

- **Final steps** The rest of the derivation consists of proving that the sentence of the row 14 can be transformed to *true* by applying some resolution rules and introducing some valid sentences of the theory of strings. (For more details see the complete derivation in the appendix A.4)
  The program we extract from the proof is hence

$$front(a) = \begin{cases} \mathit{if} & char(a) \\ \mathit{then} & \epsilon \\ \mathit{else} & hd(a).front(tl(a)). \end{cases}$$

$$last(a) = \begin{cases} \mathit{if} & char(a) \\ \mathit{then} & a \\ \mathit{else} & last(tl(a)). \end{cases}$$

2.3 Semantic Tableaux

The sequent calculus developed by Gentzen [21] is a positive calculus, it starts from certain formulas, axioms known to be logical tautologies, and derives by applications of inference rules, which preserve logically tautologies, other complex tautologies. Smullyan's system is based on a refutation calculus, that constructs a counter-model of the negation of a given formula. Smullyan proceeds as follows: one begins with the negation of a formula and attempts to produce a contradiction by applying the inference rules of Gentzen upside-down. The proofs can be represented as *trees*, where nodes of the trees are labeled with formulas. Each *branch* of such a tree should be thought of as representing the conjunction of the formulas appearing in it, and the tree itself as representing the disjunction of its branches. A branch is called **closed** if some formulas X and its negation, $\neg X$, occur in it. A tree is called closed, if every branch of it is closed. In the next section we shortly give Smullyan calculus and we illustrate it by an example. For more details about semantic tableaux and the close relationship between tableaux, and the sequence calculus proofs,the book of M. Fitting [20] can be consulted.

$$\frac{S\ ,\ F_{(\varphi\wedge\psi)}}{S\ ,\ F_{\varphi}\ \mid\ S\ ,\ F_{\psi}}\,(F\wedge) \qquad \frac{S\ ,\ T_{(\varphi\wedge\psi)}}{S\ ,\ T_{\varphi}\ ,\ T_{\psi}}\,(T\wedge)$$

$$\frac{S\ ,\ F_{(\varphi\vee\psi)}}{S\ ,\ F_{\varphi}\ ,\ F_{\psi}}\,(F\vee) \qquad \frac{S\ ,\ T_{(\varphi\vee\psi)}}{S\ ,\ T_{\varphi}\ \mid\ S\ ,\ T_{\psi}}\,(T\vee)$$

$$\frac{S\ ,\ F_{(\varphi\rightarrow\psi)}}{S\ ,\ T_{\varphi}\ ,\ F_{\psi}}\,(F\rightarrow) \qquad \frac{S\ ,\ T_{(\varphi\rightarrow\psi)}}{S\ ,\ F_{\varphi}\ \mid\ S\ ,\ T_{\psi}}\,(T\rightarrow)$$

$$\frac{S\ ,\ F_{\neg\varphi}}{S\ ,\ T_{\varphi}}\,(F\neg) \qquad \frac{S\ ,\ T_{\neg\varphi}}{S\ ,\ F_{\varphi}}\,(T\neg)$$

$$\frac{S\ ,\ F_{\forall x.\varphi}}{S\ ,\ F_{\varphi^x_a}}\,(F\forall)^{(*)} \qquad \frac{S\ ,\ T_{\forall x.\varphi}}{S\ ,\ T_{\forall x.\varphi}\ ,\ T_{\varphi^x_t}}\,(T\forall)$$

$$\frac{S\ ,\ F_{\exists x.\varphi}}{S\ ,\ F_{\exists x.\varphi}\ ,\ F_{\varphi^x_t}}\,(F\exists) \qquad \frac{S\ ,\ T_{\exists x.\varphi}}{S\ ,\ T_{\varphi^x_a}}\,(T\exists)^{(*)}$$

(*) : provide that a does not occur in S

Figure 2.1: Tableau Expansion Rules

2.3.1 Calculus

To translate the sequent calculus Smullyan has introduced two symbols T and F, called **signs**. A signed formula is of the form T_φ or F_φ, where φ is a first-order formula. Intuitively, in a given model, we think of T_φ as an assertion that φ is true and F_φ as an assertion that φ is false. Figure 2.1 contains the inference rules of semantic tableaux. There S denotes a finite set of signed formulas. Further we write $S,\ \phi$ for $S \cup \{\phi\}$ and φ^x_t to denote the formula φ where the variable x is instantiated by the term t. The operator "|" for a set of signed formulas gives the two possibles sets (called also branches) that must be proved to be closed.

2.3.2 Example

As an example we will prove the validity of the formula

$$s\ \equiv\ \forall x\, y.\ Q(x,y) \rightarrow \exists z_1\, z_2.\ Q(a,z_1)\ \wedge\ Q(z_2,b)$$

$$
\begin{tikzcd}[column sep=small]
& F_{\forall xy.\ Q(x,y)\ \rightarrow\ \exists z_1 z_2.\ Q(a,z_1) \wedge Q(z_2,b)} \arrow[d, "\mathbf{(1)}"] & \\
& T_{\forall xy.\ Q(x,y)}\ ,\ F_{\exists z_1 z_2.\ Q(a,z_1)\ \wedge\ Q(z_2,b)} \arrow[d, "\mathbf{(2)}"] & \\
& F_{\exists z_1 z_2.\ Q(a,z_1)\ \wedge\ Q(z_2,b)}\ , T_{\forall xy.\ Q(x,y)}\ , T_{\forall y.\ Q(u,y)} \arrow[d, "\mathbf{(3)}"] & \\
& F_{\exists z_1 z_2.\ Q(a,z_1)\ \wedge\ Q(z_2,b)}\ , T_{\forall xy.\ Q(x,y)}\ , T_{\forall y.\ Q(u,y)}\ , T_{Q(u,v)} \arrow[d, "\mathbf{(4)}"] & \\
& \cdots\ , F_{Q(a,w_1)\ \wedge\ Q(w_2,b)} \arrow[dl, "\mathbf{(5)}"'] \arrow[dr, "\mathbf{(5)}"] & \\
\ldots, F_{Q(a,w_1)} & & \ldots, F_{Q(w_2,b)}
\end{tikzcd}
$$

Figure 2.2: A semantic tableau proof of $\forall xy.\ Q(x,y) \rightarrow \exists z_1 z_2.\ Q(a,z_1) \wedge Q(z_2,b)$

Semantic tableaux rules	*Deductive tableau rules*
$F \rightarrow$	*ifsplit*
$T\wedge$	*andsplit*
$F\vee$	*orsplit*
$F\neg$	*duality principal*
$T\neg$	*duality principal*
Skolemization rules	*Skolemization rules*

Figure 2.3: Correspondence of some semantic tableaux rules and deductive tableaux rules

in first-order logic. For this purpose we affix the formula s with the sign F, then we apply the rules of semantic tableaux to the set $\{F_{\forall x\, y.Q(x,y) \rightarrow \exists z_1\, z_2.Q(a,z_1) \wedge Q(z_2,b)}\}$. Figure 2.2 contains a proof of s. (1) is the application of the rule $(F \rightarrow)$ to the initial set giving the set of formulas listed in line 2. The steps (2) and (3) eliminate the universal quantifiers thereby we have substitute the variables x and y with the new variables u and v. By applying twice the the existence quantifier elimination in (4) we have substituted the variables z_1 and z_2 by the new ones w_1 and w_2 and extended thereby the set of the signed formulas with the following conjunction: $F_{Q(a,w_1)\ \wedge\ Q(w_2,b)}$. In (5) we have decomposed this conjunction into two formulas $F_{Q(a,w_1)}$ and $F_{Q(w_2,b)}$ creating thereby two branches labeled with the formulas of above. The first branch is closed, because we can unify $F_{Q(a,w_1)}$ and $T_{Q(u,v)}$ by the substitution $\{u \leftarrow a,\ v \leftarrow w_1\}$. The second branch is also closed, because the formulas $F_{Q(w_2,a)}$ and $T_{Q(u,v)}$ are unifiable by the substitution $\{u \leftarrow w_2,\ v \leftarrow b\}$. So the tree is also closed and the proof is completed.

2.4 Relation between Semantic Tableaux and Deductive Tableaux

If we ignore the output columns in the deductive rules of Manna and Waldinger (represented in section 2.2.4) we get a calculus that can be used to prove the validity of sentences in a given theory. In this section we will study informal the relationship between this calculus and the semantic tableaux calculus presented in the last section.

Although proofs are differently represented in both systems, semantic tableaux proofs are represented as trees and deduction proofs are represented as tableaux, we notice a great similarity between both calculi. Moreover we can give a one-to-one translation of some deduction rules into semantic tableaux rules Consider

label	*Assertions*	*Goals*	*Rules*
1		$\forall xy.\ Q(x,y) \ \rightarrow\ \exists z_1 z_2.\ \ Q(a,z_1)\ \wedge\ Q(z_2,b)$	
2	$\forall xy.\ Q(x,y)$		$\mathit{ifsplit}(1)$
3		$\exists z_1 z_2.\ \ Q(a,z_1)\ \wedge\ Q(z_2,b)$	
4	$\forall y.\ Q(u,y)$		$\forall\text{-}E(2)$
5	$\boxed{Q(u,v)}^{-}$		$\forall\text{-}E(4)$
6		$\exists z_2.\ \ Q(a,w_1)\ \wedge\ Q(z_2,b)$	$\exists\text{-}E(3)$
7		$\boxed{Q(a,w_1)}^{+}\ \wedge\ Q(w_2,b)$	$\exists\text{-}E(6)$
8		$\boxed{Q(w_2,b)}^{+}$	$AG(5,7)$
9		*true*	$AG(5,8)$

Figure 2.4: A deductive tableau proof of $\forall xy.\ Q(x,y) \rightarrow \exists z_1 z_2.\ Q(a,z_1)\ \wedge\ Q(z_2,b)$

for example the rule *ifsplit*. This rule has the following form

	$A\ \rightarrow\ B$
A	
	B

The first row is the required row and the both other are the generated ones. If we write the goals and assertions of the deduction rules in a labeled form using the signs F and T, respectively, we can transform the above rule to following one:

$$\frac{F_{A\rightarrow B}}{T_A\ ,\ F_B}$$

This rule corresponds to the semantic tableaux rule $(F \rightarrow)$ (see figure 2.1). In figure 2.3 we have displayed a list of further corresponding rules of the both systems. Now let us turn our attention to the decomposition rules of the semantic tableaux calculus,$(F\wedge)$, $(T\vee)$ and $(T\rightarrow)$ listed in figure 2.1. To these rules there is no corresponding deduction rules. In deed looking at the front/last example given in section 2.2.5 we remark that conjunctions of goals (or, respectively, disjunctions of assertions) are not decomposed and they can not be decomposed using Manna and Waldinger's calculus. To illustrate the meaning of such conjunctions (or, respectively, disjunctions) in the tableaux in accordance to the representations of semantic tableaux proofs, we go through a deduction proof of the example treated in section 2.3.2. Figure 2.4 shows a deduction proof of this example. Row 1 contains the formula that will be proved. Analogously to the semantic tableaux proof (see figure 2.2) we have applied the rule *ifsplit* to the first row yielding thereby row 2 and row 3. Than we have applied some *skolemization* rules to generate the rows 4, 5, 6 and 7. Thereby we have however introduced the new variables u, v, w_1 and w_2. The above steps are similar to the steps achieved in the semantic tableaux proof. Now looking at row 7 we find a conjunction of two formulas, the first formula is $Q(a, w_1)$ and the second one is $Q(w_2, b)$. These formulas correspond to the formulas labeling the branches of the tree given in figure 2.2. To close the first branch (labeled with the formula $Q(a, w_1)$) we have applied the *nonclausal resolution* rule to row 5 and row 7 and produced row 8. This last row contains the last branch that will be closed. So similarly to above we have applied the *nonclausal resolution* rule to row 5 and row 8 and produced thereby the final row 9.

By this means Manna and Waldinger's tableaux can be seen as trees, whose branches are assembled as conjunction of goals (or, respectively, disjunction of assertions). A tableau is a "linear" tree. The set of the signed formulas corresponding to this tree is the collection of all formulas occuring in the rows. To close a branch of this tree Manna and Waldinger make use of the *nonclausal resolution* rules. The desired branch can be selected as one formula of both required formulas by applying the *nonclausal resolution* rules. In our example, in the last but one step we have selected the branch labeled with $Q(a, w_1)$ occuring in the conjunction of row 5. The second formula required by the *nonclausal resolution* rules can also be a formula of an other row such as $Q(u, v)$, occuring in row 5, in the last but one step of our example above.

Chapter 3

Isabelle

This chapter introduces the theorem prover Isabelle together with its formalization of higher-order logic, in which we will translate the deductive tableaux system and develop some sorting programs.

3.1 Overview of Isabelle

Isabelle is a generic theorem prover developed at Cambridge by Larry Paulson. It is a logical framework, in which object-logics are introduced by specifying their syntax and axioms using a meta-logic. Theorems in object-logics can be derived interactively by applications of tactics, or composition of rules.

3.1.1 Meta-logic

Isabelle's *meta-logic* is a fragment of the intuitionistic higher-order logic (see [44]). Its terms are *typed* λ*-expressions*: *constants*, *variables*, *abstractions* and *applications*. Types are constructed using the built-in type, called *prop* and a *type-constructor*, represented by $\Rightarrow$, that takes two types and produces a new one. Furthermore, types can be polymorphic. In order to control the polymorphism Isabelle uses the notions of *sorts*, also called *classes*. These are sets of types partially ordered by the subset relation (details about class concepts can be found in [41, 43]). Two important built-in classes are *term* and *logic*. The former contains all types of terms (e.g. in first-order logic such types are *nat*, *list*, etc) and the latter contains logical types (for example the type "o" of first-order formulas). Intuitively the logical types are all those we want reason about. The class *term* is a subclass of the class *logic*. Meta-level formulas (we will call them *propositions* in our work) have the type *prop*. They are constructed using the following built-in connectives

$$\begin{aligned}
\Longrightarrow &:: [prop,\ prop] \Rightarrow prop \\
\bigwedge &:: (\alpha :: logic \Rightarrow prop) \Rightarrow prop \\
\equiv &:: [\alpha :: logic,\ \alpha] \Rightarrow prop
\end{aligned}$$

where

- :: denotes both the meta-type membership and the sort membership

- $\Longrightarrow$ denotes the implication and expresses entailment
- $\bigwedge$ denotes the universal quantifier and expresses the generality of propositions and axiom schemata.
- $\equiv$ is the equality and needed to express definitions

The meta-predicate $Trueprop :: \alpha \Rightarrow prop$ coercs elements of judgments of the object-logics up to the propositions in the meta-logic. Intuitively $Trueprop(\phi)$ expresses that the object-level formula ϕ is derivable in its object-logic.

3.1.2 Meta-Rules

The connectives $\Longrightarrow$, $\bigwedge$ and $\equiv$ are defined by a collection of inference rules in a natural deduction style calculus. As an example the meta-implication rules are $\Longrightarrow I$ (I stands for introduction) and $\Longrightarrow E$ (E stands for elimination),

$$\frac{\begin{array}{c}[\phi]\\ \vdots\\ \psi\end{array}}{\phi \Longrightarrow \psi}\,(\Longrightarrow I) \qquad\qquad \frac{\phi \Longrightarrow \psi \qquad \phi}{\psi}\,(\Longrightarrow E)$$

which resemble $\rightarrow I$ and $\rightarrow E$ in first-order logic. The other rules are defined in the same manner. The rule calculus of an object-logic is encoded as a set of meta-propositions using the coercion function $Trueprop$. For example the modus ponens rule, $\rightarrow E$, in first-order logic can as the following proposition be presented

$$Trueprop(\phi \rightarrow \psi) \Longrightarrow (Trueprop(\phi) \Longrightarrow Trueprop(\psi))$$

In the following we may omit the occurrences of $Trueprop$ in the rules to clarify the representation. Proving an object-level theorem is done by proving the corresponding proposition in the meta-logic using the primitive meta-rules described above and the rule calculus of the corresponding object-logic. The initial proofstate of an object-level theorem ϕ is the trivial proposition $\phi \Longrightarrow \phi$ and the final proofstate of the theorem ϕ is the proposition ϕ. Isabelle's rules have the form of Horn clause

$$\phi_1 \Longrightarrow (\ldots \phi_n \Longrightarrow \phi \ldots)$$

They can be abbreviated as

$$[\![\phi_1;\ \ldots;\ \phi_n]\!] \Longrightarrow \phi$$

Further, Isabelle allows the derivation of schematic rules. These rules are first, formulated as goals with free variables in the corresponding theory. Then they will be derived and during their proofs the free variables are treated as constants. When the proofs are completed these free variables are displayed with preceding question mark indicating that they are schematic and can be subsequently instantiated with concrete objects of the corresponding object-logic to build special rules. If we want for example derive the following more general form of the modus ponens rule in first-order logic

$$\frac{P \to Q \qquad P \qquad \begin{matrix}[Q]\\ \vdots \\ R\end{matrix}}{R}\,(impE)$$

we first state the goal

$$[\![P \to Q;\ P;\ Q \Longrightarrow R]\!] \Longrightarrow R$$

and prove it using first-order rules. The resulting theorem is displayed by the system as follows

$$[\![?P \to ?Q;\ ?P;\ ?Q \Longrightarrow ?R]\!] \Longrightarrow ?R$$

In the rest of our work (except the following two paragraphs) we will not display question marks in the rules and their free variables implicitly are schematic variables. Proofs performed using primitive rules are length and hard to understand. To clarify the representation of proofs Isabelle provides derived rules and allows the use of tactics, which are ML-programs[1] that can be combined into more complex form. The **resolution** rule is the most important derived rule of Isabelle. This rule can be explained as follows: Suppose we are proving a proposition ϕ and the current proofstate is has the form

$$[\![\phi_1;\ \ldots;\ \phi_n]\!] \Longrightarrow \phi$$

where the propositions $\phi_1,\ \ldots,\ \phi_n$ are the **subgoals** of the proofstate. Our task is to solve all these subgoals. Suppose further that proving some subgoal ϕ_i could be reduced to proving some other propositions $\psi_1,\ \ldots,\ \psi_m$. This can be expressed in the rule

$$[\![\psi_1;\ \ldots;\ \psi_m]\!] \Longrightarrow \phi_i$$

In this situation we remove the subgoal ϕ_i from the proofstate and replace it by the new subgoals $\psi_1;\ \ldots;\ \psi_m$. This can generally be formalized as follows

$$\frac{[\![\psi_1;\ \ldots;\ \psi_m]\!] \Longrightarrow \psi \qquad [\![\phi_1;\ \ldots;\ \phi_n]\!] \Longrightarrow \phi}{([\![\phi_1;\ \ldots;\ \phi_{i-1};\ \psi_1;\ \ldots\psi_m;\ \phi_i;\ \ldots\phi_n]\!] \Longrightarrow \phi)s}\,(resolution)$$

The first premise of *resolution* is the rule that is to be applied. The second premise is the current proofstate and the conclusion denotes the new proofstate, where s is a substitution that unifies the propositions ψ and ϕ_i. Note that Isabelle uses higher-order unification which is an incomplete procedure and may yield an infinite set of unifiers; therefore the resolution returns a sequence (lazy list) of new proofstates.

[1] ML stands for the functional programming language ML

3.1.3 Lifting

The resolution rule raises two principal problems. First, in some applications of the resolution we are obliged to unify atomic propositions with non-atomic ones. To illustrate this consider the following example. We want resolve the first premise of the proposition

$$[\![A \Longrightarrow B; C]\!] \Longrightarrow D$$

against the modus ponens rule

$$[\![?P \rightarrow ?Q;\ ?P]\!] \Longrightarrow ?Q$$

If we try to unify the conclusion $?Q$, which stands for the atomic proposition $\mathit{Trueprop}(?Q)$, with the assumption $A \Longrightarrow B$, which stands for the non atomic proposition $\mathit{Trueprop}(A) \Longrightarrow \mathit{Trueprop}(B)$, will fail. The solution suggested in Isabelle is to use a meta-inference rule, called **lifting**. Thus the first step is to **lift** the modus ponens rule **over the assumption** A to the following new rule

$$[\![A \Longrightarrow (?P \rightarrow ?Q); A \Longrightarrow ?P]\!] \Longrightarrow (A \Longrightarrow ?Q)$$

Then we can apply the resolution rule getting the new proposition

$$[\![A \Longrightarrow (?P \rightarrow B); A \Longrightarrow ?P;\ C]\!] \Longrightarrow D$$

The next problem does concern metavariables. Suppose we want resolve the proposition

$$[\![\bigwedge x.\ P(x) \rightarrow Q(x)]\!] \Longrightarrow \forall x.\ P(x) \rightarrow Q(x)$$

against the implication introduction rule

$$[\![?P \Longrightarrow ?Q]\!] \Longrightarrow (?P \rightarrow ?Q)$$

For this purpose we **lift** the above rule **over the parameter** x, this introduces a dependence upon x. We obtain the following new rule

$$[\![\bigwedge x.\ ?P_1(x) \Longrightarrow ?Q_1(x)]\!] \Longrightarrow \bigwedge x.\ ?P_1(x) \rightarrow ?Q_1(x)$$

which can now be successfully applied to refine the first subgoal of our initial proposition. Thus the result is

$$[\![\bigwedge x.\ P(x) \Longrightarrow Q(x)]\!] \Longrightarrow \forall x.\ P(x) \rightarrow Q(x)$$

3.1.4 Proof Construction

In the LCF-discipline [22] an inference rule is a function from theorems to theorems that maps premises to conclusion; its inverse is a function called tactic mapping a goal to subgoals. A tactic returns a “validation”: a function from theorems to theorems. While forward proofs are performed by applying inference rules, tactics are applied to perform backwards proof. These can be combinaed to form more powerful proof procedures by tecticals. Applications

of the validations translate the backwards proof to forwards proof. Note that the secure type checking of ML, the meta-language of LCF, ensures soundness, this means that theorems can be only constructed from other ones by rules.
Isabelle works differently. Here we have no distinction between inputs and outputs in a rule. As we have seen the derived rule Resolution is convenient to support backward proof, where we start from a goal and replace it with some simpler subgoals until all subgoals have been resolved. Isabelle supports also forward reasoning, in which certain rules have to be glued together and then applied by using resolution. In other words the application of rules is performed in an uniform way using the resolution rule. In the next example we show how can we use resolution for forwards reasoning.

3.1.4.1 Example of forward reasoning

We wish to resolve the subgoal

$$[\![\psi_1 \to (\psi_2 \to \psi_3);\ \psi_1;\ \psi_2]\!] \Longrightarrow \psi_3$$

of the following proofstate

$$[\![[\![\psi_1 \to (\psi_2 \to \psi_3);\ \psi_1;\ \psi_2]\!] \Longrightarrow \psi_3;\ \psi]\!] \Longrightarrow G,$$

this will be done by applying modus ponens twice

$$[\![?P \to ?Q;\ ?P]\!] \Longrightarrow ?Q$$

First we resolve modus ponens with itself yielding :

$$[\![?P \to (?P_1 \to ?Q);\ ?P;\ ?P_1]\!] \Longrightarrow ?Q$$

Now we can apply resolution[2] to this derived rule and the initial goal to refine the first subgoal. We obtain the following proposition

$$\begin{array}{l}
[\![\\
\quad [\![\psi_1 \to (\psi_2 \to \psi_3);\ \psi_1;\ \psi_2]\!] \Longrightarrow ?P \to (?P_1 \to ?Q) \\
\quad [\![\psi_1 \to (\psi_2 \to \psi_3);\ \psi_1;\ \psi_2]\!] \Longrightarrow ?P \\
\quad [\![\psi_1 \to (\psi_2 \to \psi_3);\ \psi_1;\ \psi_2]\!] \Longrightarrow ?P_1 \\
\quad \psi \\
]\!] \Longrightarrow G
\end{array}$$

Now the three first premises can be proved by assumption and we get finally $[\![\psi]\!] \Longrightarrow G$.

3.1.5 Some Tactics

For interactive systems like Isabelle, tactics are very helpful and desirable. They allow a clear representation of proofs and the possibility to develop complex procedures for certain proof-tasks such as backtracking and simplification. Isabelle provides tactics, that are functions coded in the programming-language

[2] Note that we have to lift the modus ponens rule over $\psi_1 \to (\psi_2 \to \psi_3) \Longrightarrow \psi_1 \Longrightarrow \psi_2$

Tactic call in Isabelle	**Abbreviation**	**Description**
by (`assume_tac` *i*)	`ba` *i*	Solve subgoal *i* by assumption.
`by (resolve_tac` *thms i*`)`	`br` *thm i*	Refine the subgoal *i* of the proof state by resolution using the first applicable rule of the rule list *thms*.
`by(eresolve_tac` *thms i*`)`	`be` *thm i*	The same as `eresolve_tac` with subsequent application of the assumption tactic to resolve the first premise of subgoal *i* and to additionally delete that assumption from all new subgoals arising from the resolution.
`by (res_inst_tac` *inst_list thms i*`)`		Apply the resolution tactic after a partial instantiation of some schematic variables occuring in the rules.

Table 3.1: Some Isabelle tactics

ML operating on the entire proof state of the actual proof. Tactics can be combined sequentially, as alternatives or repeatedly using the basic tacticals *THEN*, *ORELSE* and *REPEAT*. Isabelle offers some generic practical tactics for automatically resolving goals in object-logics. We will explain some of the ones we have used in the synthesis of some sorting programs (see Chapter 6 Examples).

- **fast_tac** is a classical theorem prover parameterized with a set, called **classical set**, of introduction rules and elimination rules for the corresponding object-logic.
- **simp_tac, asm_simp_tac** and some other constitute the generic simplification package of Isabelle. They provide conditional and unconditional rewriting and the use of local assumptions (as by `asm_simp_tac`). The simplification tactics are controlled by sets, called **simpsets**, of rewrite and congruence rules. Theorems can be added to simple sets or delete from them to make the simplification more efficient or to filter out looping.

Further useful tactics are listed in table 3.1.

3.2 The Encoding of Higher Order Logic

Several object-logics are implemented in Isabelle. They include first-order logic **FOL**, set theory **ZF** based on the Zermelo-Fraenkel axioms, higher order logic

HOL and any other logics. As an illustration of encoding object-logics in Isabelle we briefly present here the implementation of HOL that comes with Isabelle. A full description of higher order logic can be founded in Andrews book [1] and in [10]. At this place we will omit many details and [46] should be consulted for a full account of Isabelle.
In Isabelle, object-logics are theories. A **theory** consists of a **signature**, which is a collection of declarations of classes, types and constants, plus a list of axioms. The encoding of an object-logic is accomplished by extending Isabelle's meta-logic with new axioms. Theories can be combined by merging their signatures provided, they are compatible, and building the concatenation of their axiom lists. The **Pure** theory only contains the meta-logic. The encodings in Isabelle are represented in the following sections in `typewriter` font. Those symbols for which there is no transliteration are translated as follows:

$$\lambda \ becomes \ \%$$
$$\equiv becomes ==$$
$$\bigwedge becomes \ !!$$

3.2.1 HOL

Isabelle's HOL has three primitives: implication $\rightarrow$, equality $=$, and Hilbert's description $\epsilon x.\ \phi(x)$. The connectives $\wedge$, $\vee$, $\neg$, and the quantifiers $\forall$, $\exists$ and some other are defined in terms of the primitives. Formulas are terms of the type *bool* and are constructed from the logical constants and quantifiers. Note that there is no $\leftrightarrow$ connective. Biconditional terms are built using the equality symbol. Figure 3.1 gives the concrete Isabelle syntax of some connectives and quantifiers. HOL derives natural deduction rules for the connectives and quantifiers. In figure 3.2 is a listing of some definitions and rules with their names. The meaning of the rules is obvious and their ML names are standard (a name suffixed with `I` denotes an introduction rule and a name suffixed with `E` denotes an elimination rule). The rule `selectI` gives the definition of the Hilbert ϵ-operator; `@x.P(x)` stands for some a that satisfies the predicate $P(a)$, if such exists.

&	in place of	$\wedge$
|	in place of	$\vee$
-- >	in place of	$\rightarrow$
`! x. P`	in place of	$\forall x.\ P(x)$
`? x. P`	in place of	$\exists x.\ P(x)$
`@x. P(x)`	in place of	$\epsilon x.\ P(x)$

Figure 3.1: Syntax of Logical constants and quantifiers in HOL

<table>
<tr><td>name</td><td>definition</td></tr>
<tr><td><code>True_def</code></td><td><code>True == ((%x::bool.x)=(%x.x))</code></td></tr>
<tr><td><code>All_def</code></td><td><code>All(P) == (P = (%x.True))</code></td></tr>
<tr><td><code>Ex_def</code></td><td><code>Ex(P) == P(@x.P(x))</code></td></tr>
<tr><td><code>False_def</code></td><td><code>False == (!P.P)</code></td></tr>
<tr><td><code>if_def</code></td><td><code>if(P,x,y) ==z::'a.</code>
<code>(P=True --> z=x)& (P=False --> z=y)</code></td></tr>
</table>

<table>
<tr><td>name</td><td>basic rule</td></tr>
<tr><td><code>selectI</code></td><td><code>P(x::'a) ==> P(@x.P(x))</code></td></tr>
<tr><td><code>impI</code></td><td><code>(P ==> Q) ==> P-->Q</code></td></tr>
<tr><td><code>mp</code></td><td><code>[| P-->Q; P |] ==> Q</code></td></tr>
</table>

<table>
<tr><td>name</td><td>Axiom</td></tr>
<tr><td><code>True_or_False</code></td><td><code>(P=True) | (P=False)</code></td></tr>
</table>

<table>
<tr><td>name</td><td>derived rule</td></tr>
<tr><td><code>TrueI</code></td><td><code>True</code></td></tr>
<tr><td><code>FalseE</code></td><td><code>False ==> P</code></td></tr>
<tr><td><code>notI</code></td><td><code>(P ==> False) ==>   P</code></td></tr>
<tr><td><code>notE</code></td><td><code>[| ~P; P |] ==> R</code></td></tr>
<tr><td><code>conjI</code></td><td><code>[| P; Q |] ==> P & Q</code></td></tr>
<tr><td><code>conjE</code></td><td><code>[| P & Q; [| P; Q |] ==> R |] ==> R</code></td></tr>
<tr><td><code>disjI1</code></td><td><code>P ==> P | Q</code></td></tr>
<tr><td><code>disjI2</code></td><td><code>Q ==> P | Q</code></td></tr>
<tr><td><code>disjE</code></td><td><code>[| P | Q; P ==> R; Q ==> R |] ==> R</code></td></tr>
<tr><td><code>impE</code></td><td><code>[| P --> Q; P; Q ==> R |] ==> R</code></td></tr>
<tr><td><code>allI</code></td><td><code>(!!x. ?P(x)) ==> ! x. P(x)</code></td></tr>
<tr><td><code>spec</code></td><td><code>! x. ?P(x) ==> ?P(?x)</code></td></tr>
<tr><td><code>allE</code></td><td><code>[| ! x. P(x); P(x) ==> R |] ==> R</code></td></tr>
<tr><td><code>exI</code></td><td><code>P(x) ==> ?x. P(x)</code></td></tr>
<tr><td><code>exE</code></td><td><code>[|?x. P(x); !!x. P(x) ==> Q |] ==> Q</code></td></tr>
</table>

Figure 3.2: Some Definitions and derived Rules of HOL

3.2.1.1 Types in HOL

In the following section the Greek letters α, β, ... are type variables ranging over the HOL class `term`. Isabelle's HOL supports besides the type `bool` further types. The Cartesian product type $\alpha \times \beta$ has as values pairs $< x,\ y >$, for x of type α and y of type β. Sets are essentially predicates; the type $\alpha\ set$[3] possesses values having the following syntactic form $\{x \mid \ \phi(x)\}$, for some predicate $\phi(x)$. Inductive data types are defined using the **Knaster-Tarski Theorem** [4]. They are least sets under given functions ranging over sets. One advantage from such definitions is that the least fixedpoints satisfies a general induction principle that can in a very simple way be specialized to structural induction rules for those types(e.g. natural numbers, lists, trees, etc).

3.2.1.2 Examples

We shall go through an interactive proof of a derived rule to give a feel of proof construction. All proofs presented in this section are, with the exception of minor pretty printing, taken directly from an Isabelle session. The examples that we consider here are very simple and can also be proved in first-order logic. As a first example we prove the most general form of the basic rule `mp` (see figure 3.2). We begin with stating the following goal in the theory HOL to the system.

```
- val [assum1, assum2, assum3] = goal HOL.thy "[|P-->Q; P; Q ==> R|]==> R";
R
 1. R
```

We have typed the first and second lines. The two others are system output. The first line's output of the system is always the current goal. The subgoals are displayed with numbers indicating their order in the stack, where they are maintained. The variables `assum1`, `assum2` and `assum3` are bound to the assumptions of the goal. Now we proceed by resolving[5] the sole subgoal against the $assum3$.

```
- br assum3 1;
R
 1. Q
```

In the new subgoal `R` is replaced with `Q`, this is a typical backward proof step. In the next step we apply the axiom `mp`.

```
- br mp 1;
R
 1. ?P --> Q
 2. ?P
val it = () : unit
```

[3] α ranges over ordered types.

[4] the knaster-tarski theorem is implemented in Isabelle's HOL.

[5] `(br thm i)` is an abbreviation of `by(resolve_tac [thm] i)`. This tactic refines the subgoal of the proof state using a list of rules.

As result Isabelle prints two subgoals numbered with 1 and 2. The question-mark affixing `P` in both subgoals indicates that the unification treats `?P` as meta-variable (and not as constant as `Q`) and therefore it may be instantiated. Now the subgoals can be resolved against the assumptions `assum1`, `assum2`. This is done in one step using the tactical `THEN`.

```
- by ((resolve_tac [assum1] 1 ) THEN (resolve_tac [assum2] 1));
R
No subgoals!
- val impE =  result();
  val impE = "[| ?P --> ?Q; ?P; ?Q ==> ?R |] ==> ?R"
```

The proof is completed and the proved theorem is bound to the name `impE`. We derive a rule, which is very important for our program synthesis in Isabelle. This rule enables us to construct *if_then_else* terms. The rule is called `case_split` and has the following form :

$$\frac{\begin{array}{cc} [A] & [\neg A] \\ \vdots & \vdots \\ P(b) & P(s) \end{array}}{P(if(A,b,s))}\,(case\ split)$$

We start the proof of this rule by typing the goal

```
- val assums = goal thy "[| A==>P(b); ~A==>P(s)|]
          ==> P(if(A,b,s))";
```

The following step is a case analysis. For this purpose we have already developed a tactic called *cases_tac*, which takes a string s and a number i as parameters and splits the subgoal i into two subgoals. In the first of which we assume additionally that the proposition formed from s is valid in our theory and in the second of which we assume that the negation of the proposition formed from s is valid in our theory.

```
- by (cases_tac "A" 1);
P(if(A, b, s))
 1. A ==> P(if(A, b, s))
 2. ~ A ==> P(if(A, b, s))
```

Repeated application of the simplifier solve the subgoals and the proof is complete.

```
- by (REPEAT (asm_simp_tac (HOL_ss addsimps assums) 1));
P(if(A, b, s))
No subgoals!
```

Chapter 4

Well-founded Relations

One of our concerns is to develop a theory of *well-founded* relations in Isabelle. The notion of well-founded relation is important and plays a central role in computer science. We find it in domains such as theorem proving and program verification. Well-founded relations are very helpful and present a powerful technique for establishing properties of inductively defined objects. They are often used to prove the termination and correctness of recursive programs. They are also central for many systems like Boyer/Moore's theorem prover and the deductive tableaux framework of Manna and Waldinger. In the domain of automatic theorem proving to choose or to generate a well-founded relation, is not a trivial task, because of the undecidability of *well-foundedness*, and demands some creativity. This chapter introduces well-founded relations and their formalization in Isabelle's HOL. The next section is a recall of the definition of *well-foundedness* of relations and some of its standard properties. For more details, we suggest Manna and Waldinger's Book [38] that gives a good introduction to this topic, with many examples and proofs.

4.1 Preliminaries and Notation

A **relation** r is a **set** of ordered pairs presented as follows:

$$r = \{< e_1,\ e_2 > \ \mid \quad e_1 : S_1\ \wedge\ e_2 : S_2\}$$

where S_1 and S_2 are sets and the symbol " : " stands for the membership relation. Furthermore if S_1 and S_2 represent the same set, we say that r ranges over S_1.

Definition 4.1 (r-minimal)
For a non-empty set S and a relation r over S an element a is said to be *r-minimal* in S, if there is no further element b in S that is "smaller" than a in accordance to r.Formally this can be expressed as follows:
$a : S\ \wedge\ \forall^1 x : S.\ \neg < x,\ a > : r$ □

Definition 4.2 (Minimal-predicate)
mini(S, r) is a predicate that is satisfied if the set S is non-empty and it has

[1] $\forall x : S.\ P(x)$ stands for $\forall x.\ x : S \rightarrow P(x)$

an *r-minimal* element. Formally this can be expressed as follows:

$$\exists a.\ a : S \ \rightarrow\ \exists^2 m : S.\ \forall y : S.\ \neg < y,\ m > : r. \qquad \Box$$

Definition 4.3 (Decreasing chains)
A **chain** in a set S is a sequence of elements $x_1,\ \ldots,\ x_n,\ \ldots$ in S, written as $(x_i)_{i\geq 1}$. It is decreasing in accordance to a relation r ranging over S, if the following holds: $\forall i \geq 1. < x_{i+1},\ x_i > : r.$ $\Box$

Now we are able to define well-founded relations

Definition 4.4 (Well-foundedness)
A relation r over a set S is said **well-founded** if either one of the following conditions holds:

(i) *minimal condition*
For every non-empty subset A of S, $mini(A, r)$ holds.

(ii) *no-decreasing condition*
There is no decreasing infinite chain $(x_i)_{i\geq 1}$ in A.

(iii) *induction principle condition*
For every predicate P and every a in S : $P(a)$ holds, if we can derive it from the assertion that every y in S, for which $< y,\ a > : r$ holds, satisfies $P(y)$. Formally :
$\forall P.\ (\forall x : S.(\forall y : S. < y,\ x > : r \rightarrow P(y)) \rightarrow P(x)) \rightarrow \forall a : S.\ P(a)$

$\Box$

Notation
We shall write $wf_S(r)$ for the assertion that r is a well-founded relation over S. We omit the subscript S if this is known from the context and so we write $wf(r)$ for $wf_S(r)$.

4.2 Well-founded Induction Principle

The well-founded induction principle is a generalization of other known induction principles such as the stepwise induction principle, the structural induction principle and the course-of-values induction principle.

Theorem 1 (The well-founded induction principle) *For a well-founded relation r over a set S and a predicate P, the following sentence holds:*
$(\forall x : S.\ (\forall y : S. < y,\ x > : r \rightarrow P(y)) \rightarrow P(x)) \rightarrow (\forall a : S.\ P(a))$. *This sentence is called **well-founded induction principle**. The left part of its outermost implication is called **inductive step**. The sentence $\forall y : S. < y,\ x > : r \rightarrow P(y)$ is called the **induction hypothesis**.*

[2] $\exists x : S.\ P(x)$ stands for $\exists x.\ x : S \wedge P(x)$

The well-founded induction principle will be used in proofs in the following way: To establish validity of a property $P(a)$, where a is an arbitrary element of a set S, we first choose a well-founded relation r over S and then we attempt to prove the validity of the inductive step. Finally to establish this step we assume that $P(y)$ holds for all elements y smaller than x in accordance to the well-founded relation r, induction hypothesis, and we prove the predicate $P(x)$.

To encode the theory of well-foundedness of relations (see section 4.3) we will use the induction principle condition, condition (iii) of the above definition. The reason is that the induction principle is a simple consequence for this definition (see section 4.3.1). A further argument of this choice is that if we take the no-decreasing condition as basis for our encoding then we must formulate a special theory for infinite sequences to express the no-decreasing condition, which is not necessary to express the induction principle. Eventually, the well-foundedness proofs in a theory of infinite sequences are more intuitive and simpler.
Before we consider the encoding of the theory of well-founded relations in Isabelle we give some properties and examples of such relations. An well known well-founded relation is the **less-than** ordering over the natural numbers. In the set of natural numbers we can't built a decreasing infinite chain, which is possible for integers. Further examples is **proper-substring** relation over strings, **proper-sublist** relation over lists. Well-founded relations are *irreflexive* and *asymmetric*, but not necessarily *transitive*. For example the predecessor relation $< x,\ y > : r\ \equiv\ y = x + 1$ is a *non-transitive* well-founded relation.

4.3 A Formalization of Well-founded Relations in Isabelle's HOL

The theory of well-founded relations called **WFR** is an extension of Isabelle's HOL. HOL is already augmented with several other theories , such as natural number theory, set theory, list theory, Tarski's Theorem. In figure 4.1 we display the hierarchy of HOL-theories that come with Isablle. Isabelle's **WF** a predicate `wf` is declared . This predicate expresses the well-foundedness of relations taking the induction principle condition as basis (see figure 4.2). The syntax and Semantics of the new constants of our theory WFR are given in figure 4.3. The predicate `wf_prod` is a special instance of `wf`. This is needed because we don't have currying in the present theory, that means that the function $\lambda\!< x,\ y >.\ b(x,\ y)$ can not be seen as the function $\lambda x.(\lambda y.\ b(x,\ y))$. The operator `lex_rel` is the lexicographic ordering, it built a relation from two others. The operator `induced_rel` is a relation constructor mapping a function f and a relation r into a new relation that contains all pairs, whose images are in r. The mapping `map_rel` is the reciprocal of `induced_rel`. We will see later that these constructors preserve well-foundedness.

`Nat`	natural numbers
`WF`	Well-founded Recursion
`Trancl`	Transitive closure of a relation
`Lfp`	The Knaster-Tarski Theorem (least fixed point)
`Sum`	The disjoint sum of two types
`Prod`	Ordered Pairs and the Cartesian product type
`Set`	Set theory
`Ord`	Ordered types
`HOL`	Basis Theory

Figure 4.1: HOL-theories Hierarchy

name	*type*
`wf`	`:: "('a * 'a)set => bool"`

name	*definition*
`wf_def`	`"wf(r) == !P.(!x.(!y.  <y,x>:r --> P(y)) --> P(x))--> (!x.P(x)))"`

Figure 4.2: Syntax and Semantic of the predicate wf

name	*type*
`wf_prod`	`:: "('a * 'a)set => bool"`
`map_rel`	`:: "['a=>'b, ('a*'a)set]=>('b*'b)set"`
`induced_rel`	`:: "['a=>'b, ('b*'b)set]=>('a*'a)set"`
`lex_rel`	`:: "[('a*'a)set, ('b*'b)set] => (('a * 'b) * ('a * 'b))set"`

name *definition*

```
wf_prod_def       "wf_prod(r) == !P.(!x1 x2.(!y1 y2.  <<y1, y2>,<x1,x2>>:r
                                     --> P(y1,y2))
                             --> P(x1,x2))
                          --> (!x1 x2.P(x1,x2)))"

map_rel_def       "induced_rel(f, r) == Collect(%q.?  x y.(q=<f(x) ,f(y)>  &  <x, y>:r))"

induced_rel_def   "induced_rel(f, r) == Collect(%q.?  x y.(q=<x ,y>&<f(x),f(y)>:r))"

lex_rel_def       "lex_rel(ls1, ls2) == Collect(%q.?  x1 x2 y1 y2.  (q=<<x1,x2>, <y1,y2>>) &
                                 (<x1,y1>:ls1 |(x1=y1 & <x2, y2>:ls2))))"
```

Figure 4.3: Syntax and Semantics of Relation-operators of WFR

4.3.1 Derivation of the Induction Principle

The induction principle can be expressed as a natural deduction rule :

$$\dfrac{wf(r) \qquad \begin{array}{c}[x : S;\ \forall y : S.\ < y, x >: r \to P(y)]\\ \vdots \\ P(x)\end{array}}{\forall a : S.\ P(a)}\ (w.f.\ induction\ principle)$$

The rule *w.f. induction principle* is already proved in Isabelle and come as a derived rule in the system. We want to reconsider the proof to show how convenient it is to define well-foundedness of relations by means of the induction principle condition. This rule was formulated as follows

```
- val [ass1,ass2]= goal WF.thy "[| wf(r);
                                  !!x.[| ! y. <y,x>: r --> P(y) |] ==> P(x)
                                |]  ==> !a. P(a)";
! a. P(a)
 1. ! a. P(a)
```

The premise `ass1` asserts that the relation `r` is well-founded and the premise `ass2` asserts the inductive step. Now to solve the single subgoal we can benefit from the definition of r. In a first step we expand the definition of well-foundedness `wf_def` in the assumption `ass1`.

```
- val ass1_exp = (rewrite_rule [wf_def] ass1) ;
val ass1_exp =
  ! P. (! x. (! y. <y, x> : r --> P(y)) --> P(x)) --> (! x. P(x))
```

The obtained rule can not yet be directly applied to our single subgoal. We have to specialize the predicate `P` first. This can be achieved using a theorem operator called `RS`[3] and the rule `spec`[4].

```
- val new_ass1 = ass1_exp RS spec ;
val new_ass1 =
  (! x. (! y. <y, x> : r --> ?P(y)) --> ?x(x)) --> (! x. ?P(x))
```

Looking at the subgoal we remark that it matches the conclusion of the obtained new rule `new_ass2`. So we first eliminate the implication in this rule. We proceed as before by resolving `new_ass2` against the rule `mp`.

```
- val new_ass2 = new_ass1 RS mp ;
val new_ass2 = "! x. (! y. <y, x> : r --> ?P(y)) --> ?P(x) ==> ! x. ?P(x)
```

At this point we are able to apply the rule obtained above to resolve the subgoal. First we expand the definition of well-foundedness `wf_def` in the assumption `ass1` then we resolve it against the subgoal.

```
- br new_ass2 1;
! a. P(a)
 1. ! a. (! y. <y, a> : r --> P(y)) --> P(a)
```

[3] Isabelle provides operations on meta theorems. The most useful one, besides printing commands, is the operator `RS`, which performs resolution.

[4] All the rules that we use in this proof are displayed in figure 3.2

The introduced subgoal corresponds to the inductive step `ass2` and can be automatically proved by applying the classical prover `fast_tac`, where we augment the classical set `HOL_cs` with the assumption `ass2`. Finally the result is bound to the ML-identifier `wf_induct`.

```
- by (fast_tac (HOL_cs addEs [ass2]) 1);
P(a)
No subgoals!
- val wf_induct = result();
val wf_induct =
  [| wf(?r); !!x. ! y. <y, x> : ?r --> ?P(y) ==> ?P(x) |] ==> ?P(?a)
```

4.3.2 Preservation of Well-foundedness

To check well-foundedness of an arbitrary relation is a complex task which is undecidable in general. A simple way to do this for certain relations is to describe a subclass of relations, in which relations are well-founded by construction. The subclass that we consider here contains the less-than relation over natural numbers and it is closed under the constructors `lex_rel` and `induced_rel`. This subclass is sufficient to prove the termination of some examples that we will consider later. (Additional constructors could be added e.g: irreflexive transitive closure of well-founded relation, subrelations, etc). In HOL the well-foundedness of the *less-than* ordering is derived. We now consider well-foundedness of the lexicographic ordering and of induced relations.

4.3.2.1 Lexicographic Orderings

Well-foundedness The rule expressing the well-foundedness of this relation can be formulated in a natural deduction style as follows:

$$\frac{wf(ls_1) \qquad wf(ls_2)}{wf(lex\_rel(ls_1, ls_2))}\,(wf\_lex\_rel)$$

can also be formulated and proved in Isabelle.

```
- val [wf_ls1, wf_ls2] = goal thy
        "[|wf(ls1); wf(ls2) |]==> wf(lex_rel(ls1, ls2))";
wf(lex_rel(ls1, ls2))
 1. wf(lex_rel(ls1, ls2))
```

The two assumptions of our goal are bound to the constants `wf_ls1` and `wf_ls2`. First we expand the definition of the lexicographic ordering `lex_rel_def` and we transform the subgoal into a form to which we can apply the induction principle. The transformation is performed by resolving the subgoal against the rules `allI`, `impI` and the rule `pairsE`, which formalizes pair-elimination in Isabelle: `!x1 x2. P(<x1, x2>)==> !x.  P(x)`

```
- by  ((rewrite_goals_tac [wf_def])
     THEN (EVERY1 (map rtac [allI ,impI,pairsE])));
wf(lex_rel(ls1, ls2))
 1. !!P. ! x. (! y. <y, x> : lex_rel(ls1, ls2) --> P(y)) --> P(x) ==>
         ! x1 x2. P(<x1, x2>)
```

We apply induction on both components of the pair `<x1,x2>`. (using the well-foundedness of the relations `ls1` and `ls2`)

```
- by  (wf_ind_tac  [wf_ls1, wf_ls2] 1) ;
wf(lex_rel(ls1, ls2))
 1. !!P x xa.
       [| ! x. (! y. <y, x> : lex_rel(ls1, ls2) --> P(y)) --> P(x);
          ! y. <y, x> : ls1 --> (! x2. P(<y, x2>));
          ! y. <y, xa> : ls2 --> P(<x, y>) |] ==>
       P(<x, xa>)
```

The next step consists in applying the induction hypothesis

```
- by  (EVERY1[etac allE ,etac mp ]);
wf(lex_rel(ls1, ls2))
 1. !!P x xa.
       [| ! y. <y, x> : ls1 --> (! x2. P(<y, x2>));
          ! y. <y, xa> : ls2 --> P(<x, y>) |] ==>
       ! y. <y, x, xa> : lex_rel(ls1, ls2) --> P(y)
```

Finally we can simplify the subgoal and its assumptions using the simplifier `asm_full_simp_tac`. The rest can be achieved by the classical prover `fast_tac`. We bind the result to the ML-identifier `wf_lex_rel`.

```
- by  ((asm_full_simp_tac wfr_ss 1) THEN (fast_tac HOL_cs 1));
wf(lex_rel(ls1, ls2))
No subgoals!
- val wf_lex_rel = result();
[| wf(?ls1); wf(?ls2) |] ==> wf(lex_rel(?ls1, ?ls2))
```

A Special induction principle for lexicographic orderings The general induction principle `wf_induct` can be specialized to particular examples. A special induction principle for lexicographic ordering is shown below:

```
- wf_rel_lex RS wf_induct
[| wf(ls1); wf(ls2);
   !!x. ! y. <y, x> : lex_rel(ls1, ls2) --> P(y) ==> P(x)
|] ==>P(a)
```

Further specializations are possible by specifying one of the well-founded relations `ls_1` or `ls_2`.

4.3.2.2 Induced relations

Well-foundedness Well-foundedness for induced relations can be expressed in a natural deduction style as follows:

$$\frac{wf(r :: ('b *' b)set)}{wf(induced\_rel(f ::' a =>' b, r))}\ (wf\_induced\_rel)$$

The proof is similar to the last one. It is displayed in Figure 4.4

Special induced relations The relation *shorter* over lists states that every list l_1 is shorter as a list l_2, if the latter is longer than the former. This relation is the induced relation parameterized by the function *length* and the *less-than* ordering over natural numbers.

```
- val [wf_r] = goal thy "[|wf(r)|]==> wf(induced_rel(f,r))";
wf(induced_rel(f, r))
 1. wf(induced_rel(f, r))

- by  ((rewrite_goals_tac [wf_def])
     THEN (EVERY1  (map rtac [allI ,impI])));
wf(induced_rel(f, r))
 1. !!P. ! x. (! y. <y, x> : induced_rel(f, r) --> P(y)) --> P(x)
        ==> ! x. P(x)

- by (res_inst_tac [("P", "! y x. f(x)=y --> P(x)")] mp 1);
wf(induced_rel(f, r))
 1. !!P. ! x. (! y. <y, x> : induced_rel(f, r) --> P(y)) --> P(x) ==>
         (! y x. f(x) = y --> P(x)) --> (! x. P(x))
 2. !!P. ! x. (! y. <y, x> : induced_rel(f, r) --> P(y)) --> P(x) ==>
         ! y x. f(x) = y --> P(x)

- by (fast_tac HOL_cs 1);
wf(induced_rel(f, r))
 1. !!P. ! x. (! y. <y, x> : induced_rel(f, r) --> P(y)) --> P(x) ==>
         ! y x. f(x) = y --> P(x)

- by (wf_ind_tac  [wf_r] 1);
wf(induced_rel(f, r))
 1. !!P x. [| ! x. (! y. <y, x> : induced_rel(f, r) --> P(y)) --> P(x);
              ! y. <y, x> : r --> (! x. f(x) = y --> P(x)) |] ==>
           ! xa. f(xa) = x --> P(xa)

- by ((setupcontext 1) THEN (EVERY1  (map etac [allE ,mp])));
wf(induced_rel(f, r))
 1. !!P x xa.
       [| ! y. <y, x> : r --> (! x. f(x) = y --> P(x)); f(xa) = x |] ==>
       ! y. <y, xa> : induced_rel(f, r) --> P(y)

- by (asm_full_simp_tac wfr_ss 1);
wf(induced_rel(f, r))
No subgoals!
```

Figure 4.4: Well-foundedness proof of induced relations

Chapter 5

Interpretation of Deductive Tableaux in Isabelle

5.1 Background

The topic of this chapter is the interpretation of Manna and Waldinger's deductive tableaux framework in Isabelle. Before we start with the interpretation we first want to give the arguments that motivated us to choose the framework Isabelle. We first introduce some terminology. *Object-logics* (e.g. first-order logic, modal logic) often are encoded using a *meta-logic*, the meta-logic expresses the rules of the object-logic. These rules contain variables that range over syntactic categories (e.g. *terms*, *formulas*) of the object-logic. These variables are called *metavariables* and the rules *schematic rules*. For example, in the substitution rule of first-order logic

$$\frac{x = y \quad P(x)}{P(y)}\,(subst)$$

x and y are not concrete terms but metavariables ranging over first-order terms. P is a metavariable ranging over truth-valued functions. By applying schematic rules the metavariables will be instantiated with objects (e.g. *concrete terms*, *concrete formulas*) of the object-logic.

Initially we are interested only in a system in which we can synthesize programs, but because the domain of program synthesis is relative new there aren't many systems which are designed for this activity. Therefore we first decided to study the relationship between program verification and program synthesis. The two questions that motivated our work here are: First, are program verification and program synthesis really different? Second, how can we recast a calculus designed for program verification for program synthesis? The answer of these questions is important and will give us a hint how we can implement calculi based on well known techniques coming from the domain of program verification to support program construction. Green's work [23] can be interpreted as a positive answer. Then he demonstrated there that verification and synthesis can be seen as one and the same activity. Green's system is a refutation one based on resolution. Queries may contain object-level variables which represent unspecified objects. Checking a query corresponds then to a verification proof if the query didn't contain variables and, otherwise, the variables may be become instantiated. The instances are constructed using the unifier computed during the proof. Green's approach is quite similar to the programming language **Prolog**. In both above approaches the object-level variables can be seen as metavariables which can be instantiated with first-order objects using the unification. The deciding advantage of these approaches is that the same kind of proof rule (resolution) is applied to construct proofs, independent whether the queries contain metavariables or actual terms. So a key idea to unify verification and synthesis is to use metavariables in the proofs as well as in the proof rules in a system that is based on resolution and which treats the metavariables occuring in the proofs as variables of its meta-logic and instantiate them with the corresponding objects. Although the deductive tableaux system is based on resolution and allows the use of metavariables in the proofs, it is limited in its support of program development. Then using it we can construct programs only in a bottom-up way. This is because the metavariables occuring in the proofs can be only instantiated with first-order terms , so for instance we can't substitute a metavariable by a conditional term, whose "condition predicate" is represented with a metavariable, this need second-order metavariables.

Let us discuss why **Isabelle** is suited for our work. First, the appropriate object-logic, where we will express specifications, is already encoded. Second Isabelle comes with well-developed tactics for rewriting, simplification and classical proving. Third, we can derive new rules without significant extra work. But the deciding advantage is Isabelle's meta-logic, which is based on a fragment of the intuitionistic higher order logic (see section 3.1.1), allows us to use metavariables (they are variables of the meta-logic) in rules of the logic and proofs and the higher order unification may instantiate them during resolution steps.

In the remainder of this chapter we give the basis calculus, where we simulate the deductive tableaux system. Then we derive some rules that are convenient for program synthesis in Isabelle. In the following we abbreviate the deductive tableaux system with *DTS*.

5.2 The Calculus

Restricting recursion schemata to those corresponding to structural induction in program construction, we often get inefficient and incomprehensible programs. A better approach is to use not only such schemata but also other kinds based on general well-founded induction as well. To express well-foundedness and manipulate first-order specifications we need a richer logic than first-order. What is why we have chosen Isabelle's HOL. In section 5.5.2 we will develop Isabelle tactics that simulate the deduction rules of *DTS*. Thus we will exploit HOL only to prove well-foundedness the tactics can be else also derived in FOL. Isabelle's HOL was already described in section 3.2. There figure 3.2 displays its axioms and some derived rules. Note that **set theory** is also encoded in Isabelle and may be suitable for our work, but we haven' t examined this theory here.

Before we begin with the translation of *DTS*, we formulate and derive some special rules that are convenient for program construction in HOL.

5.3 Some Special Rules

One part of the interpretation of *DTS* is to simulate its **output entry mechanism**. The output columns in *DTS* serve as an intermediate storage for the stepwise generated programs. In each step of the deduction process the variables occuring in the output entries may be instantiated using the unifier computed by applying deduction rules. The terms introduced during a deduction proof may be

- Conditional terms, by the nonclausal resolution corresponding to case analysis,
- Recursive terms, by applying the induction rule, and
- Constants of the underlying theory, by applying other rules.

In the following we will describe the realization of the intermediate store-mechanism in Isabelle. Then we give a rule that allows to introduce conditional terms in the programs. Further in section 5.3.3 we present rules, called recursions schemata, whose application introduce recursive terms in the programs and also extend the accumulator by new definitions. Because recursion schemata are central in our work, we will give an example to illustrate their applications.

5.3.1 The Accumulator

Assume that we want to construct a program satisfying the first-order specification $\forall x.\ \exists z.\ spec(x, z)$. Opposed to *DTS* where the deduction proof begins with this specification, we formulate a new specification namely, `Def(?H) --> !x. spec(x,?f(x))`, where `?H` is a metavariable which denotes the accumulator[1] variable and `?f` denotes the desired program. The new specification can be read as "What do we have to define `?H` such that the specification `!x. spec(x,?f(x))` becomes provable". The accumulator variable `?H` ranges over formulas. Program definitions that would be introduced in `?H` during a proof are universally quantified formulas. Accumulators are of the general form `Def(def_1 & ...& def_n & ?Q)`, where $n \geq 0$ and `def_i`'s are program definitions. Through the variable `?Q` the accumulator remains open to set up additional definitions. The constant `Def` that surrounds the accumulator variable `?H` can be seen as a wrapper for `?H`. It is intend to omit undesired instantiation of `?H` during the proof. The constant `Def` has the property `Def(P)==P` for every formula `P`. We will see later that only by using recursion schemata we can introduce definitions and extend them with a new accumulator variable. At this point we can establish an advantage

[1] We have chosen the name **accumulator** for the formula `Def(?H)`, because its role is similar to that of the programming techniques used to store intermediate results in additional arguments which one calls accumulators

over *DTS* namely we can initially instantiate `?H` with partially synthesized programs. But in *DTS* there is no way to do this except by modifying the initial specification.

5.3.2 Case Split

In section 3.2.1.2 the rule *case_split* was already formulated and proved. It is our only rule that introduces conditional terms in the programs. Figure 5.1 lists a collection of other case split rules.

5.3.3 Induction Schemata

The induction schemata considered here are special forms of the well-founded induction principle intended as recursion schemata for the desired programs. A listing of some recursion schemata is given in Figure 5.2. Here we give an example to explain how we apply such rules and manipulate the accumulator variable. Suppose we want to prove correctness of the specification `!l. spec(l, ?f(l))`, where `spec` denotes a higher-order formula and `?f` denotes the desired program. We type this in the system

```
- goal thy "Def(?H)-->(!l. Spec(l,?f(l)))";
Def(?H)--> (!l. spec(l, ?f(l)))
 1. Def(?H)--> (!l. spec(l, ?f(l)))
```

The first step would be always the same, namely moving the accumulator `?H` into the assumption context.[2] This will be done by invoking a tactic called `SetUpcontext` (see appendix B.2) which performs introductions such as `allI`, and `impI`, etc.

```
- by (SetUpcontext 1);
Def(?H) --> (!l. spec(l, ?f(l)))
 1. Def(?H) ==> !l. spec(l, ?f(l))
```

Now we are able to apply induction. First we must chose a convenient recursion schema. Here we suggest a recursion schema called `indschemaf1_1p1`:

```
[| Def(!l. ?f(l) = ?fbody(l));
   !l. (!t. <t, l> : ?r --> Spec(t, ?f(t))) --> Spec(l, ?fbody(l));
   wf(?r)
|] ==> !l. ?Spec(l, ?f(l))
```

The first premise of the schema defines a program named `?f`, whose body is a metavariable, this means that at this stage we know neither its concrete name nor how it is built. The second premise, corresponds to the *inductive step* and asserts that for every list `l`, if the *induction hypothesis*,

```
!t. <t, l>:?r --> ?Spec(t, ?f(t))
```

[2] To avoid this step we can normally formulate the initial goal as follows `[| Def(?H)|] ==> !l. Spec(l, ?f(l))`, but this is not allowed because the use of metavariables at the premises affects problems by lifting. See [45] for more details

holds then the desired conclusion `?Spec(l, ?f)`[3] holds, too. The last premise is a predicate which states well-foundedness of the relation `?r`. Now to apply this rule we have developed a tactic called `INDTAC` (see appendix B.2). It takes four arguments

- A special induction schema *recursion_schema*
- A list of string pairs *suggested_names*
- Another list of string pairs *instances_list*
- A natural number i

The natural number i indicates the subgoal to be resolved by induction. The first step of this tactic is to extend the accumulator `Def(?H)` for further definitions that can be introduced in the rest of the derivation. This step is accomplished by resolving the subgoal against a rule called `DefConjE`

```
[| Def(P & Q); [| Def(P); Def(Q)|] ==> R |]==> R
```

It yields to two new subgoals

```
[| Def(?H) |] ==> Def(?P & ?Q)
[| Def(?H);  Def(?P); Def(?Q)|]==> !l. spec(l, ?f(l))
```

The first subgoal can be solved by assumption. The accumulator variable `?H` becomes `?P & ?Q`. Now we remove the premises `Def(?P & ?Q)` from all subgoals of the proofstate, (in this case only one subgoal), because these premises will be superfluous. We then get the following subgoal.

```
[| Def(?P); Def(?Q)|] ==> !l. spec(l, ?f(l))
```

The second step of `INDTAC` is to give a name for our desired program, in our case we bind[4] the metavariable `?f` to the name "`fname`". Hence, `indschemaf1_1p1` becomes

```
[| Def(!l. fname(l) = ?fbody(l));
   !l. (!t. <t,l>:r --> ?Spec(t, fname(t))) --> ?Spec(l, ?fbody(l));
   wf(?r)
|] ==> !l. ?Spec(l,~fname(l))
```

Note that no question-mark is affixed the identifier "`fname`". This means that it will be treated as a constant. Now we are ready to apply the induction schema. This leads the system to unify the conclusion `!l. ?Spec(l, ?f(l))` of `indschemaf1_1p1` and the conclusion `!l. spec(l, ?f(l))` of the subgoal. Since Isabelle supports higher-order unification we can obtain an undesired substitution. Therefore we use the tactic `res_inst_tac`[5], which applies the resolution

[3] Really the desired conclusion is `?Spec(l, ?fbody(l))` but by the definition `?f(l)` is equal to `?fbody(l)`

[4] By typing in the system the following:
`(read_instantiate [("f","fname")] indschemaf1_1p1)`, where `read_instantiate` is a function supported in Isabelle to specialize theorems for concrete objects. (see [45] pages 22 and 35 for more details)

[5] A call of this tactic has the form `by (res_inst_tac inst_list thm` i`)`, where *inst_list* is list of instances that will be used to partial instantiate some variables occuring in the sentence `?Sepc(l, fname(l))` to unify it and `sepc(l, fname(l))`

tactic with some partially instantiation. In our case we instantiate the metavariable `?Spec` with the predicate `% x1 x2.  spec(x1, x2)`. So we obtain the following subgoals

```
[| Def(?P); Def(?Q)|]==>Def(!l.fname=?fbody(l))
[| Def(?P); Def(?Q)|]==>!l. (!t. <t,l>:?r --> spec(t, fname(t)))
                           -->spec(l,?fbody(l))
[| Def(?P); Def(?Q)|]==> wf(?r)
```

The first subgoal can be proved by assumption, instantiating the metavariable `?P` with `!l.  fname(l) = ?fbody(l)`. Now we have introduced the definition of the desired function in the assumption context. The remaining subgoals become

```
[|Def(!l. fname(l)=?fbody(l)); Def(?Q)|]
    ==>!l.(!t. <t,l>:?r -->spec(t, fname(t)) --> spec(l,~?fbody(l))
[|Def(!l. fname(l)=?fbody(l)); Def(?Q)|]==> wf(?r)
```

We remark that at this point the premise `Def(!l.fname(l) = ?fbody(l))` will never be never used to prove the first subgoal. Furthermore to prove the first subgoal we have to apply the tactic `SetUpcontext`. All these previous steps will be achieved if we call the induction tactic

```
- by  (INDTAC indschemaf1_1p1  [("f","fname")] [("Spec", "%x1 x2. spec(x1,x2)"] 1);
Def((!l. fname(l) = ?fbody(l)) & ?Q) --> (!l. Spec(l, fname(l)))
 1. !!l. [| Def(?Q); !t. <t, l> : ?r --> Spec(t, fname(t)) |] ==>
        Spec(l, ?fbody(l))
 2. [| Def(!l. fname(l) = ?fbody(l)); Def(?Q) |] ==> wf(?r)
```

Note now that by proving the first subgoal using the induction hypothesis we build a recursive term, namely the metavariable `?fbody` will be instantiated with a term which may contains an occurrence of "`fname`". Also note that by applying the induction we have not specified the well-founded relation, because at the moment we don't know how to decompose the input list `l`. Therefore it remains represented by a metavariable and shall be later substituted by the user.

5.4 The Front/Last Example

As a motivation for the transformation of tableau proofs into Isabelle proofs we go once again through the front/last example of section 2.2.5 the discussed proof there was suggest by Manna and Waldinger in [39]. The functions *front* and *last* meet the specification

$$\forall a\ \exists z_1 z_2.\ if\, a \neq\ \epsilon\ then\ char(z_2)\ \wedge\ a = z_1 * z_2,$$

where $\forall a.\ front(a) = z_1$ and $\forall a.\ last(a) = z_2$.
See section 2.2.5 for more information about the specification and an overview of the syntax that we use for this example. Isabelle rules that we will use here are listed in figure 3.2. The deduction rules of the tableaux system are completely displayed in Appendix A.4.

```
case_splitf1      [| A ==> P(b);  ~ A ==> P(s) |] ==> P(if(A, b, s))

case_splitf2      [| A ==> P(s1, s2);
                     ~A ==> P(t1, t2)
                  |] ==> P(if(A, s1, t1), if(A, s2, t2))

case_splitf1_list [| l = [] ==> P(s1);
                     l ~= [] ==> P(s2(hd(l), tl(l)))
                  |] ==>P(if(l = [], s1, s2(hd(l), tl(l))))

case_splitf2_list [| l = []  ==> P(s1, s2);
                     l ~= [] ==> P(t1(hd(l), tl(l)), t2(hd(l), tl(l)))
                  |] ==>P(if(l = [], s1, t1(hd(l), tl(l))),if(l = [], s2, t2(hd(l), tl(l))))
```

Figure 5.1: Case split schemata

```
indschemaf1_1p1   [| Def(!l. f(l) = fbody(l));
                     !l. (!t. <t,l> : r --> Spec(t,f(t))) -->
                                Spec(l,fbody(l));
                     wf(r) |] ==> !l. Spec(l,f(l))

indschemaf1_2p2   [| Def(!l1 l2. f(l1,l2) = fbody(l1,l2));
                     !l1 l2.(!t1 t2.<<t1,t2>,l1,l2> : r -->
                                      Spec(t1,t2,f(t1,t2))) -->
                                Spec(l1,l2,fbody(l1,l2));
                     wf_prod(r) |] ==> !l1 l2.Spec(l1,l2,f(l1,l2))

indschema_innsort [| Def(!u l. f(u,l) = fbody(u,l));
                     !v l.
                     (!u t. <t,l> : r --> Spec([u],t,f(u,t))) -->
                                Spec([v],l,fbody(v,l));
                     wf(r) |] ==> !l u. Spec([u],l,f(u,l))

indschemaf2_1p1   [| Def((!l. f(l) = fbody(l)) & (!l. g(l) = gbody(l)));
                     !l.(!t. <t,l> : r --> Spec(t,f(t),g(t))) -->
                                Spec(l,fbody(l),gbody(l));
                     wf(r) |] ==> !l. Spec(l,f(l),g(l))

indschemaf2_2p1   [| Def((!u l. f(u,l) = fbody(u,l))
                         & (!u l. g(u,l) = gbody(u,l)));
                     !l.(!u t. <t,l> : r --> Spec(t,u,f(u,t),g(u,t))) -->
                                (!u. Spec(l,u,fbody(u,l),gbody(u,l)));
                     wf(r) |] ==>!l u. Spec(l,u,f(u,l),g(u,l))

indschemaf2_1p1   [| Def((!l. f(l) = fbody(l)) & (!l. g(l) = gbody(l)));
                     !l.(!t. <t,l> : r --> Spec(t,f(t),g(t))) -->
                                Spec(l,fbody(l),gbody(l));
                     wf(r) |] ==> !l. Spec(l,f(l),g(l))
```

Figure 5.2: Recursion schemata based on Well-founded Induction Principle

5.4.1 The Initial Tableau and its Corresponding Proofstate in Isabelle

The tableaux system starts with the initial tableau $\mathcal{T}_0$[6]

label	*Assertions*	*Goals*	*Outputs*	
			$front(a)$	$last(a)$
1		*if* $a \neq \epsilon$ *then* $char(z_2) \ \wedge \ (a = z_1 * z_2)$	z_1	z_2

where a is a skolem constant of type *string* and the variables z_1 and z_2 also are of the same type. To get the corresponding proofstate in Isabelle we type the following goal

$$Def(?H) \rightarrow \forall a.\ (a \neq \ \epsilon \rightarrow char(?last(a)) \ \wedge \ a = ?front(a) * ?last(a))$$

After moving the accumulator variable in the assumption context and eliminating the universal quantifier we get the following proof state $\mathcal{P}_0$:

$$\begin{aligned} &[\![\textstyle\bigwedge a.\ Def(?H) \ \Longrightarrow (a \neq \ \epsilon \rightarrow char(?last(a)) \ \wedge \ a = ?front(a) * ?last(a))]\!] \\ &\quad \Longrightarrow Def(?H) \ \rightarrow (\forall a.\ a \neq \ \epsilon \rightarrow char(?last(a)) \ \wedge \ a = ?front(a) * ?last(a)) \end{aligned}$$

Now we start by simulating each tableau rule that will be applied to the derivation.

5.4.2 The Simulation Tactic

Application of a deduction rule in the tableaux system corresponds to refining some subgoals of the proofstate in Isabelle. Nonclausal resolution and the equality rule have the following general form (see sections 2.2.4.2 and 2.2.4.3)

$$\frac{row_1 \quad row_2}{row_3}$$

Simulation of these deduction rules will be accomplished in an uniform way in three steps:(Here we assume that the formula occuring in row_2 is the conclusion of the subgoal that we have to refine)

(1) We generate an appropriate rule depending on the subgoal that we have to refine. These rules will have the following general form

$$[\![formula\_of\_row_1; \quad formula\_of\_row_3\ ]\!] \Longrightarrow formula\_of\_row_2 \qquad (\ddagger)$$

where $formula\_of\_row_1$ and $formula\_of\_row_2$ correspond to the assertions/goals of the required rows 1 and 2. $formula\_of\_row_3$ corresponds to the assertions/goals of the generated row 3.

[6]The connective $\rightarrow$ used in our Isabelle proof correspond to the *if-then* operator used in the tableau, the tableau variables z_1 and z_2 corresponds to the variables *?fbody* and *?lbody*.

(2) $formula\_of\_row_2$ corresponds to the conclusion of the subgoal that will be refined. So by resolving the corresponding subgoal against the rule derived in the first phase we produce two additional subgoals. The first one has $formula\_of\_row_1$ as conclusion

$$[\![\ldots]\!] \Longrightarrow formula\_of\_row_1$$

and the second one has $formula\_of\_row_3$ as conclusion

$$[\![\ldots]\!] \Longrightarrow formula\_of\_row_3$$

(3) In our example row_1 normally contains an axiom of the string theory or it is deduced from other previous rows. In the first case we resolve the first subgoal produced in the second phase against the corresponding axiom. In the second case we solve this subgoal by assumption. This simulation tactic is formalized in section 5.5.2. There we have exactly specified the formulas $formula\_of\_row_1$, $formula\_of\_row_2$ and $formula\_of\_row_2$ for each deduction rule. The main idea behind this simulation tactic is that the unification step, that will be performed during the application of a deduction rule, will be performed in the simulation first in the third phase, namely by resolving the first new subgoal . In this example we will show informal how we formulate $formula\_of\_row_1$, $formula\_of\_row_2$ and $formula\_of\_row_3$ for some deduction rule.

5.4.3 Induction

At the beginning we apply the induction rule with the proper *substring* relation $x <_{string}$ as well-founded order. This leads to the following new tableau $\mathcal{T}_1$

label	*Assertions*	*Goals*	*Outputs*	
			$front(a)$	$last(a)$
1		$if\ a \neq \epsilon\ then$ $char(z_2)\ \wedge\ (a = z_1 * z_2)$	z_1	z_2
2	$if\ x <_{string} a\ then$ $if\ x \neq \epsilon\ then$ $char(last(x))\ \wedge\ x = front(x) * last(x)$			

In Isabelle we resolve the subgoal against the recursion schema `indschemaf2_1p1` (see figure 5.2). Thereby we assign the desired functions $?front$ and $?last$ to the identifiers *front* and *last* [7]. $\mathcal{P}_0$ becomes $\mathcal{P}_1$:

$$
\begin{array}{l}
[\![\\
\quad \bigwedge a.[\![Def(?Q); \\
\qquad <?x(a),\ a> :?r \rightarrow ?x(a) \neq \epsilon \rightarrow \ char(last(?x(a))) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ ?x(a) = front(?x(a)) * last(?x(a)) \\
\qquad]\!] \Longrightarrow a \neq \epsilon \rightarrow char(last(a)) \ \wedge \ a = ?fbody(a) * ?lbody(a); \\
\\
\quad [\![Def(?Q);\ Def((\forall x.\ front(x) = ?fbody(x)) \ \wedge \ (\forall x.\ last(x) = ?lbody(x)))]\!] \Longrightarrow wf(?r) \\
\\
]\!] \Longrightarrow Def((\forall x.\ front(x) = ?fbody(x)) \ \wedge \ (\forall x.\ last(x) = ?lbody(x)) \wedge ?Q) \\
\qquad \rightarrow (\forall a.\ a \neq \epsilon \rightarrow char(last(a)) \ \wedge \ a = front(a) * last(a))
\end{array}
$$

Note that analogously to the introduction of the *induction hypothesis* in row 2 in the proofstate, this assertion is also pushed in the assumption context of the first subgoal. The second subgoal of the previous proofstate states the well-foundedness of the relation $?r$. For more readability of the proofstates that will be displayed a long this example we abbreviate the induction hypothesis introduced in the first subgoal of the above proofstate means of the identifier *Ind_hyp*. Furthermore we will delete the premises *Def(?Q)* from both subgoals, then as we have seen in section 2.2.5 no further subprograms needed for the development of the functions *front* and *last*. Finally we will omit the second subgoal temporarily.

In the tableau we apply *ifsplit* on row 1 the corresponding rule in Isabelle is the implication introduction `impI`. So we obtain $\mathcal{T}_2$

label	*Assertions*	*Goals*	*Outputs*	
			$front(a)$	$last(a)$
1		$if\ a \neq \epsilon\ then$ $char(z_2) \ \wedge \ (a = z_1 * z_2)$	z_1	z_2
2	$if\ x <_{string} a\ then$ $if\ x \neq \epsilon\ then$ $char(last(x)) \ \wedge \ x = front(x) * last(x)$			
3	$a \neq \epsilon$			
4		$char(z_2) \ \wedge \ a = z_1 * z_2$	z_1	z_2

and the proofstate $\mathcal{P}_2$, in which $a \neq \epsilon$ is moved into the assumptions list of the

[7] In section 5.3.3 we have explained how we apply such rules in Isabelle.

subgoal.

$$
\begin{array}{l}
[\![\\
\quad \bigwedge a.[\![Ind\_hyp\ ;\ a \neq\ \epsilon]\!] \Longrightarrow char(?lbody(a))\ \wedge\ a = ?fbody(a) * ?lbody(a) \\
\\
]\!] \Longrightarrow Def((\forall x.\ front(x) = ?fbody(x))\ \wedge\ (\forall x.\ last(x) = ?lbody(x)) \wedge ?Q) \\
\qquad \to (\forall a.\ a \neq\ \epsilon \to char(last(a))\ \wedge\ a = front(a) * last(a))
\end{array}
$$

The simulation of the last applied deduction rules shows that the transformation of tableau proofs into Isabelle proofs could be straightforward. In the following we will demonstrate that in fact the simulation is simple but it can't be successful performed in a naive way and it needs sometimes much attention. The problem arises by using rows in the deduction-process more as once. A variable occuring in such a row could be instantiated each time with different terms. In the deductive tableaux system this is allowed but in Isabelle variables can be instantiated at most once. In the next section we want follow a naive way to simulate the deduction rules that leads to an unprovable Isabelle proofstate.

5.4.4 Case Analysis

Intuitively we proceed in tableau as well as in Isabelle by performing a case analysis over the structure of the constant a. We have two cases to consider. The constant a is a string formed from one character or it has the form $u.y$, where the operator "." denotes the string constructor *cons* for lists and y is a non-empty string. In this tableau we can immediately begin by handling both cases.

5.4.4.1 First Case (a is a character)

By the equality rule, applied to the concatenation axiom $\epsilon * w = w$ and the goal of row 4

	$\boxed{\epsilon * w}^{-} = w$			
4		$char(z_2)\ \wedge\ a = z_1 * z_2$	z_1	z_2

with the substitution $\theta = \{z_1 \leftarrow \epsilon\}$, we obtain $\mathcal{T}_3$

label	*Assertions*	*Goals*	*Outputs*	
			front(a)	*last*(a)
1		$if\, a \neq \epsilon\ then$ $char(z_2) \wedge (a = z_1 * z_2)$	z_1	z_2
2	$if\ x <_{string} a\ then$ $if\, x \neq \epsilon\ then$ $char(last(x)) \wedge x = front(x) * last(x)$			
3	$a \neq \epsilon$			
4		$char(z_2) \wedge a = z_1 * z_2$	z_1	z_2
5		$char(z_2) \wedge a = z_2$	ϵ	z_2

In Isabelle we can use the concatenation axioms of strings by referring to their identifiers. $\epsilon * w = w$ has the name `append_Nil` and $u.y_1 * y_2 = u.(y_1 * y_2)$ is called `append_Cons`. Before we consider the simulation of this tableau step. Let us give a more detailed explanation of the application of the equality rule. The required formulas are $\epsilon * w = w$ and $char(z_2) \wedge a = z_1 * z_2$. Really the application of this rule has been achieved in two phases :

(1) Generate the new formula of the form:
$char(z_2) \wedge a = w \wedge (\neg false)$ [8]

(2) Unify the terms $\epsilon * w$ and $z_1 * z_2$ and then simplify the generated row to obtain $char(z_2) \wedge a = z_2$

Now we come to the simulation. This will take three phases : First we derive the rule that will be applied to the second phase to refine our subgoal. This desired rule is of the form (‡) as we have mentioned above. Thus *formula_of_row1*, *formula_of_row2* and *formula_of_row3* are specified as follows:

- *formula_of_row*$_1$
  This formula corresponds to the assertion $\epsilon * w = w$. Because we want to force the system to perform the unification of $\epsilon * w$ and $z_1 * z_2$, we formulate *formula_of_row1* as $z_1 * z_2 = w$ (we will explain later in the third phase how the unification will be achieved).

- *formula_of_row*$_2$
  This formula is the conclusion of our actual first subgoal, namely $char(z_2) \wedge a = z_1 * z_2$

[8] *false* stands for $(\epsilon * w = w)[\epsilon * w = w/false]$. See section 2.2.4.3 for more details

- *formula_of_row*$_3$
  This corresponds to the generated formula (before the unification). It has the form $char(z_2) \ \wedge \ a = w$ (Note that we have simplified it).

The rule has then the following form :

$$[\![z_1 * z_2 = w;\ char(z_2) \ \wedge \ a = w]\!] \Longrightarrow char(z_2) \ \wedge \ a = z_1 * z_2$$

Its derivation in Isabelle is straightforward. Note that all its free variables are schematic variables. In the second phase we apply the above rule to refine the first subgoal. So we obtain the following proofstate:(the variables z_1 and z_2 are bound to ?*fbody*, ?*lbody*, respectively)

$$
\begin{array}{l}
[\![\\
\quad \bigwedge a.[\![Ind\_hyp \ ; \ a \neq \ \epsilon]\!] \Longrightarrow ?fbody(a) * ?lbody(a) = ?w(a); \\
\\
\quad \bigwedge a.[\![Ind\_hyp \ ; \ a \neq \ \epsilon]\!] \Longrightarrow char(?lbody(a)) \ \wedge \ a = ?w(a) \\
\\
]\!] \Longrightarrow Def((\forall x.\ front(x) = ?fbody(x)) \wedge \\
\qquad\qquad\quad (\forall x.\ last(x) = ?lbody(x)) \ \wedge \ ?Q) \\
\qquad\quad \to (\forall a.\ a \neq \ \epsilon \to char(last(a)) \ \wedge \ a = front(x) * last(a))
\end{array}
$$

Finally, because *formula_of_row*$_1$, the conclusion of the subgoal, corresponds to the axiom `append_Nil` we resolve the subgoal against this axiom. This leads the system to perform the desired unification of the terms $?fbody(a) * ?lbody(a)$ and $\epsilon * ?w(a)$. Thereby $?fbody(a)$ becomes ϵ.

$$
\begin{array}{l}
[\![\\
\quad \bigwedge a.[\![Ind\_hyp \ ; \ a \neq \ \epsilon]\!] \Longrightarrow char(?lbody(a)) \ \wedge \ a = ?lbody(a) \\
\\
]\!] \Longrightarrow Def((\forall x.\ front(x) = \epsilon) \ \wedge \\
\qquad\qquad\quad (\forall x.\ last(x) = ?lbody(x)) \ \wedge \ ?Q) \\
\qquad\quad \to (\forall a.\ a \neq \ \epsilon \to char(last(a)) \ \wedge \ a = front(x) * last(a))
\end{array}
$$

We continue the deduction-process by applying *reflexivity* rule $x = x$ to row 5

	$\boxed{x = x}^{-}$			
5		$char(z_2) \ \wedge \ \boxed{a = z_2}^{+}$	ϵ	z_2

with the substitution $\theta = \{z_2 \leftarrow a\}$.
The obtained new tableau $\mathcal{T}_4$ contains additional the new row 6.

label	Assertions	Goals	Outputs	
			$front(a)$	$last(a)$
1		$if\ a \neq \epsilon\ then$ $char(z_2) \wedge (a = z_1 * z_2)$	z_1	z_2
2	$if\ x <_{string} a\ then$ $if\ x \neq \epsilon\ then$ $char(last(x)) \wedge x = front(x) * last(x)$			
3	$a \neq \epsilon$			
4		$char(z_2) \wedge a = z_1 * z_2$	z_1	z_2
5		$char(z_2) \wedge a = z_2$	ϵ	z_2
6		$char(a)$	ϵ	a

Analogous to the above we derive first the following rule

$$[\![b = w;\ char(w)]\!] \Longrightarrow char(w) \wedge b = w$$

Then we apply it to the subgoal

$$\begin{array}{l}
[\![\\
\quad \bigwedge a.[\![Ind_hyp\ ;\ a \neq \epsilon]\!] \Longrightarrow a = ?lbody(a) \\
\\
\quad \bigwedge a.[\![Ind_hyp\ ;\ a \neq \epsilon]\!] \Longrightarrow char(?lbody(a)); \\
\\
]\!] \Longrightarrow Def((\forall x.\ front(x) = \epsilon) \wedge \\
\qquad\qquad (\forall x.\ last(x) = ?lbody(a)) \wedge ?Q) \\
\qquad \rightarrow (\forall a.\ a \neq \epsilon \rightarrow char(a) \wedge a = front(a) * last(a))
\end{array}$$

Finally we resolve the first subgoal against the reflexivity rule `refl`. We get the following proofstate

$$\begin{array}{l}
[\![\\
\quad \bigwedge a.[\![Ind_hyp\ ;\ a \neq \epsilon]\!] \Longrightarrow char(a); \\
\\
]\!] \Longrightarrow Def((\forall x.\ front(x) = \epsilon) \wedge \\
\qquad\qquad (\forall x.\ last(x) = a) \wedge ?Q) \\
\qquad \rightarrow (\forall a.\ a \neq \epsilon \rightarrow char(a) \wedge a = front(a) * last(a))
\end{array}$$

The first step of case analysis is finished. We remark that the desired functions *front* and *last* are not generated up to now and in Isabelle we have no more possibility to do so later on. Because in our actual proofstate there occur no variables in the bodies of the functions *last* and *front* which can be substituted later, furthermore the first subgoal is unprovable. At this point we have a one to one translation of the tableau rules. But we are unable to continue the derivation in Isabelle. Why has this happened? Until now, although we intuitively performed a case analysis, we haven't applied such a rule in the tableau. But the two intuitively last steps of the derivation correspond to the first case of the case analysis. These steps have forced a substitution of some variables of the subgoal, such as ?*fbody* and ?*lbody*, in such a way that to complete the correctness proof of the specification becomes impossible. The next question is why we can continue the deduction process in the tableau. The answer consists of two arguments: First, the substitution computed during applications of the deduction rule will be automatically applied only in the newly generated rows. For example row 4 remains unchanged after the application of the equality rule for row 5. Second, rows could be used more as once. In the second part of the case analysis we will reuse row 4 to generate new rows which can be combined with row 6 to build the desired programs.

We restart now the simulation of the two *last* steps once again. But this time first we perform a case analysis on the subgoal of the proofstate. The case analysis rule in Isabelle has the form $[\![P \Longrightarrow G;\ \neg P \Longrightarrow G]\!] \Longrightarrow G$. If we resolve the subgoal against this rule we obtain two new subgoals. The first has $?P(a)$ as additional assumption. the second $\neg ?P(a)$. Both have the same conclusion, namely $?\mathit{fbody}(a) * ?\mathit{lbody}(a)$. The predicate $?P(a)$ will later be instantiated to $\mathit{char}(a)$. This would not be sufficient because, if we prove one of these new subgoals the variables occuring in the other one would be also instantiated. What we really want is to assign two terms to the variables ?*fbody* and ?*lbody*, one term in each case. The solution that we suggest here is to substitute the variables of the subgoal by terms of the form *if_then_else*. This can be achieved by applying the `case_split` rule

$$[\![P \Longrightarrow G(?s_1, ?s_2);\ \neg P \Longrightarrow G(?t_1, ?t_2)]\!] \Longrightarrow G(if(P, ?s_1, ?t_1), if(P, ?s_2, ?t_2))$$

on the first subgoal of $\mathcal{P}_2$

$$
\begin{array}{l}
[\![\\
\quad \bigwedge a.[\![\mathit{Ind\_hyp}\ ;\ ?P(a);\ a \neq \epsilon]\!] \Longrightarrow \mathit{char}(?s_2(a))\ \wedge\ a = ?s_1(a) * ?s_2(a); \\
\\
\quad \bigwedge a.[\![\mathit{Ind\_hyp}\ ;\ \neg ?P(a);\ a \neq \epsilon]\!] \Longrightarrow \mathit{char}(?t_2(a))\ \wedge\ a = ?t_1(a) * ?t_2(a); \\
\\
]\!] \Longrightarrow \mathit{Def}((\forall x.\ \mathit{front}(x) = if(?P(x), ?s_1(x), ?t_1(x)))\ \wedge \\
\qquad\qquad (\forall x.\ \mathit{last}(x) = if(?P(x), ?s_2(x), ?t_2(x)))\ \wedge\ ?Q) \\
\qquad\quad \rightarrow (\forall a.\ a \neq \epsilon \rightarrow \mathit{char}(\mathit{last}(a))\ \wedge\ a = \mathit{front}(a) * \mathit{last}(a))
\end{array}
$$

Now we can simulate the two last steps of the tableau in the same way as we have done before. The corresponding proofstates of the tableaux $\mathcal{T}_3$ and $\mathcal{T}_4$ are

$\mathcal{P}_3$

$$
\begin{array}{l}
[\![\\
\bigwedge a.[\![Ind\_hyp\ ;\ ?P(a);\ a \neq\ \epsilon]\!] \Longrightarrow char(?s_2(a))\ \wedge\ a = \epsilon{*}?s_2(a); \\
\\
\bigwedge a.[\![Ind\_hyp\ ;\ \neg?P(a);\ a \neq\ \epsilon]\!] \Longrightarrow char(?t_2(a))\ \wedge\ a = ?t_1(a){*}?t_2(a); \\
\\
]\!] \Longrightarrow Def((\forall x.\ front(x) = if(?P(x), \epsilon, ?t_1(x))) \wedge \\
\qquad\qquad (\forall x.\ last(x) = if(?P(x), ?s_2(x), ?t_2(x))) \wedge ?Q) \\
\qquad \to (\forall a.\ a \neq\ \epsilon \to char(last(a))\ \wedge\ a = front(a) * last(a))
\end{array}
$$

and $\mathcal{P}_4$:

$$
\begin{array}{l}
[\![\\
\bigwedge a.[\![Ind\_hyp\ ;\ ?P(a);\ a \neq\ \epsilon]\!] \Longrightarrow char(a); \\
\\
\bigwedge a.[\![Ind\_hyp\ ;\ \neg?P(a);\ a \neq\ \epsilon]\!] \Longrightarrow char(?t_2(a))\ \wedge\ a = ?t_1(a){*}?t_2(a); \\
\\
]\!] \Longrightarrow Def((\forall x.\ front(x) = if(?P(x), \epsilon, ?t_1(x))) \wedge \\
\qquad\qquad (\forall x.\ last(x) = if(?P(x), x, ?t_2(x))) \wedge ?Q) \\
\qquad \to (\forall a.\ a \neq\ \epsilon \to char(last(a))\ \wedge\ a = front(a) * last(a))
\end{array}
$$

5.4.4.2 Second case ($a = u.y$ and $y \neq\ \epsilon$)

We begin as in the first case. By the equality rule applied to axiom $u.y_1 * y_2 = u.(y_1 * y_2)$ and row 4 .

	$u.y_1 * y_2 = u.(\boxed{y_1 * y_2}^{-})$			
4		$char(z_2)\ \wedge\ a = z_1 * z_2$	z_1	z_2

with the substitution $\theta = \{z_1 \leftarrow u.y_1, z_2 \leftarrow y_2\}$, we obtain the additional row 7 that forms with $\mathcal{T}_4$ the new tableau $\mathcal{T}_5$

7		$char(y_2)\ \wedge\ a = u.(y_1 * y_2)$	$u.y_1$	y_2

In Isabelle we first derive the following rule

$$[\![w1 * w2 = v.(y_1 * y_2);\ \ char(y_2)\ \wedge\ b = v.(y_1 * y_2)]\!] \Longrightarrow char(w_2)\ \wedge\ b = w_1 * w_2$$

Than we apply the same tactic as before to the second subgoal. We resolve the second goal against this rule

$$
\begin{array}{l}
[\![\\
\quad \bigwedge a.[\![Ind\_hyp\ ;\ ?P(a);\ a \neq\ \epsilon]\!] \Longrightarrow char(a); \\[1ex]
\quad \bigwedge a.[\![Ind\_hyp\ ;\ \neg ?P(a);\ a \neq\ \epsilon]\!] \Longrightarrow ?t_1(a) * ?t_2(a) = ?u(a).(?y_1(a) * ?y_2(a)); \\[1ex]
\quad \bigwedge a.[\![Ind\_hyp\ ;\ \neg ?P(a);\ a \neq\ \epsilon]\!] \Longrightarrow char(y_2(a))\ \wedge\ a = ?u(a).(?y_1(a) * ?y_2(a)) \\[1ex]
]\!] \Longrightarrow Def((\forall x.\ front(x) = if(?P(x), \epsilon, ?t_1(x)))\ \wedge \\
\qquad\qquad (\forall x.\ last(x) = if(?P(x), x, ?t_2(x)))\ \wedge\ ?Q) \\
\qquad \to (\forall a.\ a \neq\ \epsilon \to char(last(a))\ \wedge\ a = front(a) * last(a))
\end{array}
$$

We refine the second subgoal against `append_Cons`. The proofstate becomes $\mathcal{P}_5$

$$
\begin{array}{l}
[\![\\
\quad \bigwedge a.[\![Ind\_hyp\ ;\ ?P(a);\ a \neq\ \epsilon]\!] \Longrightarrow char(a); \\[1ex]
\quad \bigwedge a.[\![Ind\_hyp\ ;\ \neg ?P(a);\ a \neq\ \epsilon]\!] \Longrightarrow char(y_2(a))\ \wedge\ a = ?u(a).(?y_1(a) * ?y_2(a)) \\[1ex]
]\!] \Longrightarrow Def((\forall x.\ front(x) = if(?P(x), \epsilon, ?u(x).?y_1(x)))\ \wedge \\
\qquad\qquad (\forall x.\ last(x) = if(?P(x), x, ?y_2(x)))\ \wedge\ ?Q) \\
\qquad \to (\forall a.\ a \neq\ \epsilon \to char(last(a))\ \wedge\ a = front(a) * last(a))
\end{array}
$$

Now we can use the induction hypothesis introduced in row 2. Intuitively the variables y_1 and y_2 occuring in row 7 will be substituted by $front(x)$ and $last(x)$, where x is a variable that will instantiate later to a term that depends on a. This can be realized by application of the equality rule to row 2 and row 7.

2	$if\, x \neq\ \epsilon\ then$ $char(last(x))\ \wedge\ x = \boxed{front(x) * last(x)}^{-}$			
7		$char(y_2)\ \wedge\ a = u.(y_1 * y_2)$	$u.y_1$	y_2

with the substitution $\theta = \{y_1 \leftarrow front(x), y_2 \leftarrow last(x)\}$.

The new tableau $\mathcal{T}_6$ contains the following new row:

8		$x <_{string} a\ \wedge\ x \neq\ \epsilon$ $\wedge$ $char(last(x))\ \wedge\ a = u.x$	$u.front(x)$	$last(x)$

We first derive the following rule:

$$
\begin{array}{l}
[\![\\
\quad < w,\ b > :?r \to\ (w \neq\ \epsilon \to char(w_2)\ \wedge\ w = w_1 * w_2); \\[1ex]
\quad < w,\ b > :?r\ \ \wedge\ \ w \neq\ \epsilon\ \wedge\ char(w_2)\ \wedge\ b = v.w \\[1ex]
]\!] \Longrightarrow char(w_2)\ \wedge\ b = v.(w_1 * w_2)
\end{array}
$$

We resolve the second subgoal against this rule we obtain the following proofstate:

$$
\begin{array}{l}
[\![\\
\quad \bigwedge a.[\![Ind\_hyp\ ;\ ?P(a);\ a \neq \epsilon]\!] \Longrightarrow char(a); \\[1ex]
\quad \bigwedge a.[\![Ind\_hyp\ ;\ \neg ?P(a);\ a \neq \epsilon \\
\qquad]\!] \Longrightarrow <?w(a),\ a> :?r \to\ (?w(a) \neq \epsilon \to char(?y_2(a)) \wedge ?w(a) =?y_1(a)*?y_2(a)); \\[1ex]
\quad \bigwedge a.[\![Ind\_hyp\ ;\ \neg ?P(a);\ a \neq \epsilon \\
\qquad]\!] \Longrightarrow <?w(a),\ a> :?r \wedge ?w(a) \neq \epsilon\ \wedge\ char(?w_2(a))\ \wedge\ a =?u(a).?w(a) \\[1ex]
]\!] \Longrightarrow Def((\forall x.\ front(x) = if(?P(x), \epsilon, ?u(x).y_1(x))) \wedge \\
\qquad\qquad (\forall x.\ last(x) = if(?P(x), x, ?y_2(x))) \wedge ?Q) \\
\qquad \to (\forall a.\ a \neq \epsilon \to char(last(a))\ \wedge\ a = front(a) * last(a))
\end{array}
$$

Now we solve the second subgoal by assumption to obtain proofstate $\mathcal{P}_6$

$$
\begin{array}{l}
[\![\\
\quad \bigwedge a.[\![Ind\_hyp\ ;\ ?P(a);\ a \neq \epsilon]\!] \Longrightarrow char(a); \\[1ex]
\quad \bigwedge a.[\![Ind\_hyp\ ;\ \neg ?P(a);\ a \neq \epsilon \\
\qquad]\!] \Longrightarrow\ \ <?x(a),\ a> :?r \wedge ?x(a) \neq \epsilon\ \wedge\ char(last(?x(a))\ \wedge\ a =?u(a).?x(a)) \\[1ex]
]\!] \Longrightarrow Def((\forall x.\ front(x) = if(?P(x), \epsilon, ?u(x).front(?x(a)))) \wedge \\
\qquad\qquad (\forall x.\ last(x) = if(?P(x), x, last(?x(a)))) \wedge ?Q) \\
\qquad \to (\forall a.\ a \neq \epsilon \to char(last(a))\ \wedge\ a = front(a) * last(a))
\end{array}
$$

Now we use the induction hypothesis again to prove the recursion step of the function *last*. This will be achieved by applying nonclausal resolution to row 2 and row 8.

2	$if\ x' \neq \epsilon\ then$ $\boxed{char(last(x'))}^{-}\ \wedge\ x' = front(x') * last(x')$			
8		$x <_{string} a\ \wedge\ x \neq \epsilon$ $\wedge$ $\boxed{char(last(x))}^{+}\ \wedge\ a = u.x$	$u.front(x)$	$last(x)$

The new tableau $\mathcal{T}_7$ consists of the old one augmented with the new row 9.

9		$(x <_{string} a)$ $\wedge$ $(x \neq \epsilon)\ \wedge\ a = u.x$	$u.front(x)$	$last(x)$

We repeat the same activity as before, thereby we derive the following rule

$$
\begin{array}{l}
[\![\\
\quad < y,\ a > : r \rightarrow\ (y \neq\ \epsilon \rightarrow E\ \wedge\ y = front(y) * last(y)); \\
\\
\quad < y,\ a > : r\ \wedge\ y \neq\ \epsilon\ \wedge\ a = u.y \\
\\
]\!] \Longrightarrow E\ \wedge\ a = u.y
\end{array}
$$

The new proofstate $\mathcal{P}_7$ has the following form:

$$
\begin{array}{l}
[\![\\
\quad \bigwedge a.[\![Ind\_hyp\ ;\ ?P(a);\ a \neq\ \epsilon]\!] \Longrightarrow char(a); \\
\\
\quad \bigwedge a.[\![Ind\_hyp\ ;\ \neg ?P(a);\ a \neq\ \epsilon \\
\qquad]\!] \Longrightarrow\ \ <?x(a),\ a> :?r\ \wedge\ ?x(a) \neq\ \epsilon\ \wedge a = u(a).?x(a)) \\
\\
]\!] \Longrightarrow Def((\forall x.\ front(x) = if(?P(x), \epsilon, ?u(x).front(?x(a)))) \wedge \\
\qquad\qquad (\forall x.\ last(x) = if(?P(x), x, last(?x(a)))) \wedge ?Q) \\
\qquad \rightarrow (\forall a.\ a \neq\ \epsilon \rightarrow char(last(a))\ \wedge\ a = front(a) * last(a))
\end{array}
$$

We continue the deduction process by applying some deduction rules to some axioms of string theory such as the *decomposition-property*, $s \neq\ \epsilon \rightarrow s = hd(s).tl(s)$, and the *trichometry-property*, $s = \epsilon \vee char(s) \vee tl(s) \neq\ \epsilon$. We obtain the following tableau $\mathcal{T}_{10}$

label	*Assertions*	*Goals*	*Outputs*	
			front(a)	*last*(a)
1		*if* $a \neq\ \epsilon$ *then* $char(z_2)\ \wedge\ (a = z_1 * z_2)$	z_1	z_2
2	*if* $x <_{string} a$ *then* *if* $x \neq\ \epsilon$ *then* $char(last(x)) \wedge$ $x = front(x) * last(x)$			

3	$a \neq \epsilon$			
4		$char(z_2) \wedge a = z_1 * z_2$	z_1	z_2
5		$char(z_2) \wedge a = z_2$	ϵ	z_2
6		$char(a)$	ϵ	a
7		$char(y_2) \wedge a = u.y_1 * y_2$	$u.y_1$	y_2
8		$x <_{string} a \wedge x \neq \epsilon$ $\wedge$ $char(last(x)) \wedge a = u.x$	$u.front(x)$	$last(x)$
16 9		$(x <_{string} a)$ $\wedge$ $(x \neq \epsilon) \wedge a = u.x$	$u.front(x)$	$last(x)$
10		$(tl(a) <_{string} a)$ $\wedge$ $(tl(a) \neq \epsilon) \wedge a \neq \epsilon$	$hd(a).front(tl(a))$	$last(tl(a))$
11		$tl(a) <_{string} a \wedge a \neq \epsilon$ $\wedge$ $char(a) \neq \epsilon$	$hd(a).front(tl(a))$	$last(tl(a))$
12		$tl(a) <_{string} a \wedge \neg char(a)$	$hd(a).front(tl(a))$	$last(tl(a))$

and its corresponding proofstate $\mathcal{P}_{10}$

$$
\begin{array}{l}
[\![\\
\quad \bigwedge a.[\![Ind\_hyp\ ;\ ?P(a);\ a \neq \epsilon]\!] \Longrightarrow char(a); \\
\\
\quad \bigwedge a.[\![Ind\_hyp\ ;\neg ?P(a);\ a \neq \epsilon]\!] \Longrightarrow <t,\ l>(a)a :?r \wedge \neg char(a) \\
\\
]\!] \Longrightarrow Def((\forall x.\ front(x) = if(?P(x), \epsilon, hd(x).front(tl(x)))) \wedge \\
\qquad\qquad (\forall x.\ last(x) = if(?P(x), x, last(tl(x)))) \wedge ?Q) \\
\qquad \rightarrow (\forall a.\ a \neq \epsilon \rightarrow char(last(a)) \wedge a = front(a) * last(a))
\end{array}
$$

Now we apply the case analysis in our deduction process. This can be accomplished by applying the nonclausal resolution to the goals 6 and 12.

6		$\boxed{char(a)}^{-}$	ϵ	a
12		$tl(a) <_{string} a \wedge \neg \boxed{char(a)}^{-}$	$hd(a).f(tl(a))$	$g(tl(a))$

This yields row 13 and the corresponding new tableau $\mathcal{T}_{11}$.

13		$tl(a) <_{string} a$	$if\ char(a)$ $then\ \epsilon$ $else\ hd(a).front(tl(a))$	$if\ char(a)$ $then\ a$ $else\ last(tl(a))$

In Isabelle we have already performed the case analysis, but we have not yet specified the predicate$?P(a)$. We simulate application of last case analysis in tableau as follows: First we generate two rules of the general form

$$\llbracket\ B;\ G_1[true]\ \rrbracket \Longrightarrow G_1[B]$$
$$\llbracket\ \neg B;\ G_2[false]\ \rrbracket \Longrightarrow G_2[B]$$

where G_1 and G_2 are the assertions/goals of the rows used by the applied nonclausal resolution above. In our case these rules are :

$$\llbracket B;\ true\rrbracket \Longrightarrow B$$
$$\llbracket\neg B;\ tl(a) <_{string} a\ \wedge\ \neg false\rrbracket \Longrightarrow tl(a) <_{string} a\ \wedge\ \neg B$$

which can be simplified to

$$\llbracket B\rrbracket \Longrightarrow B$$
$$\llbracket\neg B;\ tl(a) <_{string} a\ \rrbracket \Longrightarrow tl(a) <_{string} a\ \ \wedge\ \ \neg B$$

Now we resolve the subgoals 1 and 2 against these rules, thereby $?B(a)$ becomes $char(a)$. We get

$$
\begin{array}{l}
\llbracket \\
\ \ \bigwedge a.\llbracket Ind\_hyp\ ;\ ?P(a);\ a \neq\ \epsilon\rrbracket \Longrightarrow char(a); \\
\\
\ \ \bigwedge a.\llbracket Ind\_hyp\ ;\ \neg ?P(a);\ a \neq\ \epsilon\rrbracket \Longrightarrow \neg char(a); \\
\\
\ \ \bigwedge a.\llbracket Ind\_hyp\ ;\ \neg ?P(a);\ a \neq\ \epsilon\rrbracket \Longrightarrow < tl(a),\ a > :?r; \\
\\
\rrbracket \Longrightarrow Def((\forall x.\ front(x) = if(?P(x), \epsilon, hd(x).front(tl(x)))) \wedge \\
\qquad\qquad (\forall x.\ last(x) = if(?P(x), x, last(tl(x)))) \wedge ?Q) \\
\qquad \to (\forall a.\ a \neq\ \epsilon \to char(last(a))\ \wedge\ a = front(a) * last(a))
\end{array}
$$

We solve the first two subgoals by assumption, $?P(a)$ becomes $char(a)$. We introduce now the subgoal that we have omitted at the start. The proofstate becomes $\mathcal{P}_{11}$.

$$
\begin{array}{l}
\llbracket \\
\ \ \bigwedge a.\llbracket Ind\_hyp\ ;\ \neg char(a);\ a \neq\ \epsilon\rrbracket \Longrightarrow < tl(a),\ a > :?r; \\
\\
\ \ \llbracket Def((\forall x.\ front(x) = if(char(x), \epsilon, hd(x).front(tl(x)))) \wedge \\
\qquad\quad (\forall x.\ last(x) = if(char(x), x, last(tl(x))))\ \ \rrbracket \Longrightarrow wf(?r) \\
\\
\rrbracket \Longrightarrow Def((\forall x.\ front(x) = if(char(x), \epsilon, hd(x).front(tl(x)))) \wedge \\
\qquad\qquad (\forall x.\ last(x) = if(char(x), x, last(tl(x)))) \wedge ?Q) \\
\qquad \to (\forall a.\ a \neq\ \epsilon \to char(last(a))\ \wedge\ a = front(a) * last(a))
\end{array}
$$

5.4.5 Choice of the Well-founded Relations

In tableau as well as in Isabelle we have finished the first part of the program synthesis, namely the programs of the desired functions are completely constructed. In the second part of the synthesis we have to prove the termination of the functions *front* and *last*. This takes two phases:

- Choice of the well-founded Relation $?r$.
- Proving that $?r$ satisfies the constraints generated by the induction hypothesis during the correctness proof.

In the deduction process we have already chosen a well-founded relation, namely the proper substring relation $<_{string}$. Further we assume that we have already derived the constraint $tl(a) <_{string} a$, so we can apply it to the *last* row and we obtain the final row 14 and the final tableau $\mathcal{T}_{12}$.

14		$true$	$if\ char(a)$ $then\ \epsilon$ $else\ hd(a).front(tl(a))$	$if\ char(a)$ $then\ a$ $else\ last(tl(a))$

In Isabelle the metavariable $?r$ can be instantiated with the same relation as in the tableau. Additionally we prove its well-foundedness and that it contains the pair $<tl(a),\ a>$. Doing this we get the simplified final proofstate $\mathcal{P}_{12}$

$$\begin{array}{l}[\![\forall x.\ front(x) = if(char(x), \epsilon, hd(x).front(tl(x)));\\ \ \ \forall x.\ last(x) = if(char(x), x, last(tl(x)));\\ ]\!] \Longrightarrow (\forall a.\ a \neq\ \epsilon \rightarrow char(last(a))\ \wedge\ a = front(a) * last(a))\end{array}$$

5.5 Translation of *DTS*'s Rules in Isabelle Tactics

In the above section we have given an example in full details to illustrate the basic ideas of the translation of the deductions rules in Isabelle tactics. In this section we want give a general formal account of this translation. Let $\forall a.\ \exists z.\ spec(a, z)$ let be a first-order specification of a program called f. The synthesis of f in *DTS* begins by creating a row that contains the formula $spec(a, z)$ as goal. This row forms the initial Tableau. Applications of the deduction rules create additional rows that forms with the older ones the current tableaux, called also subtableaux. The deduction process terminates if a final row[9] is created. A tableau that contains a final row is said to be "**closed**". In Isabelle we begin the correctness proof of the specification by typing the goal $Def(?H) \rightarrow \forall a.\ spec(a, ?f(a))$ in the corresponding theory. The analogs to subtableaux are proofstates in Isabelle. They are represented as propositions in the metalanguage. The initial proofstate is the trivial proposition

$$[\![Def(?H) \rightarrow \forall a.\ spec(a, ?f(a))]\!] \Longrightarrow Def(?H) \rightarrow \forall a.\ spec(a, ?f(a)) \qquad (\ddagger)$$

[9] A row whose assertion/goal is $false/true$ (see section 2.2.1)

Derived rules can be applied to the proofstate. The newly obtained proofstate is created from the old one in a "destructive" way, that means during the refinement of subgoals of the old proofstate we instantiate some free variables of the whole proofstate using the corresponding unifier, and we eventually remove some subgoals and add some new ones. The final proofstate is the proposition $(Def(?H) \to \forall a.\ spec(a, ?f(a))\theta$, where θ denotes the composition all unifier computed during the proof. Roughly speaking the difference between DTS and Isabelle is that applications of the substitutions in tableaux are locally in the new rows, this means during the application of a deduction rule the computed unifier will be applied only to the rows that will be generated. But in Isabelle substitutions will applied globally to the proofstate; moreover, generated rows in tableau will never be affected, removed or modified.
The relevant questions for our simulation are:

- What correspond the initial tableaux in Isabelle?
- How can we represent the subtableaux in Isabelle?
- How will we transform the deduction rules into Isabelle rules?

5.5.1 Representation of Tableaux in Isabelle

5.5.1.1 Initial Tableaux

The initial tableau of the correctness proof of the above specification has the form

$Assertions$	$Goals$	$Outputs$
		$f(a)$
	$spec(a, z)$	z

where a is a skolem constant and z is a new variable.
The corresponding proofstate in Isabelle is

$$[\![Def(?H) \Longrightarrow \forall a.\ spec(a, ?f(a))]\!] \Longrightarrow Def(?H) \to \forall a.\ spec(a, ?f(a))$$

This proofstate is obtained by moving the accumulator variable in the premises context in the initial proofstate (‡).

5.5.1.2 Subtableaux

A subtableau $\mathcal{T}$, has the following general form:

Assertions	*Goals*	*Outputs*
		$f(a)$
A_1		
$\vdots$		
A_n		
	G_1	s_1
	$\vdots$	
	G_m	s_m

where $s_1, \ldots, s_m$ are the actual constructed outputs. The assertions $A_1, \ldots, A_n$ either are axioms from the underlying theory or created using the deduction rule *ifsplit*, the *induction* rule or a *skolemization* rule. Normally assertions no have associated output entries. The goals $G_1, \ldots, G_m$ are produced using deduction rules such as the *equality* rule, *nonclausal resolution*, etc. Roughly speaking a representation of the tableau $\mathcal{T}$ is an Isabelle proposition $\mathcal{P}$ of the form

$$
\begin{array}{l}
[\![\\
\quad \left.\begin{array}{l}
[\![\underbrace{\ldots}_{\textit{premise list}}]\!] \Longrightarrow \underbrace{\ldots}_{\textit{conclusion}} \\
\vdots \\
[\![\ldots]\!] \Longrightarrow \ldots
\end{array}\right\} \textit{Subgoals of } \mathcal{P} \\
]\!] \Longrightarrow Def(\underbrace{\ldots}_{\textit{accumulator}}) \rightarrow \forall a.\ spec(a, ?f(a))
\end{array}
$$

In the tableau rows represent alternatives, that means if one of the assertions/goals is *false*/*true* then the tableau is closed. These alternatives would be simulated by requiring the following conditions:

1. The conclusion of each subgoal of $\mathcal{P}$ is a disjunction of the tableau goals, and
2. The premise list of each subgoal of $\mathcal{P}$ contains the tableau assertions.

$(\star)$

So the subgoals would have the form

$$[\![A_1;\ \ldots;\ A_n]\!] \Longrightarrow G_1 \vee\ \ldots\ \vee G_m$$

So we need practically only one subgoal to represent $\mathcal{T}$. Why could $\mathcal{P}$ contains more as one subgoals?. To answer this question we consider the following situation: In the deductive process we will use the goal G_1 of $\mathcal{T}$ twice. Furthermore we assume that G_1 contains a free variable u that will be instantiated with different terms in both uses. In the first use u will be instantiated with a term t_1 and in the second use it will be instantiated with a term t_2. We suppose further that the variable u occurs in the output entry s_1. Then in each use of the goal G_1 we generated a row that contains an output entry o_1 (respectively o_2), in which u is instantiated with t_1 (respectively t_2). These output entries could be assembled in a conditional term: *if* P *then* o_1 *else* o_2, if we apply a *nonclausal resolution* rule corresponding to a case analysis to their rows. Let us now return our attention to the proofstate $\mathcal{P}$. As we have mentioned above in Isabelle we can't substitute u twice. Therefore we first have to duplicate the goal G_1 in our proofstate. So we make a copy of G_1, in which we rename the variable $?u$[10] to $?u_2$. In the original goal we also rename $?u$ to $?u_1$. We obtain the new proofstate

$$
\begin{array}{l}
[\![\\
\quad [\![A_1;\ \ldots;\ A_n]\!] \Longrightarrow G_1(?u_1) \vee\ \ldots\ \vee G_m \\[1ex]
\quad [\![A_1;\ \ldots;\ A_n]\!] \Longrightarrow G_1(?u_2) \vee\ \ldots\ \vee G_m \\[1ex]
]\!] \Longrightarrow Def(\ldots) \rightarrow \forall a.\ spec(a, ?f(a))
\end{array}
$$

As u occurs in s_1, it also occurs in the accumulator of our proofstate. At this moment the occurrences of the variable $?u$ in the accumulator has no relationship with the variable $?u_1$ occuring in the first subgoal and the variable $2u_2$ occurring in the second subgoal. Thus, we first have to specify the relationship between these variables. In other words we want to substitute the variable $?u$ with a term that depends on the variables $?u_1$ and $?u_2$. On the other hand we know that output entries of rows generated from G_1 can be assembled only in conditional terms. This means that the variable $?u$ will be computed in two complementary "contexts". Hence we instantiate $?u$ in the accumulator with the conditional term *if*$(?B,\ ?u_1,\ u?_2)$, where the predicate $?B$ is the context where we will instantiate $?u_1$ with t_1. The variable $?u_2$ will be instantiated in the context $\neg ?B$. At this stage the predicate $?B$ is unknown and will be instantiated later again. Finally $\mathcal{T}$ would have the following representation

$$
\begin{array}{l}
[\![\\
\quad [\![A_1;\ \ldots;\ A_n;\ ?B]\!] \Longrightarrow G_1(?u_1) \vee\ \ldots\ \vee G_m \\[1ex]
\quad [\![A_1;\ \ldots;\ A_n;\ \neg ?B]\!] \Longrightarrow G_1(?u_2) \vee\ \ldots\ \vee G_m \\[1ex]
]\!] \Longrightarrow (Def(\ldots) \rightarrow \forall a.\ spec(a, ?f(a)))[?u/if(?B,\ ?u_1,\ ?u_2)]
\end{array}
$$

where $\phi[x/t]$ denotes the formula ϕ in which all occurrences of the free variable x replaced through the term t. This representation has two subgoals that corresponds to a case analysis. Note that all the above activities carried out to produced this new proofstate correspond to the application of the *case_split*

[10] Free variables in the proofsate will be affixed with question mark ?

rule. Let us summarize this again: In the tableau system we can reuse[11] rows. This really means a case analysis. In the front/last example of the above section we have used row 4 twice to generate from rows 6 and 12, to which we have applied a *nonclausal resolution* rule corresponding to a case analysis. In Isabelle we will simulate reusing of rows by applying the *case_split* rule. So the subgoals that may occur in the proposition $\mathcal{P}$ correspond to a case analysis. Further the following two requirements should be satisfied.

1. If $\mathcal{T}$ is closed then we can solve all subgoals of $\mathcal{P}$.
2. If we can apply a deduction rule to the tableau $\mathcal{T}$ then it should be able to perform the simulation in $\mathcal{P}$.

We illustrate this by examples.

Example 1 The proofstate

$$[\![Def(?H) \Longrightarrow \forall a.\ spec(a, ?f(a))]\!] \Longrightarrow Def(?H) \rightarrow \forall a.\ spec(a, ?f(a))$$

represents the initial tableau. Then the initial tableau contains the goal $spec(a, z)$ and has no assertion. Further if $spec(a, z)$ is *true* the subgoal of this proofstate would have the form $Def(\ldots) \Longrightarrow \forall a.\ true$ and it can be solved using the simplifier `simp_tac`.

Example 2 Consider the tableau $\mathcal{T}$

Assertions	*Goals*	*Outputs*
		$f(a)$
$A_1[P]$		
A_2		
	$G_1[Q]$	s_1
	G_2	s_2

A trivial representation of $\mathcal{T}$ is the proofstate

$$\begin{array}{l} [\![\\ \qquad [\![A_1[P];\ A_2]\!] \Longrightarrow G_1[Q] \vee G_2 \\ ]\!] \Longrightarrow Def(\ldots) \rightarrow \forall a.\ spec(a, ?f(a)) \end{array}$$

This proofstate can be simplified to $Def(\ldots) \rightarrow \forall a.\ spec(a, ?f(a))$ (i.e. proved), if one of the assertions A_i is *false* or one of the goals G_i is *true*. In the following we apply a deduction rule to the subtableau $\mathcal{T}$ and we observe how the proofstate $\mathcal{P}$ will be changed. By nonclausal resolution applied to A_1 and G_1 we

[11] A row r will be reused in the deduction process, if we have already applied a rule to it and we will later apply a further deduction rule to it again.

create the following new row, which with $\mathcal{T}$ forms together the new subtableau $\mathcal{T}_{new}$

	$(\neg A_1\theta[\mathit{false}]) \wedge G_1\theta[\mathit{true}]$	$s_2\theta$

where θ is the most general unifier of the sentences P and Q. In Isabelle, as we have illustrated above we really have two cases to consider: First, one of the formulas A_1 and G_1 will be reused; second, neither of them will be used later. This means, however, that we have to take care whether an assertion (or, respectively a goal) will be used in the deduction process more than once. This isn't possible in an **online-simulation**. Hence, we will always apply the *case_split* rule before the simulation of each deduction rule. Following this method we may simplify the simulation but the representations of tableaux may become complex and contain superfluous information. For the sake of the simplicity we assume that A_1 (respectively G_1) contains only a free variable u (respectively v) that was substituted during the application of the nonclausal resolution. We begin by making copies for A_1 and G_1

$$
\begin{array}{l}
[\![\\
\qquad [\![(A_1[P])[?u/?u_1];\ A_2;\ ?B]\!] \Longrightarrow (G_1[Q])[?v/?v_1] \vee G_2 \\
\qquad [\![(A_1[P])[?u/?u_2];\ A_2;\ \neg ?B]\!] \Longrightarrow (G_1[Q])[?v/?v_2] \vee G_2 \\
]\!] \Longrightarrow (Def(\ldots) \to \forall a.\ spec(a, ?f(a)))[(?u,\ ?v)/(if(?B, ?u_1, ?u_2),\ if(?B, ?v_1, ?v_2))]
\end{array}
$$

where $\phi[(u_1, \ldots, u_n)/(t_1, \ldots, t_n)]$ stands for $(\phi[u_1/t_1]) \ldots [u_n/t_n]$.
Now we simulate the *nonclausal resolution* rule as we already explained in the front/last example of the above[12] section. We obtain the new proofstate $\mathcal{P}_{new}$:

$$
\begin{array}{l}
[\![\\
\qquad [\![A_2;\ ?B]\!] \Longrightarrow ((\neg A_1\theta[\mathit{false}]) \wedge G_1\theta[\mathit{true}]) \vee G_2 \\
\qquad [\![(A_1[P])[?u/?u_2];\ A_2;\ \neg ?B]\!] \Longrightarrow (G_1[Q])[?v/?v_2] \vee G_2 \\
]\!] \Longrightarrow (Def(\ldots) \to \forall a.\ spec(a, ?f(a)))[(?u,\ ?v)/(if(?B, ?u_1, ?u_2),\ if(?B, ?v_1, ?v_2)]\theta'
\end{array}
$$

[12] Here we are only interested in the new proofstate, later we will give a formal description of the simulation of each deduction rule.

where the substitution θ' is θ extended by $\{u_1 \leftarrow u\theta,\ v_1 \leftarrow v\theta\}$. We display the new tableau $\mathcal{T}_{new}$ to compare it with $\mathcal{P}_{new}$

Assertions	*Goals*	*Outputs*
		$f(a)$
$A_1[P]$		
A_2		
	$G_1[Q]$	s_1
	G_2	s_2
	$(\neg A_1\theta[\mathit{false}]) \wedge G_1\theta[\mathit{true}]$	$s_1\theta$

Let us explain why $\mathcal{P}_{new}$ is a representation of $\mathcal{T}_{new}$. If the assertion A_2 is *false* (or respectively the goal G_2 is *true*) then we can solve all subgoals of $\mathcal{P}_{new}$. If the assertion $A_1[P]$ is *false* (or, respectively, the goal $G_1[P]$ is *true*) then we can solve the second subgoal using the classical prover `fast_tac` (or, respectively the simplifier `simp_tac`). But how can we solve the first subgoal in this case? Intuitively we have applied the *case_split* before, since we thought that $A_1[P]$ (or, respectively $G_1[P]$) will be used more as once. But this was not necessary, because the tableau was already closed. So it is allowed to undo the above application of the *case_split* rule. This can be done by instantiating the predicate $?B$ with *false*. In this way we can solve the first subgoal using the classical prover. On the other hand if the newly created goal $(\neg A_1\theta[\mathit{false}]) \wedge G_1\theta[\mathit{true}]$ is *false*. The first subgoal can be solved using the simplifier. Further, because the tableau in now closed, we would not have needed to apply the *case_split* rule. Therefore, to solve the second subgoal we instantiate $?B$ with *true* and apply the classical prover.

The above example shows that the new proofstate $\mathcal{P}_{new}$ represents the new tableau $\mathcal{T}_{new}$, although the requirements $(\star)$ are not satisfied. For instance the new created goal $(\neg A_1\theta[\mathit{false}]) \wedge G_1\theta[\mathit{true}]$ doesn't occur in the conclusion of the second subgoal of $\mathcal{P}_{new}$. Therefore we now suggest new weaker requirements, which, however, guarantee that if a tableau is closed then its representation is provable. We will call these requirements *invariants* and we will prove that they are preserved during the simulation. Informally our invariants say : First, the conclusion of each subgoal of a tableau representation is a disjunction of some tableau goals. This can obviously be verified in $\mathcal{P}_{new}$. Second, the premise list of each subgoal contains some tableau assertions. In $\mathcal{P}_{new}$ the first subgoal contains the assertion A_2 and the second subgoal contains the assertions A_1 and A_2. Up to now yet the above both requirements don't ensure the "closure property" (if a tableau is closed then its representation is provable). So we further require that if an assertion (respectively, a goal) doesn't occur in the premise list (respectively, the conclusion) of a subgoal then this subgoal contains a premise of the form $?B$ or $\neg ?B$. The first and second requirements ensure that if a tableau is closed, then in its representation we can solve all the subgoals containing its final row. The last requirement ensures that the remaining subgoals contain premises of the form $?B$ and $\neg ?B$ and these subgoals could be solved by instantiating $?B$ with *true* or *false*. Instantiating $?B$ with one of the truth values intuitively means undoing the application of the *case_split* rule.

Now we are ready to give a formal definition establishing the relationship between a subtableau and its corresponding proposition in Isabelle during the simulation.

Definition 5.1 (representation)
A proposition $\mathcal{P}$ *is a representation* of a subtableau $\mathcal{T}$, if it holds:

(i) The conclusion of each subgoal of $\mathcal{P}$ is a disjunction of goals of $\mathcal{T}$.

(ii) The premise list of each subgoal of $\mathcal{P}$ contains assertions of $\mathcal{T}$ and premises of the form $?B$ or $\neg ?B$, where $?B$ is a metavariable ranging over formulas.

(iii) For each assertion A (or respectively goal G) of $\mathcal{T}$ there is a list of predicates $P_1, \ldots P_n, n \geq 0$ such that

   (a) $P_i = ?B_i$ or $P_i = \neg ?B_i$, where $?B_i$ is a metavariable for $1 \leq i \leq n$

   (b) The premise list of each subgoal of $\mathcal{P}$ contains either all these predicates or a negation of one of them.(By the negation of $\neg ?B$ we mean $?B$.)

   (c) A (or respectively G) occurs in all subgoals those contain all these predicates in their premise.

□

Remark The predicates $P_1, \ldots P_n$ are introduced in the subgoal of a representation by applying the *case_split* rule. The list of these predicates for an assertion A (or respectively a goal G) could be empty, in this case A (or respectively G) will occur in all subgoals. For instance, in the above example the assertion A_2 and the goal G_2 occur in the both subgoals. Their corresponding lists of predicates are empty. The assertion A_1 and the goal G_1 occur only in the second subgoal. Their corresponding lists of predicates is $?B$. □

Theorem 2 (closure property) *If $\mathcal{P}$ is a representation of a closed tableau $\mathcal{T}$ then all subgoals of $\mathcal{P}$ can be solved.*

Proof (closing property)
First, we suppose that the tableau is closed, because it contains the assertion *false*. As the proofstate $\mathcal{P}$ is a representation of $\mathcal{T}$, there is a list of predicates $P_1, \ldots, P_n$ such that each subgoal of $\mathcal{P}$ contains either all these predicates and *false* in its premises or a negation of one of these predicates. So in this case we can solve all subgoals of $\mathcal{P}$ in the following way:

- If the subgoal contains *false* in its premises and therefore as a hypothesis is false, the sequent is provable using the classical prover `fast_tac`.

- Otherwise the subgoal contains a negation of one of the above predicates, in this case we instantiate this predicate with *false* or *true* and we apply the classical prover as before.

Now we suppose that the tableau is closed, because it contains the trivial goal *true*. In this case we proceed as before, except that the subgoals whose conclusions contains *true* can be solved using the simplifier `simp_tac`. □

Theorem 3 (case_split property) *Let $\mathcal{P}$ be a representation of a tableau $\mathcal{T}$ and A an assertions (respectively, G a goal) of $\mathcal{T}$. If we apply the case_split rule to all the subgoals of $\mathcal{P}$ containing A (respectively G) then the new obtained proposition is also a representation of $\mathcal{T}$.*

Proof (case_split property)
Follows directly from definition 5.1. □

5.5.2 Tactics for the Deduction Rules

In the following we will develop those Isabelle tactics which simulate the *DTS* rules. Let $\mathcal{T}$ be a tableau and $\mathcal{P}$ its representation in the sense of the above definition. Further let r be a deduction rule, whose application to $\mathcal{T}$ produces a new tableau $\mathcal{T}_{new}$. The simulation of r consists of two parts: First, we make copies of the required rows of r. This step will be aschieved by the *case_ split* rule. Second, we apply an appropriate Isabelle tactic that also transforms $\mathcal{P}$ into a new proofstate $\mathcal{P}_{new}$. Seond we prove that $\mathcal{P}_{new}$ is a representation of $\mathcal{T}_{new}$. From theorem 3 we know that the application of the case split rule preserves the "representation property". So in the following we will always suppose that by the simulation of a deduction rule the case split rule has been already applied.

5.5.2.1 The Induction Tactic

The induction rule of the tableaux system is based on the well-founded induction principle. Whose application to $\mathcal{T}$ introduces a new assertion, which forms with $\mathcal{T}$ the new tableau $\mathcal{T}_{new}$. This rule has the form

$Assertions$	$Goals$	$Outputs$
		$f(a)$
	$P(a, z)$	z
$x <_w a \to P(x, f(x))$		

where $<_w$ is a well-founded order. Note that the proof of well-foundedness of $<_w$ can't be performed in the tableaux system. In Isabelle we dispose of the same rule, called `wf_induct` (see section 4.3.1) from which we have derived several recursion schemata which are useful for the program synthesis. For instance `indschemaf1_1p1` (see figure 5.2) is intended for specification of a program which has only one parameter. In Isabelle we have developed a special tactic that simulates the induction rule. This tactic is called *IND_tac* and it is based on four phases:

1. Generate the appropriate recursion schema *induct_schema*, such as `indschemaf1_1p1`. As mentioned above such recursion schemata are specializations of `wf_induct` and can be then proved in a straightforward way.

2. Apply *induct_schema* to all subgoals, containing the goal $P(a, z)$ as conclusion. These subgoals have the form

$$premises\_list \Longrightarrow P(a, z) \vee \ldots$$

   We get the two subgoals respectively,

$$\begin{array}{l} [\![Def(?Q); \\ \ <?x(a), a>:?r \rightarrow P(?x(a), f(?x(a))); \\ \ premises\_list]\!] \Longrightarrow P(a, ?z(a)) \vee \ldots \end{array}$$

   and

$$\begin{array}{l} [\![Def(\forall x.\ f(x) = ?z(x)); \\ \ Def(?Q); \\ \ premises\_list]\!] \Longrightarrow \ wf(?r) \end{array}$$

   This step will be achieved using the tactic *INDTAC*, that we have implemented and explained in section 5.3.3. Note that during this step the accumulator is extended by the definition of the function f and a further formula $?Q$, through which the accumulator remains open to set up additional definitions.

3. Prove the well-foundedness of the order $<_w$. We obtain the proposition $wf(<_w)$.

4. Resolve the second subgoal against $wf(<_w)$ thereby instantiating $?r$ with $<_w$. $\mathcal{P}_{new}$ contains the same subgoals as $\mathcal{P}$, except that the subgoals containing the goal $P(a, z)$ now have the additional premise: $<?x(a), a>:<_{new} \rightarrow P(?x(a), f(?x(a)))$.

Lemma 1 (the induction tactic preserves representation) *The proofstate $\mathcal{P}_{new}$ represents the tableau $\mathcal{T}_{new}$.*

Proof (induction tactic preserves representation)
Conditions (i) and (ii) of the definition 5.1 are obviously satisfied. We now prove the condition (iii). Since $\mathcal{P}$ is a representation of $\mathcal{T}$, there is a predicates list $P_1, \ldots, P_n$ for the goal $P(a, z)$ that satisfies the conditions (a), (b) and (c) required in (iii). These predicates also satisfy the above conditions for the new assertions $<?x(a), a>:?r \rightarrow P(?x(a), f(?x(a)))$, because this assertion occurs in the same subgoals as $P(a, z)$. □

5.5.2.2 Tactics for the Skolemization Rules

These rules consists of quantifier elimination. We will describe only the G-form of the $\forall$-*elimination* rule. The other variants can be simulated in the same way. The rule is

Assertions	*Goals*	*Outputs*
	$\forall x.G(x)$	s
	$G(a)$	s

where a is new skolem constant.

The Isabelle tactic simulating this rule is based essentially on the universal quantifier elimination rule `allI` (see figure 3.2). This tactic is called $\forall\_G\_tac$ and based on two phases:

1. Derive a special instance of the rule `allI`. This instance has the following form
$$\bigwedge x.\ (?P(x) \vee ?F) \Longrightarrow \forall x.\ (?P(x) \vee ?F)$$
And it can be automatically proved using the simplifier.

2. Resolve all subgoals of the form
$$[\![premises\_list]\!] \Longrightarrow (\forall x.\ G(x)) \vee \ldots$$
against the above rule, we get the subgoal
$$\bigwedge x.\ [\![premises\_list]\!] \Longrightarrow G(x) \vee \ldots$$
The variable x is now bound and will be treated as a skolem constant in the rest of the proof.

Lemma 2 (skolomezation tactics preserve representation) *The proofstate $\mathcal{P}_{new}$ represents the tableau $\mathcal{T}_{new}$.*

Proof (skolomezation tactics preserve representation)
The proof is similar to the above proof. □

5.5.2.3 The Tactics for the Nonclausal Resolution Rule

This rule is the main deduction rule of the tableau system. Here we want to treat only the AG-form other variants can be simulated in a similary way. We display this rule again

Assertions	*Goals*	*Outputs*
$A[P]$		
	$G[Q]$	s
	$\neg(A\theta[false]) \ \wedge \ G\theta[true]$	$s\theta$

where P and Q are quantifier-free sentences and θ is their most general unifier (for the notation $A[P]$ see section 2.2.2). The tactic for the *nonclausal resolution* rule (AG-form) is called *NC_AG_tac* and is based on the following four phases:

1. Generate the following rule, called *NC_AG_rule*

$$[\![A[Q];\ ((\neg A[false]) \wedge G[true]) \vee\ F]\!] \Longrightarrow G[Q] \vee\ F$$

2. Prove *NC_AG_rule* automatically using the classical prover.

3. Refine all subgoals of the form

$$[\![A[P];\ rest\_premises]\!] \Longrightarrow G[Q] \vee \ldots$$

   by *NC_AG_rule* thereby getting the two new subgoals

$$[\![A[P]; rest\_premises]\!] \Longrightarrow A[Q]$$

   and

$$[\![A[P]; rest\_premises]\!] \Longrightarrow\ ((\neg A[false]) \wedge G[true]) \vee\ \ldots$$

4. Solve the first new subgoal by assumption. This leads the system to unify the sentences P and Q. The obtained unifier θ will be applied to the whole proofstate. The second subgoal then becomes

$$[\![A\theta[P\theta]; rest\_premises]\!] \Longrightarrow\ ((\neg A\theta[false]) \wedge G\theta[true]) \vee\ \ldots$$

   The assumption $A\theta[P\theta]$ doesn't occur in the tableau as an assertion. So it will not be used in the Isabelle proof and therefore we remove it from the premises list of this subgoal. Finally we get

$$[\![rest\_premises]\!] \Longrightarrow\ ((\neg A\theta[false]) \wedge G\theta[true]) \vee\ \ldots$$

Lemma 3 (nonclausal tactics preserve representation) *The proofstate $\mathcal{P}_{new}$ represents the tableau $\mathcal{T}_{new}$.*

Proof (nonclausal tactics preserve representation)
The first two conditions of the definition 5.1are obviously satisfied. The formulas $A[P]$ and $G[Q]$ satisfies condition (iii) of the definition, so the proofstate $\mathcal{P}$ is a representation of the tableau $\mathcal{T}$. Further the newly derived goal occurs only in the subgoals where the above formulas occured, so this new goal satisfies the condition (iii) too. The remaining assertions and goals are affected by neither the application of the *nonclausal resolution* rule on $\mathcal{T}$ nor by the application of the simulation tactic on $\mathcal{P}$. □

5.5.2.4 The Equality Tactics

The equality rule is a nonclausal version of the paramodulation rule. The tactics that simulate this rule are similar to the ones of the *nonclausal resolution* rule. Here we will give only the AG-form variant and the corresponding Isabelle tactic. The AG-form of the equality rule has the form

Assertions	*Goals*	*Outputs*
$A[l = r]$		
	$G < l' >$	s
	$\neg A\theta[false] \ \wedge \ G\theta < r\theta >$	$s\theta$

where θ is the most general unifier of the terms l and l' (for the notations $G < l' >$ see section 2.2.2). We call the tactic for this form of the *equality* rule *EQ_AG_tac*. It is very similar to *NC_AG_tac*, except that the generated rule is different:

1. Generate the following rule, called *NC_AG_rule*

$$[\![A[l' = r]; \ ((\neg A[false]) \wedge G < r >) \vee \ F]\!] \Longrightarrow G < l' > \vee \ F$$

2. Prove *NC_AG_rule* (automatically using the classical prover.)

3. Refine all subgoals of the form

$$[\![A[l = r]; \ rest\_premises]\!] \Longrightarrow G < r > \vee \ldots$$

   by the *NC_AG_rule* thereby getting the two new subgoals

$$[\![A[l = r]; rest\_premises]\!] \Longrightarrow A[l' = r]$$

   and

$$[\![A[l = r]; rest\_premises]\!] \Longrightarrow \ ((\neg A[false]) \wedge G < r >) \vee \ \ldots$$

4. Solve the first new subgoal by assumption. This leads the system to unify the terms l and l'. The obtained unifier θ will be applied to the whole proofstate. The second subgoal then becomes

$$[\![A\theta[(l = r)\theta]; rest\_premises]\!] \Longrightarrow \ ((\neg A\theta[false]) \wedge G\theta < r\theta >) \vee \ \ldots$$

   The assumption $A\theta[P\theta]$ doesn't occur in the tableau as assertion. So it will be not used in the Isabelle proof and therefore we remove it from the premises list of this subgoal. Finally we get

$$[\![rest\_premises]\!] \Longrightarrow \ ((\neg A\theta[false]) \wedge G\theta < r\theta >) \vee \ \ldots$$

similarly to the nonclausal condition, the equality tactics also preserve representations of tableaux.

Lemma 4 (equality tactics preserve representation) *$\mathcal{P}_{new}$ represents $\mathcal{T}_{new}$*

Proof (equality tactics preserve the representation of tableaux)
Analogous to the above proof. □

Theorem 4 *Let $\mathcal{T}$ be a tableau and $\mathcal{P}$ its representation. If we apply a deduction rule r to $\mathcal{T}$ getting thereby a new tableau $\mathcal{T}_{new}$, then the application of the simulation tactic of r produces a new proofstate $\mathcal{P}_{new}$ representing $\mathcal{T}_{new}$.*

Proof
By the lemmata 1, 2, 3 and 4. □

5.5.3 Output entries

Up to now we haven't yet addressed the programs constructed during the simulation. An important question is "What is the relationship between the computed programs in Isabelle and the output entries of the tableau at each stage of the simulation". In order to answer this question we first discuss some properties of program building in our approach and in *DTS*. *DTS*'s output entries are built in a bottom up manner. In the front/last example for instance we first have derived the output entries ϵ and $hd(x).\mathit{front}(tl(x))$ and then assembled them to the conditional term *if* $char(x)$ *then* ϵ *else* $hd(x).\mathit{front}(tl(x))$. In Isabelle we construct programs in a top-down manner. In the front/last example we have first instantiated the metavariable representing the body of the function *front* with the conditional term $\mathit{if}(?P(x),\ ?s_1(x),\ ?t_1(x))$ and then the terms $?P(x)$, $?s_1(x)$ and $t_1(x)$ were substituted by $char(x)$, ϵ and $hd(x).\mathit{front}(tl(x))$, respectively. The front/last example shows that at each step of the simulation, an Isabelle program can be seen as "a term with holes" (the holes are the metavariables). Furthermore if we skilfully instantiate some metavariables we can simplify the program to an output entry. In other words, during the simulation process if we have an output entry t occuring in the output column of a function f, then the current Isabelle program defining f can be simplified to t by instantiating some metavariables. Intuitively the output entries occuring in the output column of a function f represent the alternative continuations of f. Therefore, to reduce an Isabelle program to an output entry, the metavariables $?B$ that have to be instantiated are those occuring in conditional subterms $\mathit{if}(?B,\ \ldots,\ \ldots)$ of the program. Let us formulate this in the following definitions

Definition 5.10 (test-predicate)
In the conditional term $\mathit{if}(B,\ s,\ t)$ we call the predicate B a *test-predicate*. □

Definition 5.11 (reduction)
Let t_1 and t_2 be two terms. We say that t_1 can be simplified to t_2 and we write

$t_1 \rightsquigarrow t_2$, if we can replace some *test-predicates* in $t1$ by truth values and then evaluate the corresponding *if_then_else* branches yielding a term t that is equal to t_2 modulo renaming of free variables. $\square$

Examples: We borrow the examples here from the front/last example.

- $if(?P(a), [], ?t1(a)) \rightsquigarrow []$
- $if(?P(a), a, ?t1(a)) \rightsquigarrow a$
- $if(?P(a), [], Cons(hd(a), front(tl(a))) \rightsquigarrow Cons(hd(a), front(tl(a)))$

Intuitively the terms t_1 and t_2 occuring in the above definition correspond to the program constructed in Isabelle and an output of the simulated tableau, respectively. During the simulation process we apply the *case_split* rule before the simulation of any deduction rule. At this point we first want to establish a property that states if we apply the *case_split* rule then the reduction property remain preserved. Then *case_split* instantiates metavariables with conditional terms of the form $if(?B,\ ?s,\ ?t)$, where $?B$, $?s$ and $?t$ are metavariables.

Theorem 5 (case_split preserves reduction) *Let t_1 and t_2 be two terms. If $t_1 \rightsquigarrow t_2$ then $t_1[x/if(B,\ x_1,\ x_2)] \rightsquigarrow t_2$, where x is a metavariable occuring in t_1 and B, x_1, and x_2 are new metavariables.*

Proof (case_split preserves reduction)
This is obvious. $\square$

5.5.4 Theorem

The principal aim of our work is to give an interpretation of the deductive tableau system in Isabelle. At this point we have given the logic HOL to develop the interpretation. Further we have developed Isabelle tactics that simulate the deduction rules of the tableau system. The simulation is motivated by the front/last example of section 5.4 . In this section we formulate a theorem that states that, if we can synthesize a program from a first-order specification then we can also synthesize the same program from the same specification in Isabelle. For the sake of simplicity we consider tableaux with one single output column. The front/last example shows that the simulation can be easily generalized to tableaux with multiple output columns. In Isabelle we initialize the accumulator with the definition of the desired program first by applying induction rule. Therefore we are obliged to apply the induction at the beginning of the proof. Consequently we can simulate only tableaux, whose deduction proofs starts with applying the induction rule, but is no real restriction. Then the induction rule will be applied to the initial goal and normally this goal occurs in the first row of the tableau. Thus, we assume that the first step of the deduction process is an application of the induction rule too. Now we formulate the first important result of our work in the following theorem.

Theorem 6 (simulation) *Let $\forall a.\ \exists z.\ spec(a,z)$ be a first-order specification. If there is a tableau proof $\mathcal{T}$ of this specification with the output entry $f(a)=t$, then we can prove the proposition $Def(\forall a.\ f(a)=\tilde{t}) \rightarrow \forall a.\ spec(a, f(a))$ in Isabelle, where $\tilde{t} \rightsquigarrow t$.*

In the theorem we say that we can prove the proposition $Def(\forall a.\ f(a)=\tilde{t}) \rightarrow \forall a.\ spec(a, f(a))$ in Isabelle, this can mean that we give a verification proof of the program constructed in the tableau system. But this is not our intention. We are interested in synthesizing the same program as in the tableau. For this purpose we start the Isabelle proof with the specification $Def(?H) \rightarrow \forall a.\ spec(a, ?f(a))$ and as final proofstate we obtain $Def(\forall a.\ f(a) = \tilde{t}) \rightarrow \forall a.\ spec(a, f(a))$.
The proof that we suggest here is based on a simulation of the tableau proof $\mathcal{T}$ in Isabelle. We first formulate the following lemma

Lemma 5 *For any subtableau $\mathcal{T}_o$ of $\mathcal{T}$ of the form*

Assertions	*Goals*	*Outputs* $f(a)$
A_1		t_1
$\vdots$		$\vdots$
A_n		t_n
	G_1	s_1
	$\vdots$	
	G_m	s_m

there is a proofstate $\mathcal{P}_o$ of the form

$$[\![\ldots]\!] \Longrightarrow Def(H) \rightarrow \forall a.\ spec(a, f(a))$$

such that

(i) $\mathcal{P}_o$ *is a representation of* $\mathcal{T}_o$

(ii) H *is a metavariable if* $\mathcal{T}_o$ *is the initial tableau. Otherwise it is a conjunction of the form* $(\forall a.\ f(a) = body) \wedge \ldots$

(iii) If there is a substitution θ such that $A_i\theta$ is false (or, respectively $G_j\theta$ is true) then body $\theta \rightsquigarrow t_i\theta$ (or, respectively body $\theta) \rightsquigarrow s_j\theta$)

Proof (simulation)
We first prove the above theorem. Consider the final proof $\mathcal{T}$. By the above lemma there is an Isabelle proposition $\mathcal{P}$ of the form

$$[\![\ldots]\!] \Longrightarrow Def((\forall a.\ f(a) = body) \wedge \ldots) \rightarrow \forall a.\ spec(a, f(a))$$

such that

- $\mathcal{P}$ represents $\mathcal{T}$. By theorem 4 and theorem 2 follows that $\mathcal{P}$ is provable.
- Condition (iii) of the lemma implies that *body* $\rightsquigarrow t$. Then $\mathcal{T}$ contains either the assertion *false* or the goal *true* with the output entry t. In both cases *body* $\rightsquigarrow t$.

$\square$

Before we begin with proving the above lemma, we first want to discuss a technical problem that appear in condition (iii) of this lemma. Then there we apply the substitution θ to the term *body* and the output entry t_i (resp., s_j). First, a such application of θ is meaningful, if we have an assignment between the variables occuring in *body* and the variables occuring in t_i (resp., s_j). Second, as we already mentioned in section 3.1.3, variables occuring in an Isabelle proofstate can be lifted over some parameters by applying some tactics and rules, this means if a variable z occurs in t_i (resp., s_j) then its corresponding variable, say $?z$, occurs lifted over some parameters, say $\vec{x}$, in *body* in the form $?z(\vec{x})$. The condition (iii): $body\{z \leftarrow a, \ldots\} \rightsquigarrow t_i\{z \leftarrow a, \ldots\}$ (resp., $body\{z \leftarrow a, \ldots\} \rightsquigarrow s_j\{z \leftarrow a, \ldots\}$) stands for $body\{?z \leftarrow \lambda\vec{x}.\ a, \ldots\} \rightsquigarrow t_i\{z \leftarrow a, \ldots\}$ (resp., $body\{?z \leftarrow \lambda\vec{x}.\ a, \ldots\} \rightsquigarrow s_j\{z \leftarrow a, \ldots\}$). In other words the left occurrence of the substitution in condition (iii) can be seen as a replacement.

Proof

The first condition holds by theorem 2. We will omit it in the rest of the proof.

- **Base case** We start the tableau proof $\mathcal{T}$ by the initial tableau $\mathcal{T}_0$

Assertions	*Goals*	*Outputs*
		$f(a)$
	$spec(a, z)$	z

In Isabelle we type the goal $Def(?H) \rightarrow \forall a.\ spec(a, ?f(a))$ and we move the accumulator to the premise list of the subgoal of the initial proofstate. We get the proofstate $\mathcal{P}_0$

$$[\![Def(?H) \Longrightarrow \forall a.\ spec(a, ?f(a))]\!] \Longrightarrow Def(?H) \rightarrow \forall a.\ spec(a, ?f(a))$$

which satisfies the conditions (ii) and (iii).

Before we treat the step case we consider the first applied rule in the tableau system, namely the induction rule. The new subtableau $\mathcal{T}_1$ has the form

Assertions	*Goals*	*Outputs*
	$spec(a, z)$	z
$x <_w a \rightarrow spec(x, f(x))$		

In Isabelle we resolve the subgoal of the proofstate $\mathcal{T}_0$ using the induction tactic IND_tac described in section 5.5.2.1. We obtain the new proofstate $\mathcal{P}_1$

$$
\begin{array}{l}
[\![\\
\quad \bigwedge a.\ [\![Def(Q);\ \ <?x(a), a>:<_w \rightarrow spec(?x(a), f(?x(a)))]\!] \Longrightarrow spec(a, ?\mathit{fbody}(a)) \\
]\!] \Longrightarrow Def((\forall a.\ f(a) = ?\mathit{fbody}(a)) \wedge ?Q) \rightarrow \forall a.\ spec(a, f(a))
\end{array}
$$

We now verify the conditions (ii) and (iii) for the proofstate $\mathcal{P}_1$. Condition (ii) holds since the current accumulator has the required form. We suppose that there is a substitution $\theta = \{z \leftarrow t\}$ such that $spec(a, t)$ is $true$. The condition (iii) is also satisfied. Then we also have $?\mathit{fbody}(a)\theta \rightsquigarrow z\theta$, where the left occurrence of the substitution θ stands for $\{?fody \leftarrow \lambda x.\ z\theta\}$.

- **Step case** Let $\mathcal{T}_o$ be the current subtableau and $\mathcal{P}_o$, its representation, satisfying the conditions of the lemma. Our aim is to prove that if we apply a deduction rule to $\mathcal{T}_o$ and its simulation to $\mathcal{P}_o$ to get the new subtableau $\mathcal{T}_n$ and proofstate $\mathcal{P}_n$, then $\mathcal{T}_n$ and $\mathcal{P}_n$ satisfy the conditions of the lemma. Here we want to consider only the *nonclausal resolution* rule (AG-form). Its other variants and the remaining deduction rules can be treated in the same manner. Let $A[P]$ and $G[Q]$ be the required rows and t and s their output entries. The *nonclausal resolution* rule generates the row

	$(\neg A\sigma[\mathit{false}])\ \wedge\ G\sigma[\mathit{true}])$	*if* $P\sigma$ *then* $s\sigma$ *else* $t\sigma$

where σ is the most general unifier of the sentences P and Q. The new tableau $\mathcal{T}_n$ is formed from $\mathcal{T}_o$ and the above row. In Isabelle we apply the simulation tactic NC_AG_tac (see section 5.5.2.3 for explanation how this tactic works). The new proofstate $\mathcal{P}_n$ will have the form

$$[\![\ldots]\!] \Longrightarrow Def(H\sigma) \rightarrow \forall a.\ spec(a, f(a))$$

Obviously the accumulator of the new proofstate satisfies the second condition of the lemma by induction. So the new accumulator has the form $(\forall a.\ f(a) = \mathit{bodoy}_{new}) \wedge \ldots$. We now prove that the new proofstate also satisfies the third condition of the lemma. We assume that there is a substitution θ such that

$$((\neg A\sigma[\mathit{false}])\ \wedge\ G\sigma[\mathit{true}])\ \theta \text{ is } \mathit{true}$$

which implies that $(A\sigma[\mathit{false}])\theta$ is *false* and $(G\sigma[\mathit{true}])\theta$ is *true*. It remains to show that

$$\mathit{body}_{new}\theta\ \rightsquigarrow\ (\mathit{if}\ (P\sigma)\ \mathit{then}\ (s\sigma)\ \mathit{else}\ (t\sigma))\theta$$

can be transformed to

$$body_{new}\ \theta \rightsquigarrow if\,(P(\sigma\ \theta))\ then\ (s(\sigma\ \theta))\ else\ (t(\sigma\ \theta))$$

We distinguish two cases.

- $P\sigma$ is *true*: Then $G[Q](\sigma\ \theta) = G(\sigma\ \theta)[true] = true$ By induction hypothesis follows $body(\sigma\ \theta) \rightsquigarrow s(\sigma\ \theta)$, which is equivalent to $body_{new}\ \theta \rightsquigarrow s(\sigma\ \theta)$
- $P\sigma$ is *false*: Then $A[P](\sigma\ \theta) = A(\sigma\ \theta)[false] = false$ By induction hypothesis follows $body(\sigma\ \theta) \rightsquigarrow t(\sigma\ \theta)$, which is equivalent to $body_{new}\ \theta \rightsquigarrow t(\sigma\ \theta)$

$\square$

Chapter 6

Examples: Synthesis of a Family of Sorting Algorithms

In this chapter we present a further focus of our work, namely the derivation of several sorting algorithms. The synthesis of these programs isn't a new result. Derivations of such programs have appeared in the computer science literature. For instance Trougott [53] published derivations of a wide variety of recursive sorting programs. The programs were manually derived using the framework of Manna and Waldinger. Further publications in this area are J. Darlington [17, 15], Broy [6], Smith [51] and many others.

Our contribution are machine-checked derivations supported by the framework Isabelle. But this work is more than a mere simulation of tableaux synthesis, within Isabelle: it is a proper approach. We have benefited from the use of metavariables in Isabelle to distinguish between synthesis and termination of programs. We chose the field of sorting, first because it is very well understood in computer science, second for facility the comparison between our work and existing other ones in the area of program development.

6.1 The Theory

Isabelle comes with an encoded theory about lists in standard notation: The empty list is represented by `[]`. Recursive lists are build using the constructor `Cons`. The symbol `@` denotes the append function. `hd` and `tl` denote head and tail of a non-empty list. For our work we have extend this theory to a theory about lists whose elements admit an *total*[1] order. A sorted version of a list l is a list l' such that l' is a permutation of l and it is also ordered. In the following we write $sort(l)$ to denote the sorted list l' of l. To express the specification of the function *sort* we declare the predicates *perm* and *ordered*. The permutation of a list l is a list l' that has the same elements with the same occurrences as in l.

[1] A *total* order $\leq$ is a partial ordering that satisfies the following condition: For each elements x and y it holds $x \leq y$ or $y \leq x$. For example the list [1,4,20] is ordered with respect to the less-than relation over natural numbers.

This can be formulated as follows

name	*definition*
`perm_Nil`	`perm ([], [])`
`perm_append_singleton`	`perm (l1@[u]@l2, t1@[u]@t2)= perm(l1@l2, t1@t2)`
`perm_member`	`perm (l1, l2) --> !a. member(a, l1)= member(a, l2)`

Note that the third axioms of the above ones is needed to establish the falsehood of propositions of the form `perm([],l)`, where `l` is a nonempty list.

A list l is ordered with respect to an order $\leq$ if for all elements u and v in l if u occurs before v then $u \leq v$. Formally

name	*definition*
`ord_Nil`	`ordered([])`
`ord_singleton`	`ordered ([a])`
`ord_double_insertion`	`ordered(Cons(a, Cons(b, t))) = (a <= b) & ordered(Cons(b, t))`

We further declare two useful predicates *mini* and *maxi*. $mini(u,\ l)$ says that the element u is less or equal as all the elements of the list l. $maxi(u,\ l)$ says that u is greater or equal as the elements of l. In table 6.1 we have displayed some axioms and useful lemmata that will be needed later in the derivations. For completeness more syntactic details and the definitions of other predicates see appendix B.1. Now the formal specification of the function *sort* can be expressed as follows

$$\forall l.\ perm(l, sort(l))\ \wedge\ ordered(sort(l))$$

In Isabelle we reformulate this specification to

```
Def(?H) --> (!  l.  perm(l, ?sort(l)) & ordered(?sort(l)))
```

Remark All proofs represented in this section are taken directly from an Isabelle session (given in `typewrite font`). The symbols `!!` and `==>` stand for the quantifier $\bigwedge$ and the implication $\Longrightarrow$ of the meta logic. As mentioned above all rules that will be needed in the following derivations are listed in table 6.1.

6.2 A Derivation of Quicksort

The original quicksort algorithm was developed by Hoare in 1962. Quicksort is based on the *devide-and-conquer* paradigm, this means it partions the input into two halves, sorts each half, and then combins the halves to form the sorted list.

6.2.1 The Initial Goal

We start in Isabelle by typing the top level specification

```
- goal thy "Def(?H)-->(ALL l.(perm(l, ?sort(l))& ordered(?sort(l))))";
```

The system prints

name	*definition*
perm_append	`[| perm(l1, l2); perm(t1, t2)|] ==> perm(l1 @ t1, l2 @ t2)`
perm_refl	`perm(l, l)`
perm_trans	`[| perm(l1, l2); perm(l2, l3) |] ==> perm(l1, l3)`
perm_sym	`perm(l1, l2) ==> perm(l2, l1)`
perm_append_with_trans	`[| perm(t1, l1); perm(t2, l2); perm(l, t1 @ t2) |] ==> perm(l, l1 @ l2)`
ord_append_insert	`ordered(l1) & ordered(l2) & maxi(u, l1) & mini(u, l2) ==> ordered(l @ [u] @ l2)`
mini_Nil	`mini(u, [])`
mini_perm	`[| perm(l1, l2); mini(u, l1) |] ==> mini(u, l2)`
maxi_Nil	`maxi(u, [])`
maxi_perm	`[| perm(l1, l2); maxi(u, l1) |] ==> maxi(u, l2)`
perm_append_singleton1	`perm(l1 @ l2, t1 @ t2) ==> perm(l1 @ [u1] @ l2, ?t1 @ [u1] @ t2)`
list_decompose1	`[| t = []; P([] @ [hd(?t)] @ tl(t)) |] ==> P(t)`

Table 6.1: Several lemmata for the predicates *perm*, *ordered*, *mini* and *maxi*

```
Level 0
Def(?H) --> (! l. perm(l, ?sort(l)) & ordered(?sort(l)))
 1. Def(?H)  --> ! l. perm(l, ?sort(l)) & ordered(?sort(l))
```

As we have mentioned in section 5.3.3 the first step in the derivation is to move the accumulator in the assumptions context. So we type

```
- br impI 1;
```

And get the following proofstate

```
Level 1
Def(?H) --> (! l. perm(l, ?sort(l)) & ordered(?sort(l)))
 1. Def(?H) ==> ! l. perm(l, ?sort(l)) & ordered(?sort(l))
```

6.2.2 Induction

We prove the subgoal by induction. This is intuitively correct, because the desired program is a recursive defined function and as we will see later means of the induction we can built recursive terms. We type the following tactic in Isabelle

```
- BY (INDTAC indschemaf1_1p1  [("f","quicksort")] [("Spec","%l f. perm(l, f) & ordered(f)")] 1);
Level 2
Def((! l. quicksort(l) = ?fbody(l)) & ?Q) -->
(! l. perm(l, quicksort(l)) & ordered(quicksort(l)))
 1. !!l. [| Def(?Q);
           ! t. <t, l> : ?r -->
                perm(t, quicksort(t)) & ordered(quicksort(t)) |] ==>
        perm(l, ?fbody(l)) & ordered(?fbody(l))
 2. [| Def(! l. quicksort(l) = ?fbody(l)); Def(?Q) |] ==> wf(?r)
```

The application of the tactic `INDTAC` is explained in section 5.3.3.

6.2.3 Case Split

Our aim now is to make use of the induction hypothesis. At the moment this will be fruitless, because by applying the induction hypothesis we have to prove the constraint `<l,l>:?r` which is invalid for all well-founded relations. The idea is thus to decompose the list `l` into sublists such that the conclusion `perm(l, ?fbody(l))& ordered(?fbody(l))` can be reduced to some other formulas which are equivalent or stronger than it. These new formulas will be produced using lemmata about the predicates *perm* and *ordered*. They could be then provable using the induction hypothesis. Before we do this we remark that if the list `l` is empty we don't need to decompose it, because we can prove it without using the induction hypothesis. So we perform now a case analysis by calling the following tactic in Isabelle

```
- BY (res_inst_tac [("l","l"),("P","%f. perm(l,f)&ordered(f)")] case_splitf1_list 1);
```

The case split rule `case_splitf1_list` is a specialization of the rule `case_plit` (see figure 5.1) for lists. It has the form

```
[|  ?l = [] ==> ?P(?case_T);
   ~?l = [] ==> ?P(?case_F(hd(l), tl(l)))
|] ==> ?P(if(?l = [], ?case_T, ?case_F(hd(l), tl(?l))))
```

This rule introduces conditional terms in the definition of the function *quicksort*. By resolving[2] the first subgoal against the above rule we bind the metavariable `?fbody` to the conditional term[3] `if(l = [], ?case_T(l), ?case_F(l, hd(l), tl(l)))` Note also that all occurrences of `?fbody` in the whole proofstate will be replaced by this term. Further we obtain two subgoals. In the first one we have the additional premise `l=[]` and in the second one `~l=[]`.

```
Def((! l. quicksort(l) = if(l = [], ?case_T(l), ?case_F(l, hd(l), tl(l)))) &
    ?Q) -->
(! l. perm(l, quicksort(l)) & ordered(quicksort(l)))
 1. !!l. [| Def(?Q);
            ! t. <t, l> : ?r --> perm(t, quicksort(t)) & ordered(quicksort(t));
            l = [] |] ==>
         perm(l, ?case_T(l)) & ordered(?case_T(l))
 2. !!l. [| Def(?Q);
            ! t. <t, l> : ?r --> perm(t, quicksort(t)) & ordered(quicksort(t));
            l ~= [] |] ==>
         perm(l, ?case_F(l, hd(l), tl(l))) & ordered(?case_F(l, hd(l), tl(l)))
 3. [| Def(! l. quicksort(l) =
               if(l = [], ?case_T(l), ?case_F(l, hd(l), tl(l))));
       Def(?Q) |] ==>
    wf(?r)
```

6.2.3.1 The Base Case

We work on the first subgoal. First we substitute `l` in the conclusion by the empty list `[]` using the tactic `hyp_subst_tac`. Then we resolve the subgoal against

[2]After instantiation the predicate `P` by the predicate `%f. perm(l,f)& ordered(f)` and the list `l` by the concrete list `l` that occurs in the subgoal

[3]The additional argument `l` is introduced by lifting

the rule `perm([], [])& ordered([])` obtained by joining the rules `perm_Nil` and `ordered_Nil` using the theorem operator `MRS`[4] and the conjunction introduction `conjI`. Together we type

```
- BY (EVERY1 [hyp_subst_tac, rtac ([perm_Nil,ord_Nil] MRS conjI)]);
Level 4
Def((! l. quicksort(l) = if(l = [], l, ?case_F(l, hd(l), tl(l)))) & ?Q) -->
(! l. perm(l, quicksort(l)) & ordered(quicksort(l)))
 1. !!l. [| Def(?Q);
           ! t. <t, l> : ?r --> perm(t, quicksort(t)) & ordered(quicksort(t));
           l ~= [] |] ==>
        perm(l, ?case_F(l, hd(l), tl(l))) & ordered(?case_F(l, hd(l), tl(l)))
 2. [| Def(! l. quicksort(l) = if(l = [], l, ?case_F(l, hd(l), tl(l))));
       Def(?Q) |] ==>
    wf(?r)
```

Note that the true-part of the conditional term of the definition is instantiated with the term `l`, which in this case is the empty list.

6.2.3.2 The Inductive Case

Now we have to make a choice to partition the list `l` or its tail, `tl(l)`. We decide to partitione the tail of `l`. We now have some questions to answer

- Where shall we place `hd(l)` in the result of the sorting ?
- How and when shall we divide `tl(l)` ?

To the first question at the moment we only know that the result should contain `hd(l)`, may be at the beginning or may be at the end or middle .... In other words the result should have the following form $result_1@[hd(l)]@result_2$. To the second question we don't know how $result_1$ and $result_2$ depend on the partitions of `tl(l)`. Intuitively the partitions of `tl(l)` are two lists whose concatenation forms a permutation of it. So we first have to break down the conclusion of our subgoal to the form `perm(tl(l), ...)& ordered(...)`, we call this step simplification. Then we apply the lemma `perm_append_with_trans` which introduces two lists `?l1(l)` and `?l2(l)` that satisfy `perm(tl(l),?l1(l)@?l2(l))` and `perm(?l1(l)@?l2(l),...)`; we call this step partitioning.

Simplification First we split the subgoal into two subgoals using the introduction rule for the conjunction

```
- br conjI 1;
Level 5
Def((! l. quicksort(l) = if(l = [], l, ?case_F(l, hd(l), tl(l)))) & ?Q) -->
(! l. perm(l, quicksort(l)) & ordered(quicksort(l)))
 1. !!l. [| Def(?Q);
           ! t. <t, l> : ?r --> perm(t, quicksort(t)) & ordered(quicksort(t));
           l ~= [] |] ==>
        perm(l, ?case_F(l, hd(l), tl(l)))
 2. !!l. [| Def(?Q);
           ! t. <t, l> : ?r --> perm(t, quicksort(t)) & ordered(quicksort(t));
```

[4] This operator resolves a premise list of a rule against a list of theorems

```
          l ~= [] |] ==>
        ordered(?case_F(l, hd(l), tl(l)))
 3. [| Def(! l. quicksort(l) = if(l = [], l, ?case_F(l, hd(l), tl(l))));
       Def(?Q) |] ==>
    wf(?r)
```

To apply the lemma `perm_append_singleton1` to the first subgoal we shall rewrite the list `l` into the form `[]@[hd(l)]@tl(l)` using `list_decompose1`. This rule asserts that the list to be decomposed should not be empty. This holds in our situation because the list `l` is non-empty by assumption. We type the following tactic

```
- BY ((res_inst_tac [("P","%l .perm(l,?A)")] list_decompose1 1)THEN (atac 1));
Level 6
Def((! l. quicksort(l) = if(l = [], l, ?case_F(l, hd(l), tl(l)))) & ?Q) -->
(! l. perm(l, quicksort(l)) & ordered(quicksort(l)))
 1. !!l. [| Def(?Q);
          ! t. <t, l> : ?r --> perm(t, quicksort(t)) & ordered(quicksort(t));
          l ~= [] |] ==>
        perm([] @ [hd(l)] @ tl(l), ?case_F(l, hd(l), tl(l)))
 2. !!l. [| Def(?Q);
          ! t. <t, l> : ?r --> perm(t, quicksort(t)) & ordered(quicksort(t));
          l ~= [] |] ==>
        ordered(?case_F(l, hd(l), tl(l)))
 3. [| Def(! l. quicksort(l) = if(l = [], l, ?case_F(l, hd(l), tl(l))));
       Def(?Q) |] ==>
    wf(?r)
```

Now we can resolve the first subgoal against the lemma `perm_append_singleton1` and then simplify[5] it. We expect in this step that our above considerations will be realized, namely that the term `?case_F(l, hd(l), tl(l))` will instantiate to a term of the form $result_1@[hd(l)]@result_2$

```
- BY (EVERY1[rtac perm_append_singleton1, simp_tac appl_ss]);
Level 7
Def((! l. quicksort(l) =
          if(l = [], l,
             ?t1(l, hd(l), tl(l)) @ [hd(l)] @ ?t2(l, hd(l), tl(l)))) &
    ?Q) -->
(! l. perm(l, quicksort(l)) & ordered(quicksort(l)))
 1. !!l. [| Def(?Q);
          ! t. <t, l> : ?r --> perm(t, quicksort(t)) & ordered(quicksort(t));
          l ~= [] |] ==>
        perm(tl(l), ?t1(l, hd(l), tl(l)) @ ?t2(l, hd(l), tl(l)))
 2. !!l. [| Def(?Q);
          ! t. <t, l> : ?r --> perm(t, quicksort(t)) & ordered(quicksort(t));
          l ~= [] |] ==>
        ordered(?t1(l, hd(l), tl(l)) @ [hd(l)] @ ?t2(l, hd(l), tl(l)))
 3. [| Def(! l. quicksort(l) =
                if(l = [], l,
                   ?t1(l, hd(l), tl(l)) @ [hd(l)] @ ?t2(l, hd(l), tl(l))));
       Def(?Q) |] ==>
    wf(?r)
```

In fact the metavariable `?case_F(l, hd(l),tl(l))` is bound to `?t1(l,hd(l),tl(l))@ [hd(l)] @ ?t2(l,hd(l),tl(l))` and the conclusion of the first

[5] Isabelle's simplifier will always be called with a set of rewrite rules. The set that we will often used is called `appl_ss`. It contains some axioms of list theory and some derived rules.

subgoal hat the desired form `perm(tl(l),...)`. Intuitively the lists `?t1(l,hd(l),tl(l))` and `?t2(l,hd(l),tl(l))` are the results of calling *quicksort* with the partitions of `tl(l)`. So we expect in the rest of the derivation that `?t1(l,hd(l),tl(l))` and `?t2(l,hd(l),tl(l))` will be stepwise instantiated and becomes respectively `quicksort(?l1(hd(l),tl(l)))` and `quicksort(?l2(hd(l),tl(l)))` where `?l1(hd(l),tl(l))` and `?l2(hd(l),tl(l))` denotes the two halves of `tl(l)`

Partitioning We first prove informally why we can't delay the partitioning of `tl(l)`. Consider the first subgoal. At this point we can't apply the induction hypothesis to solve this subgoal. To achieved this we first deduce a new premise

```
<t(l),l> ?r ==> perm(?t(l), quicksort(?t(l)))
```

Now to resolve the first subgoal against the above premise, the formulas

```
perm(?t(l), quicksort(?t(l)))
```

and

```
perm(tl(l), ?t1(l, Cons(hd(l), tl(l))) @ ?t2(l, Cons(hd(l),  tl(l))))
```

should be unifiable, but this is not satisfied. Even if we exchange[6] arguments of *perm* in both formulas the unification remains impossible. So to apply the induction hypothesis the arguments of the predicate *perm* occuring in the conclusion should be metavariables. Thus the solution consists in dividing the the list `tl(l)` in two sublists `?l1(tl(l))` and `?l2(tl(l))` and then applying lemma `perm_append`. This can be done using the lemma `perm_append_with_trans`.

```
- BY (rtac perm_append_with_trans 1);
Level 8
Def((! l. quicksort(l) =
         if(l = [], l,
            ?t1(l, hd(l), tl(l)) @ [hd(l)] @ ?t2(l, hd(l), tl(l)))) &
    ?Q) -->
(! l. perm(l, quicksort(l)) & ordered(quicksort(l)))
 1. !!l. [| Def(?Q);
           ! t. <t, l> : ?r --> perm(t, quicksort(t)) & ordered(quicksort(t));
           l ~= [] |] ==>
        perm(tl(l), ?l1(l) @ ?l2(l)) &
        perm(?l1(l), ?t1(l, hd(l), tl(l))) &
        perm(?l2(l), ?t2(l, hd(l), tl(l)))
 2. !!l. [| Def(?Q);
           ! t. <t, l> : ?r --> perm(t, quicksort(t)) & ordered(quicksort(t));
           l ~= [] |] ==>
        ordered(?t1(l, hd(l), tl(l)) @ [hd(l)] @ ?t2(l, hd(l), tl(l)))
 3. [| Def(! l. quicksort(l) =
               if(l = [], l,
                  ?t1(l, hd(l), tl(l)) @ [hd(l)] @ ?t2(l, hd(l), tl(l))));
       Def(?Q) |] ==>
    wf(?r)
```

[6] The predicate *perm* is commutative

At this point we have divided `tl(l)` but we have not said how; the partitions `?l1(l)` and `?l2(l)` are still uninstantiated metavariables . This will be the subject of the rest of the derivation. To synthesize the partitions `?l1(tl(l))` and `?l2(tl(l))` of `tl(l)` we need more informations than those displayed in the first subgoal. Actually we have no idea about the relationship between the lists `tl(l)` and `?t1(l, hd(l), tl(l))` and `?t1(l, hd(l), tl(l))`. We therefore return to the second subgoal and exploit the ordering property by applying the lemma `ord_append_insert`.

```
- BY (rtac ord_append_insert 2);
Level 9
Def((! l. quicksort(l) =
          if(l = [], l,
             ?t1(l, hd(l), tl(l)) @ [hd(l)] @ ?t2(l, hd(l), tl(l)))) &
    ?Q) -->
(! l. perm(l, quicksort(l)) & ordered(quicksort(l)))
 1. !!l. [| Def(?Q);
            ! t. <t, l> : ?r --> perm(t, quicksort(t)) & ordered(quicksort(t));
            l ~= [] |] ==>
         perm(tl(l), ?l1(l) @ ?l2(l)) &
         perm(?l1(l), ?t1(l, hd(l), tl(l))) &
         perm(?l2(l), ?t2(l, hd(l), tl(l)))
 2. !!l. [| Def(?Q);
            ! t. <t, l> : ?r --> perm(t, quicksort(t)) & ordered(quicksort(t));
            l ~= [] |] ==>
         (ordered(?t1(l, hd(l), tl(l))) & ordered(?t2(l, hd(l), tl(l)))) &
         maxi(hd(l), ?t1(l, hd(l), tl(l))) &
         mini(hd(l), ?t2(l, hd(l), tl(l)))
 3. [| Def(! l. quicksort(l) =
                if(l = [], l,
                   ?t1(l, hd(l), tl(l)) @ [hd(l)] @ ?t2(l, hd(l), tl(l))));
       Def(?Q) |] ==>
    wf(?r)
```

In the second subgoal we now obtain more informations. Scrutinizing the conclusions of the first and second subgoals we can neatly compose parts together to form instances of the originally specification (see next proofstate). In the next step we also collect[7] all related informations together in one subgoal.

```
- BY (PRIMITIVE (fn state => (state COMP rule4)));
Level 10
Def((! l. quicksort(l) =
          if(l = [], l,
             ?t1(l, hd(l), tl(l)) @ [hd(l)] @ ?t2(l, hd(l), tl(l)))) &
    ?Q) -->
```

[7] Often it is useful to remove similar subgoals or to exchange parts of conclusions of subgoals that have the same premise list. Isabelle supports a tactic, called **PRIMITIVE**, that applies function which range over theorems to the actual proofstate. It supports also a function called **COMP** that requires two theorems th_1 and th_2 and composed them by resolving the first premise of th_2 against th_1. So to collect the desired informations we derive a special rule whose first premise matches the actual proofstate and in the other premises we formulate the desired subgoals. Then we call the tactic **PRIMITIVE** with the function that composes the our derived rule and the actual proofstate.For more details about **PRIMITIVE** and **COMP** see [45] Isabelle's users may not prefer such tactic, but we used it here only to make our derivation more readable

```
(! l. perm(l, quicksort(l)) & ordered(quicksort(l)))
 1. !!l. [| Def(?Q);
           ! t. <t, l> : ?r --> perm(t, quicksort(t)) & ordered(quicksort(t));
           l ~= [] |] ==>
        perm(tl(l), ?l1(l) @ ?l2(l)) &
        mini(hd(l), ?t2(l, hd(l), tl(l))) &
        maxi(hd(l), ?t1(l, hd(l), tl(l)))
 2. !!l. [| Def(?Q);
           ! t. <t, l> : ?r --> perm(t, quicksort(t)) & ordered(quicksort(t));
           l ~= [] |] ==>
        perm(?l1(l), ?t1(l, hd(l), tl(l))) &
        ordered(?t1(l, hd(l), tl(l)))
 3. !!l. [| Def(?Q);
           ! t. <t, l> : ?r --> perm(t, quicksort(t)) & ordered(quicksort(t));
           l ~= [] |] ==>
        perm(?l2(l), ?t2(l, hd(l), tl(l))) &
        ordered(?t2(l, hd(l), tl(l)))
 4. [| Def(! l. quicksort(l) =
               if(l = [], l,
                  ?t1(l, hd(l), tl(l)) @ [hd(l)] @ ?t2(l, hd(l), tl(l))));
      Def(?Q) |] ==>
    wf(?r)
```

As before in the conclusion of the first subgoal we miss a relationship between the metavariables `?l1(l)`, `?l2(l)`, `?t1(l, hd(l), tl(l))` and `?t2(l, hd(l), tl(l))`. A relationship is implicitly present, namely it can be seen in the second and third subgoals(there `?t1(...)` stands for a sorted list of `?l1(...)` and `?t2(...)` stands for a sorted list of `?l2(...)`). So to establish this relationship explicitly in the first subgoal we prove the second and the third subgoals using the induction hypothesis. To achieve this step we apply the tactic called `apply_induction_hyp` twice. This tactic applied to a subgoal `i` with the conclusion G attempts to simplify and also to instantiate some premises from the assumption list of the subgoal in order to get a new premise of the form *condition* $\rightarrow G$. Then it applies the modus ponens rule. The result is a new subgoal whose conclusion is the condition *condition*.

```
- BY ((apply_induction_hyp 2) THEN (apply_induction_hyp 3));
Level 11
Def((! l. quicksort(l) =
          if(l = [], l,
             quicksort(?t(l, hd(l), tl(l))) @
             [hd(l)] @ quicksort(?ta(l, hd(l), tl(l))))) &
    ?Q) -->
(! l. perm(l, quicksort(l)) & ordered(quicksort(l)))
 1. !!l. [| Def(?Q);
           ! t. <t, l> : ?r --> perm(t, quicksort(t)) & ordered(quicksort(t));
           l ~= [] |] ==>
        perm(tl(l), ?t(l, hd(l), tl(l)) @ ?ta(l, hd(l), tl(l))) &
        mini(hd(l), quicksort(?ta(l, hd(l), tl(l)))) &
        maxi(hd(l), quicksort(?t(l, hd(l), tl(l))))
 2. !!l. [| Def(?Q); l ~= [] |] ==> <?t(l, hd(l), tl(l)), l> : ?r
 3. !!l. [| Def(?Q); l ~= [] |] ==> <?ta(l, hd(l), tl(l)), l> : ?r
 4. [| Def(! l. quicksort(l) =
               if(l = [], l,
                  quicksort(?t(l, hd(l), tl(l))) @
```

```
                [hd(l)] @ quicksort(?ta(l, hd(l), tl(l)))));
      Def(?Q) |] ==>
   wf(?r)
```

6.2.4 Auxiliary Programs

The first subgoal consists of a specification of the subprograms `?t` and `?ta`, which we later will call *lesseq* and *greater*. At the moment the specification contains the occurrence of the original desired program, namely *quicksort*. But this program is not yet completely defined. So if we take this specification as basis for the synthesis of the programs *lesseq* and *greater* we may have difficulties to evaluate terms of the form `quicksort(...)` in the rest of the proof. On the other hand the predicates *mini* and *maxi* only care about the elements of the list and not about their order. Therefore we can replace the lists `quicksort(?t(...))` and `quicksort(?ta(...))` by two lists which are permutations of them. To do this we make use of the lemmata `mini_perm` and `maxi_perm`. To apply these lemmata we have first to perform conjunction introduction `conjI` twice. This will break the structure of our specification, which impores its readability. Therefore we have derived a special new rule called `mini_maxi_lemma`

```
[|  P & mini(u,l2) & maxi(u,l1);
    perm(l1, t1) & perm(l2, t2)
|]==> P & mini(u,t2) & maxi(u,t1)
```

automatically proved using the simplifier with the set of rewrite rules `appl_ss` augmented with the lemmata `mini_perm` and `maxi_perm`. We remark that this rule is artificial in the sense that we have imitated the structure of the conclusion of the subgoal that we have to prove. The first premise of this rule is the desired specification and the second premise regroupes other conditions that constitute no part of the specification. One disadvantage of such rules is their restricted application as opposed to other lemmata such as `maxi_perm` and `maxi_perm`. Our motivation to generate such special rules and to apply them is to keep the derivation more readable and with fewer subgoals. In the system we call the resolution tactic with the above lemma

```
- BY (rtac mini_maxi_lemma 1);
Level 12
Def((! l. quicksort(l) =
          if(l = [], l,
             quicksort(?t(l, hd(l), tl(l))) @
             [hd(l)] @ quicksort(?ta(l, hd(l), tl(l))))) &
    ?Q) -->
(! l. perm(l, quicksort(l)) & ordered(quicksort(l)))
 1. !!l. [| Def(?Q);
            ! t. <t, l> : ?r --> perm(t, quicksort(t)) & ordered(quicksort(t));
            l ~= [] |] ==>
         perm(tl(l), ?t(l, hd(l), tl(l)) @ ?ta(l, hd(l), tl(l))) &
         mini(hd(l), ?l2(l)) & maxi(hd(l), ?l1(l))
 2. !!l. [| Def(?Q);
            ! t. <t, l> : ?r --> perm(t, quicksort(t)) & ordered(quicksort(t));
            l ~= [] |] ==>
         perm(?l1(l), quicksort(?t(l, hd(l), tl(l)))) &
         perm(?l2(l), quicksort(?ta(l, hd(l), tl(l))))
```

```
 3. !!l. [| Def(?Q); l ~= [] |] ==> <?t(l, hd(l), tl(l)), l> : ?r
 4. !!l. [| Def(?Q); l ~= [] |] ==> <?ta(l, hd(l), tl(l)), l> : ?r
 5. [| Def(! l. quicksort(l) =
                if(l = [], l,
                   quicksort(?t(l, hd(l), tl(l))) @
                   [hd(l)] @ quicksort(?ta(l, hd(l), tl(l)))));
       Def(?Q) |] ==>
    wf(?r)
```

We now decompose the second subgoal using the conjunction introduction `conjI`. It yields two additional subgoals that can be proved using the tactic `appl_induction_hyp`. The new constraints are similar to the actual third and fourth subgoal. So we can remove it using the tactic `make_compact` which invokes essentially the tactic `PRIMITIVE`.

```
- BY (EVERY [rtac conjI 2, apply_induction_hyp 2, apply_induction_hyp 3, make_compact]);
Level 13
Def((! l. quicksort(l) =
          if(l = [], l,
             quicksort(?t(l, hd(l), tl(l))) @
             [hd(l)] @ quicksort(?ta(l, hd(l), tl(l))))) &
    ?Q) -->
(! l. perm(l, quicksort(l)) & ordered(quicksort(l)))
 1. !!l. [| Def(?Q);
            ! t. <t, l> : ?r --> perm(t, quicksort(t)) & ordered(quicksort(t));
            l ~= [] |] ==>
         perm(tl(l), ?t(l, hd(l), tl(l)) @ ?ta(l, hd(l), tl(l))) &
         mini(hd(l), ?ta(l, hd(l), tl(l))) & maxi(hd(l), ?t(l, hd(l), tl(l)))
 2. !!l. [| Def(?Q); l ~= [] |] ==> <?t(l, hd(l), tl(l)), l> : ?r
 3. !!l. [| Def(?Q); l ~= [] |] ==> <?ta(l, hd(l), tl(l)), l> : ?r
 4. [| Def(! l. quicksort(l) =
                if(l = [], l,
                   quicksort(?t(l, hd(l), tl(l))) @
                   [hd(l)] @ quicksort(?ta(l, hd(l), tl(l)))));
       Def(?Q) |] ==>
    wf(?r)
```

Informally the specification states that we partition the elements of the list `tl(l)` into lists `?t(...)` and `?ta(...)` so that `?t(...)` contains only those elements less than or equal to `hd(l)` and `?ta(...)` contains the rest[8] of `tl(l)`.We will call these partitions *lesseq* and *greater*. To synthesize these functions we will perform the same procedure, (induction, case split, base case and inductive case) as we have done for the function *quicksort*. But before we begin the derivation we remark that the specification is parameterized by special parameters, namely `hd(l)` and `tl(l)`. We would like to synthesize these programs in a general form. So in the definition of the function *quicksort* these functions could be called with the special parameters `hd(l)` and `tl(l)`, but in their definitions they will be defined for general parameters. We first have to generalize the specification. This step will be done by induction. A convenient induction schema for this specification is `indschemaf2_2p1`

[8]The elements of `?ta(...)` are all greater than `hd(l)`, because we have only considered lists with total orderings

```
[| Def((!u l. ?f(u, l) = ?fbody(u, ~l)) & (!u l. ?g(u, l) = ?gbody(u, l)))
   !l. (! u t. <t, l> : r --> ?Spec(t, u, ?f(u, t), ?g(u, t)))
       --> !u. ?Spec(l, u, ?fbody(u, l), ?gbody(u, l)
   wf(r?)
|]==> ?Spec(?l, ?u, ?f(?u, ?l), ?g(?u, ?l))
```

We eliminate the quantifier from the conclusion of this schema. Thus we twice resolve this schema against the rule `spec` using the theorem operator `RS`[9]. We get

```
[| Def((!u l. ?f(u, l) = ?fbody(u, ~l)) & (!u l. ?g(u, l) = ?gbody(u, l)))
   !l. (! u t. <t, l> : r --> ?Spec(t, u, ?f(u, t), ?g(u, t)))
       --> !u. ?Spec(l, u, ?fbody(u, l), ?gbody(u, l)
   wf(r?)
|]==> ?Spec(?l, ?u, ?f(?u, ?l), ?g(?u, ?l))
```

Now if we resolve the first subgoal against the above rule the variables `?l` and `?u` become respectively `tl(l)` and `hd(l)` but this instantiation will not be propagated to the premises of the schema, because these variables occur bound in the premises (so in this way we have generalized the specification without using the `cut` rule). Analogouly to the derivation of *quicksort* we call the induction tactic `INDTAC` argumented with the above rule.

```
- BY ((INDTAC (indschemaf2_2p1 RS spec RS spec)
              [("f2","greater"),("g2","lesseq")]
              [("Spec2","%l u g f.perm(l,f@g)&mini(u,g)&maxi(u,f)")] 1)
     THEN (rename_tac "u e" 1));
Level 14
Def((! l. quicksort(l) =
          if(l = [], l,
             quicksort(lesseq(hd(l), tl(l))) @
             [hd(l)] @ quicksort(greater(hd(l), tl(l))))) &
    ((! x xa. greater(x, xa) = ?fbody(x, xa)) &
     (! x xa. lesseq(x, xa) = ?gbody(x, xa))) &
    ?Q) -->
(! l. perm(l, quicksort(l)) & ordered(quicksort(l)))
 1. !!l u e.
       [| ! t. <t, l> : ?r --> perm(t, quicksort(t)) & ordered(quicksort(t));
          l ~= []; Def(?Q);
          ! x t. <t, u> : ?ra(l) -->
                 perm(t, lesseq(x, t) @ greater(x, t)) &
                 mini(x, greater(x, t)) & maxi(x, lesseq(x, t)) |] ==>
       perm(u, ?gbody(e, u) @ ?fbody(e, u)) &
       mini(e, ?fbody(e, u)) & maxi(e, ?gbody(e, u))
 2. !!l. [| ! t. <t, l> : ?r --> perm(t, quicksort(t)) & ordered(quicksort(t));
            l ~= [];
            Def((! x xa. greater(x, xa) = ?fbody(x, xa)) &
                (! x xa. lesseq(x, xa) = ?gbody(x, xa)));
            Def(?Q) |] ==>
         wf(?ra(l))
 3. !!l. [| l ~= [];
            Def((! x xa. greater(x, xa) = ?fbody(x, xa)) &
                (! x xa. lesseq(x, xa) = ?gbody(x, xa)));
            Def(?Q) |] ==>
         <lesseq(hd(l), tl(l)), l> : ?r
```

[9] It resolve the conclusion of the first rule against the first premise of the second rule

```
 4. !!l. [| l ~= [];
           Def((! x xa. greater(x, xa) = ?fbody(x, xa)) &
               (! x xa. lesseq(x, xa) = ?gbody(x, xa)));
           Def(?Q) |] ==>
        <greater(hd(l), tl(l)), l> : ?r
 5. [| Def(! l. quicksort(l) =
               if(l = [], l,
                  quicksort(lesseq(hd(l), tl(l))) @
                  [hd(l)] @ quicksort(greater(hd(l), tl(l)))));
       Def((! x xa. greater(x, xa) = ?fbody(x, xa)) &
           (! x xa. lesseq(x, xa) = ?gbody(x, xa)));
       Def(?Q) |] ==>
    wf(?r)
```

Before we continue our derivation we remark that some premises in some subgoals will never be used. For example

- The induction hypothesis for the function *quicksort* can be removed from the premise list of the first subgoal and the second subgoal.
- The accumulator `Def(?Q)` can be removed from all subgoals because we will not synthesize further subprograms
- The definitions of the functions *greater* in the third subgoal and *lesseq* in the fourth subgoal are also superfluous.
- The well-founded relations `?r` and `?ra(l)` don't depend on the definitions of new programs *quicksort*, *lesseq* and *greater*.

We clean up these subgoals by a special tactic called `clean_up` written for this purpose.

```
- BY clean_up ;
Level 15
Def((! l. quicksort(l) =
          if(l = [], l,
             quicksort(lesseq(hd(l), tl(l))) @
             [hd(l)] @ quicksort(greater(hd(l), tl(l))))) &
    ((! x xa. greater(x, xa) = ?fbody(x, xa)) &
     (! x xa. lesseq(x, xa) = ?gbody(x, xa))) &
    ?Q) -->
(! l. perm(l, quicksort(l)) & ordered(quicksort(l)))
 1. !!l u e.
       [| l ~= [];
          ! x t. <t, u> : ?ra(l) -->
                 perm(t, lesseq(x, t) @ greater(x, t)) &
                 mini(x, greater(x, t)) & maxi(x, lesseq(x, t)) |] ==>
       perm(u, ?gbody(e, u) @ ?fbody(e, u)) &
       mini(e, ?fbody(e, u)) & maxi(e, ?gbody(e, u))
 2. !!l. l ~= [] ==> wf(?ra(l))
 3. !!l. [| l ~= []; Def(! x xa. lesseq(x, xa) = ?gbody(x, xa)) |] ==>
        <lesseq(hd(l), tl(l)), l> : ?r
 4. !!l. [| l ~= []; Def(! x xa. greater(x, xa) = ?fbody(x, xa)) |] ==>
        <greater(hd(l), tl(l)), l> : ?r
 5. wf(?r)
```

In the following we will discuss only the main steps of the rest of the proof. For the same reasons as for *quicksort* we continue the proof by performing a structural analysis over the induction variable u. At the same time we introduce conditional terms to the bodies of the functions *lesseq* and *greater*

```
- BY (res_inst_tac [("l","u"),("P","%g f.perm(u,f@g)&mini(e,g)&maxi(e,f)")] case_splitf2_list 1);
Level 16
Def((! l. quicksort(l) =
          if(l = [], l,
             quicksort(lesseq(hd(l), tl(l))) @
             [hd(l)] @ quicksort(greater(hd(l), tl(l))))) &
    ((! x xa.
         greater(x, xa) =
         if(xa = [], ?case1_T(xa, x), ?case1_F(xa, x, hd(xa), tl(xa)))) &
     (! x xa.
         lesseq(x, xa) =
         if(xa = [], ?case2_T(xa, x), ?case2_F(xa, x, hd(xa), tl(xa))))) &
    ?Q) -->
(! l. perm(l, quicksort(l)) & ordered(quicksort(l)))
 1. !!l u e.
       [| l ~= [];
          ! x t. <t, u> : ?ra(l) -->
                 perm(t, lesseq(x, t) @ greater(x, t)) &
                 mini(x, greater(x, t)) & maxi(x, lesseq(x, t));
          u = [] |] ==>
       perm(u, ?case2_T(u, e) @ ?case1_T(u, e)) &
       mini(e, ?case1_T(u, e)) & maxi(e, ?case2_T(u, e))
 2. !!l u e.
       [| l ~= [];
          ! x t. <t, u> : ?ra(l) -->
                 perm(t, lesseq(x, t) @ greater(x, t)) &
                 mini(x, greater(x, t)) & maxi(x, lesseq(x, t));
          u ~= [] |] ==>
       perm(u, ?case2_F(u, e, hd(u), tl(u)) @ ?case1_F(u, e, hd(u), tl(u))) &
       mini(e, ?case1_F(u, e, hd(u), tl(u))) &
       maxi(e, ?case2_F(u, e, hd(u), tl(u)))
 3. !!l. l ~= [] ==> wf(?ra(l))
 4. !!l. [| l ~= [];
            Def(! x xa.
                   lesseq(x, xa) =
                   if(xa = [], ?case2_T(xa, x),
                      ?case2_F(xa, x, hd(xa), tl(xa)))) |] ==>
         <lesseq(hd(l), tl(l)), l> : ?r
 5. !!l. [| l ~= [];
            Def(! x xa.
                   greater(x, xa) =
                   if(xa = [], ?case1_T(xa, x),
                      ?case1_F(xa, x, hd(xa), tl(xa)))) |] ==>
         <greater(hd(l), tl(l)), l> : ?r
 6. wf(?r)
```

In the base case (u=[]) we make use of some lemmata such as mini_Nil, maxi_Nil and perm_Nil. Note that perm_Nil is included in the set of rewrite rules appl_ss.

```
- BY (EVERY1 [hyp_subst_tac ,
              rtac ([mini_Nil, maxi_Nil] MRS conjI RSN (2,conjI)),
      simp_tac appl_ss]);
```

```
Level 17
Def((! l. quicksort(l) =
          if(l = [], l,
             quicksort(lesseq(hd(l), tl(l))) @
             [hd(l)] @ quicksort(greater(hd(l), tl(l))))) &
    ((! x xa.
         greater(x, xa) = if(xa = [], xa, ?case1_F(xa, x, hd(xa), tl(xa)))) &
     (! x xa.
         lesseq(x, xa) = if(xa = [], xa, ?case2_F(xa, x, hd(xa), tl(xa))))) &
    ?Q) -->
(! l. perm(l, quicksort(l)) & ordered(quicksort(l)))
 1. !!l u e.
       [| l ~= [];
          ! x t. <t, u> : ?ra(l) -->
                 perm(t, lesseq(x, t) @ greater(x, t)) &
                 mini(x, greater(x, t)) & maxi(x, lesseq(x, t));
          u ~= [] |] ==>
       perm(u, ?case2_F(u, e, hd(u), tl(u)) @ ?case1_F(u, e, hd(u), tl(u))) &
       mini(e, ?case1_F(u, e, hd(u), tl(u))) &
       maxi(e, ?case2_F(u, e, hd(u), tl(u)))
 2. !!l. l ~= [] ==> wf(?ra(l))
 3. !!l. [| l ~= [];
            Def(! x xa.
                   lesseq(x, xa) =
                   if(xa = [], xa, ?case2_F(xa, x, hd(xa), tl(xa)))) |] ==>
         <lesseq(hd(l), tl(l)), l> : ?r
 4. !!l. [| l ~= [];
            Def(! x xa.
                   greater(x, xa) =
                   if(xa = [], xa, ?case1_F(xa, x, hd(xa), tl(xa)))) |] ==>
         <greater(hd(l), tl(l)), l> : ?r
 5. wf(?r)
```

Similarily to the inductive case of the function *quicksort*, in order to apply the induction hypothesis, we have to decompose the list `u` in several sublists. We also have to decide either to decompose `u` or its `tail`. We decide here again to divide the tail of `u`. Our first task is to place `hd(u)` in one of the lists `?case1_F(u, e, hd(u), tl(u))` and `?case2_F(u, e, hd(u), tl(u))`. So because `u` is a permutation of these lists `hd(u)` should occurs in one of them, looking at the second and third part of the conjunction we read of that `hd(u)` occurs in `?case1_F(u, e, hd(u), tl(u))` (respectively in `?case_F(u, e, hd(u), tl(u))`) only if `e < hd(u)` (respectively only if `hd(u) <= e`). We thus distinguish first whether `hd(u) <= e`.

```
- BY (res_inst_tac[("A","hd(u)<=e"),("P","%w v. perm(u,w@v)& mini(e,v)& maxi(e,w)")]case_splitf2 1);
Level 18
Def((! l. quicksort(l) =
          if(l = [], l,
             quicksort(lesseq(hd(l), tl(l))) @
             [hd(l)] @ quicksort(greater(hd(l), tl(l))))) &
    ((! x xa.
         greater(x, xa) =
         if(xa = [], xa,
            if(hd(xa) <= x, ?s2(xa, x, hd(xa), tl(xa)),
               ?t2(xa, x, hd(xa), tl(xa))))) &
     (! x xa.
```

```
         lesseq(x, xa) =
         if(xa = [], xa,
            if(hd(xa) <= x, ?s1(xa, x, hd(xa), tl(xa)),
               ?t1(xa, x, hd(xa), tl(xa))))) &
    ?Q) -->
(! l. perm(l, quicksort(l)) & ordered(quicksort(l)))
 1. !!l u e.
       [| l ~= [];
          ! x t. <t, u> : ?ra(l) -->
                 perm(t, lesseq(x, t) @ greater(x, t)) &
                 mini(x, greater(x, t)) & maxi(x, lesseq(x, t));
          u ~= []; hd(u) <= e |] ==>
       perm(u, ?s1(u, e, hd(u), tl(u)) @ ?s2(u, e, hd(u), tl(u))) &
       mini(e, ?s2(u, e, hd(u), tl(u))) & maxi(e, ?s1(u, e, hd(u), tl(u)))
 2. !!l u e.
       [| l ~= [];
          ! x t. <t, u> : ?ra(l) -->
                 perm(t, lesseq(x, t) @ greater(x, t)) &
                 mini(x, greater(x, t)) & maxi(x, lesseq(x, t));
          u ~= []; ~ hd(u) <= e |] ==>
       perm(u, ?t1(u, e, hd(u), tl(u)) @ ?t2(u, e, hd(u), tl(u))) &
       mini(e, ?t2(u, e, hd(u), tl(u))) & maxi(e, ?t1(u, e, hd(u), tl(u)))
 3. !!l. l ~= [] ==> wf(?ra(l))
 4. !!l. [| l ~= [];
            Def(! x xa.
                   lesseq(x, xa) =
                   if(xa = [], xa,
                      if(hd(xa) <= x, ?s1(xa, x, hd(xa), tl(xa)),
                         ?t1(xa, x, hd(xa), tl(xa))))) |] ==>
         <lesseq(hd(l), tl(l)), l> : ?r
 5. !!l. [| l ~= [];
            Def(! x xa.
                   greater(x, xa) =
                   if(xa = [], xa,
                      if(hd(xa) <= x, ?s2(xa, x, hd(xa), tl(xa)),
                         ?t2(xa, x, hd(xa), tl(xa))))) |] ==>
         <greater(hd(l), tl(l)), l> : ?r
 6. wf(?r)
```

During the application of case split we have introduced a further conditional term, namely by instantiating the variables

`?case1_F` with `%u e hdu tlu.  if(e <= hd(u),?s1(u,e,hdu,tlu),?t1(u,e,hdu,tlu))`

and

`?case2_F` with `%u e hdu tlu.  if(e <= hd(u),?s2(u,e,hdu,tlu),?t2(u,e,hdu,tlu))`

The first subgoal represents the first case `hd(u) <= e`. In this case we will introduce the element `hd(u)` in the list `?s1(u, e, hd(u), tl(u))`. The specification doesn't prescribe how we have to introduce `e` in this list, so we decided to put it always at the beginning. We regroup the steps for this case in the following tactic

```
- BY less_or_eq_inductive_case;
Level 19
Def((! l. quicksort(l) =
          if(l = [], l,
             quicksort(lesseq(hd(l), tl(l))) @
             [hd(l)] @ quicksort(greater(hd(l), tl(l))))) &
```

```
    ((! x xa.
         greater(x, xa) =
         if(xa = [], xa,
            if(hd(xa) <= x, greater(x, tl(xa)),
               ?t2(xa, x, hd(xa), tl(xa))))) &
     (! x xa.
         lesseq(x, xa) =
         if(xa = [], xa,
            if(hd(xa) <= x, [hd(xa)] @ lesseq(x, tl(xa)),
               ?t1(xa, x, hd(xa), tl(xa)))))) &
    ?Q) -->
(! l. perm(l, quicksort(l)) & ordered(quicksort(l)))
 1. !!l u e.
       [| l ~= [];
          ! x t. <t, u> : ?ra(l) -->
                 perm(t, lesseq(x, t) @ greater(x, t)) &
                 mini(x, greater(x, t)) & maxi(x, lesseq(x, t));
          u ~= []; ~ hd(u) <= e |] ==>
       perm(u, ?t1(u, e, hd(u), tl(u)) @ ?t2(u, e, hd(u), tl(u))) &
       mini(e, ?t2(u, e, hd(u), tl(u))) & maxi(e, ?t1(u, e, hd(u), tl(u)))
 2. !!l u e. [| l ~= []; u ~= []; hd(u) <= e |] ==> <tl(u), u> : ?ra(l)
 3. !!l. l ~= [] ==> wf(?ra(l))
 4. !!l. [| l ~= [];
            Def(! x xa.
                   lesseq(x, xa) =
                   if(xa = [], xa,
                      if(hd(xa) <= x, [hd(xa)] @ lesseq(x, tl(xa)),
                         ?t1(xa, x, hd(xa), tl(xa))))) |] ==>
         <lesseq(hd(l), tl(l)), l> : ?r
 5. !!l. [| l ~= [];
            Def(! x xa.
                   greater(x, xa) =
                   if(xa = [], xa,
                      if(hd(xa) <= x, greater(x, tl(xa)),
                         ?t2(xa, x, hd(xa), tl(xa))))) |] ==>
         <greater(hd(l), tl(l)), l> : ?r
 6. wf(?r)
```

The first subgoal represents the case `e < hd(u)`. In this case we introduce the element `hd(u)` in the front of the list `?t2(u, e, hd(u), tl(u))`.

```
- BY greater_inductive_case;
Level 20
Def((! l. quicksort(l) =
          if(l = [], l,
             quicksort(lesseq(hd(l), tl(l))) @
             [hd(l)] @ quicksort(greater(hd(l), tl(l))))) &
    ((! x xa.
         greater(x, xa) =
         if(xa = [], xa,
            if(hd(xa) <= x, greater(x, tl(xa)),
               [hd(xa)] @ greater(x, tl(xa))))) &
     (! x xa.
         lesseq(x, xa) =
         if(xa = [], xa,
            if(hd(xa) <= x, [hd(xa)] @ lesseq(x, tl(xa)),
               lesseq(x, tl(xa)))))) &
    ?Q) -->
```

```
(! l. perm(l, quicksort(l)) & ordered(quicksort(l)))
 1. !!l u e. [| l ~= []; u ~= [] |] ==> <tl(u), u> : ?ra(l)
 2. !!l. l ~= [] ==> wf(?ra(l))
 3. !!l. [| l ~= [];
           Def(! x xa.
                  lesseq(x, xa) =
                  if(xa = [], xa,
                     if(hd(xa) <= x, [hd(xa)] @ lesseq(x, tl(xa)),
                        lesseq(x, tl(xa))))) |] ==>
        <lesseq(hd(l), tl(l)), l> : ?r
 4. !!l. [| l ~= [];
           Def(! x xa.
                  greater(x, xa) =
                  if(xa = [], xa,
                     if(hd(xa) <= x, greater(x, tl(xa)),
                        [hd(xa)] @ greater(x, tl(xa))))) |] ==>
        <greater(hd(l), tl(l)), l> : ?r
 5. wf(?r)
```

6.2.5 Termination

Looking at the proofstate we observe that the functions *quicksort*, *lesseq* and *greater* are completely defined. The five subgoals displayed in the proofstate express their termination. We begin by establishing termination of the function *quicksort*. Our task here is to instantiate the relation `?r` with a well-founded relation that satisfies the constraints

`<lesseq(hd(l),tl(l)), l>:?r` and `<greater(hd(l),tl(l)), l>:?r`

We first informally analyze the functions *lesseq* and *greater* in the above constraints. By looking at their definitions we conclude that for a non-empty list `x` and an arbitrary element `u` the lists `lesseq(u,tl(x))` and `greater(u,tl(x))` are proper sublists of `x`. This gives us a hint to instantiate `?r` with a relation that admits at least the `proper-sublist` relation as a subrelation. On the other hand the well-foundedness proof of the instance should be "simple" and it should demand no extra work. The following relation, called `shorter`, seems a good candidate

```
induced_rel(length, <) =  {<x, y> | length(x) < length(y)}
```

As we have seen in chapter 4 `shorter` is well-founded by construction, as an induced relation and because the order `<`[10] over the natural numbers is well-founded. Further the proper sublist relation is a subrelation, because the length of a list l is always greater than the length of any of its proper sublists.
We perform this steps first in the system. We resolve the fifth subgoal against the rule `wf_shorter`, which states the well-foundedness of the relation `shorter`

```
- br wf_shorter 5;
Level 21
Def((! l. quicksort(l) =
          if(l = [], l,
             quicksort(lesseq(hd(l), tl(l))) @
             [hd(l)] @ quicksort(greater(hd(l), tl(l))))) &
```

[10]This order is represented in Isabelle by the following set $\{q \mid \exists xy.\ q = <x,\ y> \wedge x < y\}$

```
    ((! x xa.
         greater(x, xa) =
         if(xa = [], xa,
            if(hd(xa) <= x, greater(x, tl(xa)),
               [hd(xa)] @ greater(x, tl(xa))))) &
     (! x xa.
         lesseq(x, xa) =
         if(xa = [], xa,
            if(hd(xa) <= x, [hd(xa)] @ lesseq(x, tl(xa)),
               lesseq(x, tl(xa)))))) &
    ?Q) -->
(! l. perm(l, quicksort(l)) & ordered(quicksort(l)))
 1. !!l x xa. [| l ~= []; x ~= [] |] ==> <tl(x), x> : ?ra(l)
 2. !!l. l ~= [] ==> wf(?ra(l))
 3. !!l. [| l ~= [];
            Def(! x xa.
                   lesseq(x, xa) =
                   if(xa = [], xa,
                      if(hd(xa) <= x, [hd(xa)] @ lesseq(x, tl(xa)),
                         lesseq(x, tl(xa))))) |] ==>
         <lesseq(hd(l), tl(l)), l> :
         induced_rel(length, {q. ? x y. q = <x, y> & x < y})
 4. !!l. [| l ~= [];
            Def(! x xa.
                   greater(x, xa) =
                   if(xa = [], xa,
                      if(hd(xa) <= x, greater(x, tl(xa)),
                         [hd(xa)] @ greater(x, tl(xa))))) |] ==>
         <greater(hd(l), tl(l)), l> :
         induced_rel(length, {q. ? x y. q = <x, y> & x < y})
```

Now we shall prove the assertions `<lesseq(hd(l), tl(l)), l>:induced_rel(length,<)` and `<greater(hd(l), tl(l)), l>:induced_rel(length,<)` displayed in the subgoals 3 and 4. In a first step we generalize this subgoals by replacing `hd(l)` by an arbitrary element `x`.

```
- BY (EVERY
       [subgoal_tac "!x. <lesseq(x, tl(l)), l>:induced_rel(length, {q. ? x y. q = <x, y> & x < y})" 3,
        fast_tac List_cs 3, rtac allI 3,
        subgoal_tac "!x. <greater(x, tl(l)), l> :induced_rel(length,{q. ? x y. q = <x, y> & x < y})" 4,
        fast_tac List_cs 4, rtac allI 4]);
Level 22
Def((! l. quicksort(l) =
          if(l = [], l,
             quicksort(lesseq(hd(l), tl(l))) @
             [hd(l)] @ quicksort(greater(hd(l), tl(l))))) &
    ((! x xa.
         greater(x, xa) =
         if(xa = [], xa,
            if(hd(xa) <= x, greater(x, tl(xa)),
               [hd(xa)] @ greater(x, tl(xa))))) &
     (! x xa.
         lesseq(x, xa) =
         if(xa = [], xa,
            if(hd(xa) <= x, [hd(xa)] @ lesseq(x, tl(xa)),
               lesseq(x, tl(xa)))))) &
    ?Q) -->
```

```
(! l. perm(l, quicksort(l)) & ordered(quicksort(l)))
 1. !!l u e. [| l ~= []; u ~= [] |] ==> <tl(u), u> : ?ra(l)
 2. !!l. l ~= [] ==> wf(?ra(l))
 3. !!l x. [| l ~= [];
             Def(! x xa.
                    lesseq(x, xa) =
                    if(xa = [], xa,
                       if(hd(xa) <= x, [hd(xa)] @ lesseq(x, tl(xa)),
                          lesseq(x, tl(xa))))) |] ==>
          <lesseq(x, tl(l)), l> :
          induced_rel(length, {q. ? x y. q = <x, y> & x < y})
 4. !!l x. [| l ~= [];
             Def(! x xa.
                    greater(x, xa) =
                    if(xa = [], xa,
                       if(hd(xa) <= x, greater(x, tl(xa)),
                          [hd(xa)] @ greater(x, tl(xa))))) |] ==>
          <greater(x, tl(l)), l> :
          induced_rel(length, {q. ? x y. q = <x, y> & x < y})
```

The generalized subgoals 3 and 4 can be inductively solved. The steps achieving this are regrouped in the following tactic

```
- BY quicksort_termination_tac ;
Level 23
Def((! l. quicksort(l) =
          if(l = [], l,
             quicksort(lesseq(hd(l), tl(l))) @
             [hd(l)] @ quicksort(greater(hd(l), tl(l))))) &
    ((! x xa.
         greater(x, xa) =
         if(xa = [], xa,
            if(hd(xa) <= x, greater(x, tl(xa)),
               [hd(xa)] @ greater(x, tl(xa))))) &
     (! x xa.
         lesseq(x, xa) =
         if(xa = [], xa,
            if(hd(xa) <= x, [hd(xa)] @ lesseq(x, tl(xa)),
               lesseq(x, tl(xa)))))) &
    ?Q) -->
(! l. perm(l, quicksort(l)) & ordered(quicksort(l)))
 1. !!l u e. [| l ~= []; u ~= [] |] ==> <tl(u), u> : ?ra(l)
 2. !!l. l ~= [] ==> wf(?ra(l))
```

Now we turn our attention the termination of *lesseq* and *greater*. We suggest the relation, `tail_relation  {<tl(x), x>| ~x=[]}`
as an instance of the relation `?ra`. `tail_relation` is a subrelation of `shorter` and therefore is well-founded. The constraint `<tl(x), x>:tail_relation` can be straightforwardly proven using the classical prover with the classical set `List_cs`

```
- BY ((wf_tail_relation 2) THEN (fast_tac List_cs 1));
Level 23
Def((! l. quicksort(l) =
          if(l = [], l,
             quicksort(lesseq(hd(l), tl(l))) @
             [hd(l)] @ quicksort(greater(hd(l), tl(l))))) &
    ((! x xa.
```

```
         greater(x, xa) =
         if(xa = [], xa,
            if(hd(xa) <= x, greater(x, tl(xa)),
               [hd(xa)] @ greater(x, tl(xa))))) &
     (! x xa.
         lesseq(x, xa) =
         if(xa = [], xa,
            if(hd(xa) <= x, [hd(xa)] @ lesseq(x, tl(xa)),
               lesseq(x, tl(xa)))))) &
    ?Q) -->
(! l. perm(l, quicksort(l)) & ordered(quicksort(l)))
No subgoals!
```

Finally we have no more subgoals which means that proof is finished. We can also bind the final proofstate to the identifier `quicksort_defs` by typing

```
val quicksort_defs = finalize (topthm());
  "[| !!l.
         ?quicksort(l) =
         if
          (l = [], l,
           ?quicksort(?lesseq(hd(l), tl(l))) @
           Cons(hd(l), ?quicksort(?greater(hd(l), tl(l)))));
      !!x xa.
         ?greater(x, xa) =
         if
          (xa = [], xa,
           if
            (hd(xa) <= x, ?greater(x, tl(xa)),
             Cons(hd(xa), ?greater(x, tl(xa)))));
      !!x xa.
         ?lesseq(x, xa) =
         if
          (xa = [], xa,
           if
            (hd(xa) <= x, Cons(hd(xa), ?lesseq(x, tl(xa))),
             ?lesseq(x, tl(xa)))) |] ==>
   ! l. perm(l, ?quicksort(l)) & ordered(?quicksort(l))" : thm
```

The function `finalize` simplifies the final proofstate after converting the free variables to schematic variables.

6.3 A Derivation of Mergesort

Let us now derive an other common sorting program: *mergesort*. In *mergesort* we decompose the input list `l` into two partitions, sort them and interleave to obtain the final output list. The derivation of *mergesort* is really the same as the derivation of *quicksort*, except that the liberty that we had to apply rules will now be used in a different way. The two first steps are identical to *quicksort*. We therefore make a jump to the third step of proof developement (for more details see appendix B.4 and B.2).

```
Level 2
Def((! l. mergesort(l) = ?fbody(l)) & ?Q) -->
(! l. perm(l, mergesort(l)) & ordered(mergesort(l)))
```

```
 1. !!l. [| Def(?Q);
           ! t. <t, l> : ?r -->
                perm(t, mergesort(t)) & ordered(mergesort(t)) |] ==>
        perm(l, ?fbody(l)) & ordered(?fbody(l))
 2. [| Def(! l. mergesort(l) = ?fbody(l)); Def(?Q) |] ==> wf(?r)
```

To apply induction we first have to split the list `l` into sublists. Here we have decided to decompose `l` and not its tail as we have done for *quicksort*. For efficiency reasons we examine whether `l` is empty. Further if the list `l` has only one element then decomposing it will be superfluous. So in the derivation we first distinguish the cases

```
- BY (res_inst_tac [("A","(l=[] | Single(l))"),("P","%f. perm(l, f)&ordered(f)")]case_splitf1 1);
Level 3
Def((! l. mergesort(l) = if(l = [] | Single(l), ?b(l), ?s(l))) & ?Q) -->
(! l. perm(l, mergesort(l)) & ordered(mergesort(l)))
 1. !!l. [| Def(?Q);
           ! t. <t, l> : ?r --> perm(t, mergesort(t)) & ordered(mergesort(t));
           l = [] | Single(l) |] ==>
        perm(l, ?b(l)) & ordered(?b(l))
 2. !!l. [| Def(?Q);
           ! t. <t, l> : ?r --> perm(t, mergesort(t)) & ordered(mergesort(t));
           ~ (l = [] | Single(l)) |] ==>
        perm(l, ?s(l)) & ordered(?s(l))
 3. [| Def(! l. mergesort(l) = if(l = [] | Single(l), ?b(l), ?s(l)));
       Def(?Q) |] ==>
    wf(?r)
```

The base case now can automatically proved using the classical prover with the classical set `List_cs` in which we have already introduced some lemmata about the predicates *perm* and *Single*.

```
- BY (fast_tac  List_cs 1);
Level 4
Def((! l. mergesort(l) = if(l = [] | Single(l), l, ?s(l))) & ?Q) -->
(! l. perm(l, mergesort(l)) & ordered(mergesort(l)))
 1. !!l. [| Def(?Q);
           ! t. <t, l> : ?r --> perm(t, mergesort(t)) & ordered(mergesort(t));
           ~ (l = [] | Single(l)) |] ==>
        perm(l, ?s(l)) & ordered(?s(l))
 2. [| Def(! l. mergesort(l) = if(l = [] | Single(l), l, ?s(l))); Def(?Q) |] ==>
    wf(?r)
```

Now we come to partitioning the list `l`. For this purpose we use the lemma `perm_trans`. So we get the new conclusion

```
          perm(l, ?t1(l)@?t2(l))                          (1)
                    &
    perm(?t1(l)@?t2(l), ?s(l)) & ord(?s(l))               (2)
```

Formula (1) describes the relationship between the partitions and the input list. The second states that the result list `?s(l)` is a permutation of these partitions and is ordered. Intuitively we expect a relationship between the intermediate results (the results of sorting `?t1(l)` and `?t2(l)`) and `?s(l)`. Therefore we assert that there are two metavariables `?l1(l) ?l2(l)` that denote the intermediate results such that

```
perm(?t1(l), ?l1(l)) & ordered(?l1(l))
```
(3)

```
perm(?t1(l), ?l2(l)) & ordered(?l2(l))
```
(4)

```
perm(?t1(l), ?l1(l)) & ordered(?l1(l)) &  perm(?t1(l), ?l2(l)) & ordered(?l2(l))
                                  -->
                 perm(?t1(l)@?t2(l), ?s(l)) & ord(?s(l))
```
(5)

The formulas (3) and (4) say, as already mentioned, that `?l1(l)and ?l2(l)` are the sorted lists of the partitions `?t1(l)` and `?t2(l)`. Formula (5) expresses: how we can compute the whole result `?s(l)` in dependence of the partitions of `l` knowing that `?l1(l)` and `?l2(l)` are already the sorted versions of these partitions. Using some lemmata about the predicate *perm* such as `perm_trans` and `perm_append` we can simplify formula (5) to

```
                  perm(?t1(l)@?t2(l), ?s(l))
                              &
     (ordered(?l1(l)) &  ordered(?l2(l)) -->  ord(?s(l)))
```

Now the above formula stands for a specification of the function merging two ordered lists into an ordered one that forms a permutation of them. In the system we have collected a sequence of tactics into one tactic called `synthesis_merge` which defines the function *merge*.

```
- BY synthesis_merge ;
Level 5
Def((! l. mergesort(l) = if(l = [] | Single(l), l, merge(?l1(l), ?l2(l)))) &
    (! x xa.
        merge(x, xa) =
        if(x = [], xa,
           if(xa = [], x,
              if(hd(x) <= hd(xa), [] @ [hd(x)] @ merge(tl(x), xa),
                 [] @ [hd(xa)] @ merge(x, tl(xa)))))) &
    ?Q) -->
(! l. perm(l, mergesort(l)) & ordered(mergesort(l)))
 1. !!l. [| ~ (l = [] | Single(l)); Def(?Q) |] ==> perm(l, ?t1(l) @ ?t2(l))
 2. !!l. [| ! t. <t, l> : ?r --> perm(t, mergesort(t)) & ordered(mergesort(t));
            ~ (l = [] | Single(l)); Def(?Q) |] ==>
         perm(?t1(l), ?l1(l)) & ordered(?l1(l))
 3. !!l. [| ! t. <t, l> : ?r --> perm(t, mergesort(t)) & ordered(mergesort(t));
            ~ (l = [] | Single(l)); Def(?Q) |] ==>
         perm(?t2(l), ?l2(l)) & ordered(?l2(l))
 4. wf(?r)
 5. !!l x xa.
       [| ~ (l = [] | Single(l)); Def(?Q); x ~= []; xa ~= [] |] ==>
       <<tl(x), xa>, x, xa> : ?ra(l)
 6. !!l x xa.
       [| ~ (l = [] | Single(l)); Def(?Q); x ~= []; xa ~= [] |] ==>
       <<x, tl(xa)>, x, xa> : ?ra(l)
 7. !!l. ~ (l = [] | Single(l)) ==> wf_prod(?ra(l))
```

The first and second subgoals correspond to the formulas (3) and (4). The third subgoal corresponds to the formula (1). We now proceed by resolving subgoals 2 and 3 using the induction hypothesis.

```
- BY ((apply_induction_hyp 2) THEN (apply_induction_hyp 3));
Level 6
Def((! l. mergesort(l) =
          if(l = [] | Single(l), l,
             merge(mergesort(?t1(l)), mergesort(?t2(l))))) &
    (! x xa.
        merge(x, xa) =
        if(x = [], xa,
           if(xa = [], x,
              if(hd(x) <= hd(xa), [] @ [hd(x)] @ merge(tl(x), xa),
                 [] @ [hd(xa)] @ merge(x, tl(xa)))))) &
    ?Q) -->
(! l. perm(l, mergesort(l)) & ordered(mergesort(l)))
 1. !!l. [| ~ (l = [] | Single(l)); Def(?Q) |] ==> perm(l, ?t1(l) @ ?t2(l))
 2. !!l. [| ~ (l = [] | Single(l)); Def(?Q) |] ==> <?t1(l), l> : ?r
 3. !!l. [| ~ (l = [] | Single(l)); Def(?Q) |] ==> <?t2(l), l> : ?r
 4. wf(?r)
 5. !!l x xa.
       [| ~ (l = [] | Single(l)); Def(?Q); x ~= []; xa ~= [] |] ==>
       <<tl(x), xa>, x, xa> : ?ra(l)
 6. !!l x xa.
       [| ~ (l = [] | Single(l)); Def(?Q); x ~= []; xa ~= [] |] ==>
       <<x, tl(xa)>, x, xa> : ?ra(l)
 7. !!l. ~ (l = [] | Single(l)) ==> wf_prod(?ra(l))
```

Now we come to synthesizing the partitions of the list `l` . Note that in the specification displayed in the first subgoal we only require that the partitions form a permutation of the input list. Here we haven't synthesized the standard partitions, namely the first half, called *left*, and the second half, called *right*, but we have suggested two lists, the first containing all elements with an even position in the input list,(we call it *even_ops*) and the second one, (called *odd_ops*), containing all elements with an odd position in the input list. We now derive these functions.

```
- BY  synthesis_even_odd_pos ;
Level 7
Def((! l. mergesort(l) =
          if(l = [] | Single(l), l,
             merge(mergesort(even_pos(l)), mergesort(odd_pos(l))))) &
    (! x xa.
        merge(x, xa) =
        if(x = [], xa,
           if(xa = [], x,
              if(hd(x) <= hd(xa), [] @ [hd(x)] @ merge(tl(x), xa),
                 [] @ [hd(xa)] @ merge(x, tl(xa)))))) &
    ((! x. even_pos(x) =
           if(x = [], x, if(tl(x) = [], [], [hd(x)] @ even_pos(tl(tl(x)))))) &
     (! x. odd_pos(x) =
           if(x = [], x,
              if(tl(x) = [], x, [hd(tl(x))] @ odd_pos(tl(tl(x))))))) &
    ?Q) -->
(! l. perm(l, mergesort(l)) & ordered(mergesort(l)))
```

```
 1. !!l x. [| ~ (l = [] | Single(l)); x ~= []; tl(x) ~= [] |] ==>
          <tl(tl(x)), x> : ?rb(l)
 2. !!l. ~ (l = [] | Single(l)) ==> wf(?rb(l))
 3. !!l. [| ~ (l = [] | Single(l));
           Def(! x. even_pos(x) =
                    if(x = [], x,
                       if(tl(x) = [], [],
                          [hd(x)] @ even_pos(tl(tl(x)))))) |] ==>
        <even_pos(l), l> : ?r
 4. !!l. [| Def(! x. odd_pos(x) =
                    if(x = [], x,
                       if(tl(x) = [], x, [hd(tl(x))] @ odd_pos(tl(tl(x))))));
           ~ (l = [] | Single(l)) |] ==>
        <odd_pos(l), l> : ?r
 5. wf(?r)
 6. !!l x xa.
       [| ~ (l = [] | Single(l)); x ~= []; xa ~= [] |] ==>
       <<tl(x), xa>, x, xa> : ?ra(l)
 7. !!l x xa.
       [| ~ (l = [] | Single(l)); x ~= []; xa ~= [] |] ==>
       <<x, tl(xa)>, x, xa> : ?ra(l)
 8. !!l. ~ (l = [] | Single(l)) ==> wf_prod(?ra(l))
```

At this stage we are finished with the synthesis. In the remainder of the derivation we establish termination of the developed functions. Subgoals 1 and 2 express termination of the functions *even_ops* and *odd_ops*. Subgoal 3, 4 and 5 express termination of the function *merge*. The rest express termination of *mergesort*. We start with the simple functions, namely *even_pos* and *odd_pos*. We suggest the relation `shorter` as an instance of `?rb`. As we have seen in the section above this relation is well-founded. The first subgoal could then be resolved using the monotonicity of the function `length` and the transitivity of the relation $<$ over the natural numbers. In the system we call the following tactic.

```
- BY (EVERY [ rtac wf_shorter 2,
              simp_tac appl_ss 1,
              fast_tac ((List_cs addEs [length_mono]) addIs [less_trans]) 1]);
Level 8
Def((! l. mergesort(l) =
          if(l = [] | Single(l), l,
             merge(mergesort(even_pos(l)), mergesort(odd_pos(l))))) &
    (! x xa.
        merge(x, xa) =
        if(x = [], xa,
           if(xa = [], x,
              if(hd(x) <= hd(xa), [] @ [hd(x)] @ merge(tl(x), xa),
                 [] @ [hd(xa)] @ merge(x, tl(xa)))))) &
    ((! x. even_pos(x) =
           if(x = [], x, if(tl(x) = [], [], [hd(x)] @ even_pos(tl(tl(x)))))) &
     (! x. odd_pos(x) =
           if(x = [], x,
              if(tl(x) = [], x, [hd(tl(x))] @ odd_pos(tl(tl(x)))))) &
    ?Q) -->
(! l. perm(l, mergesort(l)) & ordered(mergesort(l)))
 1. !!l. [| ~ (l = [] | Single(l));
           Def(! x. even_pos(x) =
                    if(x = [], x,
```

```
                        if(tl(x) = [], [],
                           [hd(x)] @ even_pos(tl(tl(x))))) |] ==>
         <even_pos(l), l> : ?r
 2. !!l. [| Def(! x. odd_pos(x) =
                     if(x = [], x,
                        if(tl(x) = [], x, [hd(tl(x))] @ odd_pos(tl(tl(x))))));
            ~ (l = [] | Single(l)) |] ==>
         <odd_pos(l), l> : ?r
 3. wf(?r)
 4. !!l x xa.
       [| ~ (l = [] | Single(l)); x ~= []; xa ~= [] |] ==>
       <<tl(x), xa>, x, xa> : ?ra(l)
 5. !!l x xa.
       [| ~ (l = [] | Single(l)); x ~= []; xa ~= [] |] ==>
       <<x, tl(xa)>, x, xa> : ?ra(l)
 6. !!l. ~ (l = [] | Single(l)) ==> wf_prod(?ra(l))
```

We now prove termination of the function *merge*. This function is defined recursively over pairs of lists. As an instance for `?ra` we make use of the lexicographic relation, `lex_rel(shorter, shorter)`, obtained by combining the relation `shorter` with itself.

```
- BY ((rtac ([wf_shorter ,wf_shorter] MRS wf_prod_lex_rel) 6)
     THEN
     (REPEAT ((simp_tac appl_ss 4) THEN (fast_tac (List_cs addIs [length_mono]) 4))));
Level 9
Def((! l. mergesort(l) =
          if(l = [] | Single(l), l,
             merge(mergesort(even_pos(l)), mergesort(odd_pos(l))))) &
    (! x xa.
        merge(x, xa) =
        if(x = [], xa,
           if(xa = [], x,
              if(hd(x) <= hd(xa), [] @ [hd(x)] @ merge(tl(x), xa),
                 [] @ [hd(xa)] @ merge(x, tl(xa)))))) &
    ((! x. even_pos(x) =
           if(x = [], x, if(tl(x) = [], [], [hd(x)] @ even_pos(tl(tl(x)))))) &
     (! x. odd_pos(x) =
           if(x = [], x,
              if(tl(x) = [], x, [hd(tl(x))] @ odd_pos(tl(tl(x))))))) &
    ?Q) -->
(! l. perm(l, mergesort(l)) & ordered(mergesort(l)))
 1. !!l. [| ~ (l = [] | Single(l));
            Def(! x. even_pos(x) =
                     if(x = [], x,
                        if(tl(x) = [], [],
                           [hd(x)] @ even_pos(tl(tl(x))))) |] ==>
         <even_pos(l), l> : ?r
 2. !!l. [| Def(! x. odd_pos(x) =
                     if(x = [], x,
                        if(tl(x) = [], x, [hd(tl(x))] @ odd_pos(tl(tl(x))))));
            ~ (l = [] | Single(l)) |] ==>
         <odd_pos(l), l> : ?r
 3. wf(?r)
```

Finally termination of the function *mergesort* can be proved in the same way as termination of the function `even_pos` and `odd_pos`. So we call the following tactic

```
- BY (EVERY [rewrite_goals_tac [Def_def],
             rtac wf_shorter 3,
             REPEAT (fast_tac (List_cs addIs [Single_E,even_shorter_I , odd_shorter_I]) 1)]);
Level 10
Def((! l. mergesort(l) =
          if(l = [] | Single(l), l,
             merge(mergesort(even_pos(l)), mergesort(odd_pos(l))))) &
    (! x xa.
        merge(x, xa) =
        if(x = [], xa,
           if(xa = [], x,
              if(hd(x) <= hd(xa), [] @ [hd(x)] @ merge(tl(x), xa),
                 [] @ [hd(xa)] @ merge(x, tl(xa)))))) &
    ((! x. even_pos(x) =
           if(x = [], x, if(tl(x) = [], [], [hd(x)] @ even_pos(tl(tl(x)))))) &
     (! x. odd_pos(x) =
           if(x = [], x,
              if(tl(x) = [], x, [hd(tl(x))] @ odd_pos(tl(tl(x))))))) &
    ?Q) -->
(! l. perm(l, mergesort(l)) & ordered(mergesort(l)))
No subgoals!
```

The derivation of *mergesort* is now completely finished. We can simplify the definitions we have obtained and bind the whole proofstate to an identifier `mergesort_defs`

```
- val mergesort_defs  = finalize (topthm());
val mergesort_defs =
  "[| !!l.
         ?mergesort(l) =
         if
          (l = [] | l = [hd(l)], l,
           ?merge(?mergesort(?even_pos(l)), ?mergesort(?odd_pos(l))));
      !!x xa.
         ?merge(x, xa) =
         if
          (x = [], xa,
           if
            (xa = [], x,
             if
              (hd(x) <= hd(xa), Cons(hd(x), ?merge(tl(x), xa)),
               Cons(hd(xa), ?merge(x, tl(xa))))));
      !!x.
         ?even_pos(x) =
         if(x = [], x, if(tl(x) = [], [], Cons(hd(x), ?even_pos(tl(tl(x))))));
      !!x.
         ?odd_pos(x) =
         if
          (x = [], x,
           if(tl(x) = [], x, Cons(hd(tl(x)), ?odd_pos(tl(tl(x)))))) |] ==>
   ! l. perm(l, ?mergesort(l)) & ordered(?mergesort(l))" : thm
```

6.4 A Derivation of Insert Sort

insertion sort is the first simple algorithm taught in the area of sorting. The main idea behind this program is: For a non empty list l, first recursively sort the tail of l then insert the first element in the correct place with respect to the appropriate order. *insertion* sort can also be seen as a variant of *mergesort*, where instead of splitting the list into two equal halves we divide it into the singleton list consisting of the first element and the second list contains the remaining elements of the list. Here we display only the final proofstate of its derivation. The complete proof will be given in appendix B.5.

```
val insertsort_defs = finalize (topthm());
"[| ! l. ?inssort(l) = if(l = [], l, ?insertion(hd(l), ?inssort(tl(l))));
      ! x xa.
         ?insertion(x, xa) =
         if
          (xa = [], Cons(x, xa),
           if
            (x <= hd(xa), Cons(x, ?insertion(hd(xa), tl(xa))),
             Cons(hd(xa), ?insertion(x, tl(xa))))) |] ==>
   ! l. perm(l, ?inssort(l)) & ordered(?inssort(l))" : thm
```

6.5 Summary

The starting point of both above derivations was the specification

$$\forall l.\ perm(l,\ ?sort(l)) \ \wedge\ ordered(?sort(l))$$

The derivations were different proofs of this top level specification. The variety of proofs led to different programs. The deciding steps responsible for generating the three represented programs were those where we have chosen different strategies to decompose the input list. In the derivation of *quicksort* we have aimed at decomposing the tail of the list. In the derivation of the *mergesort* and *insertsort* we have aimed at decomposing the list itself. For *mergesort* we also have decided that the partitions of the list should be all of the same length while for *insertsort* we have prefered that the first partition contains only the first element of the list and the second partition the remaining elements. Taking other decisions we can discover other programs, such as *bubblesort*, *selectsort*, etc. We now turn our attention to the termination part of the derivations. Applications of the well-founded induction principle always produces a subgoal whose conclusion has the form $wf(?r)$. This predicate stands for "There is a well-founded relation ?r". During the proof and by using the induction hypothesis we have produced subgoals whose conclusions are of the form $< \ldots,\ \ldots > \ \in ?r$. Let us discuss the possibility of solving these subgoals. We recapitulate the situation once again. Our proofstate contains subgoals of the form

$$\begin{array}{l} [\![\ldots]\!] \implies\ < f_1(x_1),\ g_1(y_1) > \ \in ?r \\ \qquad\qquad \vdots \\ [\![\ldots]\!] \implies\ < f_n(x_n),\ g_n(y_n) > \ \in ?r \\ \\ [\![\ldots]\!] \implies\ wf(?r) \end{array}$$

where f_i and g_i are functions ranging over the type of x_i for $1 \le i \le n$.

- A first solution to solve the subgoals would be to instantiate $?r$ and then to solve the constraints. In this manner we have, for example, proven the termination of the function *quicksort* (see section §level ...). There we have suggested the relation *shorter* as an instance for $?r$. This relation is well-founded by construction. But solving the constraints $< lesseq(xa, tl(l)),\ l > \ \in ?r$ and $< greater(xa, tl(l)),\ l > \ \in ?r$ was not trivial. In the derivation we used the derived rules *lesseq_shorter_I* and *greater_shorter_I*, which are proved by induction.

- An other possibility is to build the well-founded relation first by collecting the constraints as

  $$\{< f_i(x),\ g_i(x) > \ \mid\ for\ all\ x,\ 1 \leq i \leq n\ \}$$

  Solving the constraints using this method is straightforward, but the proof of the well-foundedness of this relation can be hard. We have chosen a strategy to prove the termination of the functions *lesseq* and *greater*. There the unique constraint was $< tl(x),\ x > \ \in ?r$. We have build the relation $?r$ as a the set

  $$\{< tl(x),\ x > \ \mid\ for\ all\ x \neq []\}$$

  The constraint is then automatically proved using the classical prover. Fortunately the well-foundedness proof of the relation was also simple, since this relation can be seen as a subrelation of *shorter* and therefore it is well-founded.

The termination parts of our derivations obviously belong to the domain of proving the termination of algorithms, which is known to be undecidable. In this area several works appear in the computer science literature. We will here a give a rough description of a recent method of Christoph Walther. The method published in his book [55] and the papers [54, 56] is proposed to automatically prove termination of certain algorithms. The main idea behind his method is to compare the size of the arguments in each recursive call and the size of the initially given input. The size of an argument is defined in terms of the number of constructors occuring in it. For example the size of a list is its length and the size of a tree is the number of its nodes. The well-founded relation then is the so called $size - order$ which is a subrelation of $<$ over natural numbers. To compare arguments with respect to the *size* function and thus to prove inequalities he has developed a deductive calculus, called *estimation calculus*. Using this calculus and some other basic knowledge about algorithms a so called *termination hypothesis* will be produced and proved automatically using the *INKA-System* [4]. For example if we consider the recursive algorithm

$$\begin{aligned} f\ x\ \Leftarrow\ & \phi_1 \rightarrow R(f\ t) \\ & \phi_2 \rightarrow H(f\ s), \end{aligned}$$

it terminates obviously if (1) $size(t) < size(x)$ and (2) $size(s) < size(x)$ are valid. The estimation calculus attempts to prove (1) and (2) by producing two first-order formulas ψ_1 and ψ_2 such that the validity of ψ_i implies (i) (i=1,2). The obtained termination hypotheses are $\phi_1 \rightarrow \psi_1$ and $\phi_1 \rightarrow \psi_2$. The method is already implemented by Wahlter and has been successful tested for a class of algorithms, including sorting algorithms. This seems important for our work and can be implemented as an Isabelle tactic. At the moment we have no experiments with it but this would be a topic for future work. Similar to idea, namely to postpone the instantiation of the well-founded order during an induction proof, Martin Protzen [47] proposed an idea, which is quite similar to ours, namely to postpone the instantiation of well-founded ordering during the induction proof. His strategy thus extends conventional *recursion analysis* in a direction diametrically opposed to Walther [57], who tries to do as many induction hypothesis as possible before the proof. In an other respect we can see the well-founded relation considered in our above derivations as set of rewriting rules. The well-foundedness then means the termination of the corresponding rewriting system. In the domain of rewriting systems comes a well known and successful technique called *recursive path orderings* (rpo) [18]. In rpo the well-founded relation is defined by a set of inference rules. This technique is already implemented in many systems such as *SATURATE* [2]. Using rpo in termination proofs has been proved powerful and practical. To implement rpo as another Isabelle tactic will also be an object of future investigation.

Chapter 7

Conclusion

7.1 Conclusion

Using Isabelle we have illustrated how logical frameworks, which are based on higher order unification and providing the use of metavariables in the inference rules as well as in the proof itself, can be convenient for program development. We have extended an existing theory in Isabelle about lists with some predicates such as the permutation *perm* and the ordered predicate *ordered* in order to express the sort specification. Lemmata about the defined predicates have been derived in a simple way profiting from the powerful tactics coming with Isabelle. For the development of the functional sorting programs we have essentially formalized two rules. The first rule is the case split schema, which was automatically proved. It is intended to construct conditional terms. The second one is the induction schema, which is a special instance of the well-founded principle. This rule was used for building recursive terms. But for our purpose the most fruitful system property is the use of metavariables in the proofs. So to introduce the conditional terms for example we have applied the case split schema to instantiate metavariables occuring in the proof with terms of the form $if(?A,\ ?s,\ ?t)$. Other metavariables such as $?A$, $?s$ and $?t$ are introduced in the proof and will be instantiated later. Considering the derivation of *quicksort* we observe at the starting point that the function is represented with a metavariable. Later at the third level the body of *quicksort* is instantiated with a conditional term. Notice also that the substitution of each metavariable is propagated in the whole proofstate. A further feature of our method is that the terms building the desired programs are constructed top-down. This means that at each stage of the proof the syntactic structure of each term is known even if it contains metavariables. But this property does not influence the structure of the generated programs. The derivation of *quicksort* and *mergesort* demonstrated that we are able to develop both top-down and bottom-up algorithms. The derived sorting programs are of the divide-and-conquer paradigm, this was not imposed by the system. Other varieties of proofs can produce different types of programs. The front/last example of last chapter isn't of the divide-and-conquer paradigm.

An other focus of our work is the separation between the construction of programs and their termination. If we look at our proofs for the sorting programs we always distinguish two parts. The first is where the bodies of the functions are completely defined. In the second one we have solved subgoals expressing termination of the functions. We call the first part, *partial correctness* and the second part *termination*. The idea to separate these tasks is indebted to practical purposes. The main factor that makes this separation possible is once again the use of metavariables. In the proof by applying the induction we don't specify a concrete well-founded relation, they only appear in the proofs after applying the induction metavariables $?r$ that should be well-founded. The advantage of this method is that the termination part can be automatically achieved using the techniques mentioned in the section above.

7.2 Related Work

7.2.1 Manna and Waldinger

In their deductive tableaux system [35, 37, 39] Manna and Waldinger have shifted program synthesis to a theorem-proving task, instead of the approach of applying transformation rules to a given specification. To a given first-order input specification deduction rules based on nonclausal resolution are applied. The program emerges as a "by-product" of the theorem-proving effort. A brief summary of the deductive tableaux system was given in the second chapter. The main idea behind their approach is the use of first-order metavariables to represent unspecified first-order objects. By applying deduction rules the metavariables become instantiated. Programs are constructed in a bottom-up way, this means first (for recursive programs) the base case will be constructed then the recursive case and finally a conditional term assembling both cases. Notice that this is opposed to our method. One of the main rules in the deductive tableaux system is the induction rule. Its application introduces new assertions, called induction hypotheses in the proof. The use of these assertions introduces recursion in the target programs. The induction rule corresponds to the well-founded induction principle. The corresponding well-founded relations needed by applying the induction rule can be chosen from a pre-existing set of basic well-founded relations.

So in this approach as opposed to our work Manna and Waldinger don't have the possibility to generate constraints for a well-founded relation that does not exist in the basic set. Another disadvantage in Manna and Waldinger's approach concerns the proof representation namely it is very difficulty to know what an application of a deduction rule intuitively means. One reason that contributes to this blur is that the rules are based on nonclausal resolution and that conjunctions occuring in the goal column (or respectively disjunctions occuring in the assertion column) can not break down to simpler ones. They can only be simplified if one of the values *true* or *false* occurs in them. For example looking at the step that produced row 11 from the rows 2 and 10 in the front/last example of section 2.2.5, we will not recognize that this step corresponds to the application of the induction hypothesis. During our translation of the deductive tableaux system we have noticed that some derived rules are really not derivable in the sense that besides the basic rules of the tableaux system we need some other semantic properties. For example the *ifsplit* rule can only be derived if we use the duality principle.

Nevertheless the deductive tableaux system has been of the best frameworks intended for program verification and program synthesis. Using Manna and Waldinger's work many programs in several domains are manually developed. The interesting ones of them are:

- The integer square root of nonnegative numbers [32]
- Finding the quotient of two integers [32]
- The Front/last substring of strings [39]
- The unification algorithm for first-order logic [33, 14]
- The sorting algorithms [53]
- Binary-search [36]

7.2.2 Smith

In [51] Smith presented a system, called *CYPRESS*, to synthesize a class of divide-and-conquer algorithms. A set of pre-existing program schemata serves to give the overall form of the target programs. Furthermore the system needs human assistance to achieve the key decisions of the synthesis such as design strategies, well-founded ordering, etc. After suggesting the top structure of the desired program the user has to choose a decomposition operator (or respectively a composition operator). The system generates specifications for the other components, composition operator, (or respectively decomposition operator and a corresponding base case operator), and auxiliary programs. Unfortunately many of the low-level auxiliary programs can't be synthesized using the pre-existing program schemata and must be synthesized by other methods. Therefore program development in this framework is quite limited.

7.2.3 Logic program synthesis

In the approach of deductive program synthesis we develop functional programs. There are other authors (see for example [26, 50, 19]) that have accorded a great importance to synthesizing logic programs. Hogger in his work [27] argues that for programs written in a conventional algorithmic languages it is hard to deduce the logical properties that can be used for their analysis. In general the derived properties of such programs are mixed with informations about control entities which only obscure the meaning of these properties. Logic programs have important properties. They are partial, multivalued and they may return more as one value using the backtracking mechanism. The target program can be expressed in the same language as the specification. So synthesis can be see as a transformation of the specification into some other formulas that constitute the definition of the logic program. Naturally these formula are written in a restricted syntax and can be directly executed or they can be translated into logic programming languages such as Prolog or Gödel.

7.2.3.1 Kraan, Basin and Bundy

In order to develop logic programs Kraan, Basin and Bundy [30] proposed to extend the proof planner *CLAM* [8], a theorem prover based on rippling [7, 29], symbolic evaluation and simplification, with the use of metavariables to represent unspecified objects. The idea was implemented in [31] and can also be used to overcome the selection of the kind of induction schema at the begin of the synthesis. We briefly describe their method: The top level specification has the form $\forall \vec{x}.\ ?P(\vec{x}) \leftrightarrow spec(\vec{x})$ where $?P$ is metavariable used to denote the body of the target program. The derivation of $?P$ consists of two phases: First, Bundy and his group attempt to determine the kind of the recursion schema that will be applied to derive $?P$. For this purpose they build the goal $?P(\vec{x}) \rightarrow ?P(?C(\vec{x}))$ and prove it. The proof is supported in the proof planner *CLAM*. During the proof of the above goal the metavariable $?P$ remains unchanged and the metavariable $?C$ becomes instantiated , C, (e.g. $C(x) = succ(x)$, $C(x) = pred(x), \ldots$). In the second phase $?P$ will be derived. The essential step of the derivation is the application of the induction. The used recursion schema has the following form

$$\frac{?P(\vec{x_0}) \qquad \begin{array}{c} ?P(\vec{x}) \\ \vdots \\ ?P(C(\vec{x})) \end{array}}{\forall \vec{x}.\ ?P(C(\vec{x}))}$$

where C is the constructor-selector function computed in the first phase and $\vec{x_0}$ represents the base case.

If we compare this approach with our work we notice that in our proofs we apply induction without a priori fixing of its corresponding recursion schema. Both activities are always achieved simultaneously. Following the above method an extra step is always needed to determine the appropriate type of the needed induction schema. A further notable difference is that the induction schemata generated by the above method corresponds to well-founded relations whose forms are restricted to the set $\{< x,\ C(x) > \mid\ \forall x\}$ where C is a constructor-selector function. For example the relation *shorter* (see section 6.2.5) can not be expressed as such a set and therefore programs like *quicksort* and *mergesort* can not be synthesized. Nevertheless the method here attempt to fully automate the synthesis of logic programs and can therefore be seen as an important first step in this direction.

7.2.3.2 Whelk

G.A. Wiggins in his paper [58] has published a system called *Whelk* intended to transform and synthesize logic programs. In his system formulas are tagged. This means they have the form $[\![F]\!]_\phi$, where F is a sorted first-order formula and ϕ represents a logic program that satisfies the specification F. The tag ϕ is expressed in the same logic as the formula F but with some restrictions such that it can be translated into executable logic program in a logic programming language. The *Whelk* rules constitute a calculus that transforms formulas into other equivalent ones. During a *Whelk* proof the tag formula of the input specification will be stepwise instantiated into a logic program. This approach is superficially similar to proofs-as-programs adapted in many systems [12, 9, 42] based on Martin-löf type theory [40] to synthesize functional programs.

7.2.3.3 IsaWhelk

Basin in his report [3] gave a translation of *Whelk*. The interpretation is called *IsaWhelk* and is implemented in the system Isabelle. *IsaWhelk* is a one to one translation its rules are derived in the intuitionistic first-order logic *IFOL* contributed in Isabelle. *IsaWhelk* also can be seen as a guarantee of aspects of the calculus of *Whelk*. Furthermore it is simple, transparent, easy to automate and to use. Basin in this work has particularly exploited higher-oder unification in Isabelle and also benefited from the tactics distributed in Isabelle.

7.2.3.4 Darlington

Darlington in [16] has developed six sorting algorithms for lists and arrays (*quicksort, mergesort, selection sort, insertion sort, exchange sort and bubble sort*) The starting specification was $Ordered(Perm(X))$, where $Perm(X)$ denotes the set of all permutations of X and *Ordered* is a filter that selects only the ordered ones. The form of this specification is called "generate-and-test". The synthesis was based on a set of transformation rules (i.g. *fold/unfold*, ...) and a set of reduction rules for sets. While unfolding corresponds to symbolic evaluation, folding is intended to introduce recursions. On the way to develop the sorting algorithms Darlington has first generated three recursive algorithms for computing *Perm*. These three varieties of *Perm* leads to three categories of sort algorithms

- *quicksort* and *insertion sort*
- *mergesort* and *selction sort*
- *bubble sort* and *exchange sort*

The difference of the structures of these programs reflects the difference between the permutation algorithms. But notice that our derivations don't depend on the definitions of the predicates *perm* and *ordered* occuring in the top level specification.

7.2.3.5 Clark and Darlington

In [11] Darlington with Clark had attempted to analyze the relationship between algorithms derived from a common specification. The authors have also chosen to construct programs in the sorting domain. The input specification is the same first-order formula that we have considered in our work as a specification for sorting algorithms. During the development the starting specification is decomposed in parts that can be directly solved using the folding rule and the rest represents the specification of auxiliary programs. Four sorting algorithms are derived: *quicksort*, *mergesort*, *insertion sort* and *selection sort*. The quicksort was given as follows :

$$\begin{aligned} quicksort(x_1@x_2) \Leftarrow\ & quicksort(y_1)@quicksort(y_2) \\ & such\ that\ partition(x_1@x_2,\ y_1,\ y_2) \end{aligned}$$

Obviously *quicksort* can fail to terminate if we choose y_1 and y_2 as $x_1@x_2$ and []. The failure of this program is caused by the application of the folding rule without checking that the arguments y_1 and y_2 of the recursive calls are smaller than $x_1@x_2$. The previous requirement guarantee the termination of *quicksort*. Notice that these requirements appeared in our proofs after applying the induction hypothesis in the form of constraints that will be satisfied from the well-founded relation which will be specified later.

7.2.4 Constructive Synthesis

Another main approach in the program synthesis is "*proofs-as-programs*". This approach is based on the curry-howard-isomorphism [28], where propositions are identified with types. Under this isomorphism, a proposition is true if and only if the corresponding type is inhabited. So a proof of a proposition consists in constructing a term of the corresponding type. Following this method a specification can be seen as a proposition and a program that satisfies it can be seen as term of the corresponding type. Systems that support this approach are typically based on some kind of constructive logic or type theory; examples of implemented systems with this method are Nuprl [12] and the calculus of constructions [13].

Appendix A

Tableau Rules

A.1 Nonclausal Resolution Rules

A.1.1 AA-Resolution

Rule

Assertions	*Goals*	*Outputs*
$A_1[P]$		s
$A_2[Q]$		t
$A_1\theta[false] \;\vee\; A_2\theta[true]$		$if\ P\theta\ then\ t\theta\ else\ s\theta$

Side-Conditions:
1. A_1 and A_2 have no variables in common,
2. P and Q are quantifier-free and unifiable,
3. θ is the most-general unifier of P and Q.

A.1.2 AG-Resolution

Rule

Assertions	*Goals*	*Outputs*
$A[P]$		s
	$G[Q]$	t
	$\neg(A\theta[false]) \;\wedge\; G\theta[true]$	$if\ P\theta\ then\ t\theta\ else\ s\theta$

Side-Conditions: The same conditions as for the AA-resolution rule.

A.1.3 GA-Resolution

Rule

Assertions	*Goals*	*Outputs*
	$G[Q]$	t
$A[P]$		s
	$\neg(A\theta[false]) \wedge G\theta[true]$	$if\ P\theta\ then\ t\theta\ else\ s\theta$

Side-Conditions: The same conditions as for the AA-resolution rule.

A.1.4 GG-Resolution

Rule

Assertions	*Goals*	*Outputs*
	$G_1[P]$	s
	$G_2[Q]$	t
	$G_1\theta[false] \wedge G_2\theta[true]$	$if\ P\theta\ then\ \theta\ else\ \theta$

Side-Conditions:The same conditions as for the AA-resolution rule.

A.1.5 The Ifsplit Rule

Rule

Assertions	*Goals*	*Outputs*
$A \wedge B$		s
A		s
B		s

A.1.6 The Andsplit Rule

Rule

Assertions	*Goals*	*Outputs*
	$A \vee B$	s
	A	s
	B	s

A.1.7 The Orsplit Rule

Rule

Assertions	*Goals*	*Outputs*
	$if\ A\ tehn\ B$	s
A		s
	B	s

A.2 Skolemization Rules

A.2.1 ∀-Elimination rules

Rule (Assertion-form)

Assertions	*Goals*	*Outputs*
$A[\forall(\exists)\, x.\, P(x)]$		s
$A[P(f(y, z))]$		s

Side-Conditions:

(i) A contains at least one occurrence of the subsentence $\forall(\exists)\, x.\, P(x)$, which is of strict universal force in the the tableau.

(ii) y is a vector formed from all free variables of A.

(iii) z all quantifiers of strict existential force surrounded the occurrence $\forall(\exists)\, x.\, P(x)$.

(iv) f is a new skolem function.

■

Rule (Goal-form)

Assertions	*Goals*	*Outputs*
	$G[\forall(\exists)\, x.\, P(x)]$	s
	$G[P(f(y, z))]$	s

Side-Conditions: The same conditions as for the A-form.

■

A.2.2 $\exists$-Elimination rules

Rule (Assertion-form)

$Assertions$	$Goals$	$Outputs$
$A[\forall(\exists)\, x.\, P(x)]$		s
$A[P(z)]$		s

Side-Conditions:

(i) A contains at least one occurrence of the subsentence $\forall(\exists)\, x.\, P(x)$, which is of strict existential force in the tableau.

(ii) The occurrence of $\forall(\exists)\, x.\, P(x)$ in A is not within the scope of any quantifier of universal force.

(iii) z is a new variable in the tableau.

∎

Rule (Goal-form)

$Assertions$	$Goals$	$Outputs$
	$G[\forall(\exists)\, x.\, P(x)]$	s
	$G[P(z)]$	s

Side-Conditions: The same conditions as for the A-form.

∎

A.3 Simplification Rules

A.3.1 The true-false simplification rules

connective	rule	name
Negation		
	$\neg true \Longrightarrow false$	*not true*
	$\neg false \Longrightarrow true$	*not false*
Conjunction		
	$F \wedge true \Longrightarrow F$	*and true*
	$F \wedge false \Longrightarrow false$	*and false*
	$true \wedge F \Longrightarrow F$	*true and*
	$false \wedge F \Longrightarrow false$	*false and*
Disjunction		
	$F \vee true \Longrightarrow true$	*or true*
	$F \vee false \Longrightarrow F$	*or false*
	$true \vee F \Longrightarrow true$	*true or*
	$false \vee F \Longrightarrow F$	*false or*
Implication		
	if true then $G \Longrightarrow G$	*if true*
	if false then $G \Longrightarrow$ *true*	*if false*
	if F *then true* $\Longrightarrow$ *true*	*then true*
	if F *then false* $\Longrightarrow \neg F$	*then false*
Equivalence		
	$F \equiv true \Longrightarrow F$	*iff true*
	$F \equiv false \Longrightarrow \neg F$	*iff false*
	$true \equiv F \Longrightarrow F$	*true iff*
	$false \equiv F \Longrightarrow \neg F$	*false iff*
Conditional		
	if true then G *else*$H \Longrightarrow G$	*cond if true*
	if false then G *else* $H \Longrightarrow H$	*cond if false*
	if F *then true else* $H \Longrightarrow (F \vee H)$	*cond then true*

A.3.2 Other logical simplification rules

connective	rule	name
Negation		
	$\neg\neg F \Longrightarrow F$	*not not*
	$\neg(\neg F \wedge \neg G) \Longrightarrow (F \vee G)$	*not and*
	$\neg(\neg F \vee \neg G) \Longrightarrow F \wedge G$	*not or*
	$\neg(if\ F\ then\ \neg G) \Longrightarrow F \wedge G$	*not if*
	$\neg(\neg F \equiv G) \Longrightarrow F \equiv G$	*not iff left*
	$\neg(F \equiv \neg G) \Longrightarrow F \equiv G$	*not iff Longright*
	$\neg(if\ F\ then\ \neg G\ else\ \neg H \Longrightarrow if\ F\ then\ G\ else\ H$	*cond else false*
Conjunction		
	$F \wedge F \Longrightarrow F$	*and two*
	$F \wedge \neg F \Longrightarrow false$	*and not*
	$\neg F \wedge F \Longrightarrow false$	*not and*
Disjunction		
	$F \vee F \Longrightarrow F$	*or two*
	$F \vee \neg F \Longrightarrow true$	*or not*
	$\neg F \vee F \Longrightarrow true$	*not or*
Implication		
	$if\ F\ then\ F \Longrightarrow true$	*if two*
	$if\ \neg F\ then\ F \Longrightarrow F$	*if not*
	$if\ F\ then\ true \Longrightarrow true$	*then not*
	$if\ \neg F\ then\ \neg F \Longrightarrow true$	*contraposition*
Equivalence		
	$F \equiv F \Longrightarrow true$	*iff two*
	$F \equiv \neg F \Longrightarrow false$	*iff not*
	$\neg F \equiv F \Longrightarrow false$	*not iff*
Conditional		
	$if\ F\ then\ G\ else G\ \Longrightarrow G$	*cond two*
	$if\ \neg F\ then\ G\ else\ H \Longrightarrow if\ F\ then\ H\ else\ G$	*cond not*

A.4 The Front/Last Example

label	*Assertions*	*Goals*	*Outputs*		*Rules*
			front(a)	*last(a)*	
1		*if* $a \neq \epsilon$ *then* $char(z_2) \ \wedge \ (a = z_1 * z_2)$	z_1	z_2	
2	*if* $x <_{string} a$ *then* *if* $x \neq \epsilon$ *then* $\boxed{char(last(x))}^{-}$ $\wedge$ $x = front(x) * last(x)$				*Ind*
3	$\boxed{a \neq \epsilon}^{-}$				*ifsplit*(1)
4		$char(z_2) \ \wedge \ a = \boxed{z_1 * z_2}^{+}$	z_1	z_2	
5	$\epsilon * w = w$				*Ass*
6		$char(z_2) \ \wedge \ \boxed{a = z_2}^{+}$	ϵ	z_2	$EQ(4,5)$
7	$\boxed{x = x}^{-}$				*Ass*
8		$\boxed{char(a)}^{+}$	ϵ	a	$AG(6,7)$
9	*if* $char(u)$ *then* $(u.y_1) * y_2 = u.(y_1 * y_2)$				*Ass*
10		$char(u) \ \wedge$ $char(y_2) \ \wedge$ $a = u.(y_1 * y_2)$	$u.y_1$	y_2	$EQ(7,9)$
11		$x <_{string} a \ \wedge$ $x \neq \epsilon \ \wedge$ $char(u) \ \wedge$ $a = u.x \ \wedge$ $\boxed{char(last(x))}^{+}$	$u.front(x)$	$last(x)$	$EQ(2,10)$

$label$	$Assertions$	$Goals$	$Outputs$		$Rules$
			$front(a)$	$last(a)$	
12		$x <_{string} a \wedge$ $x \neq \epsilon \wedge$ $char(u) \wedge$ $\boxed{a = u.x}^{+}$	$u.front(x)$	$last(x)$	$AG(2,11)$
13	$if\ w \neq \epsilon\ then$ $\boxed{w = hd(w).tl(w)}^{-}$				Ass
14		$tl(a) <_{string} a \wedge$ $tl(a) \neq \epsilon \wedge$ $\boxed{char(hd(a))}^{+}$ $\wedge\ a \neq \epsilon$	$hd(a).front(tl(a))$	$last(tl(a))$	$AG(12,13)$
15	$if\ (w \neq \epsilon)\ then$ $\boxed{char(hd(w))}^{-}$				Ass
16		$tl(a) \neq \epsilon$ $\boxed{tl(a) <_{string} a}^{+}$ $\wedge\ a \neq \epsilon$	$hd(a).front(tl(a))$	$last(tl(a))$	$AG(14,15)$
17	$if\ (w \neq \epsilon)\ then$ $\boxed{tl(w) <_{string} w}^{-}$				Ass
18		$a \neq \epsilon$ $\wedge$ $\boxed{tl(a) \neq \epsilon)}^{+}$	$hd(a).front(tl(a))$	$last(tl(a))$	$GA(16,17)$
19	$w = \epsilon\ \vee\ char(w)$ $\vee \boxed{tl(w) \neq \epsilon}^{-}$				Ass

$label$	$Assertions$	$Goals$	$Outputs$		$Rules$
			$front(a)$	$last(a)$	
20		$\boxed{a \neq \epsilon}^{+}$ $\wedge$ $\neg(char(a))$	$hd(a).front(tl(a))$	$last(tl(a))$	$GA(18, 19)$
21		$\neg \boxed{char(a)}^{-}$	$hd(a).front(tl(a))$	$last(tl(a))$	$AG(3, 20)$
22		$true$	$if\ char(a)$ $then\ \epsilon\ else$ $hd(a).front(tl(a))$	$if\ char(a)$ $then\ a\ else$ $last(tl(a))$	$GG(8, 21)$

Appendix B

Examples

B.1 The Theory

```
APPL   = List +
classes
  ord_t < ord
consts
   wf_prod          :: "('a * 'a)set => bool"
   induced_rel      :: "['a=>'b, ('b*'b)set]=>('a*'a)set"
   lex_rel          :: "[('a*'a)set, ('b*'b)set] =>(('a * 'b) * ('a * 'b))set"
   Single           :: " 'a list => bool"
   member           :: " ['a , 'a list] => bool"
   ordered          :: "  ('a::ord_t) list => bool"
   perm             :: " ['a list , 'a list] => bool"
   maxi, mini       :: " ['a::ord_t , 'a list] => bool"
   Def              :: " bool => bool"
   length           :: "  'a list => nat"
rules
   wf_prod_def            "wf_prod(r)== (!P.
                                        (!x1 x2.
                                          (!y1 y2. <<y1, y2>,<x1,x2>>:r --> P(y1,y2)) --> P(x1,x2))
                                         --> (!x1 x2.P(x1,x2)))"
   induced_rel_def        "induced_rel(f, r)  == Collect(%q. ? x y. ( q=<x ,y> & <f(x),f(y)>:r))"
   lex_rel_def            "lex_rel(ls1, ls2)  ==
                               Collect(%q. ? x1 x2 y1 y2.
                                ((q=<<x1,x2>, <y1,y2>>) & (<x1,y1>:ls1 | (x1=y1 & <x2, y2>:ls2))))"
   leq_def_refl           " a::'a::ord_t <= a"
   leq_def_trans          " [| a::'a::ord_t <= b; b <= c |] ==> a <= c"
   leq_def_antisym        " [| a::'a::ord_t <= b; b <= a |] ==> a = b"
   l_def                  " (a::'a::ord_t < b) == (~(a = b) & a <= b) "
   leq_def_totality       " (~(a::'a::ord_t <= b)) = (b < a)"
   member_Nil             " ~(member(a ,  []))"
   member_Cons            " member(a, Cons(h, t))= ( a = h | member(a, t))"
   perm_Nil               " perm ([], [])"
   perm_append_singleton  " perm (l1@[u]@l2, t1@[u]@t2)= perm(l1@l2, t1@t2)"
   perm_member            " perm(l1, l2) -->(! a.(member(a,l1)= member(a, l2)))"
   ord_Nil                " ordered([])"
   ord_singleton          " ordered ([a])"
   ord_double_insertion   " ordered(Cons(a,Cons(b,t)))=(a<=b)&ordered(Cons(b,t))"
   maxi_Nil               " maxi(u, [])"
   mini_Nil               " mini(u, [])"
   maxi_Cons              " maxi(u, Cons(a, l)) = ((a<=u)& maxi(u, l))"
```

```
   mini_Cons            " mini(u,Cons(a, l))  = ((u<=a)& mini(u, l))"
   Single_def0          " Single([x])"
   Single_def1          " ~ Single([])"
   Single_def2          " ~l=[]-->~Single(Cons(x,l))"
   Def_def              " Def(P) == P"
   length_def_Nil       " length([])=0"
   length_def_Cons      " length(Cons(h,t))= Suc(length(t))"
end
```

B.2 Tactics

```
val Intro              = FIRST  (map (fn rule => resolve_tac [rule] 1) [allI, impI, conjI]);

fun SetUpContext n     = SELECT_GOAL (REPEAT Intro) n;

fun IndTac lemma inst n = (ALLGOALS (etac DefConjE))         (* Extend definition variable *)
                                                             (* Finish extension *)
                          THEN (res_inst_tac inst lemma n)   (* Do Induction *)
                          THEN (assume_tac n)                (* Instantiate Def *)
                          THEN (etac DefThin n)              (* Thin out Def from subgoal n *)

fun INDTAC  shema prog_names inst i =
      let val new_shema = read_instantiate prog_name shema (* Introduction of the names *)
      in  (IndTac tac inst i)                              (* Apply the induction *)
          THEN
          SetUpContext                                   (* Apply all introduction rules except exI*)
      end;

fun BY  t   =  by (t THEN (PRIMITIVE (fn state => (zero_var_indexes(state)))));

fun finalize  thm = (rule_by_tactic (setupcontext 1)
                    (rule_by_tactic (simp_tac appl_ss 1)
                    ((read_instantiate [("Q","True")] (rewrite_rule [Def_def]
                                   ((zero_var_indexes (thm))) )) RS mp)));


val  less_or_eq_inductive_case =
                        (EVERY1 [SELECT_GOAL (REPEAT_FIRST (fn i=> (rtac conjI i))),
                                 artac hd_tl_E ,
                                 rtac perm_eq_Cons_I,
                                 rtac exI ,rtac exI,
                                 rtac conjI])
                         THEN
                         (EVERY [res_inst_tac [("P","%l. (?A =perm (l))")] (append_assoc RS subst) 1,
                                rtac perm_eq_congs 1,
                                rtac perm_eq_commutativ 1 ,
                                (REPEAT(rtac perm_eq_refl 1)),
                                asm_simp_tac appl_ss  3,
                                EVERY (map (fn i => (apply_induction_hyp i)) [1,2,3]),
                                EVERY (map (fn i =>(REPEAT (etac thinrule2 i))) [1,2,3]),
                                PRIMITIVE (fn state => (state COMP rule6))]);

val  greater_inductive_case =
                      (SELECT_GOAL (REPEAT_FIRST (fn i=> (rtac conjI i))) 1)
                      THEN ((EVERY1 [res_inst_tac [("P","%u. perm(u,?A)")] hd_tl_E,
                                   atac,
```

```
                              rtac perm_eq_Cons_I,
                              rtac exI ,rtac exI ,
                              rtac conjI])
                         THEN
                         (EVERY [ rtac perm_eq_congs 1,
                              REPEAT (rtac perm_eq_refl 1),
                              asm_full_simp_tac appl_ss 2,
                              EVERY (map (fn i => (apply_induction_hyp i)) [1,3,2]),
                              EVERY (map (fn x =>(REPEAT (etac thinrule2 x))) [1,2,3]),
                              EVERY [rotate 2 1,etac thinrule 1,rotate 2 2,etac thinrule 2,
                                    etac thinrule 2,rotate 2 3,etac thinrule 3,
                                    rotate 2 4,etac thinrule 4],
                              PRIMITIVE (fn state => (state COMP rule7))]));



val quicksort_termination_tac =
                      (rewrite_goals_tac [Def_def])
                      THEN
                      (REPEAT (fast_tac (List_cs addIs [lesseq_shorter_I,greater_shorter_I]) 3));

val synthesis_merge =
      (EVERY1 [rtac conjI ,
               rtac perm_trans,
              rtac perm_append_with_trans])
      THEN  (EVERY [res_inst_tac [("P1","ordered(?z1)"),("Q1","ordered(?z2)")] (conjI RSN (2,mp)) 5,
                  PRIMITIVE (fn state => (state COMP rule)),
                  (INDTAC (indschemaf1_2p2 RS spec RS spec)
                   [("f2","merge")]
                   [("Spec2","%l1 l2 f. perm(l1@ l2,f)&((ordered(l1)&ordered(l2))-->ordered(f))")]
                   1),
                  (res_inst_tac [("P","%f. perm(x@xa,f)&((ordered(x) & ordered(xa))--> ordered(f))"),
                               ("l","x")]
                   case_splitf1_list 1),
                  EVERY1 [asm_simp_tac appl_ss,
                         rtac (perm_refl RSN (1, conjI)),
                         asm_simp_tac appl_ss,
                         (res_inst_tac
                          [("P","%f. perm(x@xa,f)&((ordered(x) & ordered(xa))--> ordered(f))"),
                           ("l","xa")] case_splitf1_list ),
                         asm_simp_tac appl_ss,
                         rtac (perm_refl RSN (1, conjI)),
                         asm_simp_tac appl_ss,
                         res_inst_tac [("A","hd(x)<=hd(xa)")] case_splitf1 ,
                         etac  thinrule ,
                         rtac conjI ,
                         artac hd_tl_E ,
                         simp_tac  list_ss ,
                         rtac perm_Cons_I],
                  REPEAT (rtac exI 1),
                  EVERY1 [ rtac conjI , rtac refl],
                  rtac impI  2,
                  rtac ord_append 2,
                  ALLGOALS (fn i => (simp_tac appl_ss i)),
                  rtac conjI 3,
                  rtac  ord_Nil 2,
                  EVERY1 [simp_tac appl_ss ,apply_induction_hyp],
```

```
REPEAT (etac allE 2),
dtac imp_and_elim  2,
etac conjE 2,
etac (mp RS mp) 2,
asm_simp_tac appl_ss 3,
rtac (ord_insertion1 RS conjunct1) 3,
artac hd_tl_I  3 ,
asm_simp_tac appl_ss 3,
rtac mini_perm 3,
apply_induction_hyp 3,
rtac mini_append 4 ,
rtac (ord_Cons1 RS ord_insertion1 RS conjunct2) 4,
artac hd_tl_I 5,
rtac leq_def_refl 4,
asm_simp_tac appl_ss 4,
artac hd_tl_E 4,
rtac (mini_Cons RS ssubst)  4,
artac conjI 4,
etac (ord_Cons1 RS ord_insertion1 RS conjunct2) 4,
artac hd_tl_I 4,
asm_simp_tac appl_ss 4,
rtac maxi_Nil 4,
etac thinrule 4,
dres_inst_tac [("P","%x.x")] (leq_def_totality RS subst) 4,
rtac conjI 4,
res_inst_tac [("P","%l. perm(?A@l,?B)"),("s","[hd(xa)]@tl(xa)")] ssubst 4,
simp_tac appl_ss 4,
artac hd_tl 4,
rtac (perm_append_singleton RS ssubst) 4,
rtac impI 5,
rtac ord_append 5,
ALLGOALS (fn i => (simp_tac appl_ss i)),
rtac conjI 6,
rtac ord_Nil 5,
simp_tac appl_ss 4,
apply_induction_hyp 4,
REPEAT (etac allE 5),
dtac imp_and_elim  5,
etac conjE 5,
etac (mp RS mp) 5,
asm_simp_tac appl_ss 6,
rtac (ord_insertion1 RS conjunct1) 6,
artac hd_tl_I  6 ,
asm_simp_tac appl_ss 6,
rtac mini_perm 6,
apply_induction_hyp 6,
rtac mini_append 7,
artac hd_tl_E  7,
rtac (ord_Cons1 RSN (2,(ord_insertion1 RS conjunct2 RSN (2, mini_Cons_I)))) 7,
artac hd_tl_I 9,
etac conjE 9,
atac 9,
(repeat_n_times (asm_full_simp_tac appl_ss  7) 2),
rtac (ord_insertion1 RS conjunct2) 7,
artac hd_tl_I 7,
etac conjE 7,
atac 7,
```

```
                  rtac  maxi_Nil 7,
                  clean_up]);

val synthesis_even_odd_pos =
    EVERY [(INDTAC (indschemaf2_1p1 RS spec)
           [("f1","even_pos"),("g1","odd_pos") ] [("Spec1","% l f g. perm(l, f@g)"),("x","l")] 1),
          res_inst_tac [("P","%f g. perm(x,f@g)"),("A","x=[]")] case_splitf2 1,
          hyp_subst_tac 1,
          rtac (append_Nil RS subst) 1,
          rtac perm_append 1,
          REPEAT (rtac perm_Nil 1),
          res_inst_tac [("A","tl(x)=[]")] case_splitf2 1,
          artac (list_decompose RS ssubst) 1,
          rtac perm_append 1,
          rtac perm_Nil 1,
          asm_simp_tac list_ss 1,
          rtac perm_refl 1,
          artac hd_tl_E 1,
          rtac perm_eq_Cons_I 1,
          REPEAT (rtac exI 1),
          rtac conjI 1,
          res_inst_tac [("P","%l. (?A =perm (l))")] (append_assoc RS subst) 1,
          rtac (perm_eq_commutativ RS perm_eq_congs) 1,
          REPEAT (rtac perm_eq_refl 1),
          artac hd_tl_E 1,
          rtac perm_eq_Cons_I 1,
          REPEAT (rtac exI 1),
          rtac conjI 1,
          rtac perm_eq_congs 1,
          REPEAT (rtac perm_eq_refl 1),
          apply_induction_hyp 1,
          etac DefThin 1,
          REPEAT (etac DefThin 2),
          etac DefConjE 3, etac DefThin 3, rotate 2 3, etac DefThin 3,
          etac DefConjE 4, rotate 1 4, etac DefThin 4,etac DefThin 4,
          REPEAT (etac DefThin 6),
          REPEAT (etac DefThin 7)];
```

B.3 The Quicksort derivation

The derivation is complexly described in section 6.2

B.4 The Mergesort derivation

```
goal thy "Def(?H)-->(ALL l.(perm(l, ?sort(l))& ordered(?sort(l))))";

(*1*)  br  impI 1;
(*2*)  BY  (INDTAC indschemaf1_1p1 [("f","mergesort")] [("Spec","%l f. perm(l, f) & ordered(f)")] 1);
(*3*)  BY  (res_inst_tac [("A","(l=[] | Single(l))"),("P","%f. perm(l, f)&ordered(f)")]case_splitf1 1);
(*4*)  BY  (fast_tac  List_cs 1);
(*5*)  BY  synthesis_merge ;
(*6*)  BY  ((apply_induction_hyp 2) THEN (apply_induction_hyp 3));
(*7*)  BY  synthesis_even_odd_pos ;

         (**** TERMINATION OF even_pos and odd_pos ****)
```

```
(*8*)  BY (EVERY [rtac wf_shorter 2,
                  simp_tac appl_ss 1,
                  fast_tac ((List_cs addEs [length_mono]) addIs [less_trans]) 1]);


         (**** TERMINATION OF MERGE ****)

(*9*)  BY ((rtac ([wf_shorter ,wf_shorter] MRS wf_prod_lex_rel) 6)
          THEN
           (REPEAT ((simp_tac appl_ss 4) THEN (fast_tac (List_cs addIs [length_mono]) 4))));

         (**** TERMINATION OF MERGESORT ****)

(*10*) BY (EVERY [rewrite_goals_tac [Def_def],
                 rtac wf_shorter 3,
                 REPEAT (fast_tac (List_cs addIs [Single_E,even_shorter_I , odd_shorter_I]) 1)]);

finalize(topthm());

Level 0
Def(?H) --> (! l. perm(l, ?sort(l)) & ordered(?sort(l)))
 1. Def(?H) --> (! l. perm(l, ?sort(l)) & ordered(?sort(l)))

Level 1
Def(?H) --> (! l. perm(l, ?sort(l)) & ordered(?sort(l)))
 1. Def(?H) ==> ! l. perm(l, ?sort(l)) & ordered(?sort(l))

Level 2
Def((! l. mergesort(l) = ?fbody(l)) & ?Q) -->
(! l. perm(l, mergesort(l)) & ordered(mergesort(l)))
 1. !!l. [| Def(?Q);
            ! t. <t, l> : ?r -->
                 perm(t, mergesort(t)) & ordered(mergesort(t)) |] ==>
         perm(l, ?fbody(l)) & ordered(?fbody(l))
 2. [| Def(! l. mergesort(l) = ?fbody(l)); Def(?Q) |] ==> wf(?r)

Level 3
Def((! l. mergesort(l) = if(l = [] | Single(l), ?b(l), ?s(l))) & ?Q) -->
(! l. perm(l, mergesort(l)) & ordered(mergesort(l)))
 1. !!l. [| Def(?Q);
            ! t. <t, l> : ?r --> perm(t, mergesort(t)) & ordered(mergesort(t));
            l = [] | Single(l) |] ==>
         perm(l, ?b(l)) & ordered(?b(l))
 2. !!l. [| Def(?Q);
            ! t. <t, l> : ?r --> perm(t, mergesort(t)) & ordered(mergesort(t));
            ~ (l = [] | Single(l)) |] ==>
         perm(l, ?s(l)) & ordered(?s(l))
 3. [| Def(! l. mergesort(l) = if(l = [] | Single(l), ?b(l), ?s(l)));
       Def(?Q) |] ==>
    wf(?r)

Level 4
Def((! l. mergesort(l) = if(l = [] | Single(l), l, ?s(l))) & ?Q) -->
(! l. perm(l, mergesort(l)) & ordered(mergesort(l)))
 1. !!l. [| Def(?Q);
            ! t. <t, l> : ?r --> perm(t, mergesort(t)) & ordered(mergesort(t));
```

```
           ~ (l = [] | Single(l)) |] ==>
        perm(l, ?s(l)) & ordered(?s(l))
 2. [| Def(! l. mergesort(l) = if(l = [] | Single(l), l, ?s(l))); Def(?Q) |] ==>
    wf(?r)

Level 5
Def((! l. mergesort(l) = if(l = [] | Single(l), l, merge(?l1.0(l), ?l2.0(l)))) &
    (! x xa.
        merge(x, xa) =
        if(x = [], xa,
           if(xa = [], x,
              if(hd(x) <= hd(xa), [] @ [hd(x)] @ merge(tl(x), xa),
                 [] @ [hd(xa)] @ merge(x, tl(xa)))))) &
    ?Q) -->
(! l. perm(l, mergesort(l)) & ordered(mergesort(l)))
 1. !!l. [| ~ (l = [] | Single(l)); Def(?Q) |] ==> perm(l, ?t1.0(l) @ ?t2.0(l))
 2. !!l. [| ! t. <t, l> : ?r --> perm(t, mergesort(t)) & ordered(mergesort(t));
           ~ (l = [] | Single(l)); Def(?Q) |] ==>
        perm(?t1.0(l), ?l1.0(l)) & ordered(?l1.0(l))
 3. !!l. [| ! t. <t, l> : ?r --> perm(t, mergesort(t)) & ordered(mergesort(t));
           ~ (l = [] | Single(l)); Def(?Q) |] ==>
        perm(?t2.0(l), ?l2.0(l)) & ordered(?l2.0(l))
 4. wf(?r)
 5. !!l x xa.
       [| ~ (l = [] | Single(l)); Def(?Q); x ~= []; xa ~= [] |] ==>
       <<tl(x), xa>, x, xa> : ?ra(l)
 6. !!l x xa.
       [| ~ (l = [] | Single(l)); Def(?Q); x ~= []; xa ~= [] |] ==>
       <<x, tl(xa)>, x, xa> : ?ra(l)
 7. !!l. ~ (l = [] | Single(l)) ==> wf_prod(?ra(l))

Level 6
Def((! l. mergesort(l) =
          if(l = [] | Single(l), l,
             merge(mergesort(?t1.0(l)), mergesort(?t2.0(l))))) &
    (! x xa.
        merge(x, xa) =
        if(x = [], xa,
           if(xa = [], x,
              if(hd(x) <= hd(xa), [] @ [hd(x)] @ merge(tl(x), xa),
                 [] @ [hd(xa)] @ merge(x, tl(xa)))))) &
    ?Q) -->
(! l. perm(l, mergesort(l)) & ordered(mergesort(l)))
 1. !!l. [| ~ (l = [] | Single(l)); Def(?Q) |] ==> perm(l, ?t1.0(l) @ ?t2.0(l))
 2. !!l. [| ~ (l = [] | Single(l)); Def(?Q) |] ==> <?t1.0(l), l> : ?r
 3. !!l. [| ~ (l = [] | Single(l)); Def(?Q) |] ==> <?t2.0(l), l> : ?r
 4. wf(?r)
 5. !!l x xa.
       [| ~ (l = [] | Single(l)); Def(?Q); x ~= []; xa ~= [] |] ==>
       <<tl(x), xa>, x, xa> : ?ra(l)
 6. !!l x xa.
       [| ~ (l = [] | Single(l)); Def(?Q); x ~= []; xa ~= [] |] ==>
       <<x, tl(xa)>, x, xa> : ?ra(l)
 7. !!l. ~ (l = [] | Single(l)) ==> wf_prod(?ra(l))

Level 7
Def((! l. mergesort(l) =
```

```
         if(l = [] | Single(l), l,
            merge(mergesort(even_pos(l)), mergesort(odd_pos(l))))) &
    (! x xa.
        merge(x, xa) =
        if(x = [], xa,
           if(xa = [], x,
              if(hd(x) <= hd(xa), [] @ [hd(x)] @ merge(tl(x), xa),
                 [] @ [hd(xa)] @ merge(x, tl(xa)))))) &
    ((! x. even_pos(x) =
           if(x = [], x, if(tl(x) = [], [], [hd(x)] @ even_pos(tl(tl(x)))))) &
     (! x. odd_pos(x) =
           if(x = [], x,
              if(tl(x) = [], x, [hd(tl(x))] @ odd_pos(tl(tl(x))))))) &
    ?Q) -->
(! l. perm(l, mergesort(l)) & ordered(mergesort(l)))
 1. !!l x. [| ~ (l = [] | Single(l)); x ~= []; tl(x) ~= [] |] ==>
           <tl(tl(x)), x> : ?rb(l)
 2. !!l. ~ (l = [] | Single(l)) ==> wf(?rb(l))
 3. !!l. [| ~ (l = [] | Single(l));
            Def(! x. even_pos(x) =
                     if(x = [], x,
                        if(tl(x) = [], [],
                           [hd(x)] @ even_pos(tl(tl(x)))))) |] ==>
         <even_pos(l), l> : ?r
 4. !!l. [| Def(! x. odd_pos(x) =
                     if(x = [], x,
                        if(tl(x) = [], x, [hd(tl(x))] @ odd_pos(tl(tl(x))))));
            ~ (l = [] | Single(l)) |] ==>
         <odd_pos(l), l> : ?r
 5. wf(?r)
 6. !!l x xa.
       [| ~ (l = [] | Single(l)); x ~= []; xa ~= [] |] ==>
       <<tl(x), xa>, x, xa> : ?ra(l)
 7. !!l x xa.
       [| ~ (l = [] | Single(l)); x ~= []; xa ~= [] |] ==>
       <<x, tl(xa)>, x, xa> : ?ra(l)
 8. !!l. ~ (l = [] | Single(l)) ==> wf_prod(?ra(l))

Level 8
Def((! l. mergesort(l) =
          if(l = [] | Single(l), l,
             merge(mergesort(even_pos(l)), mergesort(odd_pos(l))))) &
    (! x xa.
        merge(x, xa) =
        if(x = [], xa,
           if(xa = [], x,
              if(hd(x) <= hd(xa), [] @ [hd(x)] @ merge(tl(x), xa),
                 [] @ [hd(xa)] @ merge(x, tl(xa)))))) &
    ((! x. even_pos(x) =
           if(x = [], x, if(tl(x) = [], [], [hd(x)] @ even_pos(tl(tl(x)))))) &
     (! x. odd_pos(x) =
           if(x = [], x,
              if(tl(x) = [], x, [hd(tl(x))] @ odd_pos(tl(tl(x))))))) &
    ?Q) -->
(! l. perm(l, mergesort(l)) & ordered(mergesort(l)))
 1. !!l. [| ~ (l = [] | Single(l));
            Def(! x. even_pos(x) =
```

```
                     if(x = [], x,
                        if(tl(x) = [], [],
                           [hd(x)] @ even_pos(tl(tl(x))))) |] ==>
         <even_pos(l), l> : ?r
 2. !!l. [| Def(! x. odd_pos(x) =
                     if(x = [], x,
                        if(tl(x) = [], x, [hd(tl(x))] @ odd_pos(tl(tl(x))))));
            ~ (l = [] | Single(l)) |] ==>
         <odd_pos(l), l> : ?r
 3. wf(?r)
 4. !!l x xa.
       [| ~ (l = [] | Single(l)); x ~= []; xa ~= [] |] ==>
       <<tl(x), xa>, x, xa> : ?ra(l)
 5. !!l x xa.
       [| ~ (l = [] | Single(l)); x ~= []; xa ~= [] |] ==>
       <<x, tl(xa)>, x, xa> : ?ra(l)
 6. !!l. ~ (l = [] | Single(l)) ==> wf_prod(?ra(l))

Level 9
Def((! l. mergesort(l) =
          if(l = [] | Single(l), l,
             merge(mergesort(even_pos(l)), mergesort(odd_pos(l))))) &
    (! x xa.
        merge(x, xa) =
        if(x = [], xa,
           if(xa = [], x,
              if(hd(x) <= hd(xa), [] @ [hd(x)] @ merge(tl(x), xa),
                 [] @ [hd(xa)] @ merge(x, tl(xa)))))) &
    ((! x. even_pos(x) =
           if(x = [], x, if(tl(x) = [], [], [hd(x)] @ even_pos(tl(tl(x)))))) &
     (! x. odd_pos(x) =
           if(x = [], x,
              if(tl(x) = [], x, [hd(tl(x))] @ odd_pos(tl(tl(x))))))) &
    ?Q) -->
(! l. perm(l, mergesort(l)) & ordered(mergesort(l)))
 1. !!l. [| ~ (l = [] | Single(l));
            Def(! x. even_pos(x) =
                     if(x = [], x,
                        if(tl(x) = [], [],
                           [hd(x)] @ even_pos(tl(tl(x))))) |] ==>
         <even_pos(l), l> : ?r
 2. !!l. [| Def(! x. odd_pos(x) =
                     if(x = [], x,
                        if(tl(x) = [], x, [hd(tl(x))] @ odd_pos(tl(tl(x))))));
            ~ (l = [] | Single(l)) |] ==>
         <odd_pos(l), l> : ?r
 3. wf(?r)

Level 10
Def((! l. mergesort(l) =
          if(l = [] | Single(l), l,
             merge(mergesort(even_pos(l)), mergesort(odd_pos(l))))) &
    (! x xa.
        merge(x, xa) =
        if(x = [], xa,
           if(xa = [], x,
              if(hd(x) <= hd(xa), [] @ [hd(x)] @ merge(tl(x), xa),
```

```
                [] @ [hd(xa)] @ merge(x, tl(xa)))))) &
    ((! x. even_pos(x) =
          if(x = [], x, if(tl(x) = [], [], [hd(x)] @ even_pos(tl(tl(x)))))) &
     (! x. odd_pos(x) =
          if(x = [], x,
             if(tl(x) = [], x, [hd(tl(x))] @ odd_pos(tl(tl(x))))))) &
    ?Q) -->
(! l. perm(l, mergesort(l)) & ordered(mergesort(l)))
No subgoals!

val it =
  "[| !!l.
         ?mergesort(l) =
         if
          (l = [] | l = [hd(l)], l,
           ?merge(?mergesort(?even_pos(l)), ?mergesort(?odd_pos(l))));
      !!x xa.
         ?merge(x, xa) =
         if
          (x = [], xa,
           if
            (xa = [], x,
             if
              (hd(x) <= hd(xa), Cons(hd(x), ?merge(tl(x), xa)),
               Cons(hd(xa), ?merge(x, tl(xa))))));
      !!x.
         ?even_pos(x) =
         if(x = [], x, if(tl(x) = [], [], Cons(hd(x), ?even_pos(tl(tl(x))))));
      !!x.
         ?odd_pos(x) =
         if
          (x = [], x,
           if(tl(x) = [], x, Cons(hd(tl(x)), ?odd_pos(tl(tl(x)))))) |] ==>
   ! l. perm(l, ?mergesort(l)) & ordered(?mergesort(l))" : thm
```

B.5 The Insertsort derivation

```
goal thy "Def(?H)-->(ALL l.(perm(l, ?sort(l))& ordered(?sort(l))))";

                            (* INDUCTION *)

(*1*) BY induction_step ;

                            (* CASE SPLIT*)
(*2*) BY case_split_step;
                            (* l = [] *)

(*3*) br  ([perm_Nil,ord_Nil] MRS conjI) 1;

                            (* l ~= [] *)

(*4*) BY (((EVERY1 [rtac conjI ,
                    rtac perm_trans,
                    rtac ([perm_sym ,perm_sym] MRS perm_append_with_trans)])
           THEN (res_inst_tac [("P1","ordered(?z1)"),("Q1","ordered(?z2)")] (conjI RSN (2,mp)) 5))
           THEN (PRIMITIVE (fn state => (state COMP rule)))
```

```
           THEN (PRIMITIVE(fn state => (zero_var_indexes state))));

                    (* Synthesis of insertion *)
                       (* INDUCTION *)

(*5*) BY subprograms_step ;
(*6*) BY (EVERY [apply_induction_hyp 8,
                 apply_induction_hyp 9]);

              (**** TERMINATION OF INSSORT AND INSERTION ****)


(*7*)  by ((rtac wf_shorter  10) THEN (rtac wf_shorter 7));
(*8*)  by ( REPEAT ((simp_tac appl_ss 1) THEN (fast_tac (List_cs addEs [length_mono]) 1)));

finalize (topthm());



Level 0
Def(?H) --> (! l. perm(l, ?sort(l)) & ordered(?sort(l)))
 1. Def(?H) --> (! l. perm(l, ?sort(l)) & ordered(?sort(l)))

Level 1
Def((! l. inssort(l) = ?fbody3(l)) & ?Q2) -->
(! l. perm(l, inssort(l)) & ordered(inssort(l)))
 1. !!l. [| Def(?Q2);
            ! t. <t, l> : ?r3 -->
                 perm(t, inssort(t)) & ordered(inssort(t)) |] ==>
         perm(l, ?fbody3(l)) & ordered(?fbody3(l))
 2. [| Def(! l. inssort(l) = ?fbody3(l)); Def(?Q2) |] ==> wf(?r3)

Level 2
Def((! l. inssort(l) = if(l = [], ?case_T13(l, l), ?case_F7(l, hd(l), tl(l)))) &
    ?Q2) -->
(! l. perm(l, inssort(l)) & ordered(inssort(l)))
 1. !!l. [| ! t. <t, []> : ?r3 --> perm(t, inssort(t)) & ordered(inssort(t));
            Def(?Q2) |] ==>
         perm([], ?case_T14(l, [])) & ordered(?case_T13(l, []))
 2. !!l. [| Def(?Q2);
            ! t. <t, l> : ?r3 --> perm(t, inssort(t)) & ordered(inssort(t));
            l ~= [] |] ==>
         perm(l, ?case_F7(l, hd(l), tl(l))) & ordered(?case_F7(l, hd(l), tl(l)))
 3. [| Def(! l. inssort(l) =
                if(l = [], ?case_T13(l, l), ?case_F7(l, hd(l), tl(l))));
       Def(?Q2) |] ==>
    wf(?r3)

Flex-flex pairs:
%l. ?case_T14(l, l) =?= %l. ?case_T13(l, l)

Level 3
Def((! l. inssort(l) = if(l = [], l, ?case_F7(l, hd(l), tl(l)))) & ?Q2) -->
(! l. perm(l, inssort(l)) & ordered(inssort(l)))
 1. !!l. [| Def(?Q2);
            ! t. <t, l> : ?r3 --> perm(t, inssort(t)) & ordered(inssort(t));
            l ~= [] |] ==>
```

```
        perm(l, ?case_F7(l, hd(l), tl(l))) & ordered(?case_F7(l, hd(l), tl(l)))
 2. [| Def(! l. inssort(l) = if(l = [], l, ?case_F7(l, hd(l), tl(l))));
       Def(?Q2) |] ==>
    wf(?r3)

Level 4
Def((! l. inssort(l) = if(l = [], l, ?case_F(l, hd(l), tl(l)))) & ?Q) -->
(! l. perm(l, inssort(l)) & ordered(inssort(l)))
 1. !!l. [| Def(?Q);
            ! t. <t, l> : ?r --> perm(t, inssort(t)) & ordered(inssort(t));
            l ~= [] |] ==>
         perm(?l1.0(l) @ ?l2.0(l), ?case_F(l, hd(l), tl(l))) &
         (ordered(?z(l)) & ordered(?za(l)) -->
          ordered(?case_F(l, hd(l), tl(l))))
 2. !!l. [| Def(?Q);
            ! t. <t, l> : ?r --> perm(t, inssort(t)) & ordered(inssort(t));
            l ~= [] |] ==>
         perm(?l1.0(l), ?t1.0(l)) & ordered(?z(l))
 3. !!l. [| Def(?Q);
            ! t. <t, l> : ?r --> perm(t, inssort(t)) & ordered(inssort(t));
            l ~= [] |] ==>
         perm(?l2.0(l), ?t2.0(l)) & ordered(?za(l))
 4. !!l. [| Def(?Q);
            ! t. <t, l> : ?r --> perm(t, inssort(t)) & ordered(inssort(t));
            l ~= [] |] ==>
         perm(l, ?t1.0(l) @ ?t2.0(l))
 5. [| Def(! l. inssort(l) = if(l = [], l, ?case_F(l, hd(l), tl(l))));
       Def(?Q) |] ==>
    wf(?r)

Level 5
Def((! l. inssort(l) = if(l = [], l, insertion(hd(l), ?x9(l, hd(l), tl(l))))) &
    (! x xa.
        insertion(x, xa) =
        if(xa = [], Cons(x, xa),
           if(x <= hd(xa), [x] @ insertion(hd(xa), tl(xa)),
              [] @ [hd(xa)] @ insertion(x, tl(xa))))) &
    ?Q1) -->
(! l. perm(l, inssort(l)) & ordered(inssort(l)))
 1. !!l v x.
       [| l ~= []; Def(?Q1); x ~= []; v <= hd(x);
          <tl(x), x> : ?r8(l) -->
          ordered([hd(x)]) & ordered(tl(x)) -->
          ordered(insertion(hd(x), tl(x))) |] ==>
       <tl(x), x> : ?r8(l)
 2. !!l v x.
       [| l ~= []; Def(?Q1); x ~= []; v <= hd(x); ordered([v]); ordered(x);
          <tl(x), x> : ?r8(l) -->
          perm(Cons(hd(x), tl(x)), insertion(hd(x), tl(x))) |] ==>
       <tl(x), x> : ?r8(l)
 3. !!l v x.
       [| l ~= []; Def(?Q1); x ~= []; v <= hd(x); ordered([v]); ordered(x);
          <tl(x), x> : ?r8(l) -->
          ordered([hd(x)]) & ordered(tl(x)) -->
          ordered(insertion(hd(x), tl(x))) |] ==>
       <tl(x), x> : ?r8(l)
 4. !!l v x.
```

```
       [| l ~= []; Def(?Q1); x ~= []; ~ v <= hd(x);
          <tl(x), x> : ?r8(l) -->
          ordered([v]) & ordered(tl(x)) --> ordered(insertion(v, tl(x))) |] ==>
       <tl(x), x> : ?r8(l)
 5. !!l v x.
       [| l ~= []; Def(?Q1); x ~= []; ~ v <= hd(x); ordered(x);
          <tl(x), x> : ?r8(l) --> perm([v] @ tl(x), insertion(v, tl(x))) |] ==>
       <tl(x), x> : ?r8(l)
 6. !!l v x.
       [| l ~= []; Def(?Q1); x ~= []; ~ v <= hd(x); ordered(x);
          <tl(x), x> : ?r8(l) -->
          ordered([v]) & ordered(tl(x)) --> ordered(insertion(v, tl(x))) |] ==>
       <tl(x), x> : ?r8(l)
 7. !!l. [| ! t. <t, l> : ?r --> perm(t, inssort(t)) & ordered(inssort(t));
           l ~= [];
           Def(! x xa.
                  insertion(x, xa) =
                  if(xa = [], Cons(x, xa),
                     if(x <= hd(xa), [x] @ insertion(hd(xa), tl(xa)),
                        [] @ [hd(xa)] @ insertion(x, tl(xa)))));
           Def(?Q1) |] ==>
        wf(?r8(l))
 8. !!l. [| ! t. <t, l> : ?r --> perm(t, inssort(t)) & ordered(inssort(t));
           l ~= [];
           Def(! x xa.
                  insertion(x, xa) =
                  if(xa = [], Cons(x, xa),
                     if(x <= hd(xa), [x] @ insertion(hd(xa), tl(xa)),
                        [] @ [hd(xa)] @ insertion(x, tl(xa)))));
           Def(?Q1) |] ==>
        perm(tl(l), ?x9(l, hd(l), tl(l)))
 9. !!l. [| ! t. <t, l> : ?r --> perm(t, inssort(t)) & ordered(inssort(t));
           l ~= [];
           Def(! x xa.
                  insertion(x, xa) =
                  if(xa = [], Cons(x, xa),
                     if(x <= hd(xa), [x] @ insertion(hd(xa), tl(xa)),
                        [] @ [hd(xa)] @ insertion(x, tl(xa)))));
           Def(?Q1) |] ==>
        ordered(?x9(l, hd(l), tl(l)))
 10. [| Def(! l. inssort(l) =
                 if(l = [], l, insertion(hd(l), ?x9(l, hd(l), tl(l)))));
        Def(! x xa.
               insertion(x, xa) =
               if(xa = [], Cons(x, xa),
                  if(x <= hd(xa), [x] @ insertion(hd(xa), tl(xa)),
                     [] @ [hd(xa)] @ insertion(x, tl(xa)))));
        Def(?Q1) |] ==>
     wf(?r)

Level 6
Def((! l. inssort(l) = if(l = [], l, insertion(hd(l), inssort(tl(l))))) &
    (! x xa.
        insertion(x, xa) =
        if(xa = [], Cons(x, xa),
           if(x <= hd(xa), [x] @ insertion(hd(xa), tl(xa)),
              [] @ [hd(xa)] @ insertion(x, tl(xa))))) &
```

```
    ?Q1) -->
(! l. perm(l, inssort(l)) & ordered(inssort(l)))
 1. !!l v x.
       [| l ~= []; Def(?Q1); x ~= []; v <= hd(x);
          <tl(x), x> : ?r8(l) -->
          ordered([hd(x)]) & ordered(tl(x)) -->
          ordered(insertion(hd(x), tl(x))) |] ==>
       <tl(x), x> : ?r8(l)
 2. !!l v x.
       [| l ~= []; Def(?Q1); x ~= []; v <= hd(x); ordered([v]); ordered(x);
          <tl(x), x> : ?r8(l) -->
          perm(Cons(hd(x), tl(x)), insertion(hd(x), tl(x))) |] ==>
       <tl(x), x> : ?r8(l)
 3. !!l v x.
       [| l ~= []; Def(?Q1); x ~= []; v <= hd(x); ordered([v]); ordered(x);
          <tl(x), x> : ?r8(l) -->
          ordered([hd(x)]) & ordered(tl(x)) -->
          ordered(insertion(hd(x), tl(x))) |] ==>
       <tl(x), x> : ?r8(l)
 4. !!l v x.
       [| l ~= []; Def(?Q1); x ~= []; ~ v <= hd(x);
          <tl(x), x> : ?r8(l) -->
          ordered([v]) & ordered(tl(x)) --> ordered(insertion(v, tl(x))) |] ==>
       <tl(x), x> : ?r8(l)
 5. !!l v x.
       [| l ~= []; Def(?Q1); x ~= []; ~ v <= hd(x); ordered(x);
          <tl(x), x> : ?r8(l) --> perm([v] @ tl(x), insertion(v, tl(x))) |] ==>
       <tl(x), x> : ?r8(l)
 6. !!l v x.
       [| l ~= []; Def(?Q1); x ~= []; ~ v <= hd(x); ordered(x);
          <tl(x), x> : ?r8(l) -->
          ordered([v]) & ordered(tl(x)) --> ordered(insertion(v, tl(x))) |] ==>
       <tl(x), x> : ?r8(l)
 7. !!l. [| ! t. <t, l> : ?r --> perm(t, inssort(t)) & ordered(inssort(t));
           l ~= [];
           Def(! x xa.
                  insertion(x, xa) =
                  if(xa = [], Cons(x, xa),
                     if(x <= hd(xa), [x] @ insertion(hd(xa), tl(xa)),
                        [] @ [hd(xa)] @ insertion(x, tl(xa)))));
           Def(?Q1) |] ==>
        wf(?r8(l))
 8. !!l. [| l ~= [];
           Def(! x xa.
                  insertion(x, xa) =
                  if(xa = [], Cons(x, xa),
                     if(x <= hd(xa), [x] @ insertion(hd(xa), tl(xa)),
                        [] @ [hd(xa)] @ insertion(x, tl(xa)))));
           Def(?Q1); <tl(l), l> : ?r --> ordered(inssort(tl(l))) |] ==>
        <tl(l), l> : ?r
 9. !!l. [| l ~= [];
           Def(! x xa.
                  insertion(x, xa) =
                  if(xa = [], Cons(x, xa),
                     if(x <= hd(xa), [x] @ insertion(hd(xa), tl(xa)),
                        [] @ [hd(xa)] @ insertion(x, tl(xa)))));
           Def(?Q1); <tl(l), l> : ?r --> perm(tl(l), inssort(tl(l))) |] ==>
```

```
         <tl(l), l> : ?r
 10. [| Def(! l. inssort(l) = if(l = [], l, insertion(hd(l), inssort(tl(l)))));
        Def(! x xa.
               insertion(x, xa) =
               if(xa = [], Cons(x, xa),
                  if(x <= hd(xa), [x] @ insertion(hd(xa), tl(xa)),
                     [] @ [hd(xa)] @ insertion(x, tl(xa)))));
        Def(?Q1) |] ==>
     wf(?r)

Level 7
Def((! l. inssort(l) = if(l = [], l, insertion(hd(l), inssort(tl(l))))) &
    (! x xa.
        insertion(x, xa) =
        if(xa = [], Cons(x, xa),
           if(x <= hd(xa), [x] @ insertion(hd(xa), tl(xa)),
              [] @ [hd(xa)] @ insertion(x, tl(xa))))) &
    ?Q1) -->
(! l. perm(l, inssort(l)) & ordered(inssort(l)))
 1. !!l v x.
       [| l ~= []; Def(?Q1); x ~= []; v <= hd(x);
          <tl(x), x> : induced_rel(length, {q. ? x y. q = <x, y> & x < y}) -->
          ordered([hd(x)]) & ordered(tl(x)) -->
          ordered(insertion(hd(x), tl(x))) |] ==>
       <tl(x), x> : induced_rel(length, {q. ? x y. q = <x, y> & x < y})
 2. !!l v x.
       [| l ~= []; Def(?Q1); x ~= []; v <= hd(x); ordered([v]); ordered(x);
          <tl(x), x> : induced_rel(length, {q. ? x y. q = <x, y> & x < y}) -->
          perm(Cons(hd(x), tl(x)), insertion(hd(x), tl(x))) |] ==>
       <tl(x), x> : induced_rel(length, {q. ? x y. q = <x, y> & x < y})
 3. !!l v x.
       [| l ~= []; Def(?Q1); x ~= []; v <= hd(x); ordered([v]); ordered(x);
          <tl(x), x> : induced_rel(length, {q. ? x y. q = <x, y> & x < y}) -->
          ordered([hd(x)]) & ordered(tl(x)) -->
          ordered(insertion(hd(x), tl(x))) |] ==>
       <tl(x), x> : induced_rel(length, {q. ? x y. q = <x, y> & x < y})
 4. !!l v x.
       [| l ~= []; Def(?Q1); x ~= []; ~ v <= hd(x);
          <tl(x), x> : induced_rel(length, {q. ? x y. q = <x, y> & x < y}) -->
          ordered([v]) & ordered(tl(x)) --> ordered(insertion(v, tl(x))) |] ==>
       <tl(x), x> : induced_rel(length, {q. ? x y. q = <x, y> & x < y})
 5. !!l v x.
       [| l ~= []; Def(?Q1); x ~= []; ~ v <= hd(x); ordered(x);
          <tl(x), x> : induced_rel(length, {q. ? x y. q = <x, y> & x < y}) -->
          perm([v] @ tl(x), insertion(v, tl(x))) |] ==>
       <tl(x), x> : induced_rel(length, {q. ? x y. q = <x, y> & x < y})
 6. !!l v x.
       [| l ~= []; Def(?Q1); x ~= []; ~ v <= hd(x); ordered(x);
          <tl(x), x> : induced_rel(length, {q. ? x y. q = <x, y> & x < y}) -->
          ordered([v]) & ordered(tl(x)) --> ordered(insertion(v, tl(x))) |] ==>
       <tl(x), x> : induced_rel(length, {q. ? x y. q = <x, y> & x < y})
 7. !!l. [| l ~= [];
            Def(! x xa.
                   insertion(x, xa) =
                   if(xa = [], Cons(x, xa),
                      if(x <= hd(xa), [x] @ insertion(hd(xa), tl(xa)),
                         [] @ [hd(xa)] @ insertion(x, tl(xa)))));
```

```
            Def(?Q1);
            <tl(l), l> : induced_rel(length, {q. ? x y. q = <x, y> & x < y}) -->
            ordered(inssort(tl(l))) |] ==>
         <tl(l), l> : induced_rel(length, {q. ? x y. q = <x, y> & x < y})
 8. !!l. [| l ~= [];
            Def(! x xa.
                   insertion(x, xa) =
                   if(xa = [], Cons(x, xa),
                      if(x <= hd(xa), [x] @ insertion(hd(xa), tl(xa)),
                         [] @ [hd(xa)] @ insertion(x, tl(xa)))));
            Def(?Q1);
            <tl(l), l> : induced_rel(length, {q. ? x y. q = <x, y> & x < y}) -->
            perm(tl(l), inssort(tl(l))) |] ==>
         <tl(l), l> : induced_rel(length, {q. ? x y. q = <x, y> & x < y})

Level 8
Def((! l. inssort(l) = if(l = [], l, insertion(hd(l), inssort(tl(l))))) &
    (! x xa.
        insertion(x, xa) =
        if(xa = [], Cons(x, xa),
           if(x <= hd(xa), [x] @ insertion(hd(xa), tl(xa)),
              [] @ [hd(xa)] @ insertion(x, tl(xa))))) &
    ?Q1) -->
(! l. perm(l, inssort(l)) & ordered(inssort(l)))
No subgoals!

val it =
  "[| !!l. ?inssort(l) = if(l = [], l, ?insertion(hd(l), ?inssort(tl(l))));
      !!x xa.
         ?insertion(x, xa) =
         if
          (xa = [], Cons(x, xa),
           if
            (x <= hd(xa), Cons(x, ?insertion(hd(xa), tl(xa))),
             Cons(hd(xa), ?insertion(x, tl(xa))))) |] ==>
   ! l. perm(l, ?inssort(l)) & ordered(?inssort(l))" : thm
```

Bibliography

[1] P.B. Andrews. *An introduction to Mathematical Logic and type theory: To Truth Thought proof.* Acadimic press, 1986.

[2] L. Bachmair and H. Ganzinger. Rewrite techniques for transitive relations. In M.E. Stickel, editor, *Proc. 9th IEEE Symposium on Logic in Computer Science, 1994*. Springer-Verlag, 1994.

[3] David Basin. Isawhelk: Whelk interpreted in Isabelle. In *11th International Conference on Logic Programming (ICLP94)*, 1994. Paper is extended abstract. Full version available via anonymous ftp to mpi-sb.mpg.de in pub/papers/conferences/Basin-ICLP94.dvi.Z.

[4] S. Biundo, B. Hummel, D. Hutter, and C. Walther. The Karlsruhe induction theorem proving system. In Joerg Siekmann, editor, *8th Conference on Automated Deduction*, pages 672–674. Springer-Verlag, 1986. Springer Lecture Notes in Computer Science No. 230.

[5] R.S. Boyer and Yuan Yu. Automated correctness proofs of machine code programs for a commercial microprocessor. In D. Kapur, editor, *11th Conference on Automated Deduction*, pages 416–430, Saratoga Springs, NY, USA, June 1992. Published as Springer Lecture Notes in Artificial Intelligence, No 607.

[6] M. Broy. A case study in program developement: sorting. Technical Report TUM-INFO-11-78-31-350, Technische Universität München, Institut für Informatik, November 1978.

[7] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. Research Paper 567, Dept. of Artificial Intelligence, Edinburgh, 1991. In the Journal of Artificial Intelligence.

[8] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. Research Paper 507, Dept. of Artificial Intelligence, Edinburgh, 1990. Appeared in the proceedings of CADE-10.

[9] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M.E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.

[10] A. Church. A formulation of the simple theory of types. *Symbolic Logic*, 5(1):56–68, 1940.

[11] K.L. Clark and J. Darlington. Algorithm classification through synthesis. *The Computer Journal*, 23(1):61–65, 1980.

[12] R.L. Constable, S.F. Allen, H.M. Bromley, et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, 1986.

[13] Th. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[14] Nardie Daniel. Formal synthesis of a unification algorithm by the deductive-tableau method. *The Journal of Logic Programming*, 1989.

[15] Clark, Darlington. Algorith classification through synthesis. *Computer J.*, 23(1):61–65, 1979.

[16] J. Darlington. A synthesis of several sorting algorithms. *Acta Informatica*, 11:1–30, 1978.

[17] John Darlington. A synthesis of several programs. *Acta Inform.*, 11(1):1–3, 1978.

[18] N. Dershowitz. Termination of rewriting. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications*, pages 69–116. Academic Press, 1987.

[19] Y. Deville. *Logic programming: systematic program development.* Addison-Wesley Pub. Co., 1990.

[20] Melvin Fitting. *First-Order Logic and Automated Theorem Proving.* Springer-Verlag, New York, 1990.

[21] G. Gentzen. *The Collected Papers of Gerhard Gentzen.* North Holland, 1969. edited by Szabo, M.E.

[22] M.J. Gordon, A.J. Milner, and C.P. Wadsworth. *Edinburgh LCF - A mechanised logic of computation*, volume 78 of *Lecture Notes in Computer Science.* Springer Verlag, 1979.

[23] Cordell Green. Application of theorem proving to problem solving. In *Proceedings of the IJCAI-69*, pages 219–239, 1969.

[24] M. Heisel, W. Reif, and W. Stephan. Implementing verification strategies in the KIV-system. In *9th International Conference On Automated Deduction*, pages 131–140, Argonne, Illinois, 1988.

[25] J. Herbrand. Researches in the theory of demonstration. In J van Heijenoort, editor, *From Frege to Goedel: a source book in Mathematical Logic, 1879-1931*, pages 525–81. Harvard University Press, Cambridge, Mass, 1930.

[26] C.J. Hogger. Goal-directed derivation of logic programs. In *Proc. 7th Symp. on the Mathematical Foundations of Computer Science*, Polish Academy of Sciences, Zakopane, 1978.

[27] C.J. Hogger. Derivation of logic programs. *JACM*, 28(2):372–392, April 1981.

[28] W.A. Howard. The formulae-as-types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry; Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[29] D. Hutter. Guiding inductive proofs. In M.E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 147–161. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449.

[30] I. Kraan, D. Basin, and A. Bundy. Logic program synthesis via proof planning. In K.K. Lau and T. Clement, editors, *Logic Program Synthesis and Transformation*, pages 1–14. Springer-Verlag, 1993. Also available as Max-Planck-Institut für Informatik Report MPI-I-92-244 and Edinburgh DAI Research Report 603.

[31] I. Kraan, D. Basin, and A. Bundy. Middle-out reasoning for logic program synthesis. In D. S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*. MIT Press, 1993. Also available as Max-Planck-Institut für Informatik Report MPI-I-93-214 and Edinburgh DAI Research Report 638.

[32] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[33] Z. Manna and R. Waldinger. Deductive synthesis of the unification algorithm. *Science of Computer Programming*, pages 5–48, 1981.

[34] Z. Manna and R. Waldinger. *The Logical Basis for Computer Programming, Vol 1: Deductive Reasoning*. Addison Wesley, Reading, Mass, 1985.

[35] Z. Manna and R. Waldinger. *The Logical Basis for Computer Programming, Vol 2: Deductive Reasoning*. Addison Wesley, Reading, Mass, 1985.

[36] Z. Manna and R. Waldinger. The origin of a binary-search paradigm. *Science of Computer Programming*, 9:37–83, 1987.

[37] Z. Manna and R. Waldinger. *The deductive foundations of computer programming: a one-volume version of "The logical basis for computer programming"*. Addison Wesley, Reading, Mass, 1990.

[38] Z. Manna and R. Waldinger. *The deductive foundations of computer programming: a one-volume version of "The logical basis for computer programming"*. Addision-Wesely, 1990.

[39] Z. Manna and R. Waldinger. Fundamentals of the deductive program synthesis. *IEEE Transactions on Software Engineering*, January 1992.

[40] Per Martin-Löf. Constructive mathematics and computer programming. In *6th International Congress for Logic, Methodology and Philosophy of Science*, pages 153–175, Hanover, August 1979. Published by North Holland, Amsterdam. 1982.

[41] T. Nipkow and C. Prehofer. Type checking type classes. In *Proc. 20th ACM Symp. Principles of Programming Languages*, pages 409–418, 1993.

[42] B. Nordström. In *In informal Proceedings of the Nijmegen Workshop on types for Proofs and Programs*. The ALF editor, 1993.

[43] Simon Peyton Jones Paul Hudak and Philip Walder. Report on the programming language haskell: A non-strict, purely functional language. *ACM SIGPLAN Notices*, 27(5), 1992.

[44] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989.

[45] Lawrence C. Paulson. The Isabelle reference manual. Technical report, Cambridge University Computer Laboratory, Cambridge, 1993.

[46] Lawrence C. Paulson. *Isabelle : a generic theorem prover; with contributions by Tobias Nipkow*. LNCS-828. Springer, Berlin, 1994.

[47] Martin Protzen. Lazy generation of induction hypotheses. In Alan Bundy, editor, *12th International Conference on Automated Deduction*, pages 42–56. Springer Verlag, 1994. Lecture Notes in Computer Science No. 87.

[48] G. Robinson and L. Wos. Paramodulation and theorem-proving in first-order theories with equality. In D. Michie, editor, *Machine Intelligence 4*, pages 103–33. Edinburgh University Press, 1969.

[49] J.A. Robinson. A machine oriented logic based on the resolution principle. *J Assoc. Comput. Mach.*, 12:23–41, 1965.

[50] T. Sato and H Tamaki. Transformational logic programming synthesis. FGCS, 1984.

[51] D. R. Smith. The top-down synthesis of divide and conquer algorithms. *AI*, 27(1), 1985.

[52] Raymond M. Smullyan. *First Order Logic*. Springer-Verlag, Berlin, 1968.

[53] Jonathan Traugott. Deductive synthesis of sorting programs. *J. Symbolic Computation*, 7:533–572, 1989.

[54] C. Walther. Argument-bounded algorithms as a basis for automated termination proofs. In R. Lusk and R. Overbeek, editors, *9th Conference on Automated Deduction*, pages 602–621. Springer-Verlag, 1988.

[55] C. Walther. *Automatisierung von Terminierungsbeweisen.* Vieweg Verlag, Braunschweig, 1991.

[56] C. Walther. Mathematical induction. In S.C. Shapiro, editor, *Encyclopedia of Artificial Intelligence.* John Wiley and Sons, New York, 1991.

[57] C. Walther. Mechanizing mathematical induction. In C.J. Hogger B.M. Gabbay and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming.* Oxford University Press, Oxford, 1992. in press.

[58] G. A. Wiggins. Synthesis and transformation of logic programs in the Whelk proof development system. In K. R. Apt, editor, *Proceedings of JICSLP-92*, pages 351–368. M.I.T. Press, Cambridge, MA, 1992.