J. Symbolic Computation (2001) **11**, 1–32

A Higher-Order Interpretation of Deductive Tableau

Abdelwaheb Ayari David Basin

Institut für Informatik, Albert-Ludwigs-Universität Freiburg,

Georges-Köhler-Allee 52, 79110 Freiburg i. Br., Germany.

Phone: (49) (761) 203-8240 Fax: 203-8242

{ayari,basin}@informatik.uni-freiburg.de

(Received 28 March 2001)

The Deductive Tableau of Manna and Waldinger is a formal system with an associated methodology for synthesizing functional programs by existence proofs in classical first-order theories. We reinterpret the formal system in a setting that is higher-order in two respects: higher-order logic is used to formalize a theory of functional programs and higher-order resolution is used to synthesize programs during proof. Their synthesis methodology can be applied in our setting as well as new methodologies that take advantage of these higher-order features.

The reinterpretation gives us a framework for directly formalizing and implementing the Deductive Tableau system in standard theorem provers that support the kinds of higher-order reasoning listed above. We demonstrate this, as well as a new development methodology, within a conservative extension of higher-order logic in the Isabelle system. We report too on a case-study in synthesizing sorting programs.

1. Introduction

Program synthesis is the task of formally deriving programs from specifications. Unlike in traditional program development, where a program is first written and later (if ever) validated, program synthesis builds programs that meet their specification by design: a program is developed from its specification in a stepwise manner and each step guarantees the correctness of the final design. Over the last few decades, a variety of methodologies have been proposed for building programs from specifications, for example the use of constructive logics (Bates and Constable, 1985; Coquand and Huet, 1988) and the many approaches to deductive based synthesis and transformation, e.g., (Bibel, 1980; Bird, 1989; Burstall and Darlington, 1977), to name but a few.

In this paper we consider a particular formal system, with an associated methodology for deductive synthesis, proposed by Manna and Waldinger (1992). The formal system, called the *Deductive Tableau*, is a novel system for synthesizing functional programs. It is based on a two-dimensional tabular structure for reasoning about first-order formulae. Proofs consist of a series of table extensions using proof rules based on nonclausal resolution, induction, and equality reasoning. There is a loose resemblance to standard tableau and sequent systems in that the table records the "polarity" of formulae, that is

0747–7171/90/000000 + 00 $03.00/0

which formulae are goals and hypotheses in the current proof. However there are significant differences; in particular in *output columns* the table records, and the proof rules manipulate, *output terms*, which comprise parts of the synthesized program. Based on this system, Manna and Waldinger provide a methodology that supports the synthesis of functional programs from specifications.

Our goal is to give a new account of the Deductive Tableau that we believe is generally useful for analyzing, implementing, and extending program development methodologies. Our account is based on reinterpreting the Deductive Tableau in a setting that is higher-order in two separate ways: *Higher-order logic* is the logical foundation of our work and *higher-order resolution* supports a program synthesis methodology. These are independent and offer different advantages. Higher-order logic (HOL) is an expressive logic well-suited for reasoning about functional programs of the kind developed with the Deductive Tableau methodology. We use higher-order logic to formalize and analyze well-founded relations when proving that synthesized functions terminate. Higher-order resolution is a powerful way to solve for unknowns during proof. We use higher-order metavariables to model output columns and higher-order resolution to simulate the construction of output terms. Moreover, we show how higher-order resolution can be used to extend the Deductive Synthesis methodology in new directions.

We actually carry out two closely related interpretations. The first, given in Section 4, is based on an embedding: we represent tableaux as HOL formulae and simulate Deductive Tableau derivations in HOL. This embedding is faithful to the Deductive Tableau in the sense that it can be used to mimic Deductive Tableau proofs. Although this is conceptually interesting, e.g., it provides a declarative interpretation of tableaux as formulae in a (higher-order) metalogic and uses higher-order resolution as a mechanism to explain program synthesis, it is too inflexible for realistic use and inherits the weaknesses of the Deductive Tableau (described in Section 2.5). Hence, in Section 5, we give a second interpretation, which is the main one of interest in this paper. It represents a new program development methodology, which is complementary to, and in the spirit of, the Deductive Tableau approach.

We have implemented both of our interpretations in the ISABELLE system (Paulson, 1994b), which is a logical framework that supports an implementation of higher-order logic (ISABELLE/HOL) and builds natural deduction proofs using higher-order resolution. To illustrate our interpretations we work through an example due to Manna and Waldinger and report on a case study in synthesizing sorting programs.

Research Contributions

We show how the Deductive Tableau can be interpreted in a higher-order setting and that this offers both conceptual and practical advantages. Although our focus is on this one particular formal system and methodology, we believe that our approach, which is based on casting the system and development methodology within a more general framework, has general applicability and is relevant to research on other program development methodologies. Below we expand on these points.

First, our interpretations lead to a new understanding of the Deductive Tableau. Formally, we stratify the Deductive Tableau into three parts: first, a *logical foundation*, in our case ISABELLE/HOL, conservatively extended with theories of well-founded relations and standard recursive data types (such as lists and numbers); second, *derived rules*, which correspond to axioms and proof rules in the Deductive Tableau; and third, *tactics*, which simulate Deductive Tableau proof steps and automate simple kinds of reasoning. Methodologically, we show how metavariables and higher-order resolution can be used to explain program synthesis in the setting of the Deductive Tableau. Taken together, this provides a new conceptual basis for understanding the Deductive Tableau as well as other approaches to synthesis, which may, at first glance, appear rather different.

Second, the above stratification provides a simple means for establishing logical properties, like correctness, of Deductive Tableau program development and our extensions. Our logical foundation, HOL, has a standard semantics (Gordon and Melham, 1993). Since rules for simulating the Deductive Tableau are formally derived on top of this, they are also guaranteed to be sound. Finally, since tactics are programs that justify their proof steps with primitive or derived inference rules, it follows that any program developed by simulating the Deductive Tableau methodology, or using any formally developed extension of it, satisfies its specification.

Although it would be possible to reinterpret the Deductive Tableau in a first-order setting, higher-order logic allows us to prove more properties of our synthesized programs. In particular, program development requires appropriate data types (strings, numbers, etc.) and axiomatizations of them (e.g., term constructors are injective, inductive definitions have associated well-founded relations, etc.). The use of higher-order logic means that we do not need to work with a fixed collections of types and axioms whose correctness is understood outside the logic. Instead, we use ISABELLE's inductive data type package to extend our developments with new data types and their properties; by the design of the data type package (Paulson, 1994a) these extensions are conservative over HOL.

Third, the general setting of higher-order logic and synthesis by higher-order resolution makes it easy to extend the Deductive Tableau methodology and develop and experiment with new approaches to program synthesis. We explore this in the context of our second interpretation, described in Section 5. For example, we show that resolution can be used to separate program construction from reasoning about termination and that proving termination can also be approached as a synthesis problem.

Finally, our work contributes to an understanding of how one can simply and effectively implement deductive synthesis methodologies like the Deductive Tableau using generic theorem proving environments. Although there have been many methodologies proposed for program synthesis only few have seen their way into robust systems. The majority have only been implemented as prototypes or, as is more often the case, the subject of feasibility studies carried out using paper and pencil development (e.g., the Deductive Tableau proofs in Manna and Waldinger's original papers). This is understandable: considerable work is required to turn a development methodology into a computer supported development tool. A theorem proving environment is required that supports the logic underlying the methodology and manages proof construction. Moreover, non-trivial development requires support for structured theories, automated reasoning, and the like. The differences between methodologies typically makes it impossible to reuse existing systems and building a robust development system from scratch is hard work.

Instead of building an implementation from scratch, we show how to use a generic theorem prover like ISABELLE in a straightforward manner. Construction of our initial ISABELLE theory for the Deductive Tableau, which includes definitions, derivation of rules, and supporting tactics, took only a few days. Central to this was the fact that we could directly use the distributed ISABELLE/HOL theory and standard ISABELLE tactics such as rewriters and simplifiers. Afterwards, we could directly apply our development to interesting problems; we have carried out proofs in Isabelle of many of Manna and

Waldinger's published examples, and, in particular, we have synthesized a variety of standard sorting algorithms. Full proofs scripts may be found in (Ayari, 1995).

2. Deductive Tableau

Our account of the Deductive Tableau is based on Manna and Waldinger (1992), with additional input from (Manna and Waldinger, 1980; 1981; 1985; 1993).

As explained in the introduction, the Deductive Tableau can be understood as a formal system with an associated methodology for program development. A program is specified by formalizing its behavior as a $\forall/\exists$-formula in first-order logic, i.e., a formula with an initial prefix of universally followed by existentially quantified variables. The universally quantified variables in the prefix denote program inputs and each existentially quantified variable denotes a function of the inputs that computes a value for which the matrix of the formula is true. The specification is proven by applying proof rules and the desired programs are synthesized as a "by-product" during the proof.

A tableau proof is constructed with respect to some fixed first-order theory. The programs synthesized are functional programs in a simple language determined by the theory. In particular, let $\overline{x}$ denote a vector of distinct variables. Then a program $f(\overline{x})$ is defined by an equality $f(\overline{x}) = t$, where the free variables of t belong to $\overline{x}$. The term t is built in the usual first-order way from variables, constants and functions declared in the theory. To allow for recursive programs, t may also be built from f. To allow for programs with conditionals, there is an additional term construct if-then-else, where $\mathsf{if}\ \phi\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2$ is a term provided ϕ is a formula and t_1 and t_2 are terms. Since not every first-order term built in this way is executable, Manna and Waldinger introduce the notion of *primitive expressions*. The primitive expressions are quantifier free and consist of constants, functions, and predicate symbols from a *primitive set* that we know how to execute. This set depends on the theory in question; for example, in deriving a program to multiply numbers in the theory of integers, the primitive set may include constants like 0, function symbols like $+$, and predicate symbols like equality and inequality.

The deductive system is based on a tableau, which is a two-dimensional array. Each row contains a formula, which is either an *assertion* or a *goal*, and possibly first-order terms that are *output entries*. For example, a tableau consisting of two rows, an assertion A and a goal G, has the following form.

Assertions	*Goals*	*Outputs*		
		$f_1(\overline{x})$	$\ldots$	$f_n(\overline{x})$
A		t_1	$\ldots$	t_n
	G	s_1	$\ldots$	s_n

The function symbols f_i name the desired programs and the terms t_i and s_i, the output entries, are primitive expressions that comprise parts of the synthesized programs. Intuitively, one reads a Deductive Tableau as stating that the conjunction of all the assertions implies the disjunction of all the goals for the programs f_i. These programs are defined by tableaux of a particular form (final tableaux), described below.

A tableau proof starts with a specification of the form $\forall\,\overline{x}.\,\exists\,\overline{y}.\,Q(\overline{x},\overline{y})$. This states that for each vector of inputs $\overline{x}$ there are outputs $\overline{y}$ that satisfy the *input-output relation* $Q(\overline{x}, \overline{y})$. Said another way, if $\overline{y}$ is an n-tuple, then Q specifies the behavior of n functions $f_1, \dots, f_n$ of $\overline{x}$, i.e., $Q(\overline{x}, f_1(\overline{x}), \dots, f_n(\overline{x}))$. The specification is turned into an initial tableau by Skolemization, which reflects the above interpretation: each $x_i \in \overline{x}$ is turned into an eigenvariable a_i and each $y_i \in \overline{y}$ is computed by a function $f_i(\overline{a})$. Hence the initial tableau is as follows.

Assertions	*Goals*	*Outputs*		
		$f_1(\overline{a})$	$\dots$	$f_n(\overline{a})$
	$Q(\overline{a}, y_1, \dots, y_n)$	y_1	$\dots$	y_n

To simplify notation below, we restrict our discussion to tableaux arising from specifications where $\overline{x}$ and $\overline{y}$ are each single variables, i.e., we are synthesizing a single function f of one argument a.

A Deductive Tableau proof is constructed by applying rules that augment the tableau with new rows and may incrementally instantiate output variables. A tableau proof consists of a sequence of tableaux with associated proof rules. A proof is completed when a *final tableau* is generated; this is when the truth value false (respectively true) appears as a column assertion (respectively goal), i.e.,

false		t

(or, respectively,

	true	t

).

If t is the term in the output column of the final row, then the desired program is $f(a) = t$. Since the proof rules have conditions that guarantee that t is a primitive expression, f is an executable program.

Before we describe the main proof rules of the Deductive Tableau, we make some remarks that aid their understanding. A theory has an associated collection of sentences that are axioms. Any axiom may be added as an assertion to a tableau at any time during a proof. The distinction between assertions and goals reflects the usual distinction between antecedent and succedent often made in sequent systems. Consistent with this reading, a duality principle holds where any goal can be moved into the assertion column by negating it and vice versa. Moreover, free variables occurring in goals (respectively assertions) have existential (respectively universal) meaning. For example, a goal $\phi(x)$ says that ϕ holds for some x. Thus we can rename free variables of rows in a validity preserving way. Finally it is not required that each tableau row has an output entry. If a row has no output entry, then we can introduce a fresh variable as an output entry thereby obtaining a logically equivalent row. Thus, if the goal (respectively assertion) of a row without an output entry for a function f can be proven true (respectively false) then any program for f satisfies the input specification.

2.1. Nonclausal Resolution

Formulae in different rows of a tableau are combined using nonclausal resolution. This corresponds to a case analysis with simplification and introduces a conditional term in the output entries. Nonclausal resolution requires two rows and generates a new one. There are four variants of this rule depending on whether formulae come from goal or assertion columns and, due to the duality principle, these four rules are equivalent.

As an example, consider the following two rows:

$A(P)$		s
	$G(Q)$	t

The notation $A(P)$ denotes the assertion A with a subformula P and similarly $G(Q)$ denotes the goal G with a subformula Q. Nonclausal resolution, when applied to these rows, generates the new row

	$\neg A\theta(\mathsf{false}) \wedge G\theta(\mathsf{true})$	if $P\theta$ then $t\theta$ else $s\theta$

where θ is the most general unifier of P and Q. In addition, to ensure that the output entry is executable, the rule can only be applied when $P\theta$ (the application of the substitution θ to P) belongs to the set of primitive expressions.

To simplify proofs, a few optimizations are incorporated into this rule. First, the goal $\neg A\theta(\mathsf{false}) \wedge G\theta(\mathsf{true})$ is simplified before it is introduced in the tableau. Second, the conditional term built can, in some cases, also be simplified. In particular, if $s\theta$ and $t\theta$ are identical, then the conditional is simplified to $s\theta$. Moreover, consistent with the meaning of a missing output entry (where any term suffices), when one of the resolved rows is missing an output entry, then, instead of the conditional, we obtain the output entry of the other row. Finally, if both resolved rows have no output entries, then no output entry is given for the result.

There are two specialization of nonclausal resolution that resolve a row with itself and decompose the formula there into its subformulae. These are the rules *And-split* and *Or-split*. The rule *And-split* decomposes an assertion $A \wedge B$ into the two new assertions A and B; similarly, the rule *Or-split* decomposes a goal $A \vee B$ into the two new goals A and B. Based on the duality principle, using *Or-split* we can derive another splitting rule, *Imp-split*, which decomposes a goal $A \longrightarrow B$ into the assertion A and the goal B.

2.2. Equality

Besides the reflexivity axiom, which can be introduced as assertion, there is an equality rule that serves as a nonclausal version of the paramodulation rule used in resolution theorem proving. Like nonclausal resolution, there are four equivalent forms of the equality rule, depending on whether formulae come from goal or assertion columns.

As an example, consider the two rows:

$A(l = r)$		s
	$G(l')$	t

If l and l' are unifiable with θ, then we generate the new row:

	$\neg A\theta(\mathsf{false}) \wedge G\theta(r\theta)$	if $(l\theta = r\theta)$ then $t\theta$ else $s\theta$

In the first conjunct, all occurrences of the equality $l = r$ are replaced by false; in the second, some occurrences of l' are replaced by $r\theta$. Note that, in the same cases as for the nonclausal resolution rules, application of equality rules do not build conditional outputs.

2.3. Induction Rule

Induction is used to develop recursive programs. Suppose our goal is $Q(a, y)$, where y is the output entry of some function f, and a is an arbitrary element. To prove the goal by induction, we may introduce an *induction hypothesis* that states that $f(x)$ satisfies the specification Q, for all x less than a with respect to some well-founded ordering $<$.[1] The induction rule introduces this induction hypothesis as an assertion in the tableau.

$x < a \longrightarrow Q(x, f(x))$		

The ordering $<$ must be selected by the user from those well-founded ordering that are defined in the current theory, e.g., *less-than* over the natural numbers. Note that the induction hypothesis contains occurrences of the function symbol f, which is the function that we are trying to synthesize. When the induction hypothesis is used in the proof, a recursive call, $f(t)$, appears in the output entries.

2.4. The Front/Last Example

We take Manna & Waldinger's front/last example (Manna and Waldinger, 1980) as a running example. This example is simple to understand and illustrates both the Deductive Tableau methodology as well as its limitations. Later in Section 5, we return to this example to illustrate our approach and show how we overcome these limitations.

The front/last example builds upon a theory of strings. We assume that this theory formalizes the following functions and predicates. Strings are built from the empty string, Λ, and the 'cons' function, where $c.s$ is the string whose first element is the character c and remaining elements are given by the string s. For a nonempty string s, $\mathsf{head}(s)$ denotes the first character and $\mathsf{tail}(s)$ denotes the rest. The function $*$ concatenates two strings. Finally, the predicate $\mathsf{char}(s)$ is true when the string s is built from a single cons, i.e., contains a single character.

Consider synthesizing two functions, both operating on non-empty strings. The first, $\mathsf{front}(s)$, returns all but the last character of s. The second, $\mathsf{last}(s)$, returns the last character of s. These functions are specified by y_1 and y_2 in the first-order formula

$$\forall\, x.\, \exists\, y_1\, y_2.\, x \neq \Lambda \longrightarrow \mathsf{char}(y_2) \wedge x = y_1 * y_2 \,.$$

2.4.1. Derivation

The Initial Tableau

We skolemize the input specification to get an initial tableau: x is replaced by the constant a and y_1 and y_2 are placed in the output columns as the initial definitions of the functions $\mathsf{front}(a)$ and $\mathsf{last}(a)$. Thus, our initial tableau has the form:

[1] An ordering $\leq$ is a binary relation that is reflexive, antisymmetric and transitive. Its strict (irreflexive) part, $<$, is well-founded when there does not exist an infinite sequence of terms $t_1, t_2, \ldots$ where $t_{i+1} < t_i$.

Row Nr.	Assertions	Goals	Outputs	
			$\mathsf{front}(a)$	$\mathsf{last}(a)$
1		$a \neq \Lambda \longrightarrow \mathsf{char}(y_2) \wedge a = y_1 * y_2$	y_1	y_2

Induction

Guided by our intuition that front and last will be recursive functions, we apply the induction rule, proposing the proper substring order, $<_s$, as the well-founded order. This adds a new row with the induction hypothesis as an assertion.

2	$x <_s a \longrightarrow x \neq \Lambda \longrightarrow \mathsf{char}(\mathsf{last}(x)) \wedge x = \mathsf{front}(x) * \mathsf{last}(x)$			

Continuing the derivation, we simplify the goal in row 1 by applying the rule *Imp-split*. This yields the assumption $a \neq \Lambda$ and the goal $\mathsf{char}(y_2) \wedge a = y_1 * y_2$.

3	$a \neq \Lambda$			
4		$\mathsf{char}(y_2) \wedge a = y_1 * y_2$	y_1	y_2

Case analysis

Since a is a non-empty string (row 3), we can distinguish between two possible cases: a is either a single character or consists of more than one character. We begin this case analysis by introducing the following assertion from the string theory.

5	$\Lambda * w = w$			

We now apply the equality rule to goal 4 and the above assertion by unifying the terms $\Lambda * w$ and $y_1 * y_2$. This results in the unifier $\{y_1 \leftarrow \Lambda,\ w \leftarrow y_2\}$ and a new row:

6		$\mathsf{char}(y_2) \wedge a = y_2$	Λ	y_2

Goal 6 can be simplified to $\mathsf{char}(a)$ by instantiating the variable y_2 to the string a. Formally, we achieve this by introducing the reflexivity axiom $u = u$ as an assumption

7	$u = u$			

and applying resolution to this and the subformula $a = y_2$ in goal 6.

The tableau reached at this stage states that if a is a character then $\mathsf{front}(a)$ is the empty string Λ and $\mathsf{last}(a)$ is a.

8		$\mathsf{char}(a)$	Λ	a

In the second case, a consists of more than one character. Before applying the induction hypothesis, we must first massage goal 4 to a formula that will unify with it. We can do this using the concatenation axiom for strings.

9	$(u.z_1) * z_2 = u.(z_1 * z_2)$			

Now we apply the equality rule on rows 4 and 9, unifying $y_1 * y_2$ and $(u.z_1) * z_2$.

10		$\mathsf{char}(z_2) \wedge a = u.(z_1 * z_2)$	$u.z_1$	z_2

At this point the induction hypothesis can be applied. First, we apply the equality rule to row 2 and row 10 on the equality $x = \mathsf{front}(x) * \mathsf{last}(x)$ and the term $z_1 * z_2$. This unifies the term $\mathsf{front}(x) * \mathsf{last}(x)$ with $z_1 * z_2$, with the unifier $\theta = \{z_1 \leftarrow \mathsf{front}(x), z_2 \leftarrow \mathsf{last}(x)\}$. This generates the new goal

$$(\neg(x <_s a \longrightarrow x \neq \Lambda \longrightarrow (\mathsf{char}(\mathsf{last}(x)) \wedge \mathsf{false})) \wedge (\mathsf{char}(z_2) \wedge a = u.x))\theta,$$

which is simplified to:

11		$(x <_s a) \wedge (x \neq \Lambda) \wedge \mathsf{char}(\mathsf{last}(x)) \wedge a = u.x$	$u.\mathsf{front}(x)$	$\mathsf{last}(x)$

The derivation continues by using the induction hypothesis again. We apply the nonclausal resolution rule to row 2 and row 11.

12		$(x <_s a) \wedge (x \neq \Lambda) \wedge a = u.x$	$u.\mathsf{front}(x)$	$\mathsf{last}(x)$

Final steps

The rest of the derivation consists of simplifying the goal in row 12 to true using resolution rules and axioms from the theory of strings. The variables x and u are instantiated with $\mathsf{tail}(a)$ and $\mathsf{head}(a)$ respectively. The penultimate row of the proof is:

13		$\neg\mathsf{char}(a)$	$\mathsf{head}(a).\mathsf{front}(\mathsf{tail}(a))$	$\mathsf{last}(\mathsf{tail}(a))$

Finally, we apply nonclausal resolution on row 13 and row 8.

14		true	$\mathsf{if}\ \mathsf{char}(a)\ \mathsf{then}\ \Lambda$ $\mathsf{else}\ \mathsf{head}(a).\mathsf{front}(\mathsf{tail}(a))$	$\mathsf{if}\ \mathsf{char}(a)\ \mathsf{then}\ a$ $\mathsf{else}\ \mathsf{last}(\mathsf{tail}(a))$

Since the goal of this row is true, we are done. We have synthesized the programs:

$$\begin{aligned} \mathsf{front}(a) &= \mathsf{if}\ \mathsf{char}(a)\ \mathsf{then}\ \Lambda\ \mathsf{else}\ \mathsf{head}(a).\mathsf{front}(\mathsf{tail}(a)) \\ \mathsf{last}(a) &= \mathsf{if}\ \mathsf{char}(a)\ \mathsf{then}\ a\ \mathsf{else}\ \mathsf{last}(\mathsf{tail}(a)) \end{aligned}$$

2.5. Summary and Limitations

The above derivation illustrates the main ideas of the Deductive Tableau. A program is specified by a $\forall/\exists$-formula, which serves as the basis of the initial tableau. The user interactively applies proof rules to build a sequence of tableaux until a final tableau is reached, at which point the programs are given by the row's output entries.

Overall, the Deductive Tableau system is conceptually simple. Only induction and nonclausal resolution are essential; the remaining rules are derivable from these two. These rules formalize the essence of algorithmic development and its proof theoretic counterpart: case analysis on decidable predicates and recursion on well-founded orderings.

There are some drawbacks though in the methodology employed for program synthesis that stem from limitations of the formal system. Namely, proofs must be constructed in limited and not always natural ways, and not all properties relevant for reasoning about programs can be formalized in the system itself. Below we expand on this and use the above example for illustration. These limitations motivate the interpretation and methodological extensions that we propose in Section 5.

First, although the deductive system is conceptually simple, Deductive Tableau proofs themselves are based on resolution, which makes them difficult to construct and understand. Resolution produces a large search space and (in the nonclausal case) it is difficult for the user to identify which subexpressions are important for the proof. The resolution rule is generally used as an inference rule for low-level machine automated theorem proving, not human assisted interactive theorem proving. In almost all interactive theorem provers, proofs are instead built using natural deduction or sequent systems, where proofs are constructed in a top-down (goal directed) fashion.

Although the Deductive Tableau resembles standard tableaux, it is not possible to derive rules equivalent to the standard ones. Instead, nonclausal resolution can only be used to derive splitting rules that do not introduce branching, that is rules like *And-split* and *Or-split*. These can decompose conjunctions of assertions and disjunctions of goals, but other possibilities are excluded, e.g., conjunctions of goals or disjunctions of assertions. Methodologically, users of the Deductive Tableau must work around this limitation by using nonclausal resolution to operate under conjunctive goals or disjunctive assumptions by resolving with their subformulae, e.g., the transformation from row 10 to 13 in the above proof. Hence, rather then physically branching, the branches are conjunctively (or disjunctively) joined together in formulae.

This limitation is a direct consequence of the structure of Deductive Tableau proofs. Proofs are sequences instead of trees. Hence rules that require splitting a tableau into multiple subtableaux are not allowed. This limitation arises due to the way programs are built: splitting is not compatible with propagating subprograms in output entries. Suppose, for example, that we have a tableau with a row containing the conjunctive goal $G_1 \wedge G_2$ and output entry t. If we were to split the tableau, as is done conventionally in first-order theorem proving, then the result would be two new tableaux where the first contains the goal G_1 and the second G_2. Consistent with the interpretation that t is a program that satisfies both conjuncts, t_1 and t_2 must both be the same as t. If we perform splitting in a top-down proof then we have not yet constructed t_1, t_2, and t and splitting would require a new mechanism for keeping track of the constraint that all three must be identical (or propagating changes from one to the others). As will become apparent in Section 5, the use of metavariables, where instantiations are globally propagated, provides precisely this mechanism.

The induction rule in itself is fairly natural, but it must be used to develop programs in a bottom-up way and this makes it difficult to guide or follow proof development. Suppose we wish to develop a simple recursive program

$$f(a) = \mathsf{if}\ P\ \mathsf{then}\ \mathit{base\_case}\ \mathsf{else}\ \mathit{step\_case}\,.$$

To derive this, we must first derive the subprograms *base_case* and *step_case* and afterwards use resolution to build the conditional term defining f. Such bottom-up construction can be rather subtle. For example, both subprograms can stem from a common row that is used twice in two complementary contexts. In the front/last example, row 4 is used to build the output entry Λ of row 8 and the output entry $\mathsf{head}(a).\mathsf{front}(\mathsf{tail}(a))$ of row 13. These two entries are assembled in the conditional term that forms the body of front. Furthermore, it is not easy to recognize that the proof step producing row 11 from row 2 and row 10 corresponds to the application of the induction hypothesis. Similar to the above "splitting problem", we will show in Section 5 how metavariables can be used to remedy this shortcoming by standing-in for not yet developed subprograms.

The above problems are of a practical nature, but do not represent theoretical limitations. One such limitation arises from using first-order logic: the theories formalized are too weak to express and reason about the well-foundedness of orders. For example, in the induction proof above, we used the order $<_s$. However, any proof that this order is well-founded must take place outside the system. In general, reasoning about well-foundedness is difficult. Indeed, it is as difficult as reasoning about termination, which is undecidable. As we will see later, the use of higher-order logic not only allows users to state and prove the well-foundedness of their orders, but to synthesize orders in the same manner in which programs themselves are synthesized.

3. Isabelle

As noted in the introduction, we reinterpret the Deductive Tableau in a setting that implements higher-order logic and supports higher-order unification. We have chosen the ISABELLE logical framework for this task and in this section we briefly introduce notation and concepts important for our work. A general account of ISABELLE is given by Paulson (1994b).

3.1. Theories, Rules, and Resolution

ISABELLE is a logical framework that provides a metalogic in which object logics (e.g., first-order logic, set theory, etc.) are encoded and reasoned about. ISABELLE's metalogic is based on the universal/implicational fragment of a polymorphically typed higher-order logic. Universal quantification in the metalogic is represented by $\bigwedge$ and implication by $\Longrightarrow$. Object logics are encoded in ISABELLE's metalogic by declaring a theory, which consists of a signature and a set of axioms. For example, in formalizing higher-order logic as an object logic, the signature formalizes connectives like $\wedge$ of type $\mathsf{bool} \times \mathsf{bool} \to \mathsf{bool}$ and $\forall$ of type $(\alpha \to \mathsf{bool}) \to \mathsf{bool}$. Here bool is the type of higher-order logic formulae and the type of $\forall$ is polymorphic (α is a type-variable ranging over all HOL types). Proof rules are formulae in the metalogic such as

$$\bigwedge A : \mathsf{bool}.\ \bigwedge B : \mathsf{bool}.\ A \Longrightarrow (B \Longrightarrow (A \wedge B)) \qquad (\wedge\text{-}I)$$

$$\bigwedge P : \alpha \to \mathsf{bool}.\ (\bigwedge a : \alpha.\ P(a)) \Longrightarrow \forall\, x.\ P(x) \qquad (\forall\text{-}I)$$

which formalize natural deduction rules for $\wedge$-I (and introduction) and $\forall$-I (all introduction). The rule ($\forall$-I) can be understood as replacing $\forall$ quantification in the object logic with $\bigwedge$ quantification in the metalogic, which effectively transforms the universally quantified variable x to an eigenvariable a from the perspective of the metalogic.

Certain conventions are standard. Outermost (meta-)quantifiers and type annotation are usually omitted and iterated implication is written in a more readable list notation, e.g., $[\![A; B]\!] \Longrightarrow A \wedge B$. After a proof rule is axiomatized, or derived, the outermost (either explicitly or implicitly) meta-quantified variables are subsequently treated as metavariables that can be instantiated by higher-order resolution, discussed below. Isabelle displays such metavariables as symbols proceeded by "?".

Proof construction in Isabelle proceeds by higher-order resolution. Suppose we have two metaformulae of the form

$$[\![\psi_1; \ldots ; \psi_m]\!] \Longrightarrow \psi \tag{3.1}$$

$$[\![\phi_1; \ldots ; \phi_n]\!] \Longrightarrow \phi \tag{3.2}$$

where ϕ higher-order unifies with ψ_i. Then, if θ is the unifier, the result is

$$[\![\psi_1\theta; \ldots ; \psi_{i-1}\theta; \phi_1\theta; \ldots ; \phi_n\theta; \psi_{i+1}\theta; \ldots ; \psi_m\theta]\!] \Longrightarrow \psi\theta\,.$$

There are several ways to read metaformula like (3.1). First, (3.1) can be interpreted as an intuitionistic sequent where the consequent, ψ, is the goal to be proven and the antecedent, $[\![\psi_1; \ldots ; \psi_m]\!]$, lists the hypotheses or assumptions. Second, (3.1) can be interpreted as a proof-state, where the consequent is the goal to be proven and the antecedent lists the subgoals needed to establish it. Isabelle's resolution tactics work in a way that maintains these views. For example, if we want to prove a theorem ψ, then we start the proof with the initial proof-state (trivially a derived rule) $\psi \Longrightarrow \psi$, which means that to establish the goal ψ we must prove one subgoal, namely ψ. Proof proceeds then by resolving with the subgoals of the proof-state and the final proof-state *is itself* the desired theorem ψ.

Let us give a simple example that illustrates these different views and also some of the subtleties involved. Suppose we are given the initial proof-state

$$(\forall\, x.\, P(x) \wedge Q(x)) \Longrightarrow \forall\, x.\, P(x) \wedge Q(x)\,.$$

We can resolve this with the $\forall$-I rule yielding

$$(\bigwedge\, a.\, P(a) \wedge Q(a)) \Longrightarrow \forall\, x.\, P(x) \wedge Q(x)\,.$$

In the context of theorem proving we might summarize this by saying that our goal is to prove $\forall\, x.\, P(x) \wedge Q(x)$ and this goal itself is our first (and only) subgoal. After one resolution step, this subgoal is transformed to $\bigwedge\, a.\, P(a) \wedge Q(a)$, i.e., the universally quantified variable x is converted to the eigenvariable a. Now we might like to continue the derivation by resolving the new subgoal with the $\wedge$-I rule. Unfortunately resolution cannot unify this subgoal with the goal of $\wedge$-I due to the meta-quantification $\bigwedge\, a$. However, a slight variant of it can be, namely

$$[\![\bigwedge\, a.\, A(a); \bigwedge\, a.\, B(a)]\!] \Longrightarrow \bigwedge\, a.\, A(a) \wedge B(a)\,. \tag{$\wedge$-I_a}$$

(Note that A and B are implicitly outermost quantified with types $\alpha \longrightarrow \mathsf{bool}$ and a is of type α.) Resolving with this rule yields two subgoals: $\bigwedge\, a.\, A(a)$ and $\bigwedge\, a.\, B(a)$. That is, we have derived the proof-state

$$[\![\bigwedge a.\, P(a);\ \bigwedge a.\, Q(a)]\!] \Longrightarrow \forall x.\, P(x) \wedge Q(x)\,.$$

This example illustrates a common problem: A rule may be applied in contexts with arbitrary numbers of eigenvariables. Rather forcing the user to manually derive a variant rule, one for each number of parameters, ISABELLE provides a uniform mechanism for doing this. The transformation of $\wedge$-I to $\wedge$-I_a is performed automatically by ISABELLE and is called *lifting*: the rule $\wedge$-I is "lifted" over the eigenvariable a.

A similar problem occurs when resolving with a subgoal G that has an assumption H (i.e., is of the form $H \Longrightarrow G'$). Since rules in ISABELLE have atomic conclusions, no rule can unify with this (meta-)implication. To get round this problem, ISABELLE provides a second kind of lifting: a rule (3.1) is transformed to

$$[\![H \Longrightarrow \psi_1;\ \ldots\ ;\ H \Longrightarrow \psi_m]\!] \Longrightarrow H \Longrightarrow \psi$$

by lifting over the assumption (or, more generally, assumptions) H.

3.2. ISABELLE'S HIGHER-ORDER LOGIC

The ISABELLE distribution comes with an encoding of high-order logic, ISABELLE/HOL. This logic is based on three primitives: implication ($\longrightarrow$), equality ($=$), and Hilbert's description-operator "ϵ". All other constants (e.g., True and False) and operators (e.g., $\forall$) are defined in terms of these primitives and their proof rules are formally derived. ISABELLE supports a hierarchical extension of HOL with theories and the distribution comes with a collection of them, some of which we used in our work. Among the most important ones were theories of ordered types, sets, least fixedpoints and inductive definitions, and well-founded relations. We also employed particular theories of inductively defined data types including polymorphic lists and natural numbers.

4. Embedding the Deductive Tableau in Isabelle

In this section we show how to use ISABELLE to implement and experiment with the Deductive Tableau. One way to do this would be to exploit the fact that ISABELLE is a logical framework and implement a specialized logic and specialized machinery in it for the Deductive Tableau. An alternative approach is to embed the Deductive Tableau in a standard logic in ISABELLE and use standard means of proof construction to synthesize programs in a way that simulates Deductive Tableau proofs. We take this approach based on embedding and simulation. It shows that higher-order resolution is sufficient to account for program construction and it is a starting point for exploring the advantages of using higher-order logic. Exploring these advantages and more liberal development methods is the topic of Section 5.

For the remainder of this section we will restrict our attention to tableaux arising from specifications with a single universally and existentially quantified variable in the prefix, i.e., $\forall x.\, \exists y.\, Q(x, y)$. Such formulae specify a program of one input, $f(x) = t$, where $\forall x.\, Q(x, f(x))$. This restriction can easily be lifted at the cost of additional notation.

4.1. EMBEDDING

In our embedding we will synthesize functions f instead of terms (that constitute the bodies of functions) as in the Deductive Tableau. To do this, free first-order variables of the Deductive Tableau are translated into second-order variables in our encoding. Hence, we begin by defining this translation: for t a term, and a a variable, let $\uparrow^a t$ be the term built by replacing all free variables x with $?x(a)$ and let $\uparrow^a \phi$, for formulae ϕ, be defined similarly.

Now, a tableau consists of a number of rows and each row consists of either an assertion or a goal, and a single output column. We encode rows using one of two binary predicates, A and G, each of type $\mathsf{bool} \times \mathsf{bool} \to \mathsf{bool}$, where $\mathsf{A}(\phi, \psi)$ and $\mathsf{G}(\phi, \psi)$ are defined as $\psi \longrightarrow \phi$ and $\psi \longrightarrow \neg\phi$ respectively. Specifically, we define an encoding function $\lceil\cdot\rceil$ from rows to formulae where a row r with the assertion (respectively the goal) ϕ and output term t is encoded by the predicate $\lceil r\rceil$ defined as $\mathsf{A}(\uparrow^a \phi, f(a) = \uparrow^a t)$ (respectively $\mathsf{G}(\uparrow^a \phi, f(a) = \uparrow^a t)$). If a row has no output term then we simplify the second argument of A (respectively G) to True. This reflects the fact that we can introduce any term for the missing output term t and, in particular, we introduce the term $f(a)$ itself.

Given this representation of rows as atomic formulae, we now represent Tableaux as formulae built from these atoms. To accomplish this, we extend $\lceil\cdot\rceil$ to map tableaux to ISABELLE metaformulae, i.e., we map Deductive Tableau proof-states to ISABELLE proof-states. Suppose that in a Deductive Tableau proof of $\forall\, x.\, \exists\, y.\, Q(x, y)$ we have a tableau $\mathcal{T}$ with rows $r_1, \ldots, r_n$. Then $\lceil\mathcal{T}\rceil$ is

$$(\bigwedge a.\, [\![\, \lceil r_1\rceil; \ldots ; \lceil r_n\rceil\,]\!] \Longrightarrow \mathsf{False}) \Longrightarrow \forall\, x.\, Q(x, f(x))\,. \tag{4.1}$$

The antecedent of (4.1) encodes the rows of $\mathcal{T}$ and the consequent encodes the specification. Viewed as a proof state, the antecedent constitutes the single subgoal necessary to establish the specification. In the following simulation, the consequent always stay the same and each simulation step augments the antecedent with the encoding of an additional row.

4.2. SIMULATION

The above gives us a mapping between Deductive Tableau proof-states and ISABELLE proof-states. We now show that there is a mapping between rules of the Deductive Tableau and derived rules in ISABELLE and that this can be used to simulate Deductive Tableau derivations in ISABELLE/HOL.

To begin with, we will continue by overloading our function $\lceil\cdot\rceil$ so that rules of the Deductive Tableau are mapped to derived rules of ISABELLE. The essential tableau rules are nonclausal resolution and induction and these are mapped under $\lceil\cdot\rceil$ to the rules NCR and IND in Figure 1. Note that NCR encodes just the assertion-goal form of nonclausal resolution rule (the other three kinds are similar). A minor difference between NCR and its Tableau counterpart appears in the way they are applied. The former is applied in a backward way, whereas the latter is applied in forward way. Thus, NCR can be read as follows: in order to prove the goal ϕ, we prove the six subgoals (the six assumptions listed in NCR). The first two ones simulate the two rows on which the non-clausal resolution will be applied. The third subgoal ensures that the predicates P and Q are unifiable. The fourth (respectively fifth) subgoal represents the gaol (respectively output term) of the row that will be generated. Finally, the sixth subgoal simulates the row that will be generated.

Also, in the induction rule, we include the premise $\mathsf{wf}(r)$, which formalizes that the relation r used in the induction is well-founded. We have also given the rule NCR_NoOutput,

$$
\begin{array}{ll}
\mathsf{NCR} & [\![\ \mathsf{A}(A(P), f(a) = t); \\
& \quad \mathsf{G}(G(Q), f(a) = s); \\
& \quad P = Q; \\
& \quad \mathit{NewGoal} = \neg A(\mathsf{False}) \wedge G(\mathsf{True}); \\
& \quad \mathit{NewFbody} = \mathsf{if}\ P\ \mathsf{then}\ s\ \mathsf{else}\ t; \\
& \quad \mathsf{G}(\mathit{NewGoal}, f(a) = \mathit{NewFbody}) \Longrightarrow \phi \\
&]\!] \Longrightarrow \phi \\
\mathsf{IND} & [\![\ \mathsf{wf}(r); \\
& \quad \bigwedge a.\ [\![\ \forall t.\ \mathsf{G}(G(a,t), f(a) = t); \\
& \qquad\qquad \forall x.\ \mathsf{A}((x,a) \in r \longrightarrow G(x, f\,x), \mathsf{True})\ ]\!] \Longrightarrow \mathsf{False} \\
&]\!] \Longrightarrow G(a, f\,a) \\
\mathsf{NCR\_NoOutput} & [\![\ \mathsf{A}(A(P), \mathsf{True}); \\
& \quad \mathsf{G}(G(Q), f(a) = t); \\
& \quad P = Q; \\
& \quad \mathit{NewGoal} = \neg A(\mathsf{False}) \wedge G(\mathsf{True}); \\
& \quad \mathsf{G}(\mathit{NewGoal}, f(a) = t) \Longrightarrow \phi \\
&]\!] \Longrightarrow \phi
\end{array}
$$

Figure 1. Some Derived Tableau Rules in ISABELLE

which specializes NCR for the case where the output term is missing in the assertion. Each of these rules has been formally derived in ISABELLE/HOL in a straightforward way.

We say that we apply a tableau rule R to a tableau $\mathcal{T}$, and we write $\mathcal{T} \vdash_R \mathcal{T}'$, if we can apply R to two rows (or, in some cases only one row) of $\mathcal{T}$ generating a new row that, together with $\mathcal{T}$, constitutes the tableau $\mathcal{T}'$. We define a *tableau derivation of $\mathcal{T}_n$ from $\mathcal{T}_0$* to be a finite sequence $\mathcal{T}_0 \vdash_{R_0} \mathcal{T}_1 \ \ldots\ \mathcal{T}_{n-1} \vdash_{R_{n-1}} \mathcal{T}_n$. Analogously, we can apply an ISABELLE rule R to an ISABELLE proof-state $\mathcal{P}$, and we write $\mathcal{P} \vdash_R \mathcal{P}'$, if we can resolve $\mathcal{P}$ with R producing the proof-state $\mathcal{P}'$. Moreover, an ISABELLE derivation of $\mathcal{P}_n$ from $\mathcal{P}_0$ is a finite sequence $\mathcal{P}_0 \vdash_{R_0} \mathcal{P}_1 \ \ldots\ \mathcal{P}_{n-1} \vdash_{R_{n-1}} \mathcal{P}_n$.

We begin by observing that any step in a tableau derivation can be simulated by an ISABELLE derivation.

LEMMA 4.1. *Let $\mathcal{T}$ and $\mathcal{T}'$ be two tableaux and R a tableau proof rule. If $\mathcal{T} \vdash_R \mathcal{T}'$ then $\lceil \mathcal{T} \rceil \vdash_{\lceil R \rceil} \lceil \mathcal{T}' \rceil$.*

The proof of this lemma is given in Appendix A. Here, we show its application by simulating a step of the above front/last tableau derivation. The subtableau containing rows 1 to 7 is encoded by the ISABELLE proof-state (the conclusion is suppressed)

$$
\begin{array}{lll}
\bigwedge a.\ [\![& \qquad\qquad\qquad \vdots & \\
& \mathsf{G}(\mathsf{char}(?y_2(a)) \wedge a = ?y_2(a),\ \mathsf{front}(a) = \Lambda \wedge \mathsf{last}(a) = ?y_2(a)) & \text{(row 6)} \\
& \mathsf{A}(?u(a) = ?u(a),\ \mathsf{True}); & \text{(row 7)} \\
\quad]\!] & \Longrightarrow \mathsf{False}\,. &
\end{array}
$$

As we have seen in the tableau derivation, row 8 is obtained by applying the assertion-goal form of nonclausal resolution rule to rows 6 and 7. Applying the ISABELLE rule NCR_NoOutput of Figure 1 mimics this application. We resolve the above proof-state with

$\mathsf{NCR\_NoOutput}$ producing the new proof-state, where the additional formula encodes row 8 in the Deductive Tableau proof.

$$
\begin{array}{lll}
\bigwedge a.\, [\![& \vdots & \\
& \mathsf{G}(\mathsf{char}(?y_2(a)) \land a = ?y_2(a),\ \mathsf{front}(a) = \Lambda \land \mathsf{last}(a) = ?y_2(a)) & \text{(row 6)} \\
& \mathsf{A}(?u(a) = ?u(a),\ \mathsf{True}); & \text{(row 7)} \\
& \mathsf{G}(\mathsf{char}(a),\ \mathsf{front}(a) = \Lambda \land \mathsf{last}(a) = a) & \text{(row 8)} \\
]\!] & \Longrightarrow \mathsf{False}\,. &
\end{array}
$$

Call a tableau a *final tableau with output t* if it contains an assertion false (or goal true) with the output term t. Call an ISABELLE proof-state *a final* ISABELLE *proof-state with output t* if it is of the form $(\bigwedge a.\, f(a) = t) \Longrightarrow \forall x.\, Q(x, f(x))$.

We can now establish:

THEOREM 4.1. *Let $\mathcal{T}_n$ be derivable from $\mathcal{T}_0$, where $\mathcal{T}_0$ is an initial Tableau and $\mathcal{T}_n$ is a final tableau with output t. Then we can simulate the Tableau derivation in* ISABELLE *and derive a final* ISABELLE *proof-state with output t.*

PROOF. If $\mathcal{T}_0$ is an initial tableau then it contains a single row whose goal is $Q(a, y)$ and output term is y. So $\lceil \mathcal{T}_0 \rceil$ is

$$(\bigwedge a.\, \mathsf{G}(Q(a, ?y(a)), f(a) = ?y(a)) \Longrightarrow \mathsf{False}) \Longrightarrow \forall x.\, Q(x, f(x))\,. \tag{4.2}$$

From the definition of G, the antecedent

$$\bigwedge a.\, \mathsf{G}(Q(a, ?y(a)), f(a) = ?y(a)) \Longrightarrow \mathsf{False}$$

simplifies to $\bigwedge a.\, Q(a, ?y(a)) \land (f(a) = ?y(a))$, which implies $\bigwedge a.\, Q(a, f(a))$. Since $\bigwedge a.\, Q(a, f(a)) \Longrightarrow \forall x.\, Q(x, f(x))$ is trivially derivable in ISABELLE, so is $\lceil \mathcal{T}_0 \rceil$.

Now, by Lemma 4.1 it follows by induction on the derivation of $\mathcal{T}_n$ from $\mathcal{T}_0$ that $\lceil \mathcal{T}_n \rceil$ is also derivable from $\lceil \mathcal{T}_0 \rceil$, and hence is itself derivable. Since $\mathcal{T}_n$ is a final tableau with output t, it must have a row r_i with an assertion false (or goal true) and output t. In this case, the antecedent of $\lceil \mathcal{T}_n \rceil$ simplifies to the program definition $\bigwedge a.\, f(a) = t$. Hence from $\lceil \mathcal{T}_n \rceil$ we can derive a final ISABELLE proof-state with output t. □

4.3. SUMMARY

Our embedding provides an explanation of the Deductive Tableau in terms of derived rules within a standard logic. This gives us a formal correctness guarantee, established within ISABELLE/HOL. Practically, the embedding gives us a simple implementation that allows us to experiment with Deductive Tableau proofs and use the machinery of ISABELLE, e.g., tactics for rewriting and simplification, to partially automate their construction.

A similar embedding is possible within first-order logic. But higher-order logic offers the advantage that we can formally reason in it about well-founded orderings, and thus the termination of synthesized programs. This addresses one of the limitations discussed in Section 2.5. However, the other limitations have not yet been addressed. Doing this requires new ways of constructing proofs that are in the spirit of the Deductive Tableau but more appropriate for a natural deduction setting. This is the topic of the next section.

5. Program Construction in Isabelle

In Section 2.5 we pointed out that the Deductive Tableau has several limitations that stem from its underlying formal system. In the previous section we gave our first interpretation, where, since proofs are faithfully modeled in ISABELLE, these limitations remain. In this section we give a second interpretation. By exploiting the advantages of a richer formal system and proof by higher-order resolution, we show how it is possible to develop programs in new ways that are in the spirit of the Deductive Tableau methodology, but avoid these drawbacks.

The main idea of our approach is as follows. The logical foundation is a conservative extension of ISABELLE/HOL with inductively defined datatypes and a theory of well-founded relations. A program is specified in this logic. But, rather than turning the initial specification into a tableau, we operate on it directly. The existentially quantified variables are replaced by metavariables standing in for functions. The Deductive Tableau proof rules for nonclausal resolution and induction are recast as case analysis and induction rules. Our development methodology then consists, as in the Deductive Tableau, of using these rules to transform specifications in a way that reveals their algorithmic content. Moreover, the standard (natural deduction) proof rules of higher-order logic can also be used. For example, we can use them to perform goal/assertion decomposition: both those kinds that are possible in the Deductive Tableau and those that are impossible due to branching restrictions. Overall, proof construction is top-down and programs are incrementally synthesized during proof construction.

In the following we provide more details. To illustrate our approach and draw comparisons with the Deductive Tableau we develop again the front/last programs.

5.1. Input Specification

The formula $\forall x.\, \exists y.\, Q(x,y)$ specifies a program f that describes the relationship between its input x and output y $(= f(x))$. In our setting we formalize the functional relationship between x and y directly as $\forall x.\, Q(x, ?f(x))$. The function f is not given up front; instead we use the metavariable $?f$ as a place-holder for f, which is incrementally instantiated during the proof by higher-order resolution.

In the Deductive Tableau, output entries are used to record the definition of f. In our approach, we employ an analogous mechanism to record in the proof state possible recursive definitions. To this end, our initial goal is the implication

$$?H \longrightarrow \forall x.\, Q(x, ?f(x))\,. \tag{5.1}$$

The metavariable $?H$ will record the (possibly recursive) definition of f; we will explain how this works in Sections 5.3 and 5.4.

5.2. Case Analysis

In the Deductive Tableau, nonclausal resolution corresponds to a case analysis on a subformula C and builds a conditional term $\mathsf{if}\, C\, \mathsf{then}\, s\, \mathsf{else}\, t$. In our setting, we formalize this directly as case analysis. Specifically, the conditional term $\mathsf{if}\, C\, \mathsf{then}\, s\, \mathsf{else}\, t$ can be defined in HOL using Hilbert's ϵ-description operator, $\epsilon z.\, (C \longrightarrow z = s \wedge \neg C \longrightarrow z = t)$. This term is equal to s when the formula C holds and to t otherwise. We can derive the following rule, which splits on the case C and its negation $\neg C$.

$$[\![C \Longrightarrow P(s);\ \neg C \Longrightarrow P(t)]\!] \Longrightarrow P(\mathsf{if}\ C\ \mathsf{then}\ s\ \mathsf{else}\ t) \tag{5.2}$$

This proof rule states that we can prove P(if C then s else t) when we can prove $P(s)$ under the assumption C, and $P(t)$ under the assumption $\neg C$. Its derivation is simple and consists of expanding the term if C then s else t using the definition of if-then-else.

Recall that free variables (C, P, s, and t) are implicitly universally meta-quantified and that after derivation they are treated as metavariables that can be instantiated by resolution. When used for program synthesis, application of the case analysis rule constructs a program containing an if-then-else statement and the metavariables stand in for the condition and branches. This supports top-down program synthesis. Suppose we are synthesizing a program $?f$ specified by $Q(x, ?f(x))$. Application of the above proof rule by higher-order resolution lifts the proof rule over x and unifies $?f(x)$ with $\mathsf{if}\ ?C(x)\ \mathsf{then}\ ?s(x)\ \mathsf{else}\ ?t(x)$. We then obtain the subgoals $?C(x) \Longrightarrow Q(x, ?s(x))$ and $\neg ?C(x) \Longrightarrow Q(x, ?t(x))$. In proving these subgoals, instances of $?C$, $?s$, and $?t$ are synthesized and this in turn instantiates $?f$. As with the Deductive Tableau, the unconstrained use of resolution can instantiate $?C$ with an undecidable predicate (or a decidable one that we don't know how to execute). Therefore, analogous to the Deductive Tableau, the user must restrict the case analysis rule to instantiate $?C$ only with decidable predicates whose names belong to a set of executable (primitive) expressions.

This rule, which we call case_split_1 is general in that it applies for any predicates P and C, and terms s and t. However, in one sense it is not general enough: it can only be resolved with specifications of a single function f. For specifications of $n > 1$ functions, the conclusion $?P$ of (5.2) should contains n conditional terms. For example, the derived rule case_split_2,

$$[\![C \Longrightarrow P(s_1, s_2); \neg C \Longrightarrow P(t_1, t_2)]\!] \Longrightarrow P(\mathsf{if}\ C\ \mathsf{then}\ s_1\ \mathsf{else}\ t_1,\ \mathsf{if}\ C\ \mathsf{then}\ s_2\ \mathsf{else}\ t_2)\,, \tag{5.3}$$

can be used when we want to synthesize two functions, like in the front/last example. In essence, a program development rule, like case analysis or induction, corresponds to a family of development rules, differing in the number of functions synthesized. This is an unfortunate limitation of synthesis by higher-order resolution: rules are expressed as metaformula and this sometimes leads to rules that are syntactically too specialized. Said another way, there are kinds of generality that cannot be expressed as syntactic schemata formalized using (higher-order) metavariables. Our solution to this problem is simply to derive instances of the rule family as needed.

5.3. Induction Rule

Induction in the Deductive Tableau is based on induction along a well-founded relation. What constitutes a well-founded relation is external to the Deductive Tableau, but we can formalize this in Isabelle/hol. There are many equivalent definitions; we choose the following:

$$\mathsf{wf}(r) \equiv \forall\, P.\, (\forall\, x.\, (\forall\, y.\, (y, x) \in r \longrightarrow P(y)) \longrightarrow P(x)) \longrightarrow (\forall\, x.\, P(x))\,.$$

Note that r is an arbitrary binary relation. When r is additionally an ordering (which it will be in our examples) then $(y, x) \in r$ states that y is less than x in the ordering r. It is an easy (Isabelle) exercise to show that this is equivalent to other formalizations such as any nonempty set has a minimal element under r. From this definition, we can derive the following rule for well-founded induction:

$$[\![\forall x.\,(\forall y.\,(y,x) \in r \longrightarrow P(y)) \longrightarrow P(x);\ \mathsf{wf}(r)\,]\!] \Longrightarrow \forall x.\,P(x)$$

This is identical to standard rules for well-founded induction, except that we explicitly state that r is well-founded.

In order to synthesize functions, induction must be slightly specialized to reason about function specifications. In the Deductive Tableau (see Section 2.3) induction allows us to reason about a specification $Q(a,y)$ by adding an induction hypothesis

$$x < a \longrightarrow Q(x, f(x))\,.$$

In our setting, we generalize $<$ to an arbitrary well-founded relation r (so $x < a$ becomes $(x,a) \in r$) and make explicit the connection between the function f and the output term (in our setting given by a function $?f_{body}$). The result is

$$\begin{array}{ll} [\![& \forall x.\,f(x) = ?f_{body}(x); \\ & \forall a.\,(\forall x.\,(x,a) \in r \longrightarrow Q(x, f(x))) \longrightarrow Q(a, ?f_{body}(a)); \\ & \mathsf{wf}(r) \\ ]\!] & \Longrightarrow \forall x.\,Q(x, f(x))\,. \end{array}$$

This rule, which we name induction_schema_1, is easily derived from well-founded induction. During proof, resolution with this rule produces three subgoals. The first is $\forall x.\,?f(x) = ?f_{body}(x)$. This formula plays the role of the output column, where $?f_{body}(x)$ the output term. Our strategy is to discharge this subgoal by assuming it; we do this by instantiating the metavariable $?H$ given in the starting specification (5.1), whose role is precisely to collect such equalities. (This will become clearer in the example below.) After, by using the induction hypothesis, given by the second subgoal, $?f_{body}$ will be instantiated with recursive calls to $?f$. The third subgoal enforces that arguments to recursive calls are smaller in some well-founded relation $?r$.

Note that we need not prove in advance that a relation r is well-founded. We are free to discharge the second and third subgoal in any order. For example, we can delay proving the third subgoal and synthesize a relation during the proof of the second, and the relation need not be previously formalized. This amounts to first showing the partial correctness of a function and afterwards establishing termination. This possibility has also been suggested by Manna and Waldinger, but is not actually possible in their system.

To discharge the third subgoal we require means to build and reason about well-founded relations. The ISABELLE/HOL theory of well-founded relations provides a basis for this by proving operators for building well-founded relations from known well-founded relations. For example, we will later use the fact (proven in ISABELLE) that given two well-founded relations r_1 and r_2, their lexicographic combination, defined by

$$\mathsf{lex\_prod\_rel}(r_1, r_2) \equiv \{((a,b),(a',b')) \mid (a,a') \in r_1 \lor (a = a' \land (b,b') \in r_2)\}\,,$$

is also well-founded. Alternatively, a function f and a well-founded relation r induce a well-founded relation,

$$\mathsf{induced\_rel}(f, r) \equiv \{(x,y) \mid (f(x), f(y)) \in r\}\,.$$

This can be specialized by instantiating r with known well-founded relations like less over the natural numbers, i.e.,

$$\mathsf{measure}(f) \equiv \mathsf{induced\_rel}(f, \mathsf{less})\,.$$

5.4. Example: Front/Last

To aid comparison with the Deductive Tableau, we return to the now familiar front/last example. We begin our Isabelle proof by entering the following goal.

$?H \longrightarrow \forall x.\, x \neq \Lambda \longrightarrow \mathsf{char}(?y_2(x)) \wedge x = ?y_1(x) * ?y_2(x)$

Isabelle responds with the initial proof-state: the first line is the goal to be proven, and the remaining lines (here, only one) list the subgoals needed to establish it.

$?H \longrightarrow \forall x.\, x \neq \Lambda \longrightarrow \mathsf{char}(?y_2(x)) \wedge x = ?y_1(x) * ?y_2(x)$

1. $?H \longrightarrow \forall x.\, x \neq \Lambda \longrightarrow \mathsf{char}(?y_2(x)) \wedge x = ?y_1(x) * ?y_2(x)$

We proceed by successively resolving the above subgoal with several standard Isabelle/HOL rules: implication introduction, and all introduction (from Section 3.1).

$$\mathsf{By\ (Rtac\ [impI,\ allI]}\ 1);$$

This step[1] shifts the metavariable $?H$ over to the assumption side and turns the universally quantified x into a (universally meta-quantified) eigenvariable a.

$?H \longrightarrow \forall x.\, x \neq \Lambda \longrightarrow \mathsf{char}(?y_2(x)) \wedge x = ?y_1(x) * ?y_2(x)$

1. $\bigwedge a.\, ?H \Longrightarrow a \neq \Lambda \longrightarrow \mathsf{char}(?y_2(a)) \wedge a = ?y_1(a) * ?y_2(a)$

Next, we perform induction on a. We use a variant of the previous induction schema induction_schema_1 for synthesizing two functions, and we specify names for the functions.

$$\mathsf{By\ (INDTAC\ induction\_schema\_2}\ [(\text{"}f_1\text{"}, \text{"front"}), (\text{"}f_2\text{"}, \text{"last"})]\ 1);$$

In addition to resolving with the induction rule, this tactic performs additional pre and post-processing. For example, the resolution step transforms the subgoal into three subgoals, which correspond to the three assumptions of the induction schema. However, INDTAC discharges the first by unifying it with $?H$, whereby this "output metavariable" begins accumulating a recursive definition. Isabelle responds with the proof-state:

$\forall x.\, \mathsf{front}(x) = ?front_{body}(x)\ \wedge$
$\forall x.\, \mathsf{last}(x) = ?last_{body}(x)$
$\quad \longrightarrow \forall x.\, x \neq \Lambda \longrightarrow \mathsf{char}(\mathsf{last}(x)) \wedge x = \mathsf{front}(x) * \mathsf{last}(x)$

1. $\bigwedge a.\, [\![\forall t.\, (t, a) \in ?r \longrightarrow t \neq \Lambda \longrightarrow \mathsf{char}(\mathsf{last}(t)) \wedge t = \mathsf{front}(t) * \mathsf{last}(t);$
$\quad a \neq \Lambda\,]\!] \Longrightarrow \mathsf{char}(?last_{body}(a)) \wedge a = ?front_{body}(a) * ?last_{body}(a)$

2. $\mathsf{wf}(?r)$

[1] To give the reader a feel for the amount of proof effort involved, we include tactics given to the system. However, to avoid too many Isabelle specific details, our explanation will be at a higher level.

As in the Deductive Tableau, our first subgoal is essentially the original goal, but with program stubs for the functions we are synthesizing and the additional induction hypothesis. The additional subgoal states that the relation $?r$ is well-founded.

We proceed by resolving the first subgoal with the rule $\mathsf{case\_split\_2}$ to perform a case analysis: the string a consists of either one or more character. Higher-order resolution can have many unifiers and this is the case here; we therefore provide assistance by stipulating that the case-split predicate $?C$ should be instantiated with the predicate $\mathsf{char}(a)$.

$$\mathsf{By}\ (\mathsf{res\_inst\_tac}\ [(\text{"}C\text{"}, \text{"}\mathsf{char}(a)\text{"})]\ \mathsf{case\_split\_2}\ 1);$$

The new proof-state is:

$\forall x.\, \mathsf{front}(x) = \mathsf{if}\ \mathsf{char}(x)\ \mathsf{then}\ ?s_1(x)\ \mathsf{else}\ ?t_1(x)\ \wedge$
$\forall x.\, \mathsf{last}(x) = \mathsf{if}\ \mathsf{char}(x)\ \mathsf{then}\ ?s_2(x)\ \mathsf{else}\ ?t_2(x)$
$\quad \longrightarrow \forall x.\, x \neq \Lambda \longrightarrow \mathsf{char}(\mathsf{last}(x)) \wedge x = \mathsf{front}(x) * \mathsf{last}(x)$

1. $\bigwedge a.\, [\![\, \forall t.\, (t, a) \in ?r \longrightarrow t \neq \Lambda \longrightarrow \mathsf{char}(\mathsf{last}(t)) \wedge t = \mathsf{front}(t) * \mathsf{last}(t);$
$\quad a \neq \Lambda;\ \mathsf{char}(a)\,]\!] \Longrightarrow \mathsf{char}(?s_2(a)) \wedge a = ?s_1(a) * ?s_2(a)$
2. $\bigwedge a.\, [\![\, \forall t.\, (t, a) \in ?r \longrightarrow t \neq \Lambda \longrightarrow \mathsf{char}(\mathsf{last}(t)) \wedge t = \mathsf{front}(t) * \mathsf{last}(t);$
$\quad a \neq \Lambda;\ \neg\mathsf{char}(a)\,]\!] \Longrightarrow \mathsf{char}(?t_2(a)) \wedge a = ?t_1(a) * ?t_2(a)$
3. $\mathsf{wf}(?r)$

Resolution has instantiated the bodies of front and last with conditional terms. Note how the terms ($?s_1(a)$ and $?s_2(a)$) defining the base case appear in the first subgoal and those defining the recursive case appear in the second.

The base case is simple to establish. We resolve the first conjunct of subgoal 1 using the assumption $\mathsf{char}(a)$. This unifies $?s_2(a)$ with a (by instantiating $?s_2$ with the identity function). The second conjunct follows by resolution with the (trivially proven) lemma, concat_empty: $s = \Lambda * s$, which unifies $?s_1(a)$ and Λ. We combine these steps together

$$\mathsf{By}\ (\mathsf{EVERY1}\ [\mathsf{rtac\ conjI}\ ,\ \mathsf{atac}\ ,\ \mathsf{rtac\ concat\_empty}]);$$

and ISABELLE responds with:

$\forall x.\, \mathsf{front}(x) = \mathsf{if}\ \mathsf{char}(x)\ \mathsf{then}\ \Lambda\ \mathsf{else}\ ?t_1(x)\ \wedge$
$\forall x.\, \mathsf{last}(x) = \mathsf{if}\ \mathsf{char}(x)\ \mathsf{then}\ x\ \mathsf{else}\ ?t_2(x)$
$\quad \longrightarrow \forall x.\, x \neq \Lambda \longrightarrow \mathsf{char}(\mathsf{last}(x)) \wedge x = \mathsf{front}(x) * \mathsf{last}(x)$

1. $\bigwedge a.\, [\![\, \forall t.\, (t, a) \in ?r \longrightarrow t \neq \Lambda \longrightarrow \mathsf{char}(\mathsf{last}(t)) \wedge t = \mathsf{front}(t) * \mathsf{last}(t);$
$\quad a \neq \Lambda;\ \neg\mathsf{char}(a)\,]\!] \Longrightarrow \mathsf{char}(?t_2(a)) \wedge a = ?t_1(a) * ?t_2(a)$
2. $\mathsf{wf}(?r)$

We see that the base case of the proof is discharged, the base-cases of the definitions of front and last are instantiated, and the remaining goals are renumbered.

The recursive case is more interesting. Observe that the two subgoals share the variable $?r$. To prove the first goal we must use the induction hypothesis, but it must be used in a way that leaves the second goal provable. For example, we could immediately appeal to the induction hypothesis, but the result would require proving $(a, a) \in ?r$ for some well-founded relation $?r$, which is impossible.

We proceed with a bit of preliminary reasoning, which can be motivated by our algorithmic intuitions about how front and last should compute. Since a is not a singleton, we can split it into its head and tail. We can do this by transforming $a = ?t_1(a) * ?t_2(a)$, to $\mathsf{head}(a) \,.\, \mathsf{tail}(a) = \mathsf{head}(a) \,.\, ?t_1'(a) * ?t_2(a)$. Technically, we achieve this by resolving this subgoal with the rule hd_tl_E, $[\![\, t \neq \Lambda;\; P(\mathsf{head}(t) \,.\, \mathsf{tail}(t)) \,]\!] \Longrightarrow P(t)$. Doing this

$$\mathsf{By}\ ((\mathsf{res\_inst\_tac}\ [(\text{"}P\text{"}, \text{"}\lambda v.\, ?A \wedge v = ?B\text{"})]\ \mathsf{hd\_tl\_E}\ 1)\ \mathsf{THEN}\ (\mathsf{atac}\ 1));$$

yields the following proof-state:

$\forall\, x.\, \mathsf{front}(x) = \mathsf{if}\ \mathsf{char}(x)\ \mathsf{then}\ \Lambda\ \mathsf{else}\ \mathsf{head}(x) \,.\, ?t_1'(x)\ \wedge$
$\forall\, x.\, \mathsf{last}(x) = \mathsf{if}\ \mathsf{char}(x)\ \mathsf{then}\ x\ \mathsf{else}\ ?t_2(x)$
$\quad \longrightarrow \forall\, x.\, x \neq \Lambda \longrightarrow \mathsf{char}(\mathsf{last}(x)) \wedge x = \mathsf{front}(x) * \mathsf{last}(x)$

1. $\bigwedge a.\, [\![\, \forall\, t.\, (t, a) \in ?r \longrightarrow t \neq \Lambda \longrightarrow \mathsf{char}(\mathsf{last}(t)) \wedge t = \mathsf{front}(t) * \mathsf{last}(t);\; \neg\mathsf{char}(a);$
$\qquad a \neq \Lambda \,]\!] \Longrightarrow \mathsf{char}(?t_2(a)) \wedge \mathsf{head}(a) \,.\, \mathsf{tail}(a) = \mathsf{head}(a) \,.\, ?t_1'(a) * ?t_2(a)$

2. $\mathsf{wf}(?r)$

Actually resolution produced two subgoals, but the first was automatically solved using the given assumptions.

We continue by resolving with the lemma concat_cons,

$$P \wedge x = x_1 * x_2 \Longrightarrow P \wedge a \,.\, x = (a \,.\, x_1) * x_2 \,.$$

After typing By(rtac concat_cons 1); ISABELLE responds with:

$\forall\, x.\, \mathsf{front}(x) = \mathsf{if}\ \mathsf{char}(x)\ \mathsf{then}\ \Lambda\ \mathsf{else}\ \mathsf{head}(x) \,.\, ?t_1'(x)\ \wedge$
$\forall\, x.\, \mathsf{last}(x) = \mathsf{if}\ \mathsf{char}(x)\ \mathsf{then}\ x\ \mathsf{else}\ ?t_2(x)$
$\quad \longrightarrow \forall\, x.\, x \neq \Lambda \longrightarrow (\mathsf{char}(\mathsf{last}(x)) \wedge x = \mathsf{front}(x) * \mathsf{last}(x))$

1. $\bigwedge a.\, [\![\, \forall\, t.\, (t, a) \in ?r \longrightarrow t \neq \Lambda \longrightarrow \mathsf{char}(\mathsf{last}(t)) \wedge t = \mathsf{front}(t) * \mathsf{last}(t);$
$\qquad a \neq \Lambda;\; \neg\mathsf{char}(a) \,]\!] \Longrightarrow \mathsf{char}(?t_2(a)) \wedge \mathsf{tail}(a) = ?t_1'(a) * ?t_2(a)$

2. $\mathsf{wf}(?r)$

We now apply the induction hypothesis using the tactic By(apply_induction_hyp 1), which resolves with the premises of subgoal 1 and simplifies the result. This results in:

$\forall\, x.\, \mathsf{front}(x) = \mathsf{if}\ \mathsf{char}(x)\ \mathsf{then}\ \Lambda\ \mathsf{else}\ \mathsf{head}(x) \,.\, \mathsf{front}(\mathsf{tail}(x))\ \wedge$
$\forall\, x.\, \mathsf{last}(x) = \mathsf{if}\ \mathsf{char}(x)\ \mathsf{then}\ x\ \mathsf{else}\ \mathsf{last}(\mathsf{tail}(x))$
$\quad \longrightarrow \forall\, x.\, x \neq \Lambda \longrightarrow \mathsf{char}(\mathsf{last}(x)) \wedge x = \mathsf{front}(x) * \mathsf{last}(x)$

1. $\bigwedge a.\, [\![\, a \neq \Lambda;\; \neg\mathsf{char}(a) \,]\!] \Longrightarrow (\mathsf{tail}(a), a) \in ?r$

2. $\bigwedge a.\, [\![\, a \neq \Lambda;\; \neg\mathsf{char}(a) \,]\!] \Longrightarrow \mathsf{tail}(a) \neq \Lambda$

3. $\mathsf{wf}(?r)$

We have now completed our synthesis of front and last: all metavariables in the goal have been instantiated and the result is two recursive programs built from executable (primitive) predicates. All that remains from the first subgoal are some residual proof obligations. Since we have instantiated the universally quantified variable t in the induction hypothesis with tail(a), we must show that tail(a) is smaller than a under $?r$. We must also show that it is not the empty string. The first subgoal, together with the third, guarantees the termination of the front and last functions.

We proceed by formalizing a concrete well-founded relation that satisfies the requirement given in subgoal 1, namely $(\mathsf{tail}(a), a) \in ?r$. We instantiate $?r$ with the well-founded ordering $\mathsf{measure}(\mathsf{length})$, i.e., the length of $\mathsf{tail}(a)$ is less than a. After that, we automatically complete the proof using tactics for classical reasoning and simplification.

ISABELLE responds that we are done and prints the final proof state.

$$\begin{array}{l}\forall\, x.\, \mathsf{front}(x) \;=\; \mathsf{if}\ \mathsf{char}(x)\ \mathsf{then}\ \Lambda\ \mathsf{else}\ \mathsf{head}(x)\,.\,\mathsf{front}(\mathsf{tail}(x)) \wedge \\ \forall\, x.\, \mathsf{last}(x) = \mathsf{if}\ \mathsf{char}(x)\ \mathsf{then}\ x\ \mathsf{else}\ \mathsf{last}(\mathsf{tail}(x)) \\ \qquad \longrightarrow \forall\, x.\, x \neq \Lambda \longrightarrow \mathsf{char}(\mathsf{last}(x)) \wedge x = \mathsf{front}(x) * \mathsf{last}(x)\end{array}$$

The proven proposition both provides definitions of front and last and states that they fulfill their specification.

Overall, as in the Deductive Tableau, induction and case analysis provide a means of constructing programs hand in hand with their proofs of correctness. However the use of a more expressive logic (ISABELLE/HOL) allows us to internalize reasoning about termination, and the introduction of higher-order metavariables provides a means for tracking the construction of programs during top-down and branching derivations.

6. A Case Study in Sorting

The front/last development of the previous section is a simple but complete example that illustrates our second interpretation. In this section we will give a more realistic example, but at a higher-level: the development of sorting programs.

Sorting is an interesting problem. It is easy to specify, but yet there are a large number of different sorting algorithms in the literature. We have used our interpretation and development methodology to synthesize a number of standard sorting algorithms including *quick sort, merge sort, insertion sort, select sort* and *bubble sort.* In this section we will give some snapshots from our development of quick sort. Overall, quick sort was one of the more complicated sorting algorithms that we synthesized and its proof required 23 interactive steps. In the following we provide some background and a few snapshots from our ISABELLE session. We use these to highlight several of our development decisions, and to examine, in more detail, termination arguments.

6.1. The Specification

The standard specification of sorting is that a sorting program should return an ordered permutation of its input.

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

In our formalization we use ISABELLE's theory of lists, which employs standard notation: the empty list is represented by [], the symbol cons denotes the list constructor, the symbol @ denotes the append function, and head and tail are the head and tail functions.

Our definitions of the predicates perm and $\mathsf{ordered}$ are taken directly from Manna and Waldinger (1993) and formalize the standard permutation and ordered relation on lists.

$\mathsf{perm}([], [])$

$\mathsf{perm}(l_1 @ [u] @ l_2, t_1 @ [u] @ t_2) = \mathsf{perm}(l_1 @ l_2, t_1 @ t_2)$
$\mathsf{perm}(l_1, l_2) \longrightarrow \forall\, x.\, \mathsf{member}(x, l_1) = \mathsf{member}(x, l_2)$
$\mathsf{ordered}([])$
$\mathsf{ordered}([a])$
$\mathsf{ordered}(\mathsf{cons}(a, \mathsf{cons}(b, t))) = (a \leq b) \wedge \mathsf{ordered}(\mathsf{cons}(b, t))$

We begin our ISABELLE proof with the initial proof-state:

$?H \longrightarrow \forall\, l.\, \mathsf{perm}(l, ?sort(l)) \wedge \mathsf{ordered}(?sort(l))$

1. $?H \longrightarrow \forall\, l.\, \mathsf{perm}(l, ?sort(l)) \wedge \mathsf{ordered}(?sort(l))$

6.2. Induction

Our first proof step is induction on l by resolution with induction_schema_1, naming the synthesized program qsort. This results in the proof-state:

$$\forall\, l.\, \mathsf{qsort}(l) = ?qsort_{body}(l) \wedge ?H_1 \\ \quad \longrightarrow \forall\, l.\, \mathsf{perm}(l, \mathsf{qsort}(l)) \wedge \mathsf{ordered}(\mathsf{qsort}(l))$$

1. $\bigwedge\, l.\, [\![\forall\, t.\, (t, l) \in ?r \longrightarrow \mathsf{perm}(t, \mathsf{qsort}(t)) \wedge \mathsf{ordered}(\mathsf{qsort}(t));$
   $?H_1]\!] \Longrightarrow \mathsf{perm}(l, ?qsort_{body}(l)) \wedge \mathsf{ordered}(?qsort_{body}(l))$

2. $\mathsf{wf}(?r)$

Note that the metavariable $?H$, whose role is to collect definitions during synthesis, has been both instantiated with a definition stub for qsort and a new metavariable $?H_1$. This is accomplished by some preprocessing where our induction tactic first "duplicates" $?H$ by instantiating it with $?H_0 \wedge ?H_1$ (by resolving with $\wedge$-elimination). This preprocessing represents a kind of logical "proof-context engineering" and it allows us to use resolution to simulate proof under a growing context of definitions.

The induction step leaves us with two subgoals. The first says we must develop a program $?qsort_{body}$ under the assumption that we have a program qsort that sorts all lists smaller (under $?r$) than l. The second stipulates that $?r$ is a well-founded relation.

6.3. Case Analysis

Sorting algorithms work by partitioning lists and recursively processing the partitions. However, if the partition is to be non-trivial, we require that the list is non-empty. Hence we begin with a case-split on whether l is empty or not: we resolve the first subgoal with the case analysis rule of Section 5.2 and specify that the splitting condition is $l = []$. ISABELLE returns the proof-state:

$$\forall\, l.\, \mathsf{qsort}(l) = \mathsf{if}\ (l = [])\ \mathsf{then}\ ?S(l)\ \mathsf{else}\ ?T(\mathsf{head}(l), \mathsf{tail}(l)) \wedge ?H_1 \\ \quad \longrightarrow \forall\, l.\, \mathsf{perm}(l, \mathsf{qsort}(l)) \wedge \mathsf{ordered}(\mathsf{qsort}(l))$$

1. $\bigwedge\, l.\, [\![\forall\, t.\, (t, l) \in ?r \longrightarrow \mathsf{perm}(t, \mathsf{qsort}(t)) \wedge \mathsf{ordered}(\mathsf{qsort}(t));$
   $?H_1;\ l = []]\!] \Longrightarrow \mathsf{perm}(l, ?S(l)) \wedge \mathsf{ordered}(?S(l))$

2. $\bigwedge\, l.\, [\![\forall\, t.\, (t, l) \in ?r \longrightarrow \mathsf{perm}(t, \mathsf{qsort}(t)) \wedge \mathsf{ordered}(\mathsf{qsort}(t));$
   $?H_1;\ l \neq []]\!] \Longrightarrow \mathsf{perm}(l, ?T(\mathsf{head}(l), \mathsf{tail}(l))) \wedge \mathsf{ordered}(?T(\mathsf{head}(l), \mathsf{tail}(l)))$

3. $\mathsf{wf}(?r)$

Resolution has instantiated $?qsort_{body}$ with a conditional term and the first subgoal has been replaced by two, which construct terms for each case. The remaining subgoal, renumbered to 3, remains unchanged.

We simplify the conclusion of subgoal 1, yielding $\mathsf{perm}([], ?S([])) \wedge \mathsf{ordered}(?S([]))$, and we resolve this using the definitions of perm and ordered, which unifies $?S([])$ with $[]$.

6.4. An Aside on the Genealogy of Sorting

We have come to the point in the proof where most of the nontrivial and creative work takes place. The next steps determine which sorting algorithm will be synthesized.

A number of authors have observed that algorithms like sorting can be given a kind of "family tree" where the different algorithms can be related by decisions in their development (Broy, 1978; Smith, 1985). In our setting this tree represents the possible derivations in our Isabelle theory. Branches represent choice points between different development (resolution) steps and are navigated interactively: We use our programmer's intuition to guide the prover, often using auxiliary definitions and lemmas, to the desired program.

In Figure 2 we give the tree that summarizes the genealogy of the sorting algorithms that we developed in Isabelle. In all cases, the initial steps are identical to those given above. Our current choice-point corresponds to the "step case box" and determines how the input variable l is partitioned into smaller lists upon which the induction hypothesis can be applied. For our algorithms, this was *the* major choice-point in their development.

Partitioning a list requires defining different predicates, like head/tail, or front/last. These predicates need not themselves be executable; if they are needed for the program itself, then executable versions will have to be synthesized.

If we partition l into two lists, l_1 and l_2, then we can derive one of merge, insertion, or bubble sort. In merge sort l_1 (respectively l_2) is the first (respectively second) half of l. Insertion sort can be seen as a degenerate variant of merge sort, where l_1 contains only the first element of l and l_2 is the rest of l. Bubble sort uses three auxiliary programs, front, last and sift. The functions front and last are those that we have derived in Section 2.4. The function sift moves the maximum element of a list to the end of the list without changing the position of the other elements. In bubble sort l_1 contains all but the maximum elements of l and l_2 contains only the maximum element.

Alternatively, we can partition l into three lists. Quick sort splits its input list into its first element (the "pivot"), l_1, and those elements greater than the pivot, l_2, and less than the pivot, l_3. In the case of select sort, l_1 is the singleton consisting of the minimum of l, l_2 consists of the elements in l whose positions are less than the position of the minimum (this list is denoted by $\mathsf{lem}(l)$), l_3 (denoted by $\mathsf{gem}(l)$) is the list l without its minimum and the elements of l_2.

6.5. Recursive Step and Partitioning

Let us continue with the development of quick sort. To perform the partitioning we require two auxiliary predicates: minl and maxl. The predicate $\mathsf{minl}(u, l)$ says that the element u is less than or equal to all the elements of the list l. The predicate $\mathsf{maxl}(u, l)$ says that u is greater than or equal to the elements of l. In particular, we split $\mathsf{tail}(l)$ in two lists whose concatenation is a permutation of $\mathsf{tail}(l)$ and everything in the first list is

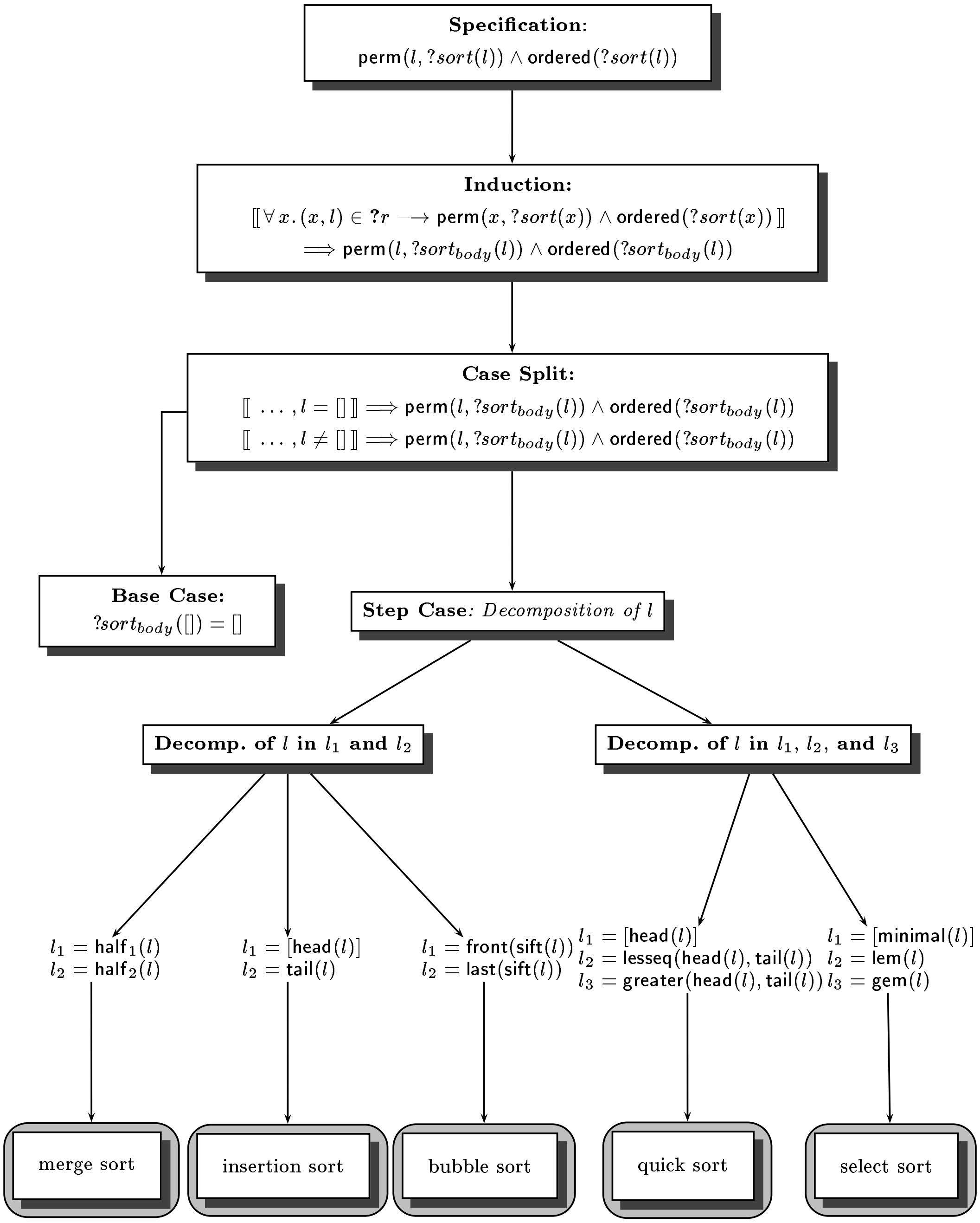

Figure 2. Derivation of Sorting Programs

less than or equal to $\mathsf{head}(l)$ and the second contains only elements greater than $\mathsf{head}(l)$. This splitting is performed by a sequence of resolution steps (that perform massaging similar to the kind we saw in the step-case of the front/last example), which result in the following proof-state:

$$\forall\, l.\, \mathsf{qsort}(l) = \mathsf{if}\ (l = [])\ \mathsf{then}\ []\ \mathsf{else}\ ?t_1(\mathsf{head}(l), \mathsf{tail}(l))\,@\,[\mathsf{head}(l)]\,@\,?t_2(\mathsf{head}(l), \mathsf{tail}(l)) \wedge ?H_1$$
$$\longrightarrow \forall\, l.\ \mathsf{perm}(l, \mathsf{qsort}(l)) \wedge \mathsf{ordered}(\mathsf{qsort}(l))$$

1. $\bigwedge l.\, [\![\forall\, t.\, (t, l) \in\ ?r \longrightarrow \mathsf{perm}(t, \mathsf{qsort}(t)) \wedge \mathsf{ordered}(\mathsf{qsort}(t));\ l \neq [];$
   $?H_1]\!] \Longrightarrow \mathsf{perm}(\mathsf{tail}(l), ?l_1(l)\,@\,?l_2(l)) \wedge$
   $\mathsf{minl}(\mathsf{head}(l), ?t_2(\mathsf{head}(l), \mathsf{tail}(l))) \wedge \mathsf{maxl}(\mathsf{head}(l), ?t_1(\mathsf{head}(l), \mathsf{tail}(l)))$
2. $\bigwedge l.\ [\![\forall\, t.\, (t, l) \in\ ?r \longrightarrow \mathsf{perm}(t, \mathsf{qsort}(t)) \wedge \mathsf{ordered}(\mathsf{qsort}(t));\ l \neq [];$
   $?H_1]\!] \Longrightarrow \mathsf{perm}(?l_1(l), ?t_1(\mathsf{head}(l), \mathsf{tail}(l))) \wedge \mathsf{ordered}(?t_1(\mathsf{head}(l), \mathsf{tail}(l)))$
3. $\bigwedge l.\ [\![\forall\, t.\, (t, l) \in\ ?r \longrightarrow \mathsf{perm}(t, \mathsf{qsort}(t)) \wedge \mathsf{ordered}(\mathsf{qsort}(t));\ l \neq []$
   $?H_1]\!] \Longrightarrow \mathsf{perm}(?l_2(l), ?t_2(\mathsf{head}(l), \mathsf{tail}(l))) \wedge \mathsf{ordered}(?t_2(\mathsf{head}(l), \mathsf{tail}(l)))$
4. $\mathsf{wf}(?r)$

The embryonic form of quick sort has taken shape. The first subgoal specifies that $\mathsf{tail}(l)$ can be decomposed into two lists $?l_1(l)$ and $?l_2(l)$, which satisfy the minl and maxl predicates. The second and third subgoals state that there are lists computed by $?t_1$ and $?t_2$ that are permutations of $?l_1(l)$ and $?l_2(l)$ and are both sorted.

The second and third subgoals are particularly easy to solve; we direct ISABELLE to unify both with their induction hypotheses.

$$\begin{aligned}\forall\, l.\, \mathsf{qsort}(l) = {} & \mathsf{if}\ (l = [])\\ & \mathsf{then}\ []\\ & \mathsf{else}\ \mathsf{qsort}(?t(\mathsf{head}(l), \mathsf{tail}(l)))\\ & \qquad @\,[\mathsf{head}(l)]\,@\\ & \quad \mathsf{qsort}(?s(\mathsf{head}(l), \mathsf{tail}(l)))\ \wedge ?H_1\\ \longrightarrow \forall\, l.\ & \mathsf{perm}(l, \mathsf{qsort}(l)) \wedge \mathsf{ordered}(\mathsf{qsort}(l))\end{aligned}$$

1. $\bigwedge l.\ [\![\forall\, t.\, (t, l) \in\ ?r \longrightarrow \mathsf{perm}(t, \mathsf{qsort}(t)) \wedge \mathsf{ordered}(\mathsf{qsort}(t));\ l \neq [];$
   $?H_1]\!] \Longrightarrow \mathsf{perm}(\mathsf{tail}(l), ?t(\mathsf{head}(l), \mathsf{tail}(l))\,@\,?s(\mathsf{head}(l), \mathsf{tail}(l))) \wedge$
   $\mathsf{minl}(\mathsf{head}(l), ?s(\mathsf{head}(l), \mathsf{tail}(l))) \wedge \mathsf{maxl}(\mathsf{head}(l), ?t(\mathsf{head}(l), \mathsf{tail}(l)))$
2. $\bigwedge l.\, [\![?H_1;\ l \neq []]\!] \Longrightarrow (?t(\mathsf{head}(l), \mathsf{tail}(l)), l) \in\ ?r$
3. $\bigwedge l.\, [\![?H_1;\ l \neq []]\!] \Longrightarrow (?s(\mathsf{head}(l), \mathsf{tail}(l)), l) \in\ ?r$
4. $\mathsf{wf}(?r)$

As a result, $?t_1$ and $?t_2$ have been replaced by recursive calls to quick sort. However, there are several residual proof obligations. We must later show that the function $?t$ (in goal 2) and $?s$ (in goal 3) are applied to arguments "less" than l under $?r$, which must be well-founded (subgoal 4). In other words, we have used the induction hypothesis twice and in each case we must show that it was on smaller instances under the not yet specified relation $?r$.

6.6. Auxiliary Synthesis

Our next step is to synthesize the functions for $?t$ and $?s$ specified by the first subgoal. These functions pick out elements in the tail of l that are less than or equal (or, in the case of $?s$, greater than) the head of l. The synthesis of these two functions proceeds, like the synthesis of other functions, by induction and case-splitting. The resulting programs synthesized are:

$$\begin{aligned}
\mathsf{greater}(x,y) &= \mathsf{if}\ (y = [])\ \mathsf{then}\ []\ \mathsf{else\ if}\ (\mathsf{head}(y) \leq x)\ \mathsf{then\ greater}(x, \mathsf{tail}(y)) \\
&\qquad\qquad\qquad\qquad \mathsf{else\ cons}(\mathsf{head}(y), \mathsf{greater}(x, \mathsf{tail}(y))) \\
\mathsf{lesseq}(x,y) &= \mathsf{if}\ (y = [])\ \mathsf{then}\ []\ \mathsf{else\ if}\ (\mathsf{head}(y) \leq x)\ \mathsf{then\ cons}(\mathsf{head}(y), \mathsf{lesseq}(x, \mathsf{tail}(y))) \\
&\qquad\qquad\qquad\qquad \mathsf{else\ lesseq}(x, \mathsf{tail}(y))
\end{aligned}$$

6.7. Termination

All that remains is to show that quick sort terminates. This is reflected by the proof state whose only remaining subgoals are:

1. $\bigwedge l.\, [\![\, l \neq [];\ \forall x\, y.\, \mathsf{lesseq}(x,y) = \ldots]\!] \Longrightarrow (\mathsf{lesseq}(\mathsf{head}(l), \mathsf{tail}(l)), l) \in\ ?r$
2. $\bigwedge l.\, [\![\, l \neq [];\ \forall x\, y.\, \mathsf{greater}(x,y) = \ldots]\!] \Longrightarrow (\mathsf{greater}(\mathsf{head}(l), \mathsf{tail}(l)), l) \in\ ?r$
3. $\mathsf{wf}(?r)$

There are various ways to solve these subgoals. One possibility is to begin with the third; for example, we could resolve it with a lemma stating that some particular relation is well-founded. However, there are several drawbacks to this. First, it is unlikely that we will have the appropriate relation already available and if we pick an inappropriate one, then the first two subgoals will not be provable. Even when an appropriate relation is available, the resulting subgoals may require considerable effort to show. For example, if we choose $?r$ to be the ordering $\{(\mathsf{tail}(x), x) \mid x \neq []\}$, then it is easy to show that $\mathsf{wf}(?r)$ holds since this ordering is a suborder of the well-founded ordering $\{(x, y) \mid \mathsf{length}(y) < \mathsf{length}(y)\}$. However, the proofs of the $\mathsf{length}(\mathsf{lesseq}(\mathsf{head}(l), \mathsf{tail}(l))) < \mathsf{length}(l)$ and $\mathsf{length}(\mathsf{greater}(\mathsf{head}(l), \mathsf{tail}(l))) < \mathsf{length}(l)$ are non-trivial.

An alternative way to solve these subgoals is to take the idea of program synthesis further and synthesize an appropriate well-founded relation during the proofs of the first two subgoals. This is consistent with the spirit of our methodology: rather than give programs (now orderings) up-front and undertake large monolithic verification tasks, we can instead combine verification and synthesis in an incremental top-down way. We take this approach here. The first two subgoals can be seen as stating constraints on the well-founded relation $?r$. The conjunction of these constraints, together with the third subgoal, constitutes a specification of $?r$. Now, to guide the system to incrementally construct an appropriate instantiation of $?r$, we apply induction on l on both subgoals 1 and 2. This yields the following subgoals (subgoals 1, 2 and 3 come from the induction on subgoal 1 of the previous proof-state and subgoal 1, 4 and 5 come from the induction on subgoal 2 of the previous proof-state):

1. $\bigwedge a.\, ([], [a]) \in\ ?r$
2. $\bigwedge a\, x\, u.\, [\![\ (\mathsf{lesseq}(a, u), \mathsf{cons}(a, u)) \in\ ?r;\ x \leq a\ ]\!] \Longrightarrow (\mathsf{cons}(x, \mathsf{lesseq}(a, u)), [a,\ x] @ u) \in\ ?r$

3. $\bigwedge a\, x\, u.\, [\![\, (\mathsf{lesseq}(a,u), \mathsf{cons}(a,u)) \in ?r;\ a < x\,]\!] \Longrightarrow (\mathsf{lesseq}(a,u), [a,\ x] @\, u) \in ?r$

4. $\bigwedge a\, x\, u.\, [\![\, (\mathsf{greater}(a,u), \mathsf{cons}(a,u)) \in ?r;\ x \leq a\,]\!] \Longrightarrow (\mathsf{greater}(a,u), [a,\ x] @\, u) \in ?r$

5. $\bigwedge a\, x\, u.\, [\![\, (\mathsf{greater}(a,u), \mathsf{cons}(a,u)) \in ?r;\ a < x\,]\!] \Longrightarrow (\mathsf{cons}(x, \mathsf{greater}(a,u)), [a,\ x] @\, u) \in ?r$

6. $\mathsf{wf}(?r)$

Subgoals 2–5 can be generalized to the following subgoal 2:

1. $\bigwedge a.\, ([], [a]) \in ?r$

2. $\bigwedge a\, b\, l_1\, l_2.\, (l_1, \mathsf{cons}(a, l_2)) \in ?r \Longrightarrow (l_1, [a,\ b] @\, l_2) \in ?r \wedge (\mathsf{cons}(b, l_1), [a,\ b] @\, l_2) \in ?r$

3. $\mathsf{wf}(?r)$

Now, the goal has been considerably simplified. We proceed by using resolution to instantiate the relation $?r$ with the measure ordering $\mathsf{measure}(?f)$ where $?f$ is a function from lists to natural numbers.

1. $\bigwedge a.\, ?f([]) < ?f([a])$

2. $\bigwedge a\, b\, l_1\, l_2.\, ?f(l_1) < ?f([a] @\, l_2) \Longrightarrow ?f(l_1) < ?f([a,\ b] @\, l_2) \wedge ?f([b] @\, l_1) < ?f([a,\ b] @\, l_2)$

It remains to find a function $?f$ having the above properties. The user can simply check that the function length is a good candidate.

6.8. Final Proof-State

After termination of quick sort and synthesis of all auxiliary programs, the final proof-state is the following.

$$
\begin{array}{l}
[\![\, \forall\, l.\, \mathsf{qsort}(l) = \mathsf{if}\ (l = [])\ \mathsf{then}\ []\quad \mathsf{else}\ \mathsf{qsort}(\mathsf{lesseq}(\mathsf{head}(l), \mathsf{tail}(l)))\ @\ [\mathsf{head}(l)] @ \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{qsort}(\mathsf{greater}(\mathsf{head}(l), \mathsf{tail}(l)));
\end{array}
$$

$$
\begin{array}{l}
\forall\, x\, y.\, \mathsf{greater}(x,y) = \mathsf{if}\ (y = [])\ \mathsf{then}\ []\ \mathsf{else\ if}\ (\mathsf{head}(y) \leq x)\ \mathsf{then}\ \mathsf{greater}(x, \mathsf{tail}(y)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ \mathsf{cons}(\mathsf{head}(y), \mathsf{greater}(x, \mathsf{tail}(y)))
\end{array}
$$

$$
\begin{array}{l}
\forall\, x\, y.\, \mathsf{lesseq}(x,y) = \mathsf{if}\ (y = [])\ \mathsf{then}\ []\ \mathsf{else\ if}\ (\mathsf{head}(y) \leq x)\ \mathsf{then}\ \mathsf{cons}(\mathsf{head}(y), \mathsf{lesseq}(x, \mathsf{tail}(y))) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ \mathsf{lesseq}(x, \mathsf{tail}(y))
\end{array}
$$

$$
]\!] \Longrightarrow \forall\, l.\, \mathsf{perm}(l, \mathsf{qsort}(l)) \wedge \mathsf{ordered}(\mathsf{qsort}(l))
$$

The initial output variable has been instantiated to the definition of quick sort and the two auxiliary functions used in this definition. The resulting theorem states that, under these definitions, the quick sort program satisfies the sorting specification. Note that the definitions given can be directly translated into one's favorite functional programming language and correspond to the standard definition of quick sort.

7. Related Work

Below we compare our work with other resolution based approaches to program synthesis. Afterwards, we step back and use the insights we have gained to shed some light on the differences between deductive and constructive synthesis approaches.

7.1. Resolution Based Synthesis

Our use of resolution to synthesize programs is related to ideas that go back to the 1960s on using first-order resolution to construct terms that represent plans or, more generally, programs (Green, 1969; Chang and Lee, 1973). This idea is now standard, especially in the logic program community where proof by resolution is used both for verification, e.g., checking that $p(t)$ follows from clauses in the program data-base, and synthesis, e.g., given the goal $p(X)$, finding a substitution t for X where $p(t)$ is provable.

Our work can be viewed as extending such a (Horn or full first-order) resolution based approach in several different directions. Instead of employing an axiomatized set of clauses as in Prolog, we use general derived rules; in our case these are motivated by, and model, the rules of the Deductive Tableau. Instead of using a fixed strategy for proof construction, like SLD resolution, we use general interactive theorem proving based on the Deductive Tableau methodology. And instead of synthesizing first-order terms using first-order resolution, we use higher-order syntax and build recursive functions by using resolution to construct proofs by induction.

In the Isabelle community the use of higher-order resolution to solve for metavariables is standard and the applications are diverse: Paulson (1994b) represents the rules of a constructive type theory in the metalogic of Isabelle and applies the rules with higher-order resolution to synthesize programs. Basin and Friedrich (1999) use resolution to synthesize circuits descriptions and, in doing so, show how a particular hardware synthesis methodology can be modeled in Isabelle. Coen (1992) developed his own formal system for reasoning about functional programs. The work of Coen is the closest to ours as both are based on applying derived rules in classical logic (in particular, rules based on induction) to synthesize programs using higher-order resolution. However, our goals are rather different: Coen was not trying to directly model a proposed formalism, but rather created his own specialized theory, with its own advantages and disadvantages. For example, he had to show the correctness of his specialized logic with respect to an appropriate semantics.

7.2. Deductive versus Constructive Synthesis

The Deductive Tableau is a classical example of the *deductive synthesis* approach to program development, which is sometimes contrasted in with *constructive synthesis* methods (Deville and Lau, 1993). The latter are typically based on constructive type theories, such as (Martin-Löf, 1982; Coquand and Huet, 1988), where a proof of a formula $\forall x.\, \exists y.\, Q(x, y)$ also yields a function f for which $Q(x, f(x))$ holds.

From the viewpoint of proof by higher-order resolution, many approaches to constructive synthesis can also be understood as deductive synthesis. For example, as mentioned above, Paulson (1994b) implements a type theory of Martin-Löf in Isabelle. In this theory, specifications are types and one shows that a program t satisfies a specification T by showing that t belongs to the type T, i.e. $t \in T$. By using higher-order resolution, the program t can initially be given by a metavariable and synthesized during proof. The point is this: Constructivity is not essential for the synthesis of programs, but rather in their *meaning*. That is, because the type theory is constructive, the programs extracted can actually be executed (which in some type-theories corresponds to proof-normalization) and their evaluation agrees with the semantics of the type theory.

Type theories and deductive synthesis methodologies have different strengths and weaknesses. Type theories are usually rich theories in which one can develop higher-order functional programs. The Deductive Tableau, at least, can only reason about first-order programs (even in our interpretations, where reasoning itself is higher-order). A drawback of constructive methods though is that one must always reason constructively, even when proving subgoals that are computationally uninteresting, i.e., have no effect on the synthesized program. Such reasoning arises when showing that programs terminate or when reasoning about equality. In the Deductive Tableau, such computationally uninteresting steps are reflected by rows without output entries.

In the Deductive Tableau, we can always reason classically; however, case-splitting is restricted to a set of primitive expressions to ensure executability of the synthesized program. This restriction is, on closer examination, quite interesting. In a constructive setting, a case-split corresponds to an application of $\vee$-elimination to an assumption $A \vee B$. Constructively, such an assumption requires a program t that decides which of A or B holds; so when we build an if-then-else constructively using case-splitting, t plays the role of the predicate tested by if, which decides which case holds. So here is an example where the additional work required in proving theorems constructively (in particular, reasoning about disjunctions) has a payoff: it yields a more flexible way of developing programs than in the Deductive Tableau since we can perform case-splits on disjunctions, provided we have shown them to be decidable. Said another way, in a constructive setting, instead of restricting ourselves to a given set of primitive predicates we can synthesize them during proof.

8. Conclusion and Future Work

We have taken the Deductive Tableau and interpreted it in two ways within higher-order logic. The first is a direct modeling and provides the basis of an implementation in ISABELLE with all the conveniences of a modern tactic based theorem prover. The second is even simpler. It distills the Deductive Tableau into two rules: case analysis and induction. Embedded within higher-order logic and supported by higher-order resolution, these rules are recast in a simple way where, together with standard (natural deduction) proof rules for HOL, programs can be synthesized in the spirit of the Deductive Tableau. This avoids the Deductive Tableau's drawbacks and enriches the methodology.

We would like to close with two areas where we see a need for further work. The first concerns the specialization of generic theorem provers for program synthesis. In this paper we have shown some of the advantages of implementing program development methodologies in a generic theorem prover. However there are also certain drawbacks. One is cosmetic and concerns how information is displayed: when we encode proof states of one formalism in another, the result is not necessarily at the right abstraction level to be presented to the user. For example, we use ISABELLE's metalogical connectives to combine representations of conceptually different parts of tableaux: the (partial) definitions of synthesized functions, metavariables standing in for possible context extensions, subgoals, and the like. The effect is that, in both of our interpretations, it can be difficult to interpret system output. Solutions to this problem should be possible based on constructing appropriate graphical user interfaces that separate conceptually different parts of the proof state and hide non-essential details.

Another drawback in using a generic prover like ISABELLE concerns the difficulties we had in implementing families of related rules. Recall the discussion in Section 5.2 of why one rule (nonclausal resolution, or induction) in the Deductive Tableau corresponds to a family of rules in our setting. This problem is essentially the same as that explored in Section 3.1: how to apply rules in contexts with varying numbers of eigenvariables. The solution to the eigenvariable problem is built into ISABELLE; the system automatically lifts rules over the variables. In our own work, we needed an analogous facility where synthesis rules are lifted over collections of function definitions. We ended up doing this lifting by hand, i.e., deriving instances of each family as required. It should be possible to find more principled solutions to this problem though, such as writing metaprograms that automatically generate and derive these rules.

A second, and more significant area for improvement concerns automation. Our proofs were interactive, but partially automated by tactics. There is considerable potential for improvement here, for example, in automating termination proofs. Recall from the previous sections that termination proofs can be viewed as synthesis proofs where we synthesize a relation that satisfies a set of given constraints. This is related to the problem of establishing the termination of term-rewriting systems. It is likely that ideas developed in the term rewriting community can be adapted to our setting. For example, the *estimation calculus* of (Walther, 1991a; Walther, 1991b) provides a means of deriving "measure orderings" (cf. $\mathsf{measure}(f)$ in Section 5.3) for recursively defined functions. It would be interesting to see if such ideas can used within ISABELLE so that the orderings derived can be automatically formalized in higher-order logic and the justification of their well-foundedness implemented as tactics.

Appendix A. Proof of Lemma 4.1

PROOF. We consider here the case where R is the assertion-goal form of nonclausal resolution. Other proof rules are handled similarly.

$\mathcal{T}$ consists of a collection of rows $r_1, \ldots, r_n$ and $\lceil \mathcal{T} \rceil$ is of the form of (4.1). Since the assertion-goal form of nonclausal resolution applies, there must be rows r_i

$A_i(P)$		t_i

and r_j

	$G_j(Q)$	s_j

to which the rule is applied. Rule application produces the row r_{n+1},

	$\neg A_i\theta(\mathsf{False}) \wedge G_j\theta(\mathsf{True})$	$\mathsf{if}\ P\theta\ \mathsf{then}\ s_j\theta\ \mathsf{else}\ t_i\theta$

where θ is the most general unifier of P and Q. Recall that by applying any tableau rule the formula (goal/assertion) as well as the output term of the new generated row are automatically simplified.

The above row together with $r_1, \ldots, r_n$, form the tableau $\mathcal{T}'$. Now consider resolution of $\lceil \mathcal{T} \rceil$ with $\lceil R \rceil$, which is NCR. Let $H(a)$ be the part of the antecedent of $\lceil \mathcal{T} \rceil$ that encodes

$r_1, \dots, r_n$, i.e. $[\![\lceil r_1 \rceil; \dots; \lceil r_n \rceil]\!]$. In resolving $\lceil \mathcal{T} \rceil$ with NCR, the rule is automatically lifted over the parameter a and assumptions $H(a)$. This resolution step instantiates the conclusion of NCR, ϕ, with False and the six subgoals of NCR are instantiated to:

1. $\bigwedge a.\, H(a) \Longrightarrow \mathsf{A}(?A(a, ?P(a)), ?f(a,\ ?a(a)) = ?t(a))$
2. $\bigwedge a.\, H(a) \Longrightarrow \mathsf{G}(?G(a, ?Q(a)), ?f(a,\ ?a(a)) = ?s(a))$
3. $\bigwedge a.\, H(a) \Longrightarrow ?P(a) = ?Q(a);$
4. $\bigwedge a.\, H(a) \Longrightarrow ?NewGoal(a) = \neg ?A(a, \mathsf{False}) \wedge ?G(a, \mathsf{True})$
5. $\bigwedge a.\, H(a) \Longrightarrow ?NewFbody(a) = \mathsf{if}\ ?P(a)\ \mathsf{then}\ ?s(a)\ \mathsf{else}\ ?t(a)$
6. $\bigwedge a.\, H(a) \Longrightarrow \mathsf{G}(?NewGoal(a), ?NewFbody(a)) \Longrightarrow \mathsf{False}$

We now show that the first 5 subgoals can be solved and the remaining subgoal constitutes the antecedent of $\lceil \mathcal{T}' \rceil$. We solve the first and second subgoals by unifying their conclusions with $\lceil r_i \rceil$ and $\lceil r_j \rceil$, which are:

$$\begin{array}{lcl} \lceil r_i \rceil & \equiv & \mathsf{A}(\uparrow^a A_i(P), f(a) =\uparrow^a t_i) \\ \lceil r_j \rceil & \equiv & \mathsf{G}(\uparrow^a G_j(Q), f(a) =\uparrow^a s_j) \end{array}$$

In doing so, the metavariables $?P$ and $?Q$ in (1) and (2) are instantiated with $\lambda a.\ \uparrow^a P$ and $\lambda a.\ \uparrow^a Q$ respectively and thus, the third subgoal becomes

$$\bigwedge a.\, H(a) \Longrightarrow \uparrow^a P =\uparrow^a Q\,.$$

Above, we assumed that θ is an unifier of P and Q; thus, we can solve the above subgoal by unifying $\uparrow^a P$ and $\uparrow^a Q$ using the substitution θ lifted over the parameter a. By "lifting over a" we mean that a substitution $\theta = \{x_1 \leftarrow t_1, \dots, x_n \leftarrow t_n\}$ is transformed to the substitution $\theta' = \{x_1 \leftarrow \lambda a.\, t_1, \dots, x_n \leftarrow \lambda a.\, t_n\}$.

In ISABELLE, substitutions are globally propagated. So, after solving the first three subgoals, the right hand side of the equations in the fourth and fifth subgoals become $\neg \uparrow^a (A_i\theta(\mathsf{False})) \wedge \uparrow^a (G_j\theta(\mathsf{True}))$ and $\mathsf{if}\ \uparrow^a (P\theta)\ \mathsf{then}\ \uparrow^a (s_j\theta)\ \mathsf{else}\ \uparrow^a (t_j\theta)$ respectively. We discharge the fourth subgoal by assigning to $?NewGoal(a)$ the simplified right-hand side of the equality and we discharge the fifth subgoal similarly.

After these steps our proof state has the antecedent

$$\bigwedge a.\, [\![H(a);\ \mathsf{G}(\uparrow^a (\neg A_i\theta(\mathsf{False}) \wedge G_j\theta(\mathsf{True})),\ \uparrow^a (\mathsf{if}\ P\theta\ \mathsf{then}\ t_i\theta\ \mathsf{else}\ s_j\theta))]\!] \Longrightarrow \mathsf{False}\,,$$

which is the antecedent of $\lceil \mathcal{T}' \rceil$. □

References

Ayari, A. (1995). A reinterpretation of the deductive tableaux system in higher-order logic. Master's thesis, University of Saarbrücken.

Basin, D., Friedrich, S. (1999). Modeling a hardware synthesis methodology in Isabelle. *Formal Methods in Systems Design*, **15**(2):99–122.

Bates, J. L., Constable, R. L. (1985). Proofs as programs. *Transactions on Programming Languages and Systems(TOPLAS)*, **7**(1):113–136.

Bibel, W. (1980). Syntax-directed, semantics-supported program synthesis. *Artificial Intelligence*, **14**:243–261.

Bird, R. (1989). Lectures on constructive functional programming. In *Constructive Methods in Computing Science*. Springer-Verlag.

Broy, M. (1978). A case study in program developement: sorting. Technical Report TUM-INFO-11-78-31-350, Technische Universität München, Institut für Informatik.

Burstall, R. M., Darlington, J. (1977). A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, **24**(1):44–67.

Chang, C.-L., Lee, R. C.-T. (1973). *Symbolic logic and mechanical theorem proving.* Academic Press.

Coen, M. D. (1992). Interactive program derivation. Technical Report 272, Cambridge University Computer Laboratory, Cambridge.

Coquand, T., Huet, G. (1988). The calculus of constructions. *Information and Computation*, **76**:95–120.

Deville, Y., Lau, K.-K. (1993). Logic program synthesis. *Journal of Logic Programming*, **12**.

Gordon, M. J. C., Melham, T. F. (1993). *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press.

Green, C. (1969). Application of theorem proving to problem solving. In *Proceedings of the IJCAI-69*, pages 219–239.

Manna, Z., Waldinger, R. (1980). A deductive approach to program synthesis. *TOPLAS*, **2**(1):90–121.

Manna, Z., Waldinger, R. (1981). Deductive synthesis of the unification algorithm. In *Science of Computer Programming*, pages 5–48.

Manna, Z., Waldinger, R. (1985). The origin of the binary-search paradigm. In Joshi, A., editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 222–224, Los Angeles, CA. Morgan Kaufmann.

Manna, Z., Waldinger, R. (1992). Fundamentals of deductive program synthesis. *IEEE Transactions on Software Engineering*, **18**(8):674–704.

Manna, Z., Waldinger, R. (1993). *The Deductive Foundations of Computer Programming.* Addison-Wesley, New York, N.Y.

Martin-Löf, P. (1982). Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam. North Holland.

Paulson, L. C. (1994a). A fixedpoint approach to implementing (co)inductive definitions. In Bundy, A., editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 148–161, Nancy, France. Springer-Verlag LNAI 814.

Paulson, L. C. (1994b). Isabelle: A generic theorem prover. *Lecture Notes in Computer Science*, **828**:xvii – 321.

Smith, D. R. (1985). The top-down synthesis of divide and conquer algorithms. *Artificial Intelligence*, **27**(1):43–96.

Walther, C. (1991a). *Automatisierung von Terminierungsbeweisen.* Vieweg Verlag, Braunschweig.

Walther, C. (1991b). Mathematical induction. In Shapiro, S., editor, *Encyclopedia of Artificial Intelligence.* John Wiley and Sons, New York.