Generic System Support for Deductive Program Development

Abdelwaheb Ayari*
David A. Basin*

ABSTRACT We report on a case study in using logical frameworks to support the formalization of programming calculi and their application to deduction-based program synthesis. Within a conservative extension of higher-order logic implemented in the Isabelle system, we derived rules for program development that can simulate those of the deductive tableau proposed by Manna and Waldinger. We have used the resulting theory to synthesize a library of verified programs, focusing on sorting algorithms. Our experience suggests that the methodology we propose is well suited both to implement and use programming calculi, extend them, partially automate them, and even formally reason about their correctness.

1 Introduction

Over the last few decades, a variety of methodologies for deductive software synthesis, transformation, and refinement from specification have been suggested, e.g., [4, 5, 8, 9, 12]. Our research investigates general frameworks that support such program development formalisms. That is, how can a framework be used to embed calculi in correctness preserving ways, be applied to the construction of programs hand in hand with their correctness proofs (e.g., synthesis as opposed to verification), simplify and extend previously proposed program development formalisms, and partially or even totally automate program construction. We are currently exploring these questions in the context of calculi for functional program development (reported here), logic programs [1, 3] and circuit synthesis. This work includes support for automated program construction [10, 11].

In this report we focus on the use of a formal metatheory to support the construction of proven correct functional programs. Our work is based on a methodology and calculus, the *deductive tableau system*, of Manna and Waldinger [12], which is a kind of first-order proof system proposed for

*Max-Planck-Institut für Informatik, Im Stadtwald, D-66123, Saarbrücken, Germany. Email: {abdu, basin}@mpi-sb.mpg.de

the synthesis of functional programs. Using the Isabelle system, a logical framework developed by Paulson [13], we have recast the deductive tableau as a formal theory that conservatively extends higher-order logic. This is done by formally deriving (as opposed to axiomatizing) proof rules that can simulate deductive tableau derivations. Moreover, deductive tableau proofs construct witnessing functions for proofs of $\forall/\exists$ formulae and, in our work, this is simulated by using higher-order metavariables to stand-in for these witnessing functions; these variables are incrementally instantiated to programs by applying proof rules using higher-order resolution. Resolution, controlled by the interactive or tactic (these are programs which construct proofs) guided application of proof rules, gives us a means not only to verify programs, but also to interactively construct them during proof.

Although our implementation uses a general purpose logical framework that supports a different kind of proof construction (natural deduction) than the deductive tableau, we can use our theory to simulate program derivations possible in the deductive tableau setting. However, because our theory is based on full higher-order logic and Isabelle supports operations on metavariables, we have considerably more flexibility in program development than the original deductive tableau framework offers. For example, we can perform splitting operations on subgoals that are not admitted in Manna and Waldinger's setting (they use nonclausal resolution and simplification to, essentially, operate under positively occurring conjunctions or negatively occurring disjunctions). Another example is that the use of higher-order logic and the Isabelle inductive data-type package allows us to construct recursive programs using well-founded induction where we use resolution to construct well-founded orderings during proofs (this arises in showing the termination of synthesized functions). This leads to a more general approach to induction than is possible in the deductive tableau where inductions are restricted to fixed collections of axiomatized orders. Moreover we gain the flexibility to develop programs first and show their termination later.

Our practical experience with our theory has been positive. Construction of our initial Isabelle theory (definitions, derivation of rules, and supporting tactics) directly utilized the distributed Isabelle HOL theory and standard tactics and, as a result, took only a few days; afterwards, we could immediately apply it to problems of interest. We have reconstructed many of Manna and Waldinger's published examples, and in particular we have synthesized a variety of standard sorting algorithms (e.g., quicksort, merge sort, and insertion sort). All proofs have been formally carried out in Isabelle; full machine checked proofs scripts may be found in [2]. This should be contrasted to the work of Manna and Waldinger; the tableaux in their published papers are generated by hand, since, for many years, they lacked an implementation.

More generally, our work suggests the potential of using frameworks like Isabelle for building prototype implementations and carrying out experiments with such deduction-based development formalisms. The use of a formal metalogic and tactic based prover like Isabelle helps to conceptually clarify the nature of the formalism under consideration (e.g., deductive tableau) since its formalization naturally stratifies into layers: the underlying logic, the derived rules, and the tactics which implement strategies for rule application and program development. Moreover the use of a framework offers certain advantages over a customized implementation; for example, we can formally establish the correctness of proof rules by deriving them. Also, by building our theories on top of standard logics, we can directly benefit from work on decision procedures, rewriting, and other automation for constructing proofs in these logics.

The remainder of this paper is organized as follows. In sections 2 and 3 we briefly review Isabelle and the deductive tableau system. Next, in section 4 we show how the deductive tableau can be interpreted in an Isabelle theory. That is, how each rule of the deductive tableau corresponds to a combination of derived rules and tactics in Isabelle. In section 5 we describe program development by presenting parts of a formal development of quicksort. In section 6 we draw conclusions.

2 Isabelle

Our work requires a theorem prover that implements higher-order logic and supports higher-order unification. We have chosen the Isabelle logical framework [13] although other frameworks such as Pfenning's ELF [14] would also suffice. Isabelle is an interactive theorem prover which serves as a logical framework; this means that its logic is a metalogic in which object logics (e.g., first-order logic, set theory, etc.) are encoded. Isabelle's metalogic is a minimal higher-order logic supporting polymorphic typing. Object logics are encoded by declaring their signature and proof rules. Afterwards, proofs are interactively constructed by applying rules using higher-order resolution. Proof construction may be automated by writing tactics to apply decision procedures, simplifiers, and other kinds of proof construction strategies.

Isabelle's metalogic is based on the universal/implicational fragment of higher-order logic. Universal quantification in the metalogic is represented by `!!` and implication by `==>` . Isabelle manipulates *rules* which are objects of the form

$$\texttt{[|}\phi_1;\ldots;\phi_n\texttt{|] ==>}\ \ \phi\,,$$

where the notation $\texttt{[|}\phi_1;\ldots;\phi_n\texttt{|]}$ `==>` ϕ is shorthand for the iterated implication ϕ_1 `==>` ... `==>` (ϕ_n `==>` ϕ). A rule can also be viewed as a proof-state, for the purposes of top-down proof construction, where ϕ is the goal to be established and the ϕ_i represent the subgoals to be proven. Under this view an initial proof state has the form ϕ `==>` ϕ, i.e., it has one subgoal, namely ϕ. The final proof state *is itself* the desired theorem ϕ.

Proof construction in Isabelle is based on higher-order resolution, which is roughly analogous to resolution in Prolog. That is, given a proof state with subgoal ψ and a rule like the one above, then we higher-order unify ϕ with ψ. If this succeeds, then unification yields a substitution σ and the proof state is updated by applying σ to it, replacing ψ with the subgoals $\sigma(\phi_1), \ldots, \sigma(\phi_n)$. Note that since unification is used to apply rules, the proof state itself may contain metavariables. We will show that this supports program transformation and synthesis during proof.

Finally, note that Isabelle supports the hierarchical development of logics and theories and a number of such theories come distributed with the system, along with supporting tactics. For our work we used higher-order logic extended with a theory of inductively defined data-types (in which standard types like lists were defined) and well-founded induction.

3 Deductive Tableau System

Here we give a brief overview of the deductive tableau system. Full details may be found in [12].

A tableau proof starts with a specification of the form $\forall x.\ \exists z.\ Q(x, z)$, where Q is a first-order formula. Such a specification states that for each input x there is an output z satisfying the formula $Q(x, z)$ called the *input-output relation*.[1] A specification is turned into an *initial tableau* by skolemization to $Q(a, f(a))$, where a is an eigenvariable and f is a skolem function which denotes the desired program. Then the term $f(a)$, occurring in the goal $Q(a, f(a))$, is replaced with a new variable, say z, which yields the following initial tableau.

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

This tableau expresses the goal that $Q(a, z)$ is to be proven in some underlying theory, e.g., a first-order theory of lists. Furthermore, the assertion column of the tableau can be augmented with formulae which are known to be valid in the theory. The desired program has, at this stage, the form $f(a) = z$ where the *output variable* z represents the current body of the function f.

[1] In general, there may be more than one universally quantified input variable and existentially quantified output variable. However, it will simplify discussion and notation to consider only one of each. Our implementation supports the general case.

A proof is constructed by applying deduction rules which add new rows to the tableau. Each step may incrementally instantiate an output variable. A deduction is successful when the truth value *false* (respectively *true*) appears in the column *assertions* (respectively *goals*). If t is the term appearing in the output column of this final row, then the desired program is $f(a) = t$, where t may depend on a. There are syntactic side conditions (involving so called *primitive expressions*) which guarantee that t is built in an executable way.

Note that, if we ignore the output columns, then the deductive tableau system simply constitutes a calculus for proving the validity of certain first-order sentences in a given theory. Moreover, there is a close relationship between deductive tableau and standard sequent calculi and semantic tableau (c.f. [7]): formulae in the goal column correspond to *goal formulae* on the right-hand side of the sequent, and those in the assertion column correspond to *hypothesis formulae* on the left-hand side of a sequent. So, for example, the conditions above for a successful deduction correspond to a sequent being provable when one of the hypotheses are false or goals are true. We will see though that the rules of the deductive tableau are different from standard sequent/tableau rules because they are based on a non-clausal resolution rule. Further, the deductive tableau system does not allow splitting tableaux; there are no rules that correspond to the sequent calculus rules for conjunction-introduction or disjunction-elimination, which cause branching in proofs.

Nonclausal Resolution

Formulae in different rows of a tableau are combined using nonclausal resolution: this allows simplification and other kinds of logical manipulation. Nonclausal resolution corresponds to a case analysis with simplification and it introduces a conditional term in the output entries. The deductive tableau system contains several such rules depending on whether formulae come from goal or assertion columns.

As an example, consider the following two rows:

$A[P]$		s
	$G[Q]$	t

In the first row $A[P]$ denotes the assertion A with a subformula P. In the second row $G[Q]$ denotes the goal G with a subformula Q. Nonclausal resolution, when applied to these rows, generates the new row

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

where θ is the most-general unifier of P and Q. The output entry is a conditional term built using the resolved rows' output entries. There are additional rules for resolution between two assumptions or two goals and other rules derivable from nonclausal resolution.

Induction Rule

Induction is used to develop recursive programs. To prove a goal $Q(a, f(a))$, for an arbitrary element a, we may introduce an *induction hypothesis* that states that the goal holds for all x strictly smaller than a with respect to a well-founded ordering $<_w$. The induction rule introduces an induction hypothesis as an assertion in the tableau.

if $(x <_w a)$ *then* $Q(x, f(x))$		

Note that the induction hypothesis contains occurrences of the function symbol f, which denotes the function that we are trying to compute. If the induction hypothesis is used in the proof, terms of the form $f(t)$ can be introduced in the output entries, yielding recursion. To apply induction, a user must choose a well-founded relation $<_w$, which is defined in the current theory, e.g., the *less-than* relation over natural numbers.

4 Interpretation in Isabelle

As discussed above, the deductive tableau is a first-order proof system. Although Isabelle contains a well-developed theory of first-order logic, we have chosen to model the deductive tableau system in an Isabelle formalization of higher-order logic. This gives us more power and flexibility in formalizing and reasoning about well-founded relations. In particular, it allows us to synthesize well-founded relations during proof, in a way analogous to how we synthesize programs.

To simulate deductive tableau it suffices to show how we mimic the initial tableau and application of the proof rules. For the initial tableau, one begins with a specification $\forall a.\ \exists z.\ spec(a, z)$, which specifies some program f. Slipping into Isabelle syntax, we begin our Isabelle proofs by typing

```
?H --> ! a.spec(a,?f(a)) .
```

We use `typewriter font` for Isabelle syntax. In Isabelle's HOL, the operator `!` represents universal quantification, and `-->` implication.[2] Variables like `f` and `H` preceded by `?` are metavariables, which can be instantiated by unification during resolution. We will call the metavariable `?H` an *output metavariable*; we use it to represent the output column of the tableau where

[2]Remember that these are connectives in the declared object logic, and should not be confused with the connectives of Isabelle's metalogic.

the constructed function `?f` is "accumulated". This output metavariable allows us not only to record in the proof-state the definition of the function `?f`, but it also reflects the logical meaning of proof in a context extended by this definition: the specification follows under the definition of `?f`. Our example in section 5 should make this clear. To enforce that output variables are only instantiated with formulae that represent executable programs, we, like in the deductive tableau, incorporate syntactic side conditions on proofs which are enforced by our tactics.

Subgoals play a role in our proofs analogous to subtableaux. Proofs proceed by refining subgoals until no more remain. We have derived rules which can be applied by tactics to simulate deductive tableau steps. Here we briefly discuss our rules corresponding to nonclausal resolution and induction.

Nonclausal Resolution

Nonclausal resolution allows subformulae from two rows to be unified and then replaced by the truth values *true* and *false* respectively; furthermore, its application builds conditional terms. In Isabelle, we define `if(C,s,t)` to be the term equal to `s` when `C` is equivalent to `True` and `t` otherwise. We may then derive the following rule which splits on the case `C` and its negation `~C`.

```
[| ?C ==> ?P(?S); ?~C ==> ?P(?T) |] ==> ?P(if(?C,?S,?T))
```

This rule says that if we have some predicate `?P(x)`, then we can prove `?P(if(?C,?S,?T))`, when we can prove `?C ==> ?P(?S)` and `?~C ==> ?P(?T)`. Its application can simulate the nonclausal resolution rule of the previous section where `?C` corresponds to the unifiable formulae P and Q and `?S` and `?T` are the output terms for the rows $A[P]$ and $G[Q]$.

Recall that Isabelle is a logical framework and rules are formulae in Isabelle's metalogic; the above case-split rule is not axiomatized, but instead is formally derived. The derivation is simple and consists of expanding the definition of `if` and propositional reasoning.

This derived rule can be applied for program synthesis and constructs a program containing an if-then statement. In particular, suppose we are synthesizing a program specified by `Spec(x,?f(x))`. Application of the above proof rule by higher-order resolution will unify `?P` to `Spec` and `?f(x)` to the function `if(?C(x),?S(x),?T(x))`. Instances for `?S(x)` and `?T(x)` (as well as `?C(x)`) will be subsequently synthesized by proving the two subgoals which will correspond to `Spec(x,?S(x))` and `Spec(x,?T(x))`. This use of rules for synthesis will become clearer in section 5.

Induction

The tableau induction rule adds induction hypotheses as assertions in the tableau proofs. We model this with a suitably formulated induction rule.

Isabelle's higher-order logic comes with a theory of relations and inductive definitions. With it, we can directly manipulate relations and reason about their well-foundedness. The well-foundedness of a relation `?r` is defined as follows.

```
wf(?r) == !P.(!x.(!y. <y,x>:?r --> P(y)) --> P(x))--> (!x.P(x))
```

The notation `<y,x>:?r` is a proposition that states that the pair `<y,x>` belongs to the relation `?r`.

From the definition of well-foundedness, it is a simple matter to formally derive the following well-founded induction rule.

```
[| !x. (!y. <y, x>:?r --> ?P(y)) --> ?P(x); wf(?r) |] ==> !x. ?P(x)
```

This rule formalizes the standard rule of noetherian induction over a well-founded ordering `r`. It says that to show `?P` holds for every `x` we can show `?P(x)` under the assumption of `?P(y)` where `y` is less than `x` in the ordering `?r`. The second assumption insists that `?r` is well-founded.

We derive a specialization of this rule for constructing a recursive program `?f` by well-founded induction.

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

We indicated above how the case-split rule is used to synthesize if-then branches. Similarly, the induction rule builds recursive programs. The conclusion `?Spec(l,?f(l))` is suitable for unifying against the translation (into our framework) of deductive tableau specifications. The additional assumption `!l. ?f(l) = ?fbody(l)` sets up a definition of `?f` which can be instantiated during the subsequent proof. Applying the induction hypothesis instantiates `?fbody` with a recursive call to `?f` and, moreover, this call must be on some smaller argument in the well-founded order defined by `?r`; hence this rule builds only terminating programs and `?r` represents an ordering upon which recursive calls are smaller. Moreover, when we resolve with this rule, we are not required to give, up front, the ordering `?r`; instead, just like with metavariables representing programs, we can incrementally synthesize this relation during subsequent proof steps.

Derived Rules from HOL

The above rules suffice to directly model deductive tableau derivations. Full details of the simulation are provided in [2] and it is proved there (Theorem 6, page 85) that with these rules and with tactics which apply them it is possible to simulate any deductive tableau derivation so that there is a direct correspondence between deductive tableau proof steps and proof steps in Isabelle and moreover the identical program is constructed.

But there is an important difference: in the Isabelle setting we are not constrained to use only the above proof rules and we can construct proofs, which synthesize programs, in ways that have no analog in the deductive tableau system. We can use standard proof rules and tactics for higher-order logic, as well as any derived rules, and this gives us considerable flexibility in constructing proofs. For example, given a goal formula which is a conjunction (or a hypothesis which is a disjunction) in the deductive tableau setting one cannot split the tableau to subtableaux corresponding to each conjunct (or disjunct). Instead one must use nonclausal resolution to perform simplification to true under the conjunctions (or to false under the disjunctions); this isn't very natural and leads to complicated formulae. In our setting, we can directly decompose conjunctions (or disjunctions) by applying the rule for conjunction introduction (or disjunction elimination). Another example of additional flexibility is using metavariables to delay giving well-founded orderings when applying the induction rule.

5 Development of Functional Programs

We have used our theory to develop in Isabelle a number of functional programs including many of Manna and Waldinger's published examples and, in particular, a number of sorting algorithms. Here we consider quicksort. Our entire proof required 23 interactive steps and full details may be found in [2] including: the entire proof script, the supporting theory, and the tactics used in assisting proof construction. In the following we provide some background and a few snapshots from our Isabelle session that illustrate case-splitting, induction, and reasoning about termination.

Sorting may be naturally specified as a relation between an input list and an output list, where the output is an ordered permutation of the input.

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

In our formalization, `l` is of type list over a domain whose elements admit a total order $\leq$.[3] The theory we employ for our development comes with the Isabelle distribution: HOL augmented with a theory of inductive definitions in which lists are defined. It employs standard notation for lists; the empty list is represented by `[]`, the symbol `@` denotes the append function, `hd` and `tl` are the head and tail functions.

Our definitions of the predicates `perm` and `ordered` are taken from Manna

[3] This is enforced through the use of type-classes; Isabelle suppresses display of this information.

and Waldinger's development, and formalize the standard permutation and ordered relation on lists.

```
perm([],[])
perm(l1@[u]@l2,t1@[u]@t2) = perm(l1@l2,t1@t2)
perm(l1,l2) --> !a. member(a, l1) = member(a,l2)
ordered([])
ordered([a])
ordered(Cons(a,Cons(b,t))) = (a <= b) & ordered(Cons(b,t))
```

After we enter the above goal, Isabelle responds with the goal to be proven (first line), and subgoals (in this case only 1, the initial goal) which must be proven to establish it:

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

Induction

Our first proof step is induction on `l` and is invoked by typing

```
by (INDTAC [("f","qsort")] 1);
```

This executes a tactic, `INDTAC`, which applies the induction schema given in section 4: after some preprocessing, this tactic uses higher-order unification to resolve the goal `!l. perm(l, ?sort(l)) & ordered(?sort(l))` with `!l. ?Spec(l, ?f(l))`, which is the conclusion of the induction rule. Resolution succeeds with `?Spec(l, m)` unified with `perm(l, m) & ordered(m)` and `?f(l)` unified with `sort(l)`, and it produces three new subgoals, corresponding to the three assumptions of the induction schema. However, `INDTAC` immediately discharges the first by unifying it with `?H`. Hence, our output metavariable begins accumulating a recursive definition; this is precisely the role that `?H` serves.[4] Our tactic also takes an argument which names the function being synthesized `qsort`. After all this, Isabelle responds with the new proof state:

```
(! l. qsort(l) = ?fbody(l)) & ?H1 -->
(! l. perm(l, qsort(l)) & ordered(qsort(l)))
 1. !!l. [| ?H1;
           ! t. <t, l> : ?r --> perm(t, qsort(t))
                                & ordered(qsort(t)) |]
         ==> perm(l, ?fbody(l)) & ordered(?fbody(l))
 2. wf(?r)
```

[4] To allow the possibility of synthesizing multiple programs (as in quicksort, where we synthesize two additional auxiliary functions) the tactic firsts "duplicates" `?H` yielding a new output variable `?H1`. This is possible because if we have an hypothesis `?H` then we can instantiate it with `?H & ?H1`. Instantiation is performed by resolution with `&`-elimination and results in the new assumptions `?H` and `?H1`. Hence, we use derived rules to simulate proof under a growing context of definitions.

We are left with two subgoals to prove. The first says we must develop a program `?fbody` (the body of `qsort`) under the assumption that we have a program `qsort` which sorts all lists smaller (under `?r`) than `l`. The second insists that `?r` is a well-founded order.

Case Split

Quicksort works by partitioning a non-empty list into those elements greater than and less than some element (usually the head of the list) and sorting the partitions recursively; empty lists are trivially sorted. This analysis requires a case split on whether `l` is empty or not. Hence, we resolve the first subgoal against the case split rule of section 4 and specify that splitting condition is `l = []`. Isabelle returns the proof state:

```
(! l. qsort(l) = if(l = [], ?S(l), ?T(hd(l), tl(l)))) & ?H1 -->
(! l. perm(l, qsort(l)) & ordered(qsort(l)))
 1. !!l. [| ?H1;
            ! t. <t, l> : ?r --> perm(t, qsort(t))
                                 & ordered(qsort(t));
            l = [] |] ==>
         perm(l, ?S(l)) & ordered(?S(l))
 2. !!l. [| ?H1;
            ! t. <t, l> : ?r --> perm(t, qsort(t))
                                 & ordered(qsort(t));
            l ~= [] |] ==>
         perm(l, ?T(hd(l), tl(l))) & ordered(?T(hd(l), tl(l)))
 3. wf(?r)
```

Resolution instantiated `?fbody` 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.

Recursive Step and Partitioning

Let us skip ahead 7 steps to see how the use of the induction hypothesis generates recursive programs.

```
(! l. qsort(l) =
         if(l = [], l,
            ?t1(hd(l), tl(l)) @ [hd(l)] @ ?t2(hd(l), tl(l))))
      & ?H1 -->
(! l. perm(l, qsort(l)) & ordered(qsort(l)))
 1. !!l. [| ?H1;
            ! t. <t, l> : ?r --> perm(t, qsort(t))
                                 & ordered(qsort(t));
            l ~= [] |] ==>
         perm(tl(l), ?l1(l) @ ?l2(l)) &
         le(hd(l), ?t2(hd(l), tl(l))) & gr(hd(l), ?t1(hd(l), tl(l)))
```

```
 2. !!l. [| ?H1;
            ! t. <t, l> : ?r --> perm(t, qsort(t))
                                 & ordered(qsort(t));
            l ~= [] |] ==>
         perm(?l1(l), ?t1(hd(l), tl(l))) & ordered(?t1(hd(l), tl(l)))
 3. !!l. [| ?H1;
            ! t. <t, l> : ?r --> perm(t, qsort(t))
                                 & ordered(qsort(t));
            l ~= [] |] ==>
         perm(?l2(l), ?t2(hd(l), tl(l))) & ordered(?t2(hd(l), tl(l)))
4. wf(?r)
```

The embryonic form of quicksort has taken shape. We have already solved the previous subgoal 1 which corresponded to the `l = []` case; we have applied simplification which replaced `?S(l)` with the empty list `l`. We then simplified the `l ~= []` case which resulted in the first 3 subgoals given above. The first specifies that `tl(l)` can be decomposed into two lists `?l1(l)` and `?l2(l)` and everything in the first list is less than or equal to `hd(l)` (this is defined by the predicate `le`) and the second contains only elements greater than `hd(l)` (stated by `gr`). The second and third subgoal state that there are lists computed by `?t1` and `?t2`, which are permutations of `?l1(l)` and `?l2(l)`, and are both sorted.

At this point, our generic sorting specification has been specialized into a specification for quicksort. This is not by chance: In the previous steps we used our "programmer's intuition" to interactively guide the derivation towards this specialization by exploiting previously proven properties of permutation and ordering. The situation in the deductive tableau setting is analogous: synthesis means guiding the proof in a way that constructs the desired program in the output column.

Subgoals 2 and 3 are particularly easy to solve; we direct Isabelle to unify both with their induction hypotheses.

```
(! l. qsort(l) =
      if(l = [], l,
         qsort(?t(hd(l), tl(l))) @ [hd(l)] @ qsort(?ta(hd(l), tl(l)))))
      & ?H1 -->
(! l. perm(l, qsort(l)) & ordered(qsort(l)))
 1. !!l. [| ?H1;
            ! t. <t, l> : ?r --> perm(t, qsort(t))
                                 & ordered(qsort(t));
            l ~= [] |] ==>
         perm(tl(l), ?t(hd(l), tl(l)) @ ?ta(hd(l), tl(l))) &
         le(hd(l), ?ta(hd(l), tl(l))) & gr(hd(l), ?t(hd(l), tl(l)))
 2. !!l. [| ?H1; l ~= [] |] ==> <?t(hd(l), tl(l)), l> : ?r
 3. !!l. [| ?H1; l ~= [] |] ==> <?ta(hd(l), tl(l)), l> : ?r
 4. wf(?r)
```

As a result, `?t1` and `?t2` have been replaced by recursive calls to quicksort. However, subgoals 2 and 3 were not completely solved; there are residual proof obligations. We must later show that the function `?t` (in goal 2) and `?ta` (in goal 3) are applied to arguments "less" then `l` in the ordering `?r`, which must be well-founded if goal 4 is to be provable. 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 ordering `?r`.

Auxiliary Synthesis and Termination

In the remaining proof we synthesize the functions for `?t` and `?ta` specified in subgoal 1; these are the functions which pick out elements in the tail of `l` which are less than or equal (or, in the case of `?ta`, greater than) the head of `l`. The instantiation of these functions is propagated to the remaining goals; for example, goal 2 becomes

```
!!l. [| l ~= [];
        ! x xa. lesseq(x, xa) =
          if(xa = [], xa, if(hd(xa) <= x,
             [hd(xa)] @ lesseq(x, tl(xa)), lesseq(x, tl(xa))))
     |] ==>  <lesseq(hd(l), tl(l)), l> : ?r
```

and to prove this we must construct a relation `?r` which contains the pair `<lesseq(hd(l), tl(l)), l>` for all non-empty `l`. Afterwards, we must show that this relation is well-founded (goal 4).

Our termination proofs exploits our use of higher-order logic. Relations are terms in higher-order logic, just like programs, and the explicit use of metavariables allows us to delay commitment to these relations when using the induction hypothesis and synthesize them later by resolution, just like programs. In this example, resolution instantiates `?r` with an ordering true when the first list contains fewer elements than the second, i.e., the relation `{<x, y> | length(x) < length(y)}`. This ordering explains in what sense arguments of recursive calls to quicksort are smaller.

This ordering is shown to be well-founded by construction. We have proven in our theory that certain relations are well-founded and that there are constructors for building new well-founded relations from given well-founded relations. For example, a well-founded relation `s` and a function `f` induce a well-founded relation containing all pairs `<f(x),f(y)>` in the relation `s`. Formally, we have shown that for any well-founded `s`, the relation denoted by `induced_rel(f,s)` is well-founded, where the new relation is defined as follows.

```
induced_rel(f, s) == {q. Ex x y.(q=<x ,y> & <f(x),f(y)>:s)}
```

Given this development, the above ordering is well-founded because we define it as `induced_rel(length, <)`.

Final Proof State

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

```
[|! l. qsort(l) =
       if(l = [], l,
          qsort(lesseq(hd(l), tl(l))) @ [hd(l)]
            @ qsort(greater(hd(l), tl(l))));
  ! x xa. greater(x, xa) =
        if(xa = [], xa,
           if(hd(xa) <= x, greater(x, tl(xa)),
                           [hd(xa)] @ greater(x, tl(xa))));
  ! x xa. lesseq(x, xa) =
        if(xa = [], xa,
           if(hd(xa) <= x, [hd(xa)] @ lesseq(x, tl(xa)),
                           lesseq(x, tl(xa)))) |]
==> (! l. perm(l, qsort(l)) & ordered(qsort(l)))
```

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

Taking Stock

The above proof sketch suggests how algorithms like quicksort can be incrementally developed by interactively applying tactics, which in turn apply proof rules using higher-order resolution. The resulting program is guaranteed to be correct because tactics applied only primitive rules or rules we previously derived. Moreover the derivation separates partial correctness and termination by delaying the choice of the well-founded relations used in showing termination until after all functions have been synthesized. Finally, although perhaps not obvious from the snapshots, our tactics incorporate procedures based on rewriting and propositional reasoning, which make substantial use of standard tactics for HOL to partially automate trivial kinds of reasoning.

6 Related Work and Conclusions

Our case study, highlighting some of the results in [2], is, to our knowledge, the first analysis of how the deductive tableau system can be embedded and extended in a standard theorem prover. It provides evidence that frameworks like Isabelle offer advantages not present in a specialized implementation. First, because Isabelle is a logical framework, we can represent rules of objects logics and prove that they are derivable. Hence, we can formally establish the correctness of rules for deductive tableau style program development. This helps us separate and clarify the underlying logic (HOL), the derived rules, and what tactics and support are necessary to apply the rules. Second, higher-order resolution allows us to directly develop programs along side their correctness proofs either by simulating deductive tableau proofs or taking advantage of the greater flexibility offered by proof in a higher-order metalogic with metavariables. For example we can directly formalize and manipulate well-orderings during proofs. Finally, we can directly utilize standard tactics and libraries that came with the system. This substantially reduces the time needed to create a usable theory with sufficient automation support (e.g., rewriting and propositional reasoning) suitable for carrying out Manna and Waldinger's examples and our own experiments.

The work closest to ours is Coen's [6] who developed his own theory called classical computational logic. He was motivated by the deductive tableau and Constructive Type Theories and his work supports deductive synthesis style proofs in his own classical logic. His goals were rather different than ours: he was not trying to directly model a proposed formalism, but rather create his own specialized theory, with its own advantages and disadvantages. For example, on the system side he had to develop his own tactics for simplification and the like; on the theory side he had to show the correctness of his specialized logic with respect to some appropriate semantics. Also relevant is the work of Regensburger [15] who developed a variant of LCF (Scott's logic of computable functions) as a conservative extension of higher-order logic and applied this to reasoning about programs operating on inductively and co-inductively defined data-types such as streams.

7 References

[1] Penny Anderson and David Basin. Deriving and applying logic program transformers. In *Algorithms, Concurrency and Knowledge (1995 Asian Computing Science Conference)*, pages 301–318, Pathumthani, Thailand, December 1995. Springer-Verlag, LNCS 1023.

[2] Abdelwaheb Ayari. A reinterpretation of the deductive tableaux system in higher-order logic. Master's thesis, University of Saarbrücken, 1995. Available at http://www.mpi-sb.mpg.de/~abdu/dts.ps.Z.

[3] David Basin. Logic frameworks for logic programs. In *4th International Workshop on Logic Program Synthesis and Transformation, (LOPSTR'94)*, pages 1–16, Pisa, Italy, June 1994. Springer-Verlag, LNCS 883.

[4] CIP System Group: F. L. Bauer et al. *The Munich Project CIP, Volume II: The Program Transformation System CIP-S*, volume 292 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[5] Rod M. Burstall and Joseph A. Goguen. Putting theories together to make specifications. In *5th IJCAI*, pages 1045 – 1058, Boston, Mass, 1977.

[6] Martin David Coen. Interactive program derivation. Technical Report 272, Cambridge University Computer Laboratory, Cambridge, November 1992.

[7] Melvin Fitting. *First-Order Logic and Automated Theorem Proving.* Springer-Verlag, New York, 1990.

[8] Berthold Hoffmann and Bernd Krieg-Brückner (Eds.). *Program Development by Specification and Transformation.* Springer LNCS 680, 1993.

[9] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, pages 31–55, 1978.

[10] Ina Kraan, David Basin, and Alan Bundy. Middle-out reasoning for synthesis and induction. To appear in the Journal of Automated Reasoning.

[11] Ina Kraan, David Basin, and Alan Bundy. Middle-out reasoning for logic program synthesis. In *10th International Conference on Logic Programming (ICLP93)*, pages 441–455, Budapest Hungary, 1993.

[12] Zohar Manna and Richard Waldinger. Fundamentals of the deductive program synthesis. *IEEE Transactions on Software Engineering*, January 1992.

[13] Lawrence C. Paulson. *Isabelle : a generic theorem prover; with contributions by Tobias Nipkow.* LNCS-828. Springer, Berlin, 1994.

[14] Frank Pfenning. Logic programming in the LF logical framework. In *Logical Frameworks*, pages 149 – 181. Cambridge University Press, 1991.

[15] Franz Regensburger. *HOLCF: Eine konservative Erweiterung von HOL um LCF.* PhD thesis, Technical University, Munich, 1984.