Interpretation of the Deductive Tableau in HOL

Abdelwaheb Ayari and David A. Basin
Max-Planck-Institut für Informatik, Im Stadtwald
D-66123, Saarbrücken, Germany
Email: {abdu, basin}@mpi-sb.mpg.de
Phone: (49) (681) 302-5435 Fax: (49) (681) 302-5401

Abstract

Our research investigates frameworks supporting the formalization of programming calculi and their application to deduction-based program synthesis. Here we report on a case study: within a conservative extension of higher-order logic implemented in the Isabelle system, we derived rules for program development which 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 in particular 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, 7, 3, 6, 10]. 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 [2] (our work includes support for completely automated program construction [9, 8]), and circuit synthesis.

In this report we present a case-study: the *deductive tableau system* of Manna and Waldinger [10], which is a kind of first-order proof system proposed for the synthesis of functional programs. Using the Isabelle system, a logical framework developed by Paulson [11], we have recast the deductive tableau as formal theory that conservatively extends higher-order logic. This is done by formally deriving (as opposed to axiomatizing) proof rules which can simulate deductive tableau derivations. Moreover, deductive tableau proofs construct witnessing functions for proofs of $\forall/\exists$ formulas and, in our work, this is simulated by using higher-order metavariables to stand-in for witnessing functions; these variables are then 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. 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. 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 possible in the deductive tableau where inductions are restricted to fixed axiomatized orders.

Our practical experience in carrying out proofs has been very positive. Construction of our initial Isabelle theory (definitions, derivation of rules, 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 many 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 and details may be found in [1]. 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. Our work suggests the potential of using frameworks like Isabelle for building prototype implementations and carrying out experiments with such development formalisms. Moreover, it can offer real advantages over a customized implementation; for example, we formally establish the correctness of proof rules by deriving them. We also benefit from work on decision procedures, rewriting, and other automation for proof construction in standard theories.

Organization In § 2 we provide a brief overview both to Isabelle and the deductive tableau. After, in § 3 we show how the tableau formalism can be interpreted in an Isabelle theory. In § 4 we describe program development and after draw conclusions.

§ 2 Background to Interpretation

Due to space considerations we only give very brief and simplified accounts of both deductive tableau and Isabelle. Full details on the former may be found in [10] and the latter in [11].

§ 2.1 Overview of Isabelle Our work requires a theorem prover that implements higher-order logic and supports higher-order unification. We have chosen Isabelle although other frameworks such as Pfenning's ELF 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 (minimal) higher-order logic. Universal quantification in the metalogic is represented by `!!` and implication by `==>` . Isabelle manipulates *rules* which are objects of the form

$$\texttt{[|}\phi_1\texttt{;} \ldots \texttt{;}\phi_n\texttt{|]} \texttt{==>} \ \phi\,, \tag{1}$$

where the notation [|$\phi_1; \ldots; \phi_n$|] ==> 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.

Isabelle supports proof construction through higher-order resolution, which is roughly analogous to resolution in Prolog. That is, given a proof state with subgoal ψ and a rule like (1), 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 development.

Finally, note that Isabelle supports the hierarchical development of logics and theories and a number of such theories come distributed with the system. 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.

§ **2.2 Deductive Tableau System** 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*. 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 tableau.

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

This tableau can be understood as follows: Our goal is to prove $Q(a, z)$ using assertions belonging to some underlying 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.

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* (resp. *true*) appears in the column *assertions* (resp. *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.

Nonclausal Resolution Formulas in different rows of a tableaux are combined using nonclausal resolution: this allows simplification and other kinds of logical manipulation. The effect of this rule corresponds to a case analysis (with simplification) and introduces a conditional term in the output entries. 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 in according 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 that is defined in the current theory, e.g., the *less-than* relation over natural numbers.

§ 3 Interpretation in Isabelle

The deductive tableau is presented as a kind of first-order proof system supporting program development. Although Isabelle contains a well-developed theory of first-order logic, we work with higher-order logic; this allows us to adequately formalize well-foundedness and reason about and construct well-founded relations during proofs.

In deductive tableau one begins with a specification $\forall a.\ \exists z.\ \mathit{spec}(a, z)$, which specifies some program f. Slipping into Isabelle syntax, we begin our Isabelle proofs by typing

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

We use `typewrite font` for Isabelle syntax. In Isabelle's HOL, the operator `!` represents universal quantification, and `-->` implication.[1] Variables like `f` and `H` preceeded by `?` are metavariables which may be instantiated by unification during resolution. We call the metavariable `H` an *output metavariable* because it represents the output column of the tableau where the constructed function `f` is "accumulated". This output metavariable allows us not only to record somewhere 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`. To enforce that output variables are only instantiated with formulas which 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. Full details of the simulation are provided in [1]; here we will briefly discuss our rules corresponding nonclausal resolution and induction.

[1] Remember that these are connectives in the declared object logic, and should not be confuse with the connectives `!!` and `==>` of Isabelle's metalogic.

§ **3.1 Nonclausal Resolution** Nonclausal resolution allows subformulas 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)`, we can prove `?P(if(?C,?S,?T))` provided we can prove both `?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 formulas 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 formulas 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 an program containing an if-then statement. In particular, suppose we are synthesizing a program to meet a particular specification `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 § 4.

§ **3.2 Induction** The tableau induction rule adds inductions 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 define well-foundedness and build well-founded relations. For example, we may define that a relation `?r` is well-founded as follows.

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

The notation `<y,x>:r` denotes that the pair `<x,y>` 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.

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

This rule says that to show `?P` holds for every `l` we can show `?P(l)` under the assumption of `?P(t)` where `t` is "less" than `l` 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 can be used for synthesizing 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. The use of the induction hypothesis will

instantiate `?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`: just like metavariables representing programs, we can incrementally synthesize this relation during subsequent proof steps.

§ 4 Development of Functional Programs

We have used our theory to develop a number of functional programs including many of Manna and Waldinger's published examples, and, in particular, sorting algorithms. Here we consider quicksort. Our entire proof required 23 interactive steps and full details may be found in [1] including: the entire proof script, the supporting theory, and the tactics used in assisting proof construction. In the following we provide a few snapshots from our Isabelle session that illustrate case-splitting and induction.

Sorting may be naturally specified as a relation between an input list and an output which is an ordered permutation.

```
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 $\leq$. Our definitions of the predicates `perm` and `ordered` are from Manna and Waldinger and formalize the standard permutation and ordered relation on lists. The theory we employ for our development (HOL augmented with a theory of inductive definitions in which lists are defined) 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.

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","quicksort")] 1);
```

This executes the tactic, `INDTAC`, which applies the induction schema given in § 3.2. In particular, after some preprocessing, it resolves 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 the unifier `?Spec(l, m) = perm(l, m) & ordered(m)` and `?f(l)=sort(l)` and 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.[2] Our tactic also takes an argument which names the function being synthesized `quicksort`. After all this, Isabelle responds with the new proof state:

[2] To allow the possibility of synthesizing multiple programs (as in quicksort) 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.

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

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 § 3.1 and specify that splitting condition is `l = []`. Isabelle returns the proof state:

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

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

```
(! l. quicksort(l) =
          if(l = [], l,
             ?t1(hd(l), tl(l)) @ [hd(l)] @ ?t2(hd(l), tl(l)))) & ?H1 -->
(! l. perm(l, quicksort(l)) & ordered(quicksort(l)))
 1. !!l. [| ?H1;
            ! t. <t, l> : ?r --> perm(t, quicksort(t)) & ordered(quicksort(t));
            l ~= [] |] ==>
         perm(tl(l), ?l1(l) @ ?l2(l)) &
         mini(hd(l), ?t2(hd(l), tl(l))) & maxi(hd(l), ?t1(hd(l), tl(l)))
 2. !!l. [| ?H1;
            ! t. <t, l> : ?r --> perm(t, quicksort(t)) & ordered(quicksort(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, quicksort(t)) & ordered(quicksort(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 = []`; we have applied simplification which replaced `?S(l)` with the empty list `l`. We then simplified the `l ~= []` case which resulted in the current first 3 subgoals. 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 mini) and the second contains only elements greater than hd(l) (stated by maxi). The second and third subgoal state that there are lists computed by ?t1 and ?t2 that are permutations of ?l1(l) and ?l2(l) which are both sorted. Hence, 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.

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

```
(! l. quicksort(l) =
      if(l = [], l,
         quicksort(?t(hd(l), tl(l))) @ [hd(l)] @ quicksort(?ta(hd(l), tl(l)))))) & ?H1 -->
(! l. perm(l, quicksort(l)) & ordered(quicksort(l)))
 1. !!l. [| ?H1;
            ! t. <t, l> : ?r --> perm(t, quicksort(t)) & ordered(quicksort(t));
            l ~= [] |] ==>
         perm(tl(l), ?t(hd(l), tl(l)) @ ?ta(hd(l), tl(l))) &
         mini(hd(l), ?ta(hd(l), tl(l))) & maxi(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 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.

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 this is proved by constructing a relation ?r containing 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).

The 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, also just like programs. In this example, we end up instantiating ?r with the ordering true when the first list contains fewer elements than the second. This ordering explains in what sense arguments of recursive calls to quicksort are smaller.

Final Proof State After termination of quicksort and all auxiliary synthesized programs have been proved, the final proof state is the following.

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

That is, under the definitions given for quicksort, greater and lesseq, we can prove that quicksort meets the specification of a sorting program. Note that the definitions given can be simply translated to ones favorite functional programming language and correspond to the standard presentation of these programs.

Tacking Stock The above proof suggests how algorithms like quicksort can be incrementally developed using interactive application of tactics which apply 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 until all functions are synthesized. Finally, although not obvious from the snapshots, many of our tactics incorporate procedures based on rewriting and propositional reasoning, making substantial use of standard tactics for HOL to partially automate trivial kinds of reasoning.

§ 5 Related Work and Conclusions

Our case study, highlighting some of the results in [1], is, to our knowledge, the first comprehensive study and formalization of how the deductive tableau formalism can be embedded and extended in a standard theorem prover. It suggests that frameworks like Isabelle offer advantages not present in a specialized implementation. First, because Isabelle is a logical framework, we were able to not just quickly encode proof rules usable for simulating deductive tableau proofs, but we formally derived them and hence we could establish their correctness. This helps to 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 allowed 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 higher-order metalogic with metavariables. For example we could directly formalize and manipulate well-orderings during proofs. Finally, we could directly utilize standard tactics and libraries that came with the system. This substantially reduced the time needed to create a usable theory with sufficient automation support (e.g., rewriting and propositional reasoning) suitable for carrying out their examples and our own experiments.

The work closest to ours is Coen's [5] 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 was required to show the correctness of his specialized logic with respect to some appropriate semantics. Also relevant is the work of Regensburger [12] who 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 datatypes such as streams.

References

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

[2] 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.

[3] 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.

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

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

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

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

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

[9] 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.

[10] Z. Manna and R. Waldinger. Fundamentals of the deductive program synthesis. *IEEE Transactions on Software Engineering*, January 1992.

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

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