An Integration of Deductive Retrieval into Deductive Synthesis

Bernd Fischer
RIACS/NASA Ames Research Center
CA-94035 Moffett Field, USA
`fisch@ptolemy.arc.nasa.gov`

Jon Whittle
RECOM/NASA Ames Research Center
CA-94035 Moffett Field, USA
`jonathw@ptolemy.arc.nasa.gov`

Abstract

Deductive retrieval and deductive synthesis are two conceptually closely related software development methods which apply theorem proving techniques to support the construction of correct programs. In this paper, we describe an integration of both methods which combines their complementary benefits and alleviates some of their drawbacks.

The core of our integration is an algorithm which automatically extracts queries from the synthesis proof state and submits them to a specialized retrieval system. Retrieved components are then used to close open subgoals in the proof. We use a higher-order framework for synthesis in which higher-order meta-variables are used to represent program fragments still to be synthesized. Hence, the introduction of a new meta-variable is an attempt to synthesize a new fragment and so highlights a possible reuse step. This observation allows us to invoke retrieval only after a substantial change *rather than at every proof step and prevents overloading the retrieval mechanism.*

Our integration raises the granularity level of synthesis by avoiding a substantial number of proof steps. It also provides a framework for adapting "near-miss" components in the case that an exact match cannot be retrieved.

1. Introduction

Deductive synthesis and deductive retrieval are two complementary yet contrasting approaches to formal software development. Deductive synthesis starts with a formal specification and uses a theorem prover to derive new programs from this specification. In its general form, it is a decomposition-based, top-down approach oriented towards creative system design. Deductive retrieval is the application of theorem provers to find existing components for specifications. As such, it is a composition-based, bottom-up approach and works with entities of smaller scale such as components or functions. Recently, retrieval systems such as NORA/HAMMR [18] and REBOUND [17] have shown tractable ways to scale retrieval up to larger component libraries.

In this paper we investigate the integration of a deductive retrieval system into a deductive synthesis framework in order to maximize the advantages of both approaches. We present our ideas in the context of deductive tableaus reinterpreted in higher-order logic as in [2, 3]. Such higher-order frameworks are well-suited for the creative aspects of deductive synthesis. They give the designer full control over the emerging system structure as program fragments yet to be synthesized are represented by meta-variables, i.e., within the logic itself and not by any extra-logical constructs. Implementations are typically built on top of interactive proof environments which automate the tedious bookkeeping tasks. Attempts are being made at full automation (e.g., [1]) but this is an elusive goal.

The main technical idea of our integration is to extract retrieval queries in the form of pre-/postcondition pairs automatically from the synthesis proof state *whenever it changes substantially*. These queries are used to retrieve library components which are then "plugged into" the proof state by instantiating meta-variables. The result is an interactive system further augmented with automated procedures. The user still controls the synthesis process and the automated retrieval process kicks in only in promising situations.

An integrated system provides great potential for alleviating drawbacks of the individual approaches. Pure synthesis systems, especially those based on general proof techniques such as deductive tableaus [11], do not scale up well, as for example observed by [8]:

> "The main problem of general approaches to program synthesis is that they force the synthesis system to derive an algorithm almost from scratch..."

An integrated, dedicated retrieval subsystem effectively allows synthesis to bottom-out in previously synthesized components instead of the built-in operators of the language only and thus helps to raise the level of granularity. Pure retrieval systems, on the other hand, face problems if no matching components can be retrieved for a given query. In an integrated system, further synthesis steps can be applied to provide a more detailed query, or to adapt retrieved but not perfectly matching components.

This potential can, however, be realized only if two critical assumptions are met: (i) the retrieval subsystem actually saves proof effort and (ii) the synthesis subsystem does not generate (too many) useless queries. In our case, the first assumption holds by construction as we utilize specialized, highly tuned retrieval systems which work fully automatically, retrieving on average more than 85% of the relevant components in a realistic amount of time. The second assumption is more difficult to achieve as many proof steps in program synthesis are mere bookkeeping steps which do not change the proof state substantially. Hence, invoking the retrieval subsystem after each step would overload it with an excessive number of equivalent queries. In the higher-order framework, a substantial change amounts to the introduction of a new meta-variable, a condition which can easily be checked automatically. This observation allows us to invoke retrieval only when appropriate.

2. Recast of the basic technology

2.1. Deductive synthesis

Deductive program synthesis is based on the Curry-Howard isomorphism [7] or "proofs-as-programs"-paradigm which asserts that a constructive proof of a specification is equivalent to a functional program which is correct with respect to this specification. A variety of different approaches has been investigated (cf. [8] for an overview). Here, we briefly describe the approaches by Manna/Waldinger and Ayari/Basin which are the basis of our work.

2.1.1. Deductive tableaus in classical first-order logics. Manna and Waldinger [12] have argued that it is "too constraining to carry out the entire proof in a constructive logic" and, moreover, that it is sufficient to restrict a proof in classical logic to be constructive only when necessary. They also developed a first-order synthesis methodology which uses a two-dimensional structure called a *deductive tableau* to represent a proof state.

Each row in the tableau contains a single sentence, either as an assertion or as a goal, and one or more optional output terms which represent the program under construction. In the initial tableau

assertions	goals	$f(a)$
	$\mathcal{Q}[a, f(a)]$	z
$\mathcal{A}_1$		
$\cdots$		
$\mathcal{A}_n$		

the only goal row contains the skolemized input specification $\forall x \exists z \cdot \mathcal{Q}[x, z]$; this is also the only row containing an output entry. All remaining rows are assertion rows; they contain the axioms $\mathcal{A}_1, \ldots, \mathcal{A}_n$ of the domain theory.

Subsequent proof steps change the tableau or extend it by new rows. The main proof rule is non-clausal resolution [11, 14] which is similar to a case analysis in an informal development and which accounts for the introduction of conditional terms in the program. Non-clausal resolution is used in several alternative versions; the GG-version shown here resolves two goal rows with two unifiable quantifier-free subexpressions $\mathcal{P}$ and $\mathcal{P}'$.

	$\mathcal{G}_1[\mathcal{P}]$	s
	$\mathcal{G}_2[\mathcal{P}']$	t
	$\mathcal{G}_1\theta[\mathit{false}/\mathcal{P}]$ $\wedge$ $\mathcal{G}_2\theta[\mathit{true}/\mathcal{P}']$	`if` $\mathcal{P}\theta$ `then` $t\theta$ `else` $s\theta$

The resolvent is a new goal row. Its goal is a conjunction of both original goals under the image of the unifier θ after replacing all occurrences of $\mathcal{P}$ ($\mathcal{P}'$) by *false* (*true*); its output term is an `if-then-else`-statement with the unified subexpression $\mathcal{P}\theta$ as guard and the two original output terms $t\theta$ and $s\theta$ as branches. The other versions are similar. The calculus also includes rules for simplification and splitting of assumptions and goals, skolemization, handling of equivalences and equalities, and a well-founded induction rule to introduce recursion.

The proof process is complete if the tableau contains a final row

	true	t

or, by duality,

false		t

provided that the output term t is ground and built up entirely from primitive (i.e., executable) function symbols; t can then be extracted as the final program.

2.1.2. Re-interpretation of deductive tableaus in higher-order logics. Ayari and Basin [3] have pointed out that the original version of deductive tableaus suffers from some inherent technical limitations. Since proofs in the deductive tableaus calculus are sequences rather than trees, rules that split a tableau into multiple sub-tableaus are not allowed and must be encoded using non-clausal resolution. Non-clausal resolution, however, while acceptable in an automated prover, is particularly unfriendly in an interactive context. Finally, the induction rule must be used in a bottom-up way and the use of a first-order logic means that reasoning about well-foundedness is not possible, so that termination proofs must preceed induction.

To overcome these limitations, Ayari and Basin use a higher-order logic as implemented in the ISABELLE-system [15]. ISABELLE provides a meta-logic in which object logics may be encoded. Universal quantification and implication of the meta-logic are represented by $\bigwedge$ and $\Longrightarrow$, respectively; iterated implication $A \Longrightarrow (B \Longrightarrow C)$ is abbreviated by $[\![\, A; B \,]\!] \Longrightarrow C$. Higher-order logic is encoded by formalizing connectives like $\wedge$ of type $\texttt{bool} \times \texttt{bool} \rightarrow \texttt{bool}$ and $\forall$ of type $(\alpha \rightarrow \texttt{bool}) \rightarrow \texttt{bool}$, where α is a type variable ranging over all HOL types.

[3] describes the development of some synthesis proofs using higher-order logic. Rather than transforming the original specification into a tableau they directly operate on the proof state

$$
\begin{array}{l}
\mathcal{A}_1 \wedge \ldots \wedge \mathcal{A}_n \rightarrow \mathcal{SPEC} \\
1) \quad [\![\, \mathcal{H}_{1,1}; \quad \ldots \quad ; \mathcal{H}_{1,n_1} \,]\!] \quad \Longrightarrow \quad \mathcal{G}_1 \\
\qquad\qquad\qquad \vdots \\
m) \quad [\![\, \mathcal{H}_{m,1}; \quad \ldots \quad ; \mathcal{H}_{m,n_m} \,]\!] \quad \Longrightarrow \quad \mathcal{G}_m
\end{array}
$$

where the existential variables are replaced by higher-order meta-variables $\mathcal{A}_i$ acting as place-holders for program fragments.[1] The tableau rows are replaced by the open subgoals. Each subgoal still has a single consequent $\mathcal{G}$ (i.e., goal in Manna/Waldinger's terminology) but may at the same time have an arbitrary number of hypotheses (or assertions) $\mathcal{H}_j$ which may include fragments $\mathcal{A}_i$.

Non-clausal resolution is replaced by higher-order resolution which is applicable to two meta-formulae

$$
\begin{array}{rcl}
[\![\, \psi_1; \ldots ; \psi_m \,]\!] & \Longrightarrow & \psi \\
[\![\, \phi_1; \ldots ; \phi_n \,]\!] & \Longrightarrow & \phi
\end{array}
$$

where ϕ higher-order unifies with ψ_i. The result for the unifier θ is the new subgoal (i.e., row)

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

This gives us a natural way to carry out program synthesis in a way consistent with the Deductive Tableau method. Proof development proceeds top-down since it starts with the antecedents, but the presence of the shared meta-variables introduces an element of bottom-up development.

The use of higher-order logic means that termination proofs are also possible within the framework. This is done via the following rule for well-founded induction:

$$
\begin{array}{l}
[\![\, \forall x \cdot (\forall y \cdot \langle y, x \rangle \in r \rightarrow P(y)) \rightarrow P(x) \,;\, \mathit{wf}(r) \,]\!] \\
\quad \Longrightarrow \forall x \cdot P(x)
\end{array}
$$

Note that the decision as to which well-founded relation is to be used may be delayed until later on in the proof if r is a meta-variable. Only when r is instantiated it must be shown that $\mathit{wf}(r)$ holds.

[1] The use of meta-variables to delay the choice of existential witnesses is also known as middle-out reasoning, a term coined by Alan Bundy.

2.2. Deductive retrieval

The purpose of software component retrieval in general is to identify and locate potentially reusable candidates within a component library. This is a core task in software reuse: after all, components have to be found before they can be reused. A wide variety of different retrieval approaches has been investigated (cf. [13] for an overview); the approaches differ substantially in which facet of a software component (e.g., documentation, syntactic structure, execution examples) they use for retrieval and, consequently, in the mechanism to establish potential reusability.

However, only deductive retrieval can exploit *exact semantic* information specific to the components. The basic idea is as follows:

1. Each component c is associated with a *contract*, a formal specification which captures the relevant component behavior in the form of pre- and postconditions, e.g.,

   $\mathit{run}\,(l : \mathit{list})\ r : \mathit{list}$
   pre true
   post $\exists\, l' : \mathit{list} \cdot l = r \frown l' \wedge \mathit{ord}(r)$

   to compute an ordered initial subsegment (i.e., run) of a list.[2]

2. Contracts also serve as queries q, e.g.,

   $\mathit{proper\text{-}segment}\,(l : \mathit{list})\ r : \mathit{list}$
   pre $l \neq [\,]$
   post $\exists\, l_1, l_2 : \mathit{list} \cdot$
   $\qquad l = l_1 \frown r \frown l_2 \wedge (l_1 \neq [\,] \vee l_2 \neq [\,])$

   can be used to retrieve any function which returns an arbitrary continuous proper sublist of the argument.

3. For each possible candidate in the library, a proof task is constructed comprising the respective pre- and postconditions.

4. A candidate qualifies if and only if the validity of the associated task can be established, usually using an automated theorem prover.

Hence, the approach retrieves proven matches only. The exact nature of component reuse is determined by the exact form of the proof tasks. The most common form is *plug-in compatibility*

$$(\mathit{pre}_q \rightarrow \mathit{pre}_c) \wedge (\mathit{pre}_q \wedge \mathit{post}_c \rightarrow \mathit{post}_q) \qquad (1)$$

which supports black box reuse—retrieved components may be reused "as is", without further proviso or modification. Other task variants support white box reuse but then

[2] $\frown$ denotes list concatenation, $[\,]$ the empty list, $[i]$ a singleton list with item i and *ord* is a user-defined predicate.

manual checks or code modifications are required in order to guarantee the applicability of the retrieved components.

The main problem in deductive retrieval is the large number of emerging proof tasks. Naive generate-and-prove implementations drown any prover, but recent efforts show how to circumvent this problem [18, 17, 5]. We assume that our integrated tool uses a retrieval system such as NORA/HAMMR [18] in which a pipeline of filters of increasing deductive strength are used to prune away proof tasks associated with non-matching components whilst maintaining a balance between fast response and high precision. Experimental evidence shows that this makes the retrieval task tractable.

3. Benefits of integration

Pure component retrieval is based on the assumption that the library already contains solutions to the query. In general, however, this assumption is not true. Components usually need to be modified before they satisfy the query specification [16]. Suppose that a query Q has no matching components in the library but that a partial solution C can be retrieved if we relax the matching requirements, e.g., by dropping preconditions. The problem then is how to adapt C to a complete solution.

Penix [16] describes an adaptation framework in which software architectures [6] are used to adapt C to satisfy Q by building a wrapper around C or composing C with another component, D. A limitation of this framework is that it is based entirely upon retrieval. D must also be available in the library. If this is not the case, there could be a potentially infinite number of iterations to search for new components, $D, D', D'', \ldots$.

By integrating deductive synthesis into this framework, the user could invoke a sequence of synthesis steps to carry out the adaptation. Indeed, it may be that after a small number of such steps, the specification is refined to the point where library components match against it. We illustrate this process on an example taken from [16, p. 94].

The task is to construct a function *find* which given a list of records and a natural number will return a record whose *key* field has the integer as its value. A problem specification for *find* is as follows:

$$\forall c : \textit{rec-list}, k : \mathbb{N} \cdot\ ?F(c,k) \in c \wedge\ ?F(c,k).\textit{key} = k$$

where $?F$ is a higher-order meta-variable. Suppose that our library contains the binary search component:

$\textit{bsearch}\ (l : \textit{rec-list}, k : \mathbb{N})\ r : \textit{rec}$
pre $\textit{rec\_ord}(l)$
post $r \in l\ \wedge\ r.\textit{key} = k$

where $\textit{rec\_ord}$ is the usual list ordering predicate specialized to lists of records.

Clearly, the postcondition of *bsearch* satisfies the specification for *find* but the precondition does not hold. However, *bsearch* can still be retrieved by using the weaker match *conditional compatibility*:

$$(\textit{pre}_{\textit{bsearch}} \wedge \textit{post}_{\textit{bsearch}}) \rightarrow \textit{post}_{\textit{find}}$$

This tells us that *bsearch* is a partial solution to the problem. To obtain a complete solution, we need a sorter which is then composed with *bsearch*. This is captured formally by defining a composition architecture, composing two components $?A$ and $?B$ sequentially into a composite system $?C$:

$$\begin{aligned} ?C_{in}(x) &\rightarrow\ ?A_{in}(x) \\ ?C_{in}(x) \wedge ?A_{out}(x,y) \wedge ?B_{out}(y,z) &\rightarrow\ ?C_{out}(x,z) \\ ?A_{in}(x) \wedge ?A_{out}(x,y) &\rightarrow\ ?B_{in}(y) \end{aligned}$$

By instantiating $?B$ with *bsearch*, a specification for the missing sorter can be obtained. At this point [16] could fail—if there is no sorter in the library, then the system cannot be completed. It may be possible to retrieve near-matches once again and repeat the process, but there is no guarantee that this process will terminate.

We propose that once *bsearch* is retrieved, the user should begin to derive the sorter using deductive synthesis rather than relying on further retrieval steps. In our framework, the use of the compositional architecture corresponds to a partial instantiation of $?F$, $?F(c,k) = \textit{bsearch}(?S(c), k)$ where $?S$ is a meta-variable representing the sorter. Note that there are two design decisions to be made here. First, the exact instantiation of $?F$ needs user interaction to specify, for example, which parameters $?S$ takes. Second, the user must (as with [16]) choose an architecture. To extract a specification for $?S$ within the synthesis framework, there must be a pre-defined tactic that uses the definition of the architecture as a lemma and invokes theorem proving tactics as in [16]. Applying such tactics would result in the following proof subgoal:

$$\begin{aligned} \textit{bsearch}(?S(c),k).\textit{key} = k \wedge \textit{bsearch}(?S(c),k) \in c \\ \wedge\ \textit{perm}(c, ?S(c)) \wedge \textit{rec\_ord}(?S(c)) \end{aligned}$$

The last two conjuncts give a specification for the sorter (where $\textit{perm}$ is introduced by resolving against a background theory) and can be used to synthesize various sorting algorithms. We show how to synthesize *quicksort* in the next section. In fact, our presentation there shows how to make further use of a component library during synthesis by retrieving auxiliary functions used in the definition of *quicksort*. This example shows how to combine deductive synthesis and retrieval in the context of software architectures. This combination alleviates the drawbacks of retrieval (what to do if nothing is retrieved?) and synthesis (how to raise the level of granularity of synthesis?).

$$
\begin{array}{l}
perm([\,],[\,]) \\
perm(l_1, l_2) \rightarrow x \in l_1 \leftrightarrow x \in l_2 \\
perm(l_1 {}^\frown l_2, t_1 {}^\frown t_2) \leftrightarrow perm(l_1 {}^\frown [a] {}^\frown l_2, t_1 {}^\frown [a] {}^\frown t_2) \\
ord([\,]) \\
ord([a]) \\
ord(a{::}b{::}l) \leftrightarrow (a \leq b) \wedge ord(b{::}l) \\
minl(a,[\,]). \\
minl(a, b::l) \leftrightarrow a \leq b \wedge minl(a,l). \\
maxl(a,[\,]). \\
maxl(a, b::l) \leftrightarrow a > b \wedge maxl(a,l).
\end{array}
$$

Figure 1. Background theory (cf. [10, 3])

4. Extracting queries from synthesis proofs

In this section we describe how we integrate deductive component retrieval into deductive program synthesis. We assume that synthesis is done interactively in the typed, higher-order framework of [3] and that retrieval is done automatically. The synthesis steps still drive the integrated development process—the user applies inference rules or tactics in the normal way; retrieval steps are invoked either automatically, whenever it seems promising, or interactively, i.e., under explicit user control. The user may choose any subset of the retrieved components and use them to close subgoals. If there are still open subgoals left, he has to go on with the application of further tactics and the process repeats. If no components can be retrieved, the user re-gains control immediately and continues with the proof process.

Note that the integration can only be semi-automatic even if retrieval works fully automatically. Integrating retrieved components involves a number of major design decisions that must be resolved by the user. These are highlighted in the following text. Even so, there is great potential for retrieval to bypass many theorem proving steps. Auxiliary function synthesis may require proofs that are as or more complicated than the top-level function synthesis. This effort can be eliminated almost totally by integrating deductive retrieval.

The core of our integration is an algorithm which automatically extracts retrieval queries from the synthesis proof state. Each meta-variable in the proof state represents a program fragment which must either be synthesized or can alternatively be retrieved. Hence, each time a new meta-variable is introduced, there are further possibilities for retrieval and so we extract a *(pre, post)*-condition specification which is then submitted to the retrieval system.

In the following we will illustrate the various steps in the query extraction algorithm by synthesizing *quicksort* from the standard specification[3]

$$\forall l \cdot perm(l, ?S(l)) \wedge ord(?S(l))$$

of a generic sorting program over a background theory comprising *perm* and *ord* as specified in Figure 1. Following [3], the initial ISABELLE proof state is:

$$
\begin{array}{l}
\mathcal{A} \rightarrow \forall l \cdot perm(l, ?S(l)) \wedge ord(?S(l)) \\
1. \quad \mathcal{A} \Longrightarrow \forall l \cdot perm(l, ?S(l)) \wedge ord(?S(l))
\end{array}
$$

Here, and in the general framework of [3], there are in fact two kinds of meta-variables:

- Accumulator variables such as $\mathcal{A}$ which range over formulas and hold a possibly incomplete definition of a synthesized function.
- Proper meta-variables such as *?S* which range over lambda terms and represent a part of the output of the desired program. Only such meta-variables are relevant in retrieval. Henceforth, any references to meta-variables exclude accumulator variables.

For clarity of explanation, we present the query extraction algorithm in several stages. Each stage makes a number of assumptions about the current proof state; some of these assumptions will be removed or further explained as we progress. Suppose that the current proof state comprises n open subgoals $\mathcal{S}_i$ and that we want to retrieve code for each of the i_m meta-variables $?M_{i_j}$, $1 \leq j \leq i_m$ in $\mathcal{S}_i$.[4] Then the assumptions are as follows:

1. For each i, j, all occurrences of $?M_{i_j}$ appear in consequents of the open subgoals. This restriction will not be lifted in this paper because a meta-variable which occurs both in the antecedent and consequent of a subgoal induces a recursive query specification.
2. For each i, j, all applications of $?M_{i_j}$ are identical (modulo α-conversion on the meta-level), i.e., for each occurrence of $?M_{i_j}$, the parameters of $?M_{i_j}$ are the same. This restriction will not be lifted in this paper due to space restrictions.
3. There is no meta-variable that appears in more than one open goal, i.e., the sets $M^i = \{?M_{i_j} : 1 \leq j \leq i_m\}$ are disjoint for $1 \leq i \leq n$. This restriction will be lifted in Section 4.4 below.
4. There is only a single meta-variable in each subgoal, i.e., for each i, M^i is a singleton. This restriction will be lifted in Section 4.3 below.

[3] We omit types for sake of clarity.

[4] Note that there may be other meta-variables in the proof for which we do not wish to retrieve code.

5. For each i, j, all occurrences of $?M_{i_j}$ are simple, i.e., all its arguments are distinct object-level variables. This restriction will be lifted in Section 4.2 below.

The basic steps of the algorithm, however, are the same in all cases.

1. Unskolemize all occurrences of the selected meta-variable(s) $?M$.
2. Form a header for the retrieval query.
3. Derive the query precondition from the antecedent of the subgoal(s) $\mathcal{S}$.
4. Derive the query postcondition from the consequent of the subgoal(s) $\mathcal{S}$.
5. Search the component library.
6. Instantiate $?M$ in the proof state.

We illustrate these steps using the *quicksort* example.

4.1. The base algorithm

In this section we tackle the case in which all of our assumptions hold, i.e., there is a single meta-variable confined to simple identical occurrences in a single open goal. Following [3], the initial proof state is refined to the one given in Figure 2. This new proof state is obtained by applying induction on the well-founded ordering $?R$, followed by a case analysis on l. The induction rule instantiates the original meta-variable *?S* with the partial definition for `qsort` and introduces a new accumulator variable $\mathcal{A}_1$; this is recorded in the binding for the accumulator variable $\mathcal{A}$. The case analysis introduces two new meta-variables $?T$ and $?F$ which automatically triggers the query extraction algorithm. However, only $?T$ satisfies all assumptions. We thus select the first subgoal to further illustrate the algorithm.

The initial algorithm step is unskolemization, i.e., the meta-variable application $?T(l)$ is replaced by an existentially quantified object-level variable r. The consequent of the first subgoal then becomes

$$\exists r : \mathit{list} \cdot \mathit{perm}(l, r) \wedge \mathit{ord}(r)$$

This step reverts the initial introduction of the meta-variables which can be considered as skolemization; it also introduces a name by which the component's result can be referenced within the first-order query.

The query header is easy to form. We just combine an arbitrary name for the component with the input variables and the output variable obtained by unskolemization. If the occurrence of the meta-variable is simple, its arguments are precisely the input variables for the query. The types of the variables are easily obtained from the current proof state. The query header for our example is:

$\mathit{query}\ (l : \mathit{list})\ r : \mathit{list}$

The third step is to derive the query precondition from the subgoal antecedent. The strongest possible precondition is the conjunction over the entire list of hypotheses. However, some of them may be unfit or irrelevant for retrieval purposes. We thus eliminate accumulator variables, literals which contain meta-variables other than the meta-variable involved in retrieval, and literals which contain no free occurrences of the input variables. Hypothesis elimination is pragmatically motivated but semantically justified by the *weakening rule* $(\mathcal{A} \Longrightarrow \mathcal{C}) \Longrightarrow (\llbracket \mathcal{A}; \mathcal{B} \rrbracket \Longrightarrow \mathcal{C})$ which is a theorem of ISABELLE's meta-logic. In the example, hypothesis elimination leaves $l = [\,]$ as the only (but already rather strong) precondition.

Due to the unskolemization step, the subgoal's consequent essentially forms the first-order postcondition of the query. Only some minor "cosmetic" modifications are still required. We drop all (universal meta-level) quantifiers of the input variables and the existential object-level quantifier introduced by unskolemization. In the example, the postcondition is therefore

$$\mathit{perm}(l, r) \wedge \mathit{ord}(r)$$

The variables do not remain free, however; they are rebound universally during the construction of the proof task.

The complete query is now

$\mathit{query}\ (l : \mathit{list})\ r : \mathit{list}$
pre $l = [\,]$
post $\mathit{perm}(l, r) \wedge \mathit{ord}(r)$

which can be submitted to the retrieval subsystem. Let us assume that our library contains two components which construct and copy lists, respectively. These components may involve any non-trivial amount of administrative overhead (e.g., memory management) but we further assume that their specifications

$\mathit{list} :: \mathit{constructor}\ ()\ r' : \mathit{list}$
pre true
post $r' = [\,]$

and

$\mathit{copy}\ (l' : \mathit{list})\ r' : \mathit{list}$
pre true
post $r' = l'$

only reflect their basic functionalities. Let us also assume that both components passed the early filter mechanisms of the retrieval system. The final retrieval step is an "all-out match", i.e., an attempt to prove that the components actually satisfy the query. The full proof task consists of (1) along with parameter quantification and identification. Hence, the proof tasks for $\mathit{list} :: \mathit{constructor}$ and copy are

$$
\begin{array}{ll}
\multicolumn{2}{l}{\forall l \cdot \texttt{qsort}(l) = \texttt{if } l = [\,] \texttt{ then } ?T(l) \texttt{ else } ?F(\texttt{hd}(l), \texttt{tl}(l)) \wedge \mathcal{A}_1} \\
\multicolumn{2}{l}{\quad \to \forall l \cdot \mathit{perm}(l, \texttt{qsort}(l)) \wedge \mathit{ord}(\texttt{qsort}(l))} \\
1. & \bigwedge l \cdot [\![\forall t \cdot \langle t, l \rangle \in ?R \to \mathit{perm}(t, \texttt{qsort}(t)) \wedge \mathit{ord}(\texttt{qsort}(t)); \mathcal{A}_1; l = [\,]]\!] \\
& \Longrightarrow \mathit{perm}(l, ?T(l)) \wedge \mathit{ord}(?T(l)) \\
2. & \bigwedge l \cdot [\![\forall t \cdot \langle t, l \rangle \in ?R \to \mathit{perm}(t, \texttt{qsort}(t)) \wedge \mathit{ord}(\texttt{qsort}(t)); \mathcal{A}_1; l \neq [\,]]\!] \\
& \Longrightarrow \mathit{perm}(l, ?F(\texttt{hd}(l), \texttt{tl}(l))) \wedge \mathit{ord}(?F(\texttt{hd}(l), \texttt{tl}(l))) \\
3. & \mathit{wf}(?R)
\end{array}
$$

Figure 2. Proof state after induction and casesplit.

respectively:

$$
\begin{array}{l}
\forall l, r, r' : \mathit{list} \cdot r = r' \to \\
\quad ((l = [\,] \to \mathsf{true}) \\
\quad \wedge (l = [\,] \wedge r' = [\,] \to \mathit{perm}(l, r) \wedge \mathit{ord}(r))) \\
\forall l, l', r, r' : \mathit{list} \cdot l = l' \wedge r = r' \to \\
\quad ((l = [\,] \to \mathsf{true}) \\
\quad \wedge (l = [\,] \wedge r' = l' \to \mathit{perm}(l, r) \wedge \mathit{ord}(r)))
\end{array}
$$

Both tasks can easily be proven by an automatic first-order theorem prover such as SPASS [23], i.e., both components are retrieved. Note that the first task is valid even though the constructor's signature does not match the query. The final selection of one of the (in general many) retrieved components is one of the major design decisions in integration which cannot be automated. It is similar to the selection of one of the (possibly infinity number of) higher-order unifiers in [3].[5] However, in our case it may also be based on the non-functional aspects (e.g., memory or performance characteristics) of the components. Assume we thus chose to instantiate $?T$ with `copy`. The `qsort`-fragment then becomes

$$\texttt{qsort}(l) = \texttt{if } l = [\,] \texttt{ then copy}(l) \texttt{ else } ?F(\texttt{hd}(l), \texttt{tl}(l))$$

In this case the primary reuse effects are rather small as the instantiation of $?T$ with `copy` saves no proof steps over the direct solution $?T = \lambda x \cdot [\,]$ as done in [3]. However, there are still secondary benefits. Using the explicit library component instead of the "inlined" $\lambda x \cdot [\,]$ improves the adaptability of the synthesized program. If for example the lists are refined into arrays, the refinement proof can be restricted to the specifications of the actual components and the original synthesis proof need not to be repeated.

4.2. Signature cleaning and signature matching

If all meta-variables are simple (cf. assumption 5), the identification of the query's input variables is trivial. This assumption rarely holds and so we will lift it here. We try to construct an appropriately cleaned, simple approximation of the still unknown component signature. Here, we apply *abstraction*, i.e., we replace non-variable object-level terms by fresh variables and move the original instantiations (via lifting) into the hypotheses such that they can be used as additional preconditions which reflect the restricted calling pattern. For example, the subgoal

$$\bigwedge l \cdot \mathcal{H} \Longrightarrow p(?F(\texttt{hd}(l), \texttt{tl}(l)))$$

becomes

$$\bigwedge l, i, m \cdot [\![\mathcal{H}; i = \texttt{hd}(l); m = \texttt{tl}(l)]\!] \Longrightarrow p(?F(i, m))$$

Abstraction does not introduce new meta-variables but (of course) new meta-quantified object-level variables. Let us illustrate this on the *quicksort*-example. Consider the proof state in Figure 3 where $?F$ has been instantiated with the characteristic recursive `qsort`-call and $?L_1$ and $?L_2$ represent the yet unknown partitions.[6] Let us also temporarily assume that $?L_2$ will be instantiated with an appropriate function `greater` in order to meet the fourth assumption. After abstraction and some simplifications due to this instantiation, the subgoal becomes

$$
\begin{array}{l}
\bigwedge l, i, m \cdot [\![\ldots; l \neq [\,]; i = \texttt{hd}(l); m = \texttt{tl}(l)]\!] \\
\quad \Longrightarrow \mathit{perm}(m, ?L_1(i, m) \frown \texttt{greater}(i, m)) \\
\qquad \wedge\, \mathit{maxl}(i, ?L_1(i, m))
\end{array}
$$

From this subgoal, the query specification

$$
\begin{array}{l}
\mathit{query}\ (i : \mathit{item}, m : \mathit{list})\ r : \mathit{list} \\
\mathsf{pre}\ \ l \neq [\,] \wedge i = \texttt{hd}(l) \wedge m = \texttt{tl}(l) \\
\mathsf{post}\ \mathit{perm}(m, r \frown \texttt{greater}(i, m)) \wedge \mathit{maxl}(i, r)
\end{array}
$$

can be extracted as before. Note that this specification still contains the original, non-abstracted variable l which has to be bound universally during the construction of the complete proof task to retain correctness. Before it can be checked against a library candidate with a different signature

[5] E.g., *list::constructor* and *copy* correspond to two solutions of the corresponding higher-order unification problem $?F([\,]) \doteq [\,]$.

[6] Here we omit subgoals which exclusively deal with termination issues, e.g., (3) in Figure 3.

$$
\begin{array}{l}
\forall l \cdot \mathtt{qsort}(l) = \mathtt{if}\ l = [\,]\ \mathtt{then}\ \mathtt{copy}(l)\ \mathtt{else}\ \mathtt{qsort}(?L_1(\mathtt{hd}(l), \mathtt{tl}(l)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad {}^\frown[\mathtt{hd}(l)]^\frown \mathtt{qsort}(?L_2(\mathtt{hd}(l), \mathtt{tl}(l)) \wedge \mathcal{A}_1 \\
\quad \to \forall l \cdot \mathit{perm}(l, \mathtt{qsort}(l)) \wedge \mathit{ord}(\mathtt{qsort}(l)) \\
1. \quad \bigwedge l \cdot [\![\forall t \cdot \langle t, l\rangle \in ?R \to \mathit{perm}(t, \mathtt{qsort}(t)) \wedge \mathit{ord}(\mathtt{qsort}(t)); \mathcal{A}_1; l \neq [\,]]\!] \\
\qquad \Longrightarrow \mathit{perm}(\mathtt{tl}(l), ?L_1(\mathtt{hd}(l), \mathtt{tl}(l))^\frown ?L_2(\mathtt{hd}(l), \mathtt{tl}(l))) \wedge \\
\qquad \mathit{minl}(\mathtt{hd}(l), ?L_2(\mathtt{hd}(l), \mathtt{tl}(l))) \wedge \mathit{maxl}(\mathtt{hd}(l), ?L_1(\mathtt{hd}(l), \mathtt{tl}(l))) \\
\qquad\qquad\qquad \vdots \qquad\qquad\qquad\qquad\qquad\qquad \vdots
\end{array}
$$

Figure 3. Proof state after instantiation of recursive call.

$$
\begin{array}{l}
\mathit{lesseq}\ (l' : \mathit{list}, i' : \mathit{item})\ r' : \mathit{list} \\
\textsf{pre true} \\
\textsf{post}\ \forall j : \mathit{item} \cdot \\
\qquad j \in l' \wedge j \leq i' \to \#(j, l') = \#(j, r')
\end{array}
$$

(where $\#(x, k)$ returns the number of occurrences of x in k), the *signature mismatch* has to be resolved: a straightforward matching would identify a *list*-parameter with an *item*-parameter and thus render the task unprovable. Signature matching can exploit type information to prevent such situations. [4] develops an approach based on constructive isomorphisms of λ-terms which fits well into our higher-order framework.

Ultimately, the signature of an auxiliary function is the result of a design decision; in [3] a special "two-placed variant" of the induction rule is used to introduce `lesseq` and `greater` as two-place functions. Our techniques can thus only be an approximation. If they fail to retrieve a matching component, the user has either to provide a different call pattern by partially instantiating the meta-variable, as in [3], or—in the worst case—to re-synthesize the missing component.

4.3. Multiple meta-variables

Our fourth assumption states that there is only a single meta-variable in each open subgoal. In general, this restriction is also unrealistic but it can fortunately be lifted relatively easily. We still assume that the meta-variables cannot be shared across subgoals.

All meta-variables are unskolemized as normal. A query is then formed for each meta-variable $?M_{i_1}, \ldots, ?M_{i_{i_m}}$ under consideration. To form the postcondition of the query for $?M_{i_j}$, we remove the existential quantifier over $?M_{i_j}$ only. This means that the other meta-variables remain existentially quantified.

Section 4.2 describes how to retrieve `lesseq` from the proof state, assuming that `greater` has already been found. Consider what happens if we try to retrieve for the meta-variables $?L_1$ and $?L_2$ simultaneously. We obtain two queries corresponding to $?L_1$ and $?L_2$ respectively:

$$
\begin{array}{l}
\mathit{query}\text{-}1\ (i : \mathit{item}, m : \mathit{list})\ r_1 : \mathit{list} \\
\textsf{pre}\ \ l \neq [\,] \wedge i = \mathtt{hd}(l) \wedge m = \mathtt{tl}(l) \\
\textsf{post}\ \exists\, r_2 : \mathit{list} \cdot \mathit{perm}(m, r_1{}^\frown r_2) \\
\qquad\qquad \wedge\ \mathit{minl}(i, r_2) \wedge \mathit{maxl}(i, r_1)
\end{array}
$$

$$
\begin{array}{l}
\mathit{query}\text{-}2\ (i : \mathit{item}, m : \mathit{list})\ r_2 : \mathit{list} \\
\textsf{pre}\ \ l \neq [\,] \wedge i = \mathtt{hd}(l) \wedge m = \mathtt{tl}(l) \\
\textsf{post}\ \exists\, r_1 : \mathit{list} \cdot \mathit{perm}(m, r_1{}^\frown r_2) \\
\qquad\qquad \wedge\ \mathit{minl}(i, r_2) \wedge \mathit{maxl}(i, r_1)
\end{array}
$$

The component specification for `lesseq` given in Section 4.2 and a similar specification for `greater` match these queries. There are potentially a large number of other components that would also match the queries, however. The key insight is to notice that the instantiations of the two meta-variables must be satisfied simultaneously. Using the two queries independently would therefore be unwise. For example, consider two simpler queries with postconditions:

$$
\begin{array}{l}
\exists r_1 : \mathit{list} \cdot \mathit{perm}(m, r_1{}^\frown r_2) \\
\exists r_2 : \mathit{list} \cdot \mathit{perm}(m, r_1{}^\frown r_2)
\end{array}
$$

Each query used independently would retrieve, amongst other things, *list::constructor*. Clearly, however, this instantiation could not satisfy both queries simultaneously. We thus follow a different approach. The idea is that given a list of queries, $Q_1, \ldots, Q_m$ for meta-variables $?M_1, \ldots, ?M_m$, we use Q_i as a filter for $Q_{i+1}, \ldots, Q_m$. The library is first searched using Q_1 which retrieves a number of possible instantiations for $?M_1$. Next, the library is searched using Q_2 but the existential variable in Q_2 corresponding to $?M_1$ is instantiated with library components retrieved in the previous step. Each possible instantiation for $?M_1$ is checked. Any instantiation for which there are no components matching the second query are dropped. This process is repeated and the end result is a set of instantiations satisfying all queries. Note that this set could be substantially smaller than if the queries had been used independently.

In the *quicksort* example, the first query retrieves `lesseq`. After substituting this in the second query we get:

$$query\text{-}2\,(i : item, m : list)\ r : list$$
$$\textsf{pre}\ \ l \neq [\,] \wedge i = \texttt{hd}(l) \wedge m = \texttt{tl}(l)$$
$$\textsf{post}\ perm(m, \texttt{lesseq}(i, m)^\frown r_2) \wedge minl(i, r_2)$$

Note also that the third conjunct in this query has been eliminated. This is the result of a post-processing step which removes any conjuncts which no longer contain either the output variable of the query or any existentially quantified variables. Since such conjuncts appear in the first query where they were proved to hold with the appropriate instantiation, they must also hold in the second query under this instantiation. So they can be removed from consideration.

This second query can now be used to retrieve `greater`. In [3], the synthesis of `lesseq` and `greater` has to be done by an inductive proof. This involves substantial time and effort on behalf of the user which can be saved by using deductive retrieval.

4.4. Meta-variables shared across multiple open subgoals

The third assumption states that each meta-variable appears in a unique open subgoal. There are (at least) two solutions to this problem. The first is to form a new query for each open subgoal within which the meta-variable appears. Since these queries must be satisfied simultaneously, each query can be used as a filter for the next query in the same way as was done above. Alternatively, we could extract a single query from all the open subgoals. Suppose that there are open subgoals $G_1, \ldots, G_n$ and that we can use our methods to extract pre- and post-conditions for each G_i. Then a single query extracting information from all the subgoals is:

$$fullquery\,(\ldots)$$
$$\textsf{pre}\ \ pre_1 \vee \cdots \vee pre_n$$
$$\textsf{post}\ (pre_1 \rightarrow post_1) \wedge \ldots \wedge (pre_n \rightarrow post_n)$$

What are the trade-offs of these two approaches? This question can be answered in terms of the shape and number of the corresponding proof obligations. Ignoring the preconditions (since these involve easy proofs), let us look at the obligations based on matching the query and component postconditions. $fullquery$ gives rise to a single proof obligation for each library component but this obligation is rather large. In the separate query case, there are, in the worst case, nk proof obligations, where k is the size of the library. Each library component must be checked to match with the postconditions for each G_i. In practice, however, many of these matches will fail. If a match for a library component fails, this component can be eliminated from consideration and matches no longer need to be checked for the remaining G_i. Hence, there will in all likelihood be substantially fewer than nk match obligations. Also, the proof obligations will be much smaller. In this context, it is unclear whether it is easier to handle a large number of small proofs or a single large proof. Hence, further experiments are required to determine which is the better strategy for deductive retrieval.

5. Related work

Manna and Waldinger have already shown [12, Section VII] that in theory a notion of component reuse can easily be built-into deductive tableaus. The initial tableau is just extended by rows containing the specifications (or even the implementations) of the library components as assumption. These rows can then be used in the derivation process the same way as the usual axioms of the background theory. This idea can be translated in a straightforward way into Ayari and Basin's higher-order re-formulation of deductive tableau.

In practice, however, this simple idea has severe disadvantages. In a fully automatic synthesis system, the additional rows significantly extend the search space. In an interactive synthesis system, finding the right component specification for a resolution step is left to the human; however, since this is exactly the original component retrieval task, nothing is gained.

The AMPHION system [22] follows the Manna-Waldinger-approach to combine synthesis and retrieval but works in the very domain-specific setting of astronomical subroutines. AMPHION is successful because of this domain restriction and because the synthesis granularity is actually fairly small—each line of specification yields, on average, only three lines of code. The integration we propose is far more general.

The KIDS system [20] provides some degree of integration of synthesis and retrieval in the sense that abstract algorithms can be pre-defined as schemas which are then re-used in later synthesis problems. We believe that the benefits of integration will only be realized if *concrete* algorithm re-use is also enabled. KIDS has a facility for reusing concrete components via a process for deriving specification morphisms using unskolemization [21]. These morphisms can be applied to existing components to produce new operators (e.g., to derive a composition operator from a decomposition operator). However, the choice of source component must be done manually, i.e., the retrieval problem emerges in the same way as in the Manna/Waldinger-approach. [19] hints at how to integrate automated retrieval into CYPRESS, a KIDS-predecessor. The tactic `operator_match` attempts to match synthesis subgoals to predefined operators. However, the tactic is applied exhaustively and so does not scale up to larger component libraries. By incorporating the pre-filters (cf. Section 2.2) we overcome this scalability problem. [19] also fails to control *when* `operator_match` is applied. We chose to integrate our retrieval techniques into the higher-order framework because the introduction of new meta-variables yields a convenient notion of what constitutes a substantial change of proof state, and hence a control on when to attempt retrieval. However, our approach is in principle also applicable to schema-based synthesis.

Our own prior work on deductive retrieval clearly demonstrates the necessity (and feasibility) of a highly optimized retrieval system. An experiment using more than 100 list processing components (comparable to the auxiliary functions used in the *quicksort* example) showed the following results averaged over approximately 120 queries of different granularity. The average number of matching components was 15.45 with a standard deviation of $\sigma = 22.82$; however, in more than 27% of the queries, only a single component matched. In the latest version of NORA/HAMMR, early pre-filters reject almost 75% of the invalid proof tasks within approx. 20 seconds per query;[7] subsequent preprocessing steps improve the average proof times by a factor of 1.5–3, depending on the prover. The average response time per query thus drops by a factor of almost 3. At the same time, the recall (i.e., prover success rate) improves from 30–60% to 60–85%, again depending on the prover. If we put all these numbers together, it means that the average expected time to retrieve the first matching component (which is sufficient for synthesis) drops from 19 minutes to 3.5 minutes.

6. Conclusion

In this paper, we have presented an integration of deductive retrieval into a deductive synthesis system. The key to a practical integration is to identify promising proof states where retrieval can be applied. This avoids overloading the retrieval subsystem with an excessive number of redundant queries. Experiments show that a specialized retrieval subsystem is required to overcome the infeasibility of a naive "all-out match".

There are three main advantages to integration. First, the number of interactive proof steps is substantially reduced; in the *quicksort*-example, retrieval of the auxiliary functions saves two complex inductive sub-proofs. Second, the granularity level of synthesis is raised from language constructs to components. Finally, integration supports the adaptation of retrieved "near-misses" and thus overcomes a major problem of pure retrieval systems.

References

[1] A. Armando, A. Smaill, and I. Green. "Automatic synthesis of recursive programs: The proof-planning paradigm", in [9], pp. 2–9.

[2] A. Ayari and D. Basin. "Generic system support for deductive program development", in T. Margaria and B. Steffen (eds.), *Proc. 2nd Intl. Workshop TACAS*, *LNCS 1055*, pp. 313–328, Passau, Springer, 1996.

[3] A. Ayari and D. Basin. "A higher-order interpretation of deductive tableau", Tech. report, Univ. Freiburg, 1999.

[4] R. DiCosmo. *Isomorphisms of Types: from λ-calculus to information retrieval and language design*, Birkhäuser, Boston, 1995.

[5] B. Fischer. "Specification-based browsing of software component libraries", in D. F. Redmiles and B. Nuseibeh (eds.), *Proc. 13th ASE*, pp. 74–83, Honolulu, 1998. IEEE.

[6] D. Garlan and M. Shaw. "An Introduction to Software Architecture", in V. Ambriola and G. Tortora (eds.), *Advances in Software Engineering and Knowledge Engineering*, pp. 1–40. World Scientific Publishing Co., 1992.

[7] W. Howard. "The formulas-as-types notion of construction", in J. P. Seldin and J. R. Hindley (eds.), *To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus, and Formalism*, pp. 479–490. Academic Press, 1980.

[8] C. Kreitz. "Program synthesis", in W. Bibel and P. H. Schmitt (eds.), *Automated Deduction - A Basis for Applications*, pp. 105–134. Kluwer, Dordrecht, 1998.

[9] M. Lowry and Y. Ledru (eds.), *Proc. 12th ASE*, Lake Tahoe, 1997. IEEE.

[10] Z. Manna and R. Waldinger. "*The Deductive Foundations of Computer Programming*", Addison-Wesley, New York, 1993.

[11] Z. Manna and R. J. Waldinger. "A deductive approach to program synthesis", *ACM TOPLAS*, 2(1):90–121, 1980.

[12] Z. Manna and R. J. Waldinger. "Fundamentals of deductive program synthesis", *IEEE TSE*, 18(8):674–704, 1992.

[13] A. Mili, R. Mili, and R. Mittermeir. "A survey of software reuse libraries", *Annals of Software Engineering*, 5:349–414, 1998.

[14] N. V. Murray. "Completely non-clausal theorem proving", *AI*, 18:67–85, 1982.

[15] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, *LNCS 828*, Springer, 1994.

[16] J. Penix. *Automated Component Retrieval and Adaptation Using Formal Specifications*, PhD thesis, Univ. Cincinnati, 1998.

[17] J. Penix and P. Alexander. "Efficient specification-based component retrieval", *Automated Software Engineering*, 6(2):139–170, 1999.

[18] J. M. P. Schumann and B. Fischer. "NORA/HAMMR: Making deduction-based software component retrieval practical", in [9], pp. 246–254.

[19] D. R. Smith. "The top-down synthesis of divide and conquer algorithms", *AI*, 27(1):43–96, 1985.

[20] D. R. Smith. "KIDS: A semi-automatic program development system", *IEEE TSE*, 16(9):1024–1043, 1990.

[21] D. R. Smith. "Constructing specification morphisms", *JSC*, 15:571–606, 1993.

[22] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood. "Deductive composition of astronomical software from subroutine libraries", in A. Bundy, (ed.) *Proc. 12th CADE*, *LNAI 814*, pp. 341–355, Nancy, Springer, 1994.

[23] C. Weidenbach. "SPASS—version 0.49", *JAR*, 18(2):247–252, 1997.

[7] All times were measured on a SUN Ultra 1/170.