Improving the Usability of HOL through Controlled Automation Tactics

Eunsuk Kang and Mark D. Aagaard

Electrical and Computer Engineering
University of Waterloo
Waterloo, ON, Canada
{ekang,maagaard}@uwaterloo.ca

Abstract. This paper introduces the concept of *controlled automation* as a balanced medium between high-level automated reasoning and low-level primitive tactics in HOL. We created a new tactic that subsumes many existing low-level tactics for logical operations and three new tactics that simplify common uses of term rewriting: definition expansion, simplification, and equational rewriting. To implement the tactics, we extended HOL with a facility to label assumptions and operate uniformly on both goals and assumptions. We select automatically and predictably which low-level tactic to apply by examining the structure of the selected assumption or goal. A simple and uniform set of hints enable users to provide the minimal information needed to guide the tactics. We performed two case studies and achieved a 60% reduction in the number of unique tactics used.

1 Introduction

HOL 4 [4] is a powerful theorem proving environment, providing a wide range of proof techniques such as tactics, term rewriting, and decision procedures. However, the vast richness of HOL can be overwhelming. A novice user often spends a significant amount of time navigating through the HOL reference manual. Even experienced HOL users occasionally encounter difficulties remembering the name or syntax of a tactic. Based on these observations, our goal was to automate common proof tasks in HOL and eliminate much of the need to search documentation when doing common reasoning. By doing so, we allow the users to focus on devising strategies to conduct the proof.

Tactics in HOL can be classified into two types. Low-level tactics are used to decompose a goal into smaller subgoals or transform it into another form through term rewriting. Most of these low-level tactics carry out one specific operation at a time, such as eliminating a conjunctive operator or expanding the definition of a constant. In comparison, high-level tactics are capable of performing multiple operations in a single step, often attempting to reduce a goal as much as possible. High-level tactics can be risky when used without careful discretion because they occasionally perform more operations than what the user wants them to do. In some other cases, they simply fail to do what the user expects. Low-level tactics provide finer control, but they are numerous and difficult to remember.

In this paper, we introduce four new tactics: `ELIM_TAC`, `EXPAND_TAC`, `REDUC_TAC`, and `EQUATE_TAC`. The purpose of our tactics is to provide a balanced medium between

high-level and low-level tactics. These tactics replace many of the existing low-level tactics and inference rules in HOL, significantly reducing the number of functions that the user needs to remember throughout a proof. At the same time, all of the tactics operate on the basic principle that they should perform exactly what the user instructs them to do — no more, no less — thereby giving the user the complete control of proof steps. In short, these tactics provide what we call *controlled automation*.

Each of the four tactics is specialized in terms of the types of low-level proof tasks that it automates. `ELIM_TAC` is intended for performing primitive logical operations, such as the elimination of a conjunctive operator. We classify term rewriting into three different categories; definitional expansion, term simplification, and equational rewriting. The three types of rewriting tasks are carried out by `EXPAND_TAC`, `REDUC_TAC`, and `EQUATE_TAC`, respectively.

The implementation of our tactics is based on three underlying ideas. First, we introduce a new data structure called *hints*, which allows the user to provide the minimal amount of information needed to guide our tactics. A major portion of our work focused on establishing hints as an easy-to-use, intuitive control mechanism for HOL users. Secondly, we develop a system of labelling and identifying an assumption using a string in order to allow the user to operate directly on the assumption. Lastly, we provide a precise control mechanism for the user to select between different rewriting strategies, depending on the type of the rewriting task.

We performed two case studies to evaluate the effectiveness of the new tactics in improving the usability of HOL, and solicited feedback from novice users in a graduate course on formal verification. We started with a small example where HOL is used to solve a logic puzzle from the popular novel series *Harry Potter*. We then moved on to a much larger study, where we formally proved the equivalence of several superscalar microprocessor correctness statements [1]. The statistics gathered from these studies show an approximately 60% reduction in the number of HOL functions that a user needs to use in a proof, and a 25 to 40% decrease in the size of a proof script. These results demonstrate that our work simplifies and reduces the amount of interaction between the user and the theorem prover.

In Section 2, we describe `ELIM_TAC` and how it replaces many of the most commonly used low-level tactics or rules in HOL. Section 3 discusses the three tactics for rewriting: `EXPAND_TAC`, `REDUC_TAC`, and `EQUATE_TAC`. Sections 4 and 5 contain the description of the case studies and related work, respectively.

2 Controlled Automation for Logical Operations

2.1 Problem Description

The origins of our work come from two observations that we made about typical HOL proof scripts. First, many proofs could be shortened if assumptions could be manipulated with the same logical operations that can be applied to the goal. Second, the logical operation to apply to an assumption or goal can often be predicted based on the outermost operator of the assumption or goal. From these observations, we set ourselves the goal of developing a tactic that the user would aim at the goal or an assumption and say “do the obvious thing”.

Any reasonably complicated proof in HOL likely involves the manipulation of assumptions. Before the user can completely prove the goal or subgoals using automated reasoners, it is often necessary to transform assumptions into a particular form through the application of low-level tactics or inference rules. However, HOL does not contain functions that allow the user to apply a forward inference rule *directly* to a specific assumption. Instead, an assumption is either moved into the goal and modified via tactics; or is transformed into a theorem, modified via forward inference rules, and then put back onto the assumption list. There are several disadvantages to this limitation. The multi-step processes to manipulate assumptions distract the user from the main reasoning of the proof. The names of forward inference rules are often seemingly unrelated to the names of the corresponding tactic. As the number of assumptions on the goalstack increases, it becomes difficult to identify the specific assumption to be manipulated. To overcome these inconveniences, we designed `ELIM_TAC` to operate uniformly on both the assumptions and the goal and developed a mechanism on top of HOL to name assumptions and then identify assumptions by name.

A goal-directed proof in HOL typically involves decomposing the goal or assumptions into smaller terms. For example, conjunctive goals (e.g. `P` $\wedge$ `Q`) are decomposed into two goals, one for `P` and one for `Q`. We observed that it is often possible to guess the next logical operation to perform by examining the outermost operator of the goal or assumption. To reduce the number of tactics that the user must remember, `ELIM_TAC` uses simple rules to choose the tactic to apply based upon the outermost logical connective or quantifier of the identified goal or assumption.

Different tactics have different arguments for users to pass information to the tactic (e.g a witness term for the elimination of an existential quantifier). Therefore, any process to automate the low-level tactics must include an intuitive, easy-to-use method for users to pass information to the underlying tactics. Relying on heuristics to guess the requisite information would violate our intention of giving the user complete control.

The remainder of this section discusses our approaches to solving the challenge of developing robust mechanisms to:

1. Choose a particular tactic to apply based on the logical structure of a term
2. Enable users to pass requisite information to underlying tactic
3. Operate operate uniformly on a goal or an assumption
4. Ensure that tactics perform the expected operation or fail immediately.

2.2 `ELIM_TAC`

`ELIM_TAC` combines many commonly used low-level tactics. We developed it with one ultimate purpose: save users time and effort in picking out which tactic to apply. The type of `ELIM_TAC` is: `string -> hint list -> tactic`. The first argument is a label that identifies the goal or an assumption as a *target* on which the tactic will carry out its operation. The empty string `""` denotes the goal.

The second argument is a list of *hints*, which serve as a control mechanism for the user to provide the tactic with the necessary information for carrying out a logical operation. The choice of hints depends on the type of the operation that is to be performed on the target expression (e.g. the nullary constructor `SKOLEM` instructs `ELIM_TAC` to

perform skolemnization on an existential quantified term). Table 1 contains a summary of the data constructors that are applicable to `ELIM_TAC`. Later in this paper, we extend the definition of `hint` with additional constructors that are used in our rewrite tactics (Table 2 in Section 3.3).

Table 1. Descriptions of data constructors for `hint`

Constructor	Type in ML	Logical Operation
`ASM`	`string -> hint`	$\Rightarrow$ elimination
`MATCH`	`term -> hint`	$\wedge$, $\vee$ elimination
`INSTANCE`	`term -> hint`	$\forall$ elimination
`WITNESS`	`term -> hint`	$\exists$ elimination
`SKOLEM`	`hint`	Skolemnization

Often, `ELIM_TAC` does not require any hints. When the logical operation to perform can be inferred from the structure of the goal or assumption, an empty list of hints suffices. In the example sequent below, the current goal is an implicative formula `P` $\Rightarrow$ `Q`. Applying `ELIM_TAC` pushes the antecedent into the current list of assumptions (note that the string "a" on the third line represents the label for the assumption, P):

$\vdash P \Rightarrow Q$
By **`ELIM_TAC "" []`**
a $\bullet P$
$\vdash Q$

$\Rightarrow$ elimination in goal

The sequents below illustrate the other uses of `ELIM_TAC` for goal decomposition without hints ($\wedge$, $\vee$, and $\forall$ elimination).

$\vdash P \wedge Q$
By **`ELIM_TAC "" []`**
$\vdash P$ $\quad \vdash Q$

$\wedge$ elimination in goal

$\vdash P \vee Q$
By **`ELIM_TAC "" []`**
$\vdash P$

$\vee$ elimination in goal

$\vdash \forall x.u$
By **`ELIM_TAC "" []`**
$\vdash u[x'/x]$

$\forall$ elimination in goal

Next, we introduce an example of a situation where the user is required to provide an extra bit of information to `ELIM_TAC`. In order to eliminate an existential quantifier, we specify a witness term with the `WITNESS` hint.

$\vdash \exists x.t$

By `ELIM_TAC "" [WITNESS ‘u‘]`

$\vdash t[u/x]$

$\exists$ elimination in goal

We have already presented the use of `ELIM_TAC` for eliminating the disjunctive operator in a goal. By default, the tactic selects the left of the two disjuncts to spawn as the subgoal. However, if the user wishes to extract the right disjunct instead, how should this operation be specified in `ELIM_TAC`? The constructor `MATCH` accepts a higher-order pattern and attempts to match it against either one of the two disjuncts in the goal. If a match is found, it spawns the matched disjunct as a subgoal. However, if no such match is found, the tactic *immediately fails* and displays an error message, instead of arbitrarily selecting a disjunct for subgoal generation. It is important to note that this failure mechanism is a critical part of all controlled automation techniques. If an operation violates the user's intention in any way, a controlled automation tactic must halt its execution; by doing so, it prevents any further operations that might lead to undesirable results for the user.

$\vdash P \vee Q$

By `ELIM_TAC "" [MATCH ‘Q‘]`

$\vdash Q$

$\vee$ elimination in goal

The sequents presented in this section encompass all of the most common logical operations that one may wish to perform on a goal. In Sections 2.3 and 2.4, we show how we extend the capabilities of `ELIM_TAC` in order to automate even a wider range of primitive operations in HOL.

2.3 Assumption Labelling

Black and Windley [2] discuss various ways to select one or more specific assumptions in HOL. Arguably the most straight-forward and convenient mechanism for the user is identifying assumptions by their position in the assumption list. From an informal survey of students in a graduate-level introductory course to theorem proving, we found that most of them preferred to create their own tactic that allowed them to select a specific assumption by an explicit numeric index, rather than using built-in capabilities in HOL. However, this mechanism of identification is highly undesirable since it is sensitive to changes in a proof script. In the long term, it will likely be detrimental to the maintenance of the proof script.

In order to provide a mechanism that is convenient for the user and yet insensitive to changes in a proof, we label assumptions with strings. In order to identify a target assumption or goal, the user simply passes in the label for the assumption (or the `""` for the goal) to `ELIM_TAC`. We also provided functions to allow the user to label a newly added assumption or change the label of an existing assumption on the goalstack.

Our implementation of labelling did not involve any changes to the existing HOL code. The mapping between an assumption and a label on the goalstack is accomplished through a hash table — the key is the term that represents the assumption, and the hashed item is the string label. The one-to-one mapping between the key and the hashed item ensures that each assumption has its own unique label, thereby separating the ordering of assumptions on the goalstack from their labels. When new assumptions are added to the goalstack without explicit labels from the user, each of them is entered into the hash table with a default label. In order to create a default label for an assumption, we use simple heuristics in which we extract a representative string based on the size and the first two alphanumeric characters of the term. We designed our heuristics in such a way that it would reduce the occurrences of duplicate default labels as much as possible without requiring the user to enter tediously long labels to refer to assumptions. However, in case of duplicate default labels, all subsequent duplicate labels after the first one are suffixed with a numeric index. This operation maintains the one-to-one mapping requirement in the hash table, but also introduces the dependency of labels on the ordering in which their corresponding assumptions are added to the goalstack. Based on our experiences with `ELIM_TAC`, the occurrences of duplicate default labels are rare. However, in order to ensure that assumption labels stay insensitive to changes in a proof script all the time, we encourage the user to label a new assumption with a unique, more meaningful name.

2.4 Extending `ELIM_TAC` to Work with Assumptions

Given the underlying infrastructure for assumption labelling, we now describe the full capabilities of `ELIM_TAC` as a versatile tool for both reducing a goal and transforming an assumption into a desired form. In this section, we present sequents for uses of `ELIM_TAC` when operating on an assumption. We also describe how hints are used to fully specify the user's intention when more than one primitive operations may be possible.

At times, it is useful to be able to extract one specific conjunct out of a term with an arbitrary number of conjuncts. Let us assume that the user intends to perform the elimination of a conjunctive operator on an assumption `'t1 ∧ t2 ∧ t3'`. Applying `ELIM_TAC` to the assumption without any hints results in the extraction of the leftmost conjunct, `'t1'`.

```
  a   • t1 ∧ t2 ∧ t3
      ⊢ P
By    ELIM_TAC "a" []
  a   • t2 ∧ t3
 t1   • t1
      ⊢ P
```

∧ elimination in assumption

Another common primitive task in HOL is extracting an *inner* conjunct out of an expression. Carrying this operation out by using only the built-in functions in HOL

may take several steps, depending on the level of nesting for that particular conjunct. Instead, we specify a pattern that matches a particular subterm (i.e. `'t2'`) using the `MATCH` hint. The meaning of `MATCH` depends on the context in which `ELIM_TAC` is currently being used. When applied to a goal, `MATCH` is used for the decomposition of a disjunctive expression; when applied to an assumption, `MATCH` plays the role in extracting a particular conjunct out of the assumption.

```
 a   • t1 ∧ t2 ∧ t3
     ⊢ P
By   ELIM_TAC "a" [MATCH 't2']
 a   • t1 ∧ t3
t2   • t2
     ⊢ P
```

∧ elimination in assumption using `MATCH`

In the next example, we show how one can use `ELIM_TAC` to perform the modus ponens rule involving two assumptions. Given the assumptions, "a1" and "a2" in the below sequent, the intuitive next step is to match the antecedent in `'t1 ==> t2'` against the other assumption and obtain the goal, `'t2'`. The data constructor `ASM` indicates which assumption should be matched against the antecedent of an implicative assumption.

```
a1   • t1 ⇒ t2
a2   • t1
     ⊢ P
By   ELIM_TAC "a1" [ASM "a2"]
a1   • t2
a2   • t1
     ⊢ P
```

⇒ elimination in assumption using `ASM`

When applied to an existentially quantified assumption without hints, `ELIM_TAC` performs skolemnization. Using only the built-in tactics in HOL, the most straightforward path to skolemnization on an assumption requires four different steps. Here, using `ELIM_TAC`, we accomplish this operation in a single step:

```
 a   • ∃x.t
     ⊢ P
By   ELIM_TAC "a" []
 a   • t
     ⊢ P
```

∃ elimination in assumption

When an existential quantified expression resides as a subterm within an assumption with two or more conjuncts (e.g. `'t1 ∧ ?x. t2 ∧ t3'`), it is sometimes desirable to perform skolemnization directly on the quantified subterm. We introduce a new constructor called `SKOLEM`, which instructs `ELIM_TAC` to search for an existentially quantified subterm and apply skolemnization directly on it.

a $\bullet\, t1 \wedge (\exists x.t2) \wedge t3$
$\vdash P$
By **`ELIM_TAC "a" [SKOLEM]`**
a $\bullet\, t1 \wedge t2 \wedge t3$
$\vdash P$

∃ **elimination in assumption using SKOLEM**

Lastly, eliminating the universal quantifier from an assumption involves instantiating the quantified variable with a specific constant. The user must provide this constant through the hint constructor `INSTANCE`:

a $\bullet\, \forall x.t$
$\vdash P$
By **`ELIM_TAC "a" [INSTANCE 'u']`**
a $\bullet\, t[u/x]$
$\vdash P$

∀ **elimination in assumption using INSTANCE**

In this section, we have described the operations that a user can perform using the combination of `ELIM_TAC` with hints. One possible criticism against `ELIM_TAC` is that the user is still required to memorize the name and the syntax of the different data constructors for the type `hint`. In response, we have tried to keep the number of the data constructors to a minimum without compromising the robustness of `ELIM_TAC` in its ability to perform a wide range of primitive operations. Also, the names and the syntax of the constructors are fairly intuitive and easy to remember. An experiment with students in the graduate course in theorem proving showed that once they became familiarized with `ELIM_TAC`, they preferred working with our tactic over having to remember a wide range of built-in HOL tactics and inference rules.

3 Controlled Automation for Term Rewriting

3.1 Problem Description

HOL contains a wide set of rewrite tactics and as well as several hundreds of previously proven theorems that can be used "off-the-shelf" as rewrite rules. Despite the richness of rewriting capabilities available in HOL, term rewriting is not always a straightforward matter, and the learning curve is stiff for a novice user.

In this section, we identify two major challenges with term rewriting in HOL. First, we observe that it is generally difficult to control the exact location within a term to

which a rewrite rule must be applied. The most commonly used rewriting tactics such as `REWRITE_TAC` and `ONCE_REWRITE_TAC` walk over the entire term in a left-to-right order and applies the rewrite rule to the first rewritable subterm that it encounters. Sometimes, these tactics fail to meet the user's expectations; they either rewrite a wrong subterm or rewrite more subterms than the user has in mind.

Secondly, HOL does not provide an efficient way to apply a rewrite rule directly to an assumption or rewrite a goal using an assumption. The former case is closely related to the discussion in Section 2.3 - that is, the lack of a built-in mechanism for identifying or directly operating on one or more specific assumptions. `ASM_REWRITE_TAC` attempts to rewrite the goal using all of the existing assumptions, but is not useful when the user wishes to rewrite using only a subset of them. `FILTER_ASM_REWRITE_TAC` improves upon the former tactic by allowing the user to select a set of assumptions as rewrite rules, but requires a condition predicate, which can be tedious to create. A simpler mechanism for rewriting using assumptions is desirable.

In the process of developing controlled automation tactics for term rewriting, we have classified common rewriting tasks into three different categories:

1. Definitional expansion
2. Term simplification
3. Other equational rewriting

Instead of creating a single tactic that performs all three types of rewriting, we designed a separate tactic for each one of them. The rationale behind the separation of tactics is that it is difficult to devise a single rewrite tactic that "does it all" without falling into the danger of violating the user's intentions. For instance, it is sometimes desirable to simplify an arithmetic expression as much as possible, whereas in other situations, the user may intend to perform a more delicate rewrite operation. In either case, it is nearly impossible to guess what the user *might* wish to do for the next step of a proof.

In Sections 3.2 and 3.3, we describe our approaches to solving the aforementioned challenges - specifying a rewritable subterm and applying rewriting operations directly to assumptions. Then, in Section 3.4, illustrate how the three rewrite tactics evolve on top of our solutions to these challenges.

3.2 Specifying the Location of Rewriting

One solution to the challenge of specifying the exact location of rewriting within a term is combining the built-in tactic `GEN_REWRITE_TAC` with *conversions*. A conversion in HOL is a function, with a type `term -> thm`, that takes a term t and produces a theorem $\vdash t = t'$. A composition of multiple conversions can be used to specify the search strategies for term rewriting when passed to `GEN_REWRITE_TAC`.

For instance, let us assume that the user wishes to rewrite the left hand side of the equation in `'x - (z + y) = x - (y + z)'` using the built-in theorem `ADD_SYM`: $\vdash \forall m\ n.\ m + n = n + m$.

The conversion `RATOR_CONV` applies a rewrite rule to the operator of a function application, and `ONCE_DEPTH_CONV` applies the rule only once to an applicable subterm; the latter conversion is necessary in order to ensure the termination of rewriting.

In the following sequent, the two conversions are composed using the infix operator `o` and passed as an argument to the tactic.

```
    ⊢x - (z + y) = x - (y + z)
By  GEN_REWRITE_TAC (RATOR_CONV o ONCE_DEPTH_CONV) [] [ADD_SYM]
    ⊢x - (y + z) = x - (y + z)
```

We developed an alternative solution where the user simply specifies a rewritable subterm using higher-order pattern matching. The main goal of our approach is to eliminate the necessity for built-in conversions in rewriting. As a first step, we define a tactic named `NEW_REWRITE_TAC` with a type `hint list -> tactic`. This tactic shares the same data type `hint` as `ELIM_TAC` does. However, in the context of rewriting, the data constructor `MATCH` is used to specify the higher-order match. In addition, we add a new constructor `THM : thm -> hint`, which takes a rewrite rule as the argument. By combining these two constructors into a list, the user instructs `NEW_REWRITE_TAC` to rewrite only the left hand side of the equation.

```
    ⊢x - (z + y) = x - (y + z)
By  NEW_REWRITE_TAC [MATCH 'z + y', THM ADD_SYM]
    ⊢ x - (y + z) = x - (y + z)
```

As an another example, let us assume that for whatever reasons, the user wishes to rewrite the left hand side of the equation `SUC (x + y) = SUC (x + y)` using `ADD_SYM`. In this case, the original higher-order matching technique cannot be used to specify only the left hand side of the term, as any pattern would match both of the two sides. In order to ensure that our term rewriting approach is complete, we extended our matching algorithm to allow the user to point at a rewritable subterm using the underscore character `'_'`. The following sequent illustrates this example:

```
    ⊢SUC (x + y) = SUC (x + y)
By  NEW_REWRITE_TAC [MATCH '_ = a', THM ADD_SYM]
    ⊢SUC (y + x) = SUC (x + y)
```

As a whole, the meaning of `MATCH '_ = a'` is equivalent to "*extract a subterm that matches the pattern provided in the hint but rewrite only the part of the subterm that is highlighted by the underscore character.*" Note that since the matching algorithm used in our approach is based on higher-order pattern matching, the fragment `'a'` in `'_ = a'` matches the right hand side of the entire term (i.e. `'SUC (x + y)'`). This mechanism is convenient because the user does not need to spell out the entire term to provide the exact match.

3.3 Integrating Rewriting with Assumption Handling

Directly Rewriting an Assumption In Section 2.3, we introduced the system of identifying an assumption using string labels. We now extend `NEW_REWRITE_TAC` to allow term rewriting on an assumption. The ML type of the tactic is also updated to:

```
NEW_REWRITE_TAC : string -> hint list -> tactic .
```

As previously mentioned, the empty string `""` denotes the goal on the current goalstack.

Rewriting a Goal using Assumptions Since `NEW_REWRITE_TAC` shares the same data type `hint` as `ELIM_TAC` does, we take advantage of the data constructor `ASM` to allow the user to specify the assumption to be applied to a goal as a rewrite rule. Then, the application of the tactic

```
NEW_REWRITE_TAC "" [ASM "a1"] .
```

rewrites the goal using the assumption with the label "a1."

In some cases, it is also desirable to include multiple assumptions as rewrite rules. In order to support this capability, we introduce a new data constructor for `hint` called `ASMS`. The constructor has an ML type `string list -> hint`. Similarly, to allow the user to specify multiple built-in or previously proven theorem as rewrite rules, we extend `hint` with another constructor called `THMS`. The complete list of data constructors of `hint` and their purposes are shown in Table 2:

Table 2. Descriptions of complete data constructors for `hint`

Constructor	Type in ML	Logical Operation
`ASM`	`string -> hint`	$\Rightarrow$ elimination, modus ponens, rewrite rule
`ASMS`	`string list -> hint`	Rewrite rules
`INSTANCE`	`term -> hint`	$\forall$ elimination
`MATCH`	`term -> hint`	$\wedge$, $\vee$ elimination, rewrite match
`SKOLEM`	`hint`	Skolemnization
`THM`	`thm -> hint`	Rewrite rule
`THMS`	`thm list -> hint`	Rewrite rules
`WITNESS`	`term -> hint`	$\exists$ elimination

3.4 Controlled Rewrite Tactics

Given the supporting mechanisms for specifying a subterm and rewriting with assumptions, we finally introduce controlled automation tactics that are specialized for term rewriting. In this process, we replace `NEW_REWRITE_TAC`, which was discussed in the previous section, with three new tactics, `EXPAND_TAC`, `REDUC_TAC`, and `EQUATE_TAC`. Syntactically, these tactics share the same ML type `string -> hint list -> tactic`. We believe that the range of rewriting operations supported by our tactics is wide enough such that user will rarely need to resort to the built-in rewrite tactics in HOL.

Definitional Expansion EXPAND_TAC performs the definitional expansion of one or more constants in a goal or an assumption. A constant is expandable if there exists a built-in or user-defined theorem $\vdash c = t$, where c is the constant itself, and t is the term that represents the definition of the constant. More generally, if c is an n-arity function, an expression equivalent to an application of c to its n parameters, $p_1, p_2, ..., p_n$ is expandable if there exists a theorem $\vdash (c\ p_1\ p_2\ ...\ p_n) = t'$, where t' is a well-typed term in the HOL-specific variant of simply typed λ-calculus. One or more of the parameters, $p_1, p_2, ..., p_n$, may be universally quantified.

When invoked without any hints, EXPAND_TAC traverses the matching subterm within the target goal or assumption in the top-down manner and unfolds the definition of the first expandable constant or expression that it encounters. Internally, unnoticed by the user, EXPAND_TAC searches through the database of existing definitions in HOL and determines whether a constant is expandable. This automatic searching capability provides convenience to the user but is not always desirable; in this case, hints act as a useful control mechanism for specifying the behaviour of the tactic.

As an alternative to specifying the rewrite rule through THM, narrowing down the scope of the subterm search by using MATCH constructor has an identical effect. For example, let us assume that two theorems, `factorial` and `pow` have already been defined.

```
factorial: |- (factorial 0 = 1) /\
              !n. factorial (SUC n) = (n + 1) * factorial n
pow: |- (!k. pow k 0 = 1) /\
        !k n. pow k (SUC n) = k * pow k n
```

As a next step, the user expands the occurrence of `factorial` in the left hand side of the inequality:

```
    ⊢ factorial (SUC n) > pow 2 (SUC n)
By  EXPAND_TAC "" []
    ⊢ (n + 1) * factorial n > pow 2 (SUC n)
```

However, the user changes his or her mind and decides to unfold the definition of `pow` instead:

```
    ⊢ factorial (SUC n) > pow 2 (SUC n)
By  EXPAND_TAC "" [THM pow]
    ⊢ factorial (SUC n) > 2 * pow 2 n
```

It is not necessarily the case that the name of the ML binder (meta-level identifier) is equal to the name of the constant (HOL term) that is currently being expanded. Alternatively, the user can specify the subterm to be rewritten using a higher-order pattern:

```
    ⊢ factorial (SUC n) > pow 2 (SUC n)
By  EXPAND_TAC "" [MATCH ‘a > _‘]
    ⊢ factorial (SUC n) > 2 * pow 2 n
```

The latter alternative may be preferred in some cases since it eliminates the user's mandate of having to remember the name of the ML binder for a specific theorem.

Term Simplification REDUC_TAC recursively applies a set of built-in simplification rules about numbers, lists, and propositions as well as user-provided theorems to a goal or an assumption until no rewrite rule remains applicable. In essence, REDUC_TAC is a wrapper for the built-in HOL simplifier SIMP_TAC with an external interface that gives the user greater control of where to apply simplification. REDUC_TAC is most beneficial when the user wishes to simplify only a particular subterm within a goal or an assumption. Consider the following sequent:

```
 a   •c + 1 - 1 = a + b
     ⊢c + 1 - 1 = SUC (a + b - 1)
By   REDUC_TAC "" [MATCH 'SUC x']
 a   •c + 1 - 1 = a + b
     ⊢c + 1 - 1 = a + b
```

Applying REDUC_TAC to the entire term would reduce `'c + 1 - 1'` to `'c'`, which, in this example, is not the desirable outcome.

Equational Rewriting The last of the trio, EQUATE_TAC, is intended for rewriting operations that do not fall into the two other categories - definitional expansion and term simplification. EQUATE_TAC may be used to perform any equational rewriting; in this sense, its behaviour is nearly identical to GEN_REWRITE_TAC. However, the two tactics differ significantly in their user interfaces. The latter tactic accepts a set of conversions for search strategies as well as a list of rewrite rules and theorems. EQUATE_TAC achieves nearly the same level of customizability with a much simpler external interface and does not require the user to commit to memory numerous conversion functions

4 Case Studies

In order to evaluate the effectiveness of our approach in providing controlled automation, we have carried out two separate case studies in HOL. We first begin with a small proof that is based on one of the logic puzzles in the popular novel *Harry Potter and the Sorcerer's Stone*. The details of the puzzle are irrelevant for the purpose of our discussion. The original version of the proof that we carried out is about 280 lines long in terms of the size of the proof script, and required 8 intermediate lemmas.

The original proof script served as the control subject in our case study. We then carried out the same proof from scratch, but the main difference now was that our four tactics were available for use by the human prover. It is important to note that we did not modify the general, high-level strategies for carrying out the proof between the two proof scripts. Rather, the focus of our study was to measure the effectiveness of our approach in automating low-level details in a proof. After the proof was re-done, we gathered statistics such as the size of the proof script and the number of tactics or inference rules that the user had to use in order to successfully carry out the proof.

The first row in Table 3 illustrates the statistics from the study of the Harry Potter puzzle. The data exhibit a considerable amount of reduction in the size of the proof script. We attribute this improvement mainly to our assumption handling system, which allows the user to apply an inference rule directly to an assumption in one step. Using built-in tools in HOL, the same operation would take two to three steps on average.

Table 3. Reduction in the Size of Proof Scripts and Number of Functions Used

	Size (LOC)			No. Functions		
	Original	Modified	Reduction	Original	Modified	Reduction
Harry Potter	282	162	43%	13	5	62%
Microbox	2254	1678	26%	30	12	60%

Perhaps more significantly, the outcome of our study in Table 3 shows a large decrease in the number of tactics or inference rules that were used during the proof; the figure "5" in the last column includes the counting of our tactics. Throughout most parts of the proof, the user was able to perform logical or rewriting operations using our controlled automation tactics. There were only a few occasions where it was necessary to resort to built-in functions in HOL.

We also carried out a larger case study (2200 line proof script) in which we redid the Microbox [1] proofs about correctness statements for superscalar microprocessors. The second row in Table 3 contains the statistics for the comparison between the original and the modified version of the proof script for the microprocessor correctness statements. Similar to the Harry Potter proof, we observed a significant improvement both in terms of the size of the proof script and the number of functions used. The figure "12" in the last column of the table includes all of the four controlled tactics that we have described in this paper. The remaining eight were built-in HOL functions that did not fit into our definition of controlled automation. These functions included high-level automated reasoners (e.g.PROVE_TAC) and miscellaneous facilities such as tactics for renaming assumptions or dropping unnecessary assumptions from the goalstack. Based on the case studies, we believe that our approach holds considerable promise in making theorem proving in HOL a much simpler, less time-consuming task for both novice and expert users.

5 Related Work

The purpose of our tactics is not to add expressive or deductive power to the HOL theorem prover. Rather, our goal is to enable users to quickly perform many of the common logical or rewriting operations.

Harrison compared declarative and procedural styles of theorem proving [7]. Delahaye later compared declarative, procedural, and term-based proofs (for constructive logics) [3]. Each of these different styles of proof is best suited for different types of reasoning (e.g., forward vs. backward, short proofs vs. long proofs, elegantly crafted

vs. done-and-forgotten). Our work lies within the world of procedural proof, and relies upon three principal ideas: assumption labelling for simple and robust access to assumptions, hints as a uniform mechanism to pass information to tactics, and packaging low-level rewrite strategies according to user-level purposes: definition expansion, simplification and reduction, and equational substitution.

Trybulec's Mizar theorem prover has long allowed users to label proof steps [12] [10]. Harrison adapted the idea of labels to do assumption labelling in his Mizar mode for HOL [5] and in HOL-light [6]. Hickey's MetaPRL supports the labelling of proof nodes to denote the type of reasoning that led to the node (e.g. primary line of reasoning, well-formedness, or antecedent to an assertion) [8].

One of the most valuable hints in our work is `MATCH`, which allows the user to select a subterm to operate on. Martin and colleagues used patterns and meta-variables in the Angel tactic language to identify subterms on which to operate and to extract subterms from the program to be passed as arguments to tactics [9]. Toyn used patterns to steer tactics and to select which tactic to apply [11]. In ISAR, Wenzel uses meta-variables in proof scripts to refer to terms and includes automatic binding of standard variables, such as *???goal* [13]. All of these uses of patterns would certainly be beneficial in hints or new tactics, but some of the uses could require significant changes to the way that HOL parses terms.

6 Conclusion and Future Work

In this paper, we have presented the concept of *controlled automation* as a balanced medium between high-level automated reasoning and low-level primitive operations in the HOL theorem prover. In order to show how we achieve this type of automation, we have introduced four new HOL tactics, which we named `ELIM_TAC`, `EXPAND_TAC`, `REDUC_TAC`, and `EQUATE_TAC`. We have also described special data structures called `hints`, which serve as an intuitive, easy-to-use mechanism for users to provide these tactics with information for carrying out the desired operations. Two case studies that we have conducted demonstrate that our tactics significantly improve on the degree of automation in HOL by reducing the number of functions that a user needs to remember.

We have introduced these tactics (in addition to existing tactics and rules in HOL) to students who are enrolled in the introductory level course in formal verification. This year's offering is the first time that we have done so. A major assignment in the course is theorem proving using HOL, and the students are given the freedom of using our tactics. So far, an informal survey of students indicates that they prefer to use the new tactics whenever possible over the existing ones in HOL. Many students have positively expressed that they do not have to cope with the burden of remembering the names of numerous tactics and inference rules in HOL. Our current plan is to continue introducing our tactics in future offerings of the course and receive feedback from students on how we can improve our tools to better assist novice users in the process of learning interactive theorem proving.

Acknowledgements

This research was funded in part by the Natural Science and Engineering Research Council of Canada (NSERC) and Canadian Foundation for Innovation (CFI). The authors are indebted to Nancy Day and the students in CS-745: Computer-Aided Verification for their feedback on earlier versions of the tactics.

References

1. M. D. Aagaard, N. A. Day, and M. Lou. Relating multi-step and single-step microprocessor correctness statements. In M. D. Aagaard and J. W. O'Leary, editors, *FMCAD*, volume 2517 of *LNCS*, pages 123–141. Springer, Nov. 2002.
2. P. E. Black and P. J. Windley. Automatically synthesized term denotation predicates: A proof aid. In *Theorem Proving in Higher Order Logics*, pages 46–57. Springer Verlag, 1995.
3. D. Delahaye. Free-style theorem proving. In *Theorem Proving in Higher Order Logics*, pages 164–181. Springer Verlag, 2002.
4. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, New York, 1993.
5. J. Harrison. A Mizar mode for HOL. In *Theorem Proving in Higher Order Logics*, pages 203–220. Springer Verlag, 1996.
6. J. Harrison. The HOL light system reference. http://www.cl.cam.ac.uk/∼jrh13/hol-light/reference_220.pdf, 2006.
7. J. R. Harrison. Proof style. In *BRA Types workshop*, pages 154–172. Springer, 1996.
8. J. J. Hickey. *The MetaPRL Logical Programming Environment*. PhD thesis, Cornell, 2001.
9. A. P. Martin, P. H. B. Gardiner, and J. C. P. Woodcock. A tactical calculus. *Formal Aspects of Computing*, 8(4):479–489, 1996.
10. P. Rudnicki and A. Trybulec. On equivalents of well-foundedness. *Jour. of Automated Reasoning*, 23(3-4):197–234, Nov. 1999.
11. I. Toyn. A tactic language for reasoning about z specifications. In *3rd BCS-FACS Northern Formal Methods Workshop*, Sept. 1998.
12. A. Trybulec and H. A. Blair. Computer assisted reasoning with MIZAR. In *Int'l Joint Conf. on Artificial Intelligence*, pages 26–28. Morgan Kaufmann, 1985.
13. M. Wenzel. Isar – A generic interpretative approach to readable formal proof documents. In *Theorem Proving in Higher Order Logics*, pages 167–183. Springer Verlag, 1999.