A Prototype Interface between CLAM and HOL*

Richard Boulton, *Alan Bundy* *Konrad Slind*, *Mike Gordon*

Dept. of Artificial Intelligence
University of Edinburgh

Computer Laboratory
University of Cambridge

June 1997

Abstract

This paper describes a prototype interface between the CLAM proof planner and the HOL interactive theorem prover. CLAM uses artificial intelligence planning techniques to find proofs at a high-level, especially where inductive proofs are required. HOL is a theorem prover for a classical higher-order logic and has been widely used in both academia and industry for verification of hardware and software. The interface sends HOL goals to CLAM for planning, and translates plans back into HOL tactics that solve the initial goals.

1 Introduction

HOL [4] is a general-purpose proof system for classical, higher-order predicate calculus; it has been used to formalize many areas of interest to computer scientists and mathematicians. The HOL system has often been criticized on the basis that it does not provide a high level of proof automation. Such remarks are often based on ignorance, since the HOL system provides powerful simplifiers, automatic first order provers (both tableaux and model elimination), a semi-decision procedure for a useful fragment of arithmetic, and a cooperating decision procedure mechanism. However, HOL lacks automation for many important areas, and moreover, there is always more that can be done to automate the proof process. A good case in point is *induction*. Induction is certainly a central proof method, but in HOL, as in many other systems, the user must interactively control the application of induction.

*Research supported by the Engineering and Physical Sciences Research Council of Great Britain under grants GR/L03071 and GR/L14381.

CLAM [3] is a proof planning system for Oyster, a tactic-based implementation of the constructive type theory of Martin-Löf. CLAM works by using formalized pre- and post-conditions of Oyster tactics as the basis of plan search. These specifications of tactics are called *methods*. When a plan for a goal is found, the expectation is that the resulting tactic will solve the goal. Experience shows that the search space for plans is often tractable: CLAM has been able to automatically plan many proofs. One emphasis on research with CLAM has been the automation of inductive proofs.

Therefore, it seems interesting to investigate a link between CLAM and HOL. The intention is to:

- Investigate the practical utility for hardware and software verification of Artificial Intelligence techniques
- Build bridges between AI and Formal Methods
- Test the HOL/CLAM combination on industrially significant verification problems
- Reduce the time it takes a HOL user to verify a system
- Simplify the interaction between the user and the theorem prover

The prototype design described in this paper avoids the need to modify CLAM to suit the classical higher-order logic (simple type theory) used by the HOL system. Instead, correspondences between syntactic features of HOL's logic and the constructive type theory of Oyster/CLAM are exploited.

2 The Interface

The version of the HOL system used in the interface is built on top of Standard ML. In this experiment CLAM is called as a child process of HOL. The sequence of operations is illustrated in Figure 1. First, a temporary file is created from within the HOL (Standard ML) process. The HOL formula (goal) to be proved is translated into the abstract syntax of Oyster's logic and, together with supporting definitions and induction schemes, is written to the file (in concrete syntax) as a sequence of Prolog goals. A CLAM process is then initiated from within ML with this file as its input.

Figure 1: Calling CLAM from HOL

$$
\begin{array}{lcl}
[\![\forall(x:ty).\ t]\!]_{form} & = & [\![x]\!]_{var} : [\![ty]\!]_{type} \Rightarrow [\![t]\!]_{form} \\
[\![\exists(x:ty).\ t]\!]_{form} & = & [\![x]\!]_{var} : [\![ty]\!]_{type} \# [\![t]\!]_{form} \\
[\![\neg t]\!]_{form} & = & [\![t]\!]_{form} \Rightarrow \textbf{void} \\
[\![t_1 \wedge t_2]\!]_{form} & = & [\![t_1]\!]_{form} \# [\![t_2]\!]_{form} \\
[\![t_1 \vee t_2]\!]_{form} & = & [\![t_1]\!]_{form} \backslash [\![t_2]\!]_{form} \\
[\![t_1 \Longrightarrow t_2]\!]_{form} & = & [\![t_1]\!]_{form} \Rightarrow [\![t_2]\!]_{form} \\
[\![(t_1 : ty) = t_2]\!]_{form} & = & [\![t_1]\!]_{term} = [\![t_2]\!]_{term} \text{ in } [\![ty]\!]_{type} \\
[\![t]\!]_{form} & = & [\![t]\!]_{term} \quad \text{if } t \text{ is not one of the forms above}
\end{array}
$$

$$
\begin{array}{lcl}
[\![f\ x_1\ \ldots\ x_n]\!]_{term} & = & \begin{cases} [\![f]\!]_{const}([\![x_1]\!]_{term}, \ldots, [\![x_n]\!]_{term}) & \text{if } f \text{ is a constant} \\ [\![f]\!]_{term} \text{ of } [\![x_1]\!]_{term} \text{ of } \ldots \text{ of } [\![x_n]\!]_{term} & \text{otherwise} \end{cases} \\
[\![\lambda x.\ t]\!]_{term} & = & \text{lambda}([\![x]\!]_{var}, [\![t]\!]_{term}) \\
[\![x]\!]_{term} & = & [\![x]\!]_{var} \\
[\![c]\!]_{term} & = & [\![c]\!]_{const}
\end{array}
$$

$$
\begin{array}{lcl}
[\![tyv]\!]_{type} & = & [\![tyv]\!]_{vartype} \\
[\![tyc]\!]_{type} & = & [\![tyc]\!]_{consttype} \\
[\![ty_1 \rightarrow ty_2]\!]_{type} & = & [\![ty_1]\!]_{type} \Rightarrow [\![ty_2]\!]_{type} \\
[\![(ty_1, \ldots, ty_n)tyc]\!]_{type} & = & [\![tyc]\!]_{consttype}([\![ty_1]\!]_{type}, \ldots, [\![ty_n]\!]_{type})
\end{array}
$$

Figure 2: Translating HOL terms and types to Oyster syntax

The contents of the source file cause a plan to be generated and saved in a target file (assuming a plan is found). The output from the CLAM process appears as output in the HOL session. When the CLAM process terminates, ML regains control and parses the plan file. After deletion of the temporary files, the plan is translated to a tactic which is applied to the HOL goal. The interface will soon be modified to use sockets instead of temporary files.

The CLAM process used has been modified to provide some independence from Oyster and the built-in types and induction schemes of the CLAM library.

The syntax of Oyster/CLAM formulas and of CLAM proof plans has been specified as an extended BNF grammar. The extensions include layout information for pretty-printing. ML code for parsers and pretty-printers is generated from this specification using the ML-Syn tool [1].

3 Translation of the object language

Figure 2 shows the translations from the HOL logic to Oyster syntax used by the experimental interface. Emphasized brackets $[\![$ and $]\!]$ are used to denote the translation functions. In HOL, formulas and terms are the same syntactic category, but it is helpful to distinguish them for purposes of translation. Certain kinds of formula (Boolean-valued terms) exist in HOL that are not explicitly handled by the translation in Figure 2. Types in HOL are distinct from terms. Many of the syntactic forms shown in the translation are derived forms. In fact, HOL terms have only four basic syntactic forms: variables, constants, function application (often referred to as combination) and λ-abstraction.

The translation of (term) constants, $[\![\_]\!]_{const}$, and of (term) variables, $[\![\_]\!]_{var}$, consists of mapping HOL names to names that are suitable for use in Prolog. Names in HOL's logic may begin with either an upper-case or lower-case letter and be followed by any number of letters, digits, underscores, and primes ($'$). Since primes are not allowed in CLAM names they are replaced by underscores. Symbolic names are also permitted in HOL. These are replaced by alphanumeric representations, e.g., '`+`' becomes '`PLUS`'. CLAM names must begin with a lower-case letter. So, if after replacement the name begins with an upper case letter or a digit, the name is prefixed with 'hol'. Under this mapping two HOL names could be translated to the same CLAM name but this is unlikely to arise in practice. The names of HOL type constants are translated in a similar way but for them the 'hol' prefix is always used. If HOL names are used according to the conventions of the built-in theories, this leads to CLAM terms in which the names of constants begin with 'hol' and the names of variables are as they appear in the original HOL term.

The CLAM name used for each HOL name is stored and used later in the process of translating CLAM plans to HOL tactics.

Type variables in HOL have a different lexical form to variables at the term level. It seems sufficient to replace their initial prime character with 't_'. The polymorphism in HOL is handled by translating the type variables to Oyster type variables. However, this is not sufficient. In HOL, type variables are implicitly universally quantified. In Oyster they have to be bound. So, at the top level, the variables introduced for HOL type variables are quantified by assuming that they inhabit the first type universe, $\mathbf{u(1)}$.

As an example, the HOL formula expressing the associativity of the function for appending two lists is:

```
∀l1 l2 l3. APPEND l1 (APPEND l2 l3) = APPEND (APPEND l1 l2) l3
```

It is translated to:

```
t_a:u(1)=>l1:hollist(t_a)=>l2:hollist(t_a)=>l3:hollist(t_a)=>
   holAPPEND(l1,holAPPEND(l2,l3)) =
   holAPPEND(holAPPEND(l1,l2),l3) in hollist(t_a)
```

Now we address the translation of plans to tactics.

4 Plans and methods

As mentioned above, CLAM writes proof plans, which are currently represented by Prolog terms, to a file. The plan parser derived from ML-Syn parses these terms to an ML concrete datatype of *plans*. A plan is just some wrapper syntax for a *method*; accordingly, we will describe the concrete syntax of (a subset of) the methods employed by CLAM:

$$
\begin{array}{rcl}
\textit{method} & ::= & \textbf{induction}([\ (\ \textit{identifier}\ \textbf{:}\ \textit{typ}\ \textbf{)-}\ \textit{typ}\ ]) \\
 & | & \textbf{generalise}(\textit{typ}\ \textbf{,}\ \textit{identifier}\ \textbf{:}\ \textit{typ}\ \textbf{)} \\
 & | & \textbf{new}[\ (\textit{identifier}\ \textbf{,})^*\ \textit{identifier}\ ] \\
 & | & \textit{identifier}\ \textbf{(}\ (\textit{argument}\ \textbf{,})^*\ \textit{argument}\textbf{)} \\
 & | & [\ (\textit{method}\ \textbf{,})^*\ \textit{method}\ ] \\
 & | & \textit{method}\ \textbf{then}\ \textit{method} \\
 & & \\
\textit{argument} & ::= & \textit{method} \\
 & | & \textbf{in} \\
 & | & []\ \ |\ \ [\ (\textit{number}\ \textbf{,})^*\ \textit{number}\ ]
\end{array}
$$

Identifiers are strings of letters, digits and underscores, beginning with a lower-case letter, and numbers are just sequences of decimal digits. The corresponding abstract syntax is the following:

```
method = Induction of typ * typ
       | Generalise of typ * typ
       | New of identifier list
       | MethodApp of identifier * argument list
       | Methods of method list
       | MethodThen of method * method
and
argument = Method of method
         | In
         | NumList of number list
```

The form '*identifier* **:** *typ*' is itself a *typ*, hence the abstract syntax for the **induction** and **generalise** methods.

Also note that *typ* is the syntactic category of CLAM types. (Everything in the Oyster/CLAM logic is a type.) Translations from the types and terms of HOL to CLAM types were given in Section 3.

5 Tactics

Tactics are a well-known method for backward proof. The original conception of Milner [5], which is still that of tactics in HOL, is that a tactic can be represented by the type

$$goal \longrightarrow goal\ list * justification,$$

i.e., a tactic decomposes a goal into subgoals plus a justification function. The justification function takes the theorems resulting from the solved subgoals and performs inference with them to return a new theorem that *achieves* the original goal. Thus the justification has type

$$thm\ list \longrightarrow thm.$$

A theorem $\Gamma \vdash M$ achieves a goal Δ, N when $M =_\alpha N$ and also each element of Γ is equal, modulo α convertibility, to an element of Δ. A tactic t *solves* a goal g when $t\ g$ creates an empty list of subgoals and a validation function f such that $f[\,]$ achieves g.

Writing new tactics is one of the principal methods for extending the inference ability of the HOL system. Section 6 gives a translation from proof plans into the following purpose-written HOL tactics:

$ELEM\_TAC$: *tactic*. Finishes off a branch in the proof. A series of basic reasoners are invoked, in order of (roughly) increasing cost. First, the goal is checked to see whether it is an instance of reflexivity; then whether it becomes true by use of various simple rewrite rules; then whether it is a tautology; and finally, whether it is an instance of a theorem of linear arithmetic.

OCC_RW_TAC : *int list* $\longrightarrow$ *string* $\longrightarrow$ *tactic*. Provides rewriting at a specified subterm in the goal. *Director strings*, *i.e.*, lists of numbers, are used to index the subterm that is to be rewritten. The strings are intended to handle first order syntax, where functions are n-ary, hence there is an on-the-fly translation of relevant parts of the goal from binary to n-ary. Note: director strings are in reverse order: for example, the director string [1,2] is interpreted as 'take the first subterm of the second subterm of the term'. The second argument is the name of the rewrite rule in a rewrite rule database. This tactic is required for CLAM's symbolic evaluation method and rippling [2] method.

$WFERT\_EQ\_TAC$: *direction* $*$ *int list* $\longrightarrow$ *tactic*. Implements weak fertilization [2]. *Fertilization* is the name given to the use of the induction hypothesis after the goal has been manipulated into a form in which the hypothesis can be used. When the hypothesis immediately solves the goal, the fertilization is *strong*. If not, the method is known as *weak fertilization*. An example of this is when the hypothesis and goal are equations and one side of the hypothesis appears as a subterm of the goal but the other side does not. The hypothesis can be used (by rewriting the subterm) but further work is required to prove the goal. The *direction* is either RIGHT or LEFT. The other argument is a director string, which is used to find the subterm in the goal to rewrite with the inductive hypothesis. If the direction is RIGHT, the inductive hypothesis $M = N$ is swapped around so that the goal is rewritten by $N = M$ at the occurrence; otherwise, the direction is LEFT and the inductive hypothesis is not switched around.

$SPEC\_TAC : term * term \longrightarrow tactic.$ [1] Generalizes the goal, by replacing the first term by the second (a variable), and then universally quantifying the variable in the goal. If the term to be generalized has some occurrences in the goal that are not free, then some outermost universal quantifications will be eliminated so that the generalization will work.

$IND\_TAC : (term * term)\ list \longrightarrow tactic.$ This is a somewhat involved and fragile operation, and we expect the interface to change. The list of pairs $(v_1, M_1), \ldots, (v_n, M_n)$ is used to identify the induction variables, to find the appropriate induction theorem, and finally to control the names used in the inductive cases of the proof. The sub-steps are as follows:

1. The list $v_1, \ldots, v_n$ designates the induction variables. We use it to manipulate the universal prefix of the goal so that it matches the binding structure of the scheme. That is, if the goal is

$$\Delta, \forall w_1 \ldots w_k . M$$

and $\{u_1, \ldots, u_j\} = \{w_1, \ldots, w_k\} - \{v_1, \ldots, v_n\}$, then the result of this step will be

$$\Delta, \forall v_1 \ldots v_n .\ \forall u_1 \ldots u_j .\ M.$$

2. The list $(v_1 : ty_1), \ldots, (v_n : ty_n)$ is used to form a tuple type $ty_1 \times \ldots \times ty_n$, and this is used as a key with which to lookup an induction scheme

$$\forall P : ty_1 \times \ldots \times ty_n \longrightarrow \text{bool}.\ < ind.\ cases > \supset \forall v_1 \ldots v_n .\ P(v_1, \ldots, v_n).$$

in the induction scheme database. This is a fragile step, since the tuple type will not, in general, uniquely identify an induction scheme. The current design is unable to handle induction schemes for simultaneous induction or those for a recursive datatype with more than one recursive constructor.

3. Rename variables in the selected induction scheme into those that CLAM used in planning. The list $M_1 \ldots M_n$ enumerates *templates* used by CLAM in the induction cases. These are used to obtain, for

[1] This is not the same as the built-in *SPEC_TAC* of HOL: the version described here provides slightly more functionality.

each case in the induction, a list of quantification prefixes that CLAM used. Currently, this design only works for datatypes having one base case and one inductive case. For example, if we have, by the previous steps, manipulated a goal into the form

$$\Delta, \forall l\ u_1 \ldots u_j.\ Q\ l\ u_1 \ldots u_j$$

and the method used by CLAM is `Induction(l, 'a list, v0 :: v1)` and the selected induction scheme is list induction

$$\forall P.P[\,]\ \wedge (\forall h\ t.\ P\ t \supset P(h :: t)) \supset \forall l.\ P\ l,$$

the instantiated and renamed induction scheme will be

$$\begin{array}{l}(\forall u_1 \ldots u_j.\ Q[\,]\ u_1 \ldots u_j)\ \wedge \\ (\forall v_0\ v_1\ u_1 \ldots u_j.\ Q\ v_1\ u_1 \ldots u_j\ \ \supset Q(v_0 :: v_1) u_1 \ldots u_j) \\ \supset \forall l\ u_1 \ldots u_j.\ Q\ l\ u_1 \ldots u_j.\end{array}$$

Only now can the induction scheme be applied in HOL.

It is crucial that the names used by CLAM are tracked in HOL, because of generalization: when CLAM generalizes a goal, it does so with an explicit term, which can have occurrences of variables from induction templates. For HOL to make the same generalization step, its goal must have corresponding occurrences of the corresponding term. In our design 'corresponding' means 'identical'.

6 Translation of plans to tactics

The translation goes in two stages; there is a translation from the type of methods into a type of tactic abstract syntax (*tacticAST*). Then there are translations from tacticAST into tactics and into concrete syntax (pretty-printing). Thus we get the picture of the conjoined systems shown in Figure 3.

Translation into tactics (*i.e.*, their internal representations) allows the plan to be applied to the goal without parsing and evaluating ML code. The generation of concrete syntax allows the tactic to be inserted in ML tactic scripts and used in other HOL sessions.

With the intermediate form, another correctness criterion emerges: the pretty-printed tactic ought to be behaviourally equivalent to the directly translated tactic:

$$eval_{ML} \circ pp = to\_tactic.$$

(In this statement, $eval_{ML}$ serves the purpose of an ML compiler: it takes concrete syntax to executable code.) This is a loose statement; among other lacunae, we are ignoring the ML context: there should be a requirement that the concrete

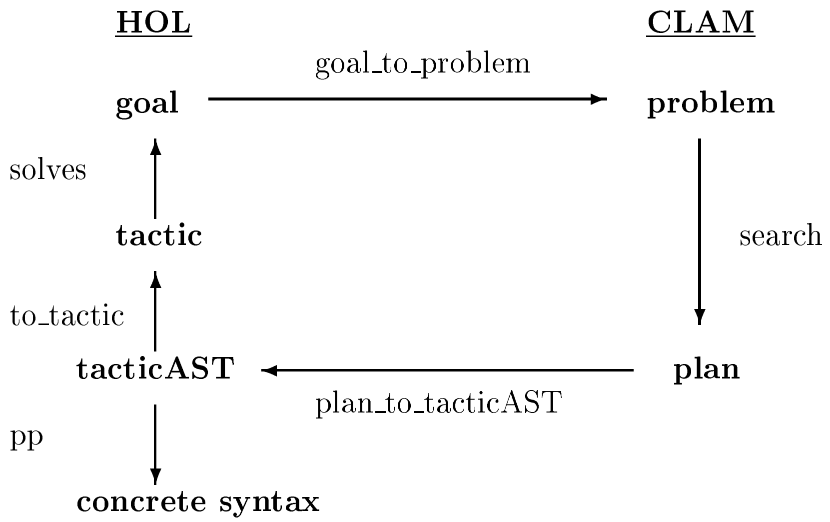

Figure 3: System Structure

syntax should compile and execute properly in any 'well-behaved' extension of the ML environment current at the time of invoking CLAM on the original goal. This will allow tactic scripts to be used between sessions, *e.g.*, using a script generated in a HOL-CLAM session in a session with only HOL present.

Another point to be made is that the translation of CLAM plans to HOL is achieved by parsing a plan file generated by CLAM into a type of plans inside HOL. Thus a refined diagram for the system as it currently stands is Figure 4.

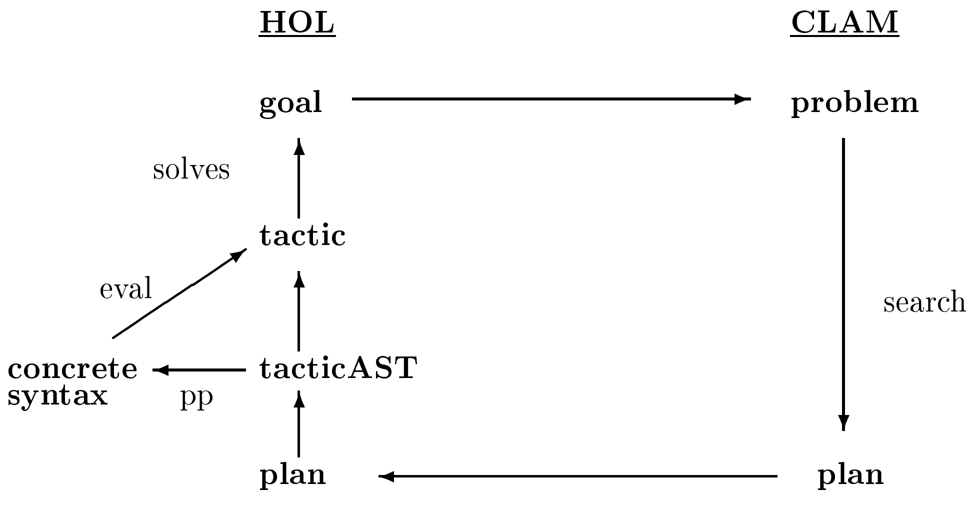

Figure 4: System Structure (refined)

6.1 tacticAST

The abstract syntax of tactics can be defined as follows:

```
datatype tacticAST = NO_TAC
                   | ALL_TAC
                   | PRIM of pprinter * tactic
                   | THEN of tacticAST * tacticAST
                   | THENL of tacticAST * tacticAST list
```

An item of type *tacticAST* can be seen as carrying a pretty-printer and a tactic at each leaf node, so that compound concrete syntax and tactics can be generated easily.

We now address the translation from methods to tacticAST. The translation $[\![\_]\!]$ in Figure 5 seems to work for simple examples. The notation $[\![L]\!]^*$ (where L is expected to be a non-empty list $[m_1, \ldots, m_n]$) equals

$$THEN([\![m_1]\!], \ldots, THEN([\![m_{n-1}]\!], [\![m_n]\!]) \ldots).$$

$$
\begin{array}{rcl}
\left[\!\!\left[\begin{array}{c}\textbf{ind\_strat}(\textbf{induction}(id, ty, tm),\\ [m_1, \ldots, m_n])\end{array}\right]\!\!\right] & = & THENL(IND[(id : ty, mk\_hol\_term\ tm)],\\
 & & \qquad [[\![m_1]\!], \ldots, [\![m_n]\!]])\\
[\![\textbf{elementary}\ m]\!] & = & ELEM\_TAC\\
[\![\textbf{reduction}\ [path, [s, (style, lr)]]]\!] & = & OCC\_RW\_TAC(path, s)\\
[\![\textbf{wave}\ [dir, path, [s, (style, lr), \_]]]\!] & = & OCC\_RW\_TAC(path, s)\\
[\![\textbf{weak\_fertilize}\ [\textbf{right}, \textbf{in}, path, v]]\!] & = & WFERT\_EQ\_TAC(RIGHT, path@[2, 1])\\
[\![\textbf{weak\_fertilize}\ [\textbf{left}, \textbf{in}, path, v]]\!] & = & WFERT\_EQ\_TAC(LEFT, path@[1, 1])\\
[\![m_1\ \textbf{then}\ [m]]\!] & = & THEN([\![m_1]\!], [\![m]\!])\\
[\![m\ \textbf{then}\ [m_1, \ldots, m_n]]\!] & = & THENL([\![m]\!], [[\![m_1]\!], \ldots, [\![m_n]\!]])\\
[\![\textbf{generalise}(tm, id, ty)]\!] & = & SPEC\_TAC(mk\_hol\_term(tm), (id : ty))\\
[\![\textbf{base\_case}(L)]\!] & = & [\![L]\!]^*\\
[\![\textbf{step\_case}(m)]\!] & = & [\![m]\!]\\
[\![\textbf{sym\_eval}(L)]\!] & = & [\![L]\!]^*\\
[\![\textbf{intro}(L)]\!] & = & [\![L]\!]^*\\
[\![\textbf{normalize\_term}(L)]\!] & = & [\![L]\!]^*\\
[\![\textbf{ripple}[dir, m]]\!] & = & [\![m]\!]\\
[\![\textbf{unblock\_then\_fertilize}[str, m]]\!] & = & [\![m]\!]\\
[\![\textbf{unblock\_fertilize\_lazy}(m)]\!] & = & ALL\_TAC\\
[\![\textbf{fertilize}[str, m]]\!] & = & [\![m]\!]\\
[\![\textbf{fertilize\_then\_ripple}\ L]\!] & = & [\![L]\!]^*\\
[\![\textbf{fertilize\_left\_or\_right}[dir, m]]\!] & = & [\![m]\!]\\
[\![L]\!] & = & [\![L]\!]^*
\end{array}
$$

Figure 5: Translation from methods to tactics

7 Examples

When CLAM is given the following goal

$$\forall m\ n.\ m * n = n * m$$

and the recursion equations for addition and multiplication in Peano arithmetic:

$$\begin{aligned} 0 + n &= n \\ SUC\ m + n &= SUC(m + n) \\ 0 * n &= 0 \\ SUC\ m * n &= m * n + n \end{aligned}$$

it returns the plan shown in Appendix A. Notice that no auxiliary lemmas are used in the planning process: this makes the task quite challenging.[2] The interest in this example is not primarily the theorem, which is quite simple; rather, it lies in how CLAM found the proof, by making multiple and nested inductions and generalizations, which overcame the artificially impoverished setting CLAM was asked to work in. On the HOL side, the result of direct translation of the plan from tacticAST to tactic solves the goal. The corresponding concrete syntax, given in Appendix B, when compiled, also solves the goal, as required by the triangle in Figure 4. That the translation could follow the twists and turns of this plan gives us some confidence that plans for more difficult goals will also be translated correctly into HOL tactics.

The following HOL theorems have also been planned and proved using the interface to CLAM:

$$\begin{aligned} \forall m\ n\ p.\ \ m + (n + p) &= (m + n) + p \\ \forall m\ n.\ \ m + n &= n + m \\ \forall m\ n\ p.\ \ m * (n * p) &= (m * n) * p \\ \forall l_1\ l_2\ l_3.\ \texttt{APPEND}\ l_1\ (\texttt{APPEND}\ l_2\ l_3) &= \texttt{APPEND}\ (\texttt{APPEND}\ l_1\ l_2)\ l_3 \\ \forall l_1\ l_2.\ \texttt{LENGTH}\ (\texttt{APPEND}\ l_1\ l_2) &= \texttt{LENGTH}\ l_1 + \texttt{LENGTH}\ l_2 \\ \forall f\ l_1\ l_2.\ \texttt{MAP}\ f\ (\texttt{APPEND}\ l_1\ l_2) &= \texttt{APPEND}\ (\texttt{MAP}\ f\ l_1)\ (\texttt{MAP}\ f\ l_2) \end{aligned}$$

It is important for any free variables in the HOL goal to be universally quantified before calling the CLAM interface because formulas with free variables are ill-formed in Oyster/CLAM.

8 Conclusions

This initial experiment has resulted in a combination of HOL and CLAM that allows a restricted set of HOL goals to be planned by CLAM and the resulting plan to be returned and applied in the HOL session.

[2] CLAM does, of course, make use of lemmas when searching for plans.

The current implementation has a number of limitations:

- CLAM currently has to be able to prove the theorems from its basic knowledge; there is as yet no provision for lemmas to be used as additional wave-rules, *etc.* *Wave-rules* are the annotated rules used by CLAM's rippling method[3] [2], a form of term rewriting. The annotations are used to restrict the application of rewite rules and to ensure termination.

- The mechanism is heavyweight. A new CLAM process is used for each call to CLAM. All supporting definitions and induction schemes have to be passed to CLAM on each call. This will be rectified by using sockets to communicate between the two processes. A CLAM process will be executed and left running. Definitions and induction schemes will be communicated only once.

- The syntax of CLAM methods known to the parser is not completely general; it has been developed only as far as is required by the examples.

- The translation from plans to tactics must be expanded to cover more complex plans.

- CLAM's representation of the chosen induction scheme must hold more information so that HOL can uniquely identify which scheme CLAM has chosen.

- Currently, fertilization is naively implemented: the particular inductive hypothesis used by CLAM is not used by the translation. So far, there have not been any problems, but we can imagine situations involving fertilization where our naive approach will fail.

- A more complicated protocol between the proof systems may be useful. For example, CLAM is always asking itself whether it has planned enough to solve the current abstract subgoal. It can be that asking HOL for advice at such points can be useful. Thus it may be that dialogues like the following example can be useful:

  HOL: Give me a tactic to solve `g0`.
  CLAM: OK ... Say, can you prove `g1`?
  HOL: Uh, yes, using a tactic invoked as "`foo 42`".
  CLAM: Then here's the plan for `g0`.

  Recursive but terminating dialogues would be a natural extension. A particular situation in which this kind of dialogue might be useful is when CLAM's rippling method gets blocked due to the absence of a required

[3] `http://www.dai.ed.ac.uk/staff/personal_pages/bundy/ripple.html`

wave-rule. When this happens, CLAM could ask HOL if it knows of an existing theorem for the wave-rule.

- For our simple examples, the nature of the underlying logic (constructive or classical) hasn't mattered. This can be explained by examining the translation of plans to tactics: the generated inference steps (chiefly induction, generalization, and rewriting) behave identically in both constructive and classical settings. It may be theoretically interesting to investigate whether CLAM can generate a plan where the fundamental differences between the logics become exposed.

References

[1] R. J. Boulton. Syn: A single language for specifying abstract syntax trees, lexical analysis, parsing and pretty-printing. Technical Report 390, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, UK, March 1996.

[2] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993.

[3] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7(3):303–324, September 1991.

[4] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[5] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

A Example Proof Plan

```
ind_strat
 (induction([(m:pnat)-s(v0)]) then
  [base_case
      ([sym_eval
           ([normalize_term([reduction([1,1],[times1,equ(left)])])])]),
   step_case
      (ripple
          (direction_out,
           wave(direction_out,[1,1],[times2,equ(left)],[])) then
       [unblock_then_fertilize
           (weak,
            unblock_fertilize_lazy([idtac]) then
            fertilize
               (weak,
                fertilize_then_ripple
                   (fertilize_left_or_right
                       (left,[weak_fertilize(left,in,[1],v1)]))))])])
then
[ind_strat
  (induction([(n:pnat)-s(v0)]) then
   [base_case
       ([sym_eval
            ([normalize_term([reduction([2,1],[times1,equ(left)])])]),
         elementary(identity)]),
    step_case
       (ripple
           (direction_out,
            wave(direction_out,[2,1],[times2,equ(left)],[])) then
        [unblock_then_fertilize
            (weak,
             unblock_fertilize_lazy([idtac]) then
             fertilize
                (weak,
                 fertilize_then_ripple
                    (fertilize_left_or_right
                        (right,
                         [weak_fertilize(right,in,[1],v1)]))))])]) then
[sym_eval([normalize_term([reduction([2,1],[plus1,equ(left)])])]) then
 [elementary(identity)]],
ind_strat
  (induction([(n:pnat)-s(v2)]) then
   [base_case
       ([sym_eval
            ([normalize_term
                 ([reduction([1,1,1],[times1,equ(left)]),
                   reduction([1,1],[plus1,equ(left)]),
                   reduction([2,1],[times1,equ(left)])])]),
         elementary(identity)]),
    step_case
       (ripple
           (direction_out,
            wave(direction_out,[2,1],[times2,equ(left)],[]) then
            [wave(direction_out,[1,1,1],[times2,equ(left)],[])]) then
        [unblock_then_fertilize
            (weak,
             unblock_fertilize_lazy([idtac]) then
             fertilize
                (weak,
                 fertilize_then_ripple
                    (fertilize_left_or_right
                        (right,
                         [weak_fertilize(right,in,[1],v3)]))))])]) then
```

```
[generalise(times(v2,v0),v4:pnat) then
 [ind_strat
     (induction([(v4:pnat)-s(v5)]) then
      [base_case
          ([sym_eval
               ([normalize_term
                    ([reduction([1,1,1],[plus1,equ(left)]),
                      reduction([1,2,1],[plus1,equ(left)])])])]),
       step_case
          (ripple
              (direction_out,
               wave(direction_out,[1,2,1],[plus2,equ(left)],[]) then
               [wave(direction_out,[2,1],[plus2,equ(left)],[]) then
                [wave(direction_out,[1,1,1],[plus2,equ(left)],[]) then
                 [wave(direction_out,[1,1],[plus2,equ(left)],[])]]])
           then
           [unblock_then_fertilize
               (weak,
                unblock_fertilize_lazy([idtac]) then
                fertilize
                   (weak,
                    fertilize_then_ripple
                       (fertilize_left_or_right
                           (right,[weak_fertilize(right,in,[1],v6)]))
                    then elementary(identity)))])]) then
  [ind_strat
      (induction([(v0:pnat)-s(v5)]) then
       [base_case
           ([sym_eval
                ([normalize_term
                     ([reduction([1,1],[plus1,equ(left)])])])]),
        step_case
           (ripple
               (direction_out,
                wave(direction_out,[1,1],[plus2,equ(left)],[])) then
            [unblock_then_fertilize
                (weak,
                 unblock_fertilize_lazy([idtac]) then
                 fertilize
                    (weak,
                     fertilize_then_ripple
                        (fertilize_left_or_right
                            (left,
                             [weak_fertilize(left,in,[1],v6)]))))])])
   then
   [ind_strat
       (induction([(v2:pnat)-s(v5)]) then
        [base_case
            ([sym_eval
                 ([normalize_term
                      ([reduction([2,1],[plus1,equ(left)])])]),
              elementary(identity)]),
         step_case
            (ripple
                (direction_out,
                 wave(direction_out,[2,1],[plus2,equ(left)],[])) then
             [unblock_then_fertilize
                 (weak,
                  unblock_fertilize_lazy([idtac]) then
                  fertilize
                     (weak,
                      fertilize_then_ripple
                         (fertilize_left_or_right
                             (right,[weak_fertilize(right,in,[1],v6)]))
                      then elementary(identity)))])]),
```

```
generalise(s(v5),v7:pnat) then
[ind_strat
    (induction([(v2:pnat)-s(v8)]) then
     [base_case
         ([sym_eval
              ([normalize_term
                   ([reduction([1,1,1],[plus1,equ(left)]),
                     reduction([2,1],[plus1,equ(left)])])]),
           elementary(intro(new[v7]) then [identity,wfftacs])]),
      step_case
         (ripple
             (direction_out,
              wave(direction_out,[2,1],[plus2,equ(left)],[]) then
              [wave(direction_out,[1,1,1],[plus2,equ(left)],[])])
          then
          [unblock_then_fertilize
              (weak,
               unblock_fertilize_lazy([idtac]) then
               fertilize
                  (weak,
                   fertilize_then_ripple
                      (fertilize_left_or_right
                          (right,[weak_fertilize(right,in,[1],v9)]))
                   then
                   elementary
                      (intro(new[v7]) then
                       [identity,wfftacs]))])])]]]]]
```

B Example Tactic

```
QIND_TAC (RENAME (theorem "num" "INDUCTION") [[], ["v0"]]) [`(m :num)`] THENL
[OCC_RW_TAC [1,1] "times1",
 OCC_RW_TAC [1,1] "times2" THEN ALL_TAC THEN WFERT_EQ_TAC (LEFT,[1, 1, 1])]
THENL
[QIND_TAC (RENAME (theorem "num" "INDUCTION") [[], ["v0"]]) [`(n :num)`]
 THENL
 [OCC_RW_TAC [2,1] "times1" THEN ELEM_TAC,
  OCC_RW_TAC [2,1] "times2" THEN ALL_TAC THEN WFERT_EQ_TAC (RIGHT,[1, 2, 1])]
 THEN OCC_RW_TAC [2,1] "plus1" THEN ELEM_TAC,
 QIND_TAC (RENAME (theorem "num" "INDUCTION") [[], ["v2"]]) [`(n :num)`]
 THENL
 [OCC_RW_TAC [1,1,1] "times1" THEN
  OCC_RW_TAC [1,1] "plus1" THEN OCC_RW_TAC [2,1] "times1" THEN ELEM_TAC,
  OCC_RW_TAC [2,1] "times2" THEN OCC_RW_TAC [1,1,1] "times2" THEN
  ALL_TAC THEN WFERT_EQ_TAC (RIGHT,[1, 2, 1])] THEN
 QSPEC_TAC (`(v2 :num) * (v0 :num)`,`(v4 :num)`) THEN
 QIND_TAC (RENAME (theorem "num" "INDUCTION") [[], ["v5"]]) [`(v4 :num)`]
 THENL
 [OCC_RW_TAC [1,1,1] "plus1" THEN OCC_RW_TAC [1,2,1] "plus1",
  OCC_RW_TAC [1,2,1] "plus2" THEN
  OCC_RW_TAC [2,1] "plus2" THEN
  OCC_RW_TAC [1,1,1] "plus2" THEN OCC_RW_TAC [1,1] "plus2" THEN
  ALL_TAC THEN WFERT_EQ_TAC (RIGHT,[1, 2, 1]) THEN ELEM_TAC] THEN
 QIND_TAC (RENAME (theorem "num" "INDUCTION") [[], ["v5"]]) [`(v0 :num)`]
 THENL
 [OCC_RW_TAC [1,1] "plus1",
  OCC_RW_TAC [1,1] "plus2" THEN ALL_TAC THEN WFERT_EQ_TAC (LEFT,[1, 1, 1])]
 THENL
 [QIND_TAC (RENAME (theorem "num" "INDUCTION") [[], ["v5"]]) [`(v2 :num)`]
  THENL
  [OCC_RW_TAC [2,1] "plus1" THEN ELEM_TAC,
   OCC_RW_TAC [2,1] "plus2" THEN
   ALL_TAC THEN WFERT_EQ_TAC (RIGHT,[1, 2, 1]) THEN ELEM_TAC],
  QSPEC_TAC (`(SUC :num -> num) (v5 :num)`,`(v7 :num)`) THEN
  QIND_TAC (RENAME (theorem "num" "INDUCTION") [[], ["v8"]]) [`(v2 :num)`]
  THENL
  [OCC_RW_TAC [1,1,1] "plus1" THEN OCC_RW_TAC [2,1] "plus1" THEN ELEM_TAC,
   OCC_RW_TAC [2,1] "plus2" THEN OCC_RW_TAC [1,1,1] "plus2" THEN
   ALL_TAC THEN WFERT_EQ_TAC (RIGHT,[1, 2, 1]) THEN ELEM_TAC]]]
```