A Prototype Interface Between CLAM and HOL

An Interface between CLAM and HOL*

Richard Boulton[1], Konrad Slind[2], Alan Bundy[1], Mike Gordon[2]

[1] Department of Artificial Intelligence, University of Edinburgh
80 South Bridge, Edinburgh EH1 1HN, Scotland
[2] University of Cambridge Computer Laboratory
New Museums Site, Pembroke Street
Cambridge CB2 3QG, England

Abstract. This paper describes an interface between the CLAM proof planner and the HOL interactive theorem prover. The interface sends HOL goals to CLAM for planning, and translates plans back into HOL tactics that solve the initial goals.

1 Introduction

HOL [6] 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 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[1]. 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.

CLAM [4] is a proof planning system for Oyster, a tactic-based implementation of the constructive type theory of Martin-Löf. Both Oyster and CLAM are implemented in Prolog. CLAM works by using formalized pre- and post-conditions of Oyster tactics as the basis of plan search. These high-level 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 [4, 3].

Our main goal in combining these two systems is to investigate the practical utility for hardware and software verification of Artificial Intelligence planning techniques. We aim to simplify the interaction between the user and the theorem prover and hence reduce the time it takes HOL users to verify systems. Eventually, we plan to test the combined system on industrially significant verification problems.

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

The system design treats CLAM as a black box and thus avoids the need to modify the CLAM implementation to suit the classical higher-order logic used by the HOL system. Instead, correspondences between syntactic features of HOL's logic and the constructive type theory of Oyster/CLAM are exploited. This allows a beneficial partition of concerns: the consistency of HOL is unaffected by interaction with CLAM; furthermore, advances in the theory and practice of proof planning in CLAM can be pursued without regard to impact on HOL.

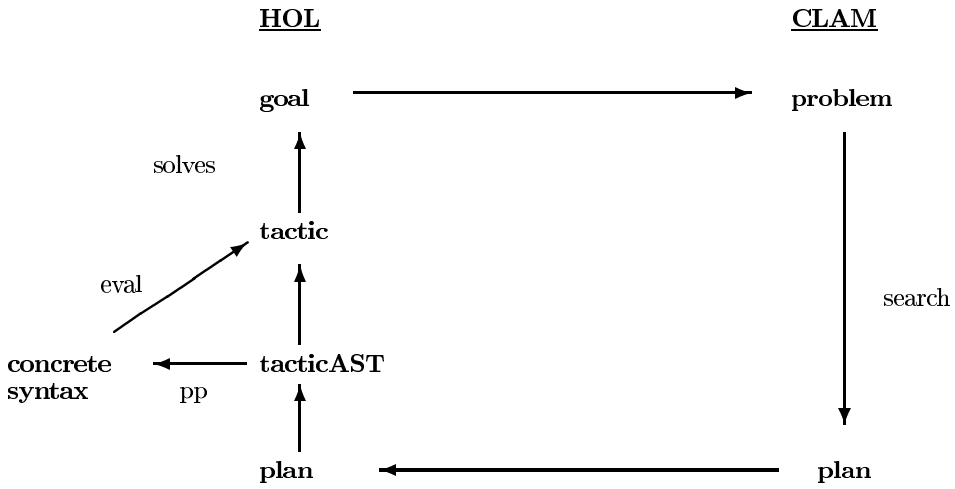

Fig. 1. System Structure

A diagram for the system is Fig. 1. The data flow is represented by arrows. 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 passed to the CLAM process. CLAM then attempts to plan the goal. If a plan is found, it is passed back to the HOL process. The plan is then translated to an intermediate form from which both tactics and (equivalent) concrete syntax can be generated. Finally, the tactic is applied to the HOL goal.

The combined system can be understood by examining the following interfaces:

- The translation from the HOL object language to Type Theory.
- The translation of plans to tactics.
- The correspondences that are maintained between the two systems. These include mappings between the rules, definitions, and induction schemes used by both systems, as well as the correspondences between explicit terms used in the proofs performed by both systems.
- The interprocess communication scheme.

2 Translation of the Object Language

Fig. 2 shows the translations from the HOL logic to Oyster syntax. Emphasized brackets ⟦ and ⟧ are used to denote the translation functions. The main syntactic distinction between the two systems is that types in HOL serve to categorize terms, and HOL formulae are just terms with boolean type. (In spite of this, we distinguish the two kinds of HOL term in our translation.) In contrast, type theory identifies types and formulae, using terms to denote proofs of formulae. The terms-as-proofs aspect of Oyster is completely ignored in our work. In summary, then, we have the translations $⟦\_⟧_{form}$, $⟦\_⟧_{term}$, and $⟦\_⟧_{type}$ for formulae, terms, and types. The translation of plans to tactics will need to map from Oyster terms to HOL terms; this will be denoted by $⟦\_⟧^{-1}_{term}$. The translation of (term) constants, $⟦\_⟧_{const}$, and of (term) variables, $⟦\_⟧_{var}$, along with the translation of type constants $⟦\_⟧_{consttype}$, and type variables $⟦\_⟧_{vartype}$ maps HOL names to names that are suitable for use in Prolog. We defer the discussion to Section 5.

$$
\begin{array}{ll}
⟦\mathrm{T}⟧_{form} & = \mathbf{\{true\}} \\
⟦\mathrm{F}⟧_{form} & = \mathbf{void} \\
⟦\forall(x : ty).\ t⟧_{form} & = ⟦x⟧_{var} : ⟦ty⟧_{type} \Rightarrow ⟦t⟧_{form} \\
⟦\exists(x : ty).\ t⟧_{form} & = ⟦x⟧_{var} : ⟦ty⟧_{type} \# ⟦t⟧_{form} \\
⟦\neg t⟧_{form} & = ⟦t⟧_{form} \Rightarrow \mathbf{void} \\
⟦t_1 \wedge t_2⟧_{form} & = ⟦t_1⟧_{form} \ \# \ ⟦t_2⟧_{form} \\
⟦t_1 \vee t_2⟧_{form} & = ⟦t_1⟧_{form} \setminus ⟦t_2⟧_{form} \\
⟦t_1 \Longrightarrow t_2⟧_{form} & = ⟦t_1⟧_{form} \Rightarrow ⟦t_2⟧_{form} \\
⟦(t_1 : \mathbf{bool}) = t_2⟧_{form} & = ⟦t_1⟧_{form} \Leftrightarrow ⟦t_2⟧_{form} \\
⟦(t_1 : ty) = t_2⟧_{form} & = ⟦t_1⟧_{term} = ⟦t_2⟧_{term} \text{ in } ⟦ty⟧_{type} \\
⟦t⟧_{form} & = ⟦t⟧_{term} \quad \text{if } t \text{ is not one of the forms above}
\end{array}
$$

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

$$
\begin{array}{ll}
⟦tyv⟧_{type} & = ⟦tyv⟧_{vartype} \\
⟦tyc⟧_{type} & = ⟦tyc⟧_{consttype} \\
⟦ty_1 \rightarrow ty_2⟧_{type} & = ⟦ty_1⟧_{type} \Rightarrow ⟦ty_2⟧_{type} \\
⟦(ty_1, \ldots, ty_n)tyc⟧_{type} & = ⟦tyc⟧_{consttype}(⟦ty_1⟧_{type}, \ldots, ⟦ty_n⟧_{type})
\end{array}
$$

Fig. 2. Translating HOL terms and types to Oyster syntax

False is translated to the empty type and true to the special type used to represent true in Oyster. Conjunction is translated to a product type, disjunction to a disjoint union type, implication to a function type, and negation to a

function type between the argument of the negation and the empty type. Universal quantification becomes a dependent function and existential quantification a dependent product. Equality between booleans is translated to if-and-only-if and other HOL equalities to equalities in CLAM. Decidability issues for the latter create some problems in planning. For example, CLAM's simplification (elementary) method only simplifies equalities over types it knows to have a decidable equality. The types communicated by HOL are all unknown to CLAM, so it knows nothing about their decidability. Hence it often fails to solve simple goals. Other HOL terms are translated almost directly into the corresponding type-theoretic constructs.

Certain kinds of formula (Boolean-valued terms) exist in HOL that are not explicitly handled by the translation in Fig. 2, *e.g.*, unique existence formulas, which can simply be split up into the separate conjuncts; and applications of HOLs indefinite description operator. Another point is that 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 and λ-abstraction.

The ML-style polymorphism of HOL terms is handled by translating HOL 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, **u(1)**. As Felty and Howe [5] point out, the domain should really be restricted to the inhabited types of **u(1)** since HOL types have to be non-empty. However, for the kinds of proof under consideration this will be of no consequence and, as as described in Sect. 3.3, can not lead to inconsistency in HOL.

3 Translation of Plans to Tactics

In this section, we will first describe the syntax of plans, and then proceed to discuss the basis tactics that had to be written to match the methods of CLAM. The section finishes with the translation between plans and tactics.

3.1 Plans

As mentioned previously, CLAM computes proof plans (which are currently represented by Prolog terms). Notionally, a plan is a hierarchy of *methods*, with the high-level methods representing steps in a proof that a human mathematician might use, e.g., induction, while low-level methods correspond to more detailed (and directed) syntactic manipulations. CLAM uses a wide variety of methods as it searches for solutions, among them being induction, generalization, rippling, and fertilization.

In the induction method, automatically inserted annotations are used to mark the differences between the induction conclusion and the induction hypothesis. The aim of the rippling method is to move these differences within the conclusion

so that the induction hypothesis can be applied to the conclusion, for example, by moving them to the outside of the term or into positions that correspond to universally quantified variables in the hypothesis. Wave-rules provide the means of moving the annotations. They are themselves annotated and the annotations in the wave-rules must correspond to the annotations in the induction conclusion for the rules to be applicable. The annotations include a direction of movement for the differences which ensures that rippling will terminate.

The following BNF describes the concrete syntax of a subset of the methods employed by CLAM:

method ::= **induction(***ident*, [((*ident* **:** *typ* **)-** *typ*,)* (*ident* **:** *typ* **)-** *typ*])
| **generalise(***typ* **,** *ident* **:** *typ* **)**
| **new**[(*ident* **,**)* *ident*]
| *ident* **(** (*argument* **,**)* *argument***)**
| [(*method* **,**)* *method*]
| *method* **then** *method*

argument ::= *method* | **in** | [(*number* **,**)* *number*]

In the above, an *ident* is a string of letters, digits and underscores, beginning with a lower-case letter, and a *number* is just a sequence of decimal digits.

3.2 Tactics

Tactics are a well-known method for backward proof. The original conception of Milner [7], 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.

Before describing the basis tactics in our translation, we must discuss some terminology.

- *Director strings*, *i.e.*, lists of numbers, are used to index subterms in the goal, usually for a rewriting tactic. 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'.

– *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 said to be *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.

Basis 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. (Actually the last two steps are integrated, by calling Boulton's HOL implementation of Nelson and Oppen's cooperating decision procedure mechanism [1].)

OCC_RW_TAC : *string* $\longrightarrow$ *int list* $\longrightarrow$ *tactic*. Provides rewriting (by a perhaps conditional equality) at a specified subterm in the goal. The first argument is the name of the rewrite rule in a rewrite rule database. This tactic is required for CLAM's symbolic evaluation, rippling [3], and weak fertilization methods.

ANT_RW_TAC : *string* $\longrightarrow$ *int list* $\longrightarrow$ *direction* $\longrightarrow$ *tactic*. Provides *implicational replacement* at a specified subterm in the goal. As in OCC_RW_TAC, director strings index the subterm that is to be rewritten. The first argument is the name of the rewrite rule in a rewrite rule database. This tactic is required for CLAM's symbolic evaluation, rippling, and weak fertilization methods. In an implicational replacement, suppose the theorem $\vdash M \supset N$ is given: there are two cases. When the direction is RIGHT, a match θ is found between the subterm and N (thus the subterm is identical to $\theta(N)$). Then the subterm is replaced by $\theta(M)$. When the direction is LEFT, things are reversed: the subterm is matched with M and $\theta(M)$ is replaced by $\theta(N)$.

$WFERT\_EQ\_TAC$: *direction* $*$ *int list* $\longrightarrow$ *tactic*. Implements weak fertilization [3] of equality goals. The *direction* is either RIGHT or LEFT. The other argument is a director string, which is used to locate the subterm in the goal to rewrite with the induction hypothesis. If the direction is RIGHT, the induction 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 induction hypothesis is not switched around.

$WFERT\_IMP\_TAC$: *direction* $*$ *int list* $\longrightarrow$ *tactic*. Implements weak fertilization of implicational goals, essentially by calling ANT_RW_TAC with assumptions until a replacement succeeds. This approach is too non-deterministic to work in complex settings, but it will have to suffice until more complete information about how CLAM names hypotheses can be extracted from proof plans.

SFERT_TAC : *tactic*. Implements strong fertilization: the first assumption that solves the goal is taken.

SPEC_TAC : $term * term \longrightarrow tactic$. 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.

ASM_EQ_TAC : *tactic*. All equational assumptions involving a variable on at least one side (such that the variable does not also occur on the opposite side) are eliminated by substituting for the variable throughout the goal.

CASE_SPLIT_TAC : $term \longrightarrow tactic$. Performs a case split on the specified term.

IND_TAC : $string \longrightarrow (term * term)\ list \longrightarrow tactic$. The first argument is used to get the induction scheme from the database. The second argument is a list of pairs $(v, M)_1, \ldots, (v, M)_n$; these are used to identify the induction variables and to control the names used in the inductive cases of the proof. The following are the steps that *IND_TAC* takes:

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. 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 each case in the induction, a list of quantification prefixes that CLAM used. For example, if we have, by the previous step, 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 was

```
Induction("listInduct", [(l, v0::v1)])
```

where `listInduct` denotes

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

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.

Notice that *bound variable names matter*: 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 the current setup, only the application of induction can introduce new variables into a proof, so it suffices to closely track the instantiations of the induction scheme, as we described above.

3.3 Translation of Plans to Tactics

This 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 (via pretty-printing). 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 scripts and used in other HOL sessions (*e.g.*, ones where CLAM may not be present). If the translation from plans to tactics is successful (which it normally is) the tactic is applied to the original HOL goal. Since CLAM uses heuristics, the tactic application may be unsuccessful; however, in practice it is very rare for CLAM to return an inappropriate plan. Most importantly, an inappropriate plan can not lead to a non-theorem being 'proved' in HOL because HOL invokes its own tactics (guided by the plan) in checking the proof.

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.

The translation from methods to `tacticAST` — denoted by $[\![\_]\!]$ — is given in Fig. 3. The notation $[\![L]\!]^*$ (where L is expected to be a non-empty list $[m_1, \ldots, m_n]$) equals

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

4 Examples

The following theorems give some idea of the sort of proofs which are currently proved fully automatically in the combined system. Some are known to be difficult to automate. The interest in many of these examples is not primarily the

theorem, which is usually fairly simple, but rather in how CLAM found the proof, by making multiple and nested inductions and generalizations. (Usually CLAM is working in a very sparse setting and must overcome this by recursively planning the lemmas it needs.)

$$\forall m\ n\ p.\ \ m + (n + p) = (m + n) + p$$
$$\forall m\ n.\ \ m + n = n + m$$
$$\forall m\ n\ p.\ \ m * (n * p) = (m * n) * p$$
$$\forall l_1\ l_2\ l_3.\ \texttt{APPEND}\ l_1\ (\texttt{APPEND}\ l_2\ l_3) = \texttt{APPEND}\ (\texttt{APPEND}\ l_1\ l_2)\ l_3$$
$$\forall l_1\ l_2.\ \texttt{LENGTH}\ (\texttt{APPEND}\ l_1\ l_2) = \texttt{LENGTH}\ l_1 + \texttt{LENGTH}\ l_2$$
$$\forall f\ l_1\ l_2.\ \texttt{MAP}\ f\ (\texttt{APPEND}\ l_1\ l_2) = \texttt{APPEND}\ (\texttt{MAP}\ f\ l_1)\ (\texttt{MAP}\ f\ l_2)$$
$$\forall x\ y.\ \texttt{REVERSE}\ (\texttt{APPEND}\ x\ y) = \texttt{APPEND}\ (\texttt{REVERSE}\ y)\ (\texttt{REVERSE}\ x)$$
$$\forall x\ m\ n.\ \texttt{APPEND}\ (\texttt{REPLICATE}\ x\ m)\ (\texttt{REPLICATE}\ x\ n) = \texttt{REPLICATE}\ x\ (m + n)$$
$$\forall x\ m\ n.\ \texttt{FLAT}\ (\texttt{REPLICATE}\ (\texttt{REPLICATE}\ x\ n)\ m) = \texttt{REPLICATE}\ x\ (m * n)$$

The functions here are curried. LENGTH computes the length of a list, APPEND concatenates two lists, MAP applies a function to every element of a list, REVERSE reverses a list, FLAT flattens a list of lists into one list (by iterated concatenation), and REPLICATE x n generates a list of n copies of x.
The following are some implicational theorems that can be proved in the system:

$$\forall x\ y.\ (x * y = 0) \supset (x = 0) \vee (y = 0)$$
$$\forall x\ y\ z.\ \neg(\texttt{LESS}\ z\ x) \wedge \neg(\texttt{LESS}\ x\ y) \supset \neg(\texttt{LESS}\ z\ y)$$
$$\forall x\ y\ z.\ \texttt{LESS}\ x\ y \supset \texttt{LESS}\ x\ (y + z)$$
$$\forall x\ l.\ \texttt{MEMBER}\ x\ (\texttt{SORT}\ l) \supset \texttt{MEMBER}\ x\ l$$
$$\forall x\ l.\ \texttt{MEMBER}\ x\ l\ \supset \texttt{MEMBER}\ x\ (\texttt{SORT}\ l)$$

These examples use the following definitions/rules:

$$\forall x.\ \texttt{LESS}\ x\ 0 = \texttt{F}$$
$$\forall y.\ \texttt{LESS}\ 0\ (\texttt{SUC}\ y) = \texttt{T}$$
$$\forall x\ y.\ \texttt{LESS}\ (\texttt{SUC}\ x)\ (\texttt{SUC}\ y) = \texttt{LESS}\ x\ y$$
$$\forall m.\ \texttt{MEMBER}\ m\ [\,] = \texttt{F}$$
$$\forall el\ h\ l.\ (el = h) \supset (\texttt{MEMBER}\ el\ (\texttt{CONS}\ h\ l) = \texttt{T})$$
$$\forall el\ h\ l.\ \neg(el = h) \supset (\texttt{MEMBER}\ el\ (\texttt{CONS}\ h\ l) = \texttt{MEMBER}\ el\ l)$$
$$\forall n.\ \texttt{INSERT}\ n\ [\,] = [n]$$
$$\forall n\ h\ t.\ \texttt{LESS}\ n\ h \supset (\texttt{INSERT}\ n\ (\texttt{CONS}\ h\ t) = \texttt{CONS}\ n\ (\texttt{CONS}\ h\ t))$$
$$\forall n\ h\ t.\ \neg(\texttt{LESS}\ n\ h) \supset (\texttt{INSERT}\ n\ (\texttt{CONS}\ h\ t) = \texttt{CONS}\ h\ (\texttt{INSERT}\ n\ t))$$
$$\texttt{SORT}\ [\,] = [\,]$$
$$\texttt{SORT}(\texttt{CONS}\ h\ t) = \texttt{INSERT}\ h\ (\texttt{SORT}\ t)$$

and the induction scheme

$$\forall P.$$
$$(\forall y.\ P\ 0\ y) \supset (\forall x.\ P\ x\ 0) \supset (\forall x\ y.\ P\ x\ y \supset P\ (\texttt{SUC}\ x)\ (\texttt{SUC}\ y))$$
$$\supset \forall x\ y.\ P\ x\ y$$

has been added to the usual set of induction schemes that CLAM is allowed to use (which includes the induction theorems for lists and natural numbers).

5 Correspondences

Identifying the correspondences needed between the two systems — and how they should be maintained — is an important matter. The following is a summary of the information that needs to be maintained.

1. The two systems need to agree on which definitions, wave rules, and induction schemes are used. The design currently uses a database in HOL indexed by names. The naming scheme is independent of whatever internal bindings are used in either system. For example, suppose HOL sends CLAM a wave rule, which is subsequently used in a plan. When the plan is returned to HOL, the name appearing in the plan is used to index into the database; the effect is that neither CLAM nor HOL has to know about the location of objects in the other system.
2. The object language *lexi* of the two systems must be respected. Constant 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.
3. The CLAM name used for each HOL name is stored and used in $[\![\_]\!]^{-1}_{term}$ when translating CLAM plans to HOL tactics.
4. 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_'. As an example, the HOL formula expressing the associativity of the function for appending two lists is:

$$\forall l_1\, l_2\, l_3.\ \texttt{APPEND}\ l_1\ (\texttt{APPEND}\ l_2\ l_3) = \texttt{APPEND}\ (\texttt{APPEND}\ l_1\ l_2)\ l_3.$$

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)
```

5. The CLAM implementation we use has been modified to provide some independence from Oyster and the built-in types and induction schemes of the CLAM library. Before using CLAM in a proof attempt, supporting definitions and induction schemes are sent from HOL. These are translated as for the goals. Additional rewrite rules may also be sent. CLAM processes these according to their structure (relative to the definitions it knows about) and records them for further use. They may be used as reduction rules, wave-rules, etc. Thus one can — from HOL — control the setting (wave rules, definitions, and induction schemes) in which CLAM works. Each goal can be planned in an independent setting, which makes some aspects of correspondence maintenance easy to handle.

6 Interprocess Communication

In the current system, the HOL user starts a CLAM process (or connects to one that is already running) from within a HOL session. The output from the CLAM process either appears as output in the HOL session or in a separate window, depending on how the CLAM process was invoked. Once the CLAM process has initialized, communication between the CLAM and HOL processes is via *sockets*. (Sockets are an inter-process communication mechanism provided by many versions of the Unix operating system. A socket behaves much like a bidirectional Unix pipe would.) In our current set-up, the socket may be either a local file-system socket or a socket connection over the Internet.

Recalling the control flow from Fig. 1, first the HOL formula (goal) to be proved is translated into the abstract syntax of Oyster's logic and the result is written to the socket (in concrete syntax) as a Prolog goal. The CLAM process waits for a message from HOL and, on receiving one it recognizes, executes it and either returns the result back down the socket or sends a handshaking message. After the goal has been sent, the CLAM process is instructed to plan a proof for it.If a plan is found it is returned to the HOL process via the socket. The HOL process then regains control and parses the plan into an ML data structure. Then the translation into tactics from Section 3.3 is performed, and finally the tactic is applied to the goal.

The use of sockets to communicate between the two processes allows the CLAM process to be left running between proof attempts, so definitions, induction schemes, etc., have to be communicated only once. An earlier version of the interface used files to communicate, thus requiring a fresh CLAM process to be executed for each proof attempt, with all the overhead that entails.

7 Summary

This experiment has resulted in a combination of HOL and CLAM that allows some HOL goals to be planned by CLAM and the resulting plan to be returned and applied in the HOL session. We think our work is of interest because of the following points:

- In our 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 various forms of rewriting) behave identically in both constructive and classical settings. Instead of an interesting but difficult interpretation of type theory in the HOL logic, our design is simple while still being secure. However, notice that it can happen that CLAM may not find a plan for a simple HOL theorem because of the constructive character of Oyster.
- The system only works because we maintain correspondences between the names of (term and type) constants, wave rules, definitions, and induction schemes in both systems. These can be labelled as "static". More dynamic correspondences also need to be maintained between variables and other terms introduced by CLAM in planning, and the variables and terms appearing in the resulting tactic.
- The details of the translations and lexical requirements can be re-used by other logic mechanizations wishing to employ CLAM's proof planning resources.
- The interprocess communication infrastructure we have built seems to be usable in a variety of other systems.
- 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 [2]. Hence the mappings both to and from CLAM have largely been automatically generated. Clearly, this technology will be of use to others attempting work similar to ours.

We see the following as future work.

- CLAM's handling of induction schemes ought to be generalized to allow more induction schemes. One opportunity is to extend the representation to the schemes returned when the TFL package is used to define functions by wellfounded recursion [9, 8].
- The current design allows only one chance for CLAM to plan a proof, and it seems obvious that a more complicated protocol between the proof systems may be useful, so that a kind of 'dialogue' occurs between the two systems. We have designed such a protocol, and are about to prototype it.

References

1. R. J. Boulton. Combining decision procedures in the HOL system. In *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, volume 971 of *Lecture Notes in Computer Science*. Springer, 1995.
2. 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.

3. 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.
4. 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.
5. A. P. Felty and D. J. Howe. Hybrid interactive theorem proving using Nuprl and HOL. In W. McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction (CADE-14)*, volume 1249 of *Lecture Notes in Artificial Intelligence*, Townsville, North Queensland, Australia, July 1997. Springer.
6. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
7. 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.
8. K. Slind. Function Definition in Higher Order Logic. In *Proceedings of the Conference on Theorem Proving in Higher Order Logics (TPHOLs96)*, volume 1125 of *Lecture Notes in Computer Science*, Abo, Finland, August 1996. Springer Verlag.
9. K. Slind. Derivation and Use of Induction Schemes in Higher-Order Logic. In *Proceedings of the Conference on Theorem Proving in Higher Order Logics (TPHOLs97)*, volume 1275 of *Lecture Notes in Computer Science*, Murray Hill, New Jersey, USA, August 1997. Springer Verlag.

$$
\begin{aligned}
[\![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{casesplit}[\textbf{Disjunction}[\neg M, M]]]\!] &= CASE\_SPLIT\_TAC([\![M]\!]^{-1}_{term}) \\
\\
\left[\!\!\left[\begin{array}{l}\textbf{ind\_strat}(\textbf{induction}(id,\\ \quad [(v,M)_1,\ldots,(v,M)_n]),[m_1,\ldots,m_k])\end{array}\right]\!\!\right] &= \begin{array}{l} THENL(IND\_TAC(id,\\ \quad [(v,[\![M]\!]^{-1}_{term})_1,\ldots,(v,[\![M]\!]^{-1}_{term})_n]),\\ \quad [[\![m_1]\!],\ldots,[\![m_k]\!]])\end{array} \\
[\![\textbf{generalise}(tm,id,ty)]\!] &= SPEC\_TAC([\![tm]\!]^{-1}_{term},[\![id:ty]\!]^{-1}_{term}) \\
[\![\textbf{reduction } (=,[path,[s,(style,lr)]])]\!] &= OCC\_RW\_TAC\ s\ path\ lr \\
[\![\textbf{reduction } (\equiv,[path,[s,(style,lr)]])]\!] &= OCC\_RW\_TAC\ s\ path\ lr \\
[\![\textbf{reduction } (imp,[path,[s,(style,lr)]])]\!] &= ANT\_RW\_TAC\ s\ path\ lr \\
[\![\textbf{wave } [dir,path,[s,(=,lr),\_]]]\!] &= OCC\_RW\_TAC\ s\ path\ lr \\
[\![\textbf{wave } [dir,path,[s,(\equiv,lr),\_]]]\!] &= OCC\_RW\_TAC\ s\ path\ lr \\
[\![\textbf{wave } [dir,path,[s,(imp,lr),\_]]]\!] &= ANT\_RW\_TAC\ s\ path\ lr \\
[\![\textbf{unblock}(wave\_front,path,s,\equiv,lr)]\!] &= OCC\_RW\_TAC\ s\ path\ lr \\
[\![\textbf{weak\_fertilize } [\textbf{RIGHT},\textbf{in},path,\_]]\!] &= WFERT\_EQ\_TAC(path@[2,1],\textbf{RIGHT}) \\
[\![\textbf{weak\_fertilize } [\textbf{LEFT},\textbf{in},path,\_]]\!] &= WFERT\_EQ\_TAC(path@[1,1],\textbf{LEFT}) \\
[\![\textbf{weak\_fertilize } [\textbf{LEFT},\Rightarrow,path,\_]]\!] &= WFERT\_IMP\_TAC(path@[1],\textbf{LEFT}) \\
[\![\textbf{weak\_fertilize } [\textbf{RIGHT},\Rightarrow,path,\_]]\!] &= WFERT\_IMP\_TAC(path@[2],\textbf{RIGHT}) \\
[\![\textbf{fertilize } (\textbf{strong})]\!] &= SFERT\_TAC \\
[\![\textbf{normal}(\textbf{univ\_intro},\_)]\!] &= GEN\_TAC \\
[\![\textbf{normal}(\textbf{imply\_intro},\_)]\!] &= DISCH\_TAC \\
[\![\textbf{normal}(\textbf{conjunct\_elim},\_)]\!] &= ASM\_CONJ\_TAC \\
[\![\textbf{equal } \_]\!] &= ASM\_EQ\_TAC \\
[\![\textbf{elementary } m]\!] &= ELEM\_TAC \\
[\![\textbf{base\_case}(L)]\!] &= [\![L]\!]^* \\
[\![\textbf{step\_case}(m)]\!] &= [\![m]\!] \\
[\![\textbf{sym\_eval}(L)]\!] &= [\![L]\!]^* \\
[\![\textbf{intro}(L)]\!] &= [\![L]\!]^* \\
[\![\textbf{normalize}(L)]\!] &= [\![L]\!]^* \\
[\![\textbf{normalize\_term}(L)]\!] &= [\![L]\!]^* \\
[\![\textbf{ripple}[dir,m]]\!] &= [\![m]\!] \\
[\![\textbf{ripple\_and\_cancel}(L)]\!] &= [\![L]\!]^* \\
[\![\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{aligned}
$$

Fig. 3. Translation from methods to tactics