Effective Support for Mutually Recursive Types

Peter V. Homeier

Computer and Information Science Department, University of Pennsylvania
Philadelphia, Pennsylvania 19104-6389 USA
http://www.cis.upenn.edu/~homeier
homeier@saul.cis.upenn.edu

Abstract. For purposes of formal analysis, it is common to form a model of a system within a logic. This sometimes requires the introduction of new types which are mutually recursive. HOL90 has possessed for several years now two excellent libraries for mutually recursive types. Despite their powerful functionality, they are discovered to be difficult to use in practice. The input specifications of the mutually recursive types are laborious, the support for defining functions on these types is limited, and there is no built-in automated support for proving theorems about these types and functions, beyond proving the induction theorem. We address these software engineering issues in this paper, by the presentation of a new library, `mutual`, which includes all the definitional power of the others with a succinct interface and tools to facilitate the practical creation of function definitions and proofs. Researchers can now find this HOL90 software available from the Web.

1 Introduction

Modeling systems in HOL for study of their properties often requires the creation of new types in the logic. One of HOL's strengths has been its powerful yet completely definitional and sound tools for creating and using new types, notably the excellent type definition package by Tom Melham [1]. This package provides facilities for specifying new recursive types in a concise syntax, automatically constructs the definitions required, and proves various theorems needed for using the new types, such as the type axiom, the structural induction theorem, the one-to-one and distinctiveness properties of the constructors, and the cases theorem. In addition to these theorems, the package also provides a tool for defining new functions on the new types, and a tactic for proving theorems about the new functions and types. This package has the appealing and enduring advantages of being easy to use, efficiently implemented, and completely sound.

In fact, if one were to look for a flaw in this package, the only place where one might reasonably criticize it might be in its scope. The package can only create one new recursive type at a time. This is fine for many applications, but there is a significant class of systems which evidence several types, where each type is defined in terms of itself and the others. These are called *mutually recursive types.* An example is the syntax structures of a programming language, where the syntax often is mutually recursive in interesting ways.

There are programming techniques that can be used to define these mutually recursive types using the standard type definition package. One new type is defined, which is a disjoint sum of all the mutually recursive types, with a tag to discriminate between the types. But these methods can be awkward to use, and do not provide the simplicity and ease-of-use that many users are familiar with from the standard package.

In 1991 Myra VanInwegen was working on her Ph.D. thesis [2] with Elsa Gunter, creating a definition of the syntax and semantics of SML within the HOL logic. SML is a language with mutually recursive syntax. To aid in representing this syntax by definitions of mutually recursive types, Gunter and VanInwegen created the `mutrec` library in the summer of 1991 [3]. This library was a significant addition to the functionality of HOL90, and provided impetus for users to switch to HOL90. Nevertheless, Gunter saw the need for additional functionality, and in the summer of 1992, Gunter jointly with Healfdene Goguen followed this library with the `nested_rec` library, with the ability to handle more general specifications of new types, including the use of pre-existing type operators such as `list`, `prod`, and `sum` in the specifications.

These new libraries provided new functionality that was greatly needed by many users of HOL who did not have the expertise to use the programming techniques mentioned before. However, these libraries came in a relatively rough condition, compared with the standard type definition package. Despite their useful functionality, these libraries were hard to use in practice, requiring laborious specifications of the types. In addition, the tool provided for creating definitions of new functions on the new types was restricted. With the most frequent impact, there was no tool provided analogous to the standard type definition package's `INDUCT_THEN` tactic, which helped to automate proofs of properties concerning a new type. One needed to use the induction theorem directly and manually, with a reduction in both ease and clarity.

In this paper we describe a new library for HOL, called `mutual`, which builds upon the functionality provided by the `nested_rec` library, providing tools to ease the creation and use of mutually recursive types, including nested recursion. The problems mentioned above are addressed, among other issues. This library makes direct use of the `nested_rec` library for creating the definitions, but adds functions to provide a more convenient and practical interface.

This new library adds no significantly new definitional functionality. Nevertheless, it can be considered a strict improvement over the pre-existing libraries. The thesis of this paper is that "ease-of-use" is an important feature of any package, which may be overlooked in the drive for increased functionality. The `mutual` library may be considered an illustrative example of this thesis.

The organization of this paper is as follows. Section 2 discusses previous approaches. In Section 3 we describe how to load the `mutual` library. Section 4 demonstrates the facilities for creating new definitions of mutually recursive types, including nested recursion. Section 5 describes the tool for defining new mutually recursive functions on those new types. Section 6 describes a tactic for proofs by mutual structural induction, and in Section 7 we conclude.

2 Previous Work

The fundamental tool for defining new types in HOL is `new_type_definition`, an ML function. This function requires the user to supply a theorem of the existence of values of the new type, and in addition create a bijection and its inverse between the new type and its representation. This involves a good deal of low-level detailed work that could be characterized as remote from the user's intuitive conception of the type.

Probably the most commonly-used mechanism for defining new recursive types in HOL is the recursive type definition package by Tom Melham, as described in Chapter 20 of [1]. This provides ML functions to define a single new concrete recursive type, with its constructor functions. The package also provides tools to produce theorems that state the axiomatization of the type, its induction principle, the disjointness and one-to-one principles of its constructors, and the cases theorem. New recursive functions in the HOL logic can be defined on the structure of this new type. In addition, the package provides the `INDUCT_THEN` tactic for proving properties about the new type and functions by structural induction.

Say we wished to define binary trees as either leaves or nodes with two child trees. A typical type definition in HOL88 would be

```
#let btree_Axiom =
#    define_type
#       `btree_Axiom` `btree = LEAF * | NODE btree btree`;;
btree_Axiom =
|- !f0 f1.
    ?! fn.
     (!x. fn(LEAF x) = f0 x) /\
     (!b1 b2. fn(NODE b1 b2) = f1(fn b1)(fn b2)b1 b2)
```

The same type definition in HOL90 would be

```
- val btree_Axiom =
=      define_type{
=         name = "btree_Axiom",
=         type_spec=`btree = LEAF of 'a | NODE of btree => btree`,
=         fixities = [Prefix,Prefix] };
val btree_Axiom =
  |- !f0 f1.
       ?!fn.
         (!x. fn (LEAF x) = f0 x) /\
         (!b1 b2. fn (NODE b1 b2) = f1 (fn b1) (fn b2) b1 b2)
```

This package has enjoyed great popularity, in no small part due to the excellent quality of the user interface provided and the efficient implementation of the tools. Last but not least, the documentation is complete and quite clear. Its obvious value has mandated its inclusion in the core HOL system, rather than as a library, to be readily available to all users.

This excellent package has only one significant limitation; it does not directly support mutually recursive types. To address this need, the `mutrec` library was created for HOL90 by Myra VanInwegen and Elsa Gunter in 1991. It provides a means to define mutually recursive types.

This brought the creation of mutually recursive types within the reach of many HOL users. However, Elsa Gunter was not satisfied with the functionality of this library, and working jointly with Healfdene Goguen, followed it a year later with an even more powerful library, `nested_rec`, which added the ability to refer to the new types being defined within some type operators, such as `list`, `sum`, and `prod`, so long as the proper theorems describing their axiomatization were also supplied.

Both these libraries, `mutrec` and `nested_rec`, were powerful additions to the set of tools in HOL for modeling general systems within the logic. However, these libraries also had certain weaknesses as well, in that they were not as well polished and easy to use as the standard recursive type definition package.

The most important areas needing improvement are these:

1. The specification of the input grammar is verbose, hard to compose and read, easy to get wrong, and very different from the simple input that the standard recursive type definition package requires.
2. When defining new functions on the new types, the functions are limited to exactly one argument, which must be one of the types defined.
3. No tactics are provided to aid in proofs by induction on the structure of the mutually recursive types, beyond proving the induction theorem.

Of these three, the first is the most obvious need; yet the last may be the most important, because for every new type definition, there may be many new functions defined, and for each new function defined, there may be many new properties proved about it.

3 Loading the Library

The `mutual` library is designed to reside in the `contrib` directory. Once installed, we load the `mutual` library by

```
load_library_in_place (find_library "mutual");
```

This will load several other libraries as well, including `mutrec` and `nested_rec`. Loading the `mutual` library will create the functors

`DefineMutualTypesFunc` and `StringDefineMutualTypesFunc`,

and also the structure `mutualLib`. The functors are used to create new mutually recursive types; they vary only in whether they take a `term frag list` or a `string` as the input specification. The structure `mutualLib` has the signature

```
structure mutualLib :
  sig
    val define_mutual_functions
        : {def:term, fixities:fixity list option,
           name:string, rec_axiom:thm}
          -> thm
    val MUTUAL_INDUCT_THEN : thm -> thm_tactic -> tactic
    val list_Axiom : thm
    val prod_Axiom : thm
    val sum_Axiom : thm
  end
```

This includes a function to define functions on the mutual types, a tactic to perform mutual structural induction, and three useful theorems for defining nested mutually recursive types. Opening this structure makes these values available at the top level:

```
- open mutualLib;
open mutualLib
val define_mutual_functions = fn
  : {def:term, fixities:fixity list option,
     name:string, rec_axiom:thm} -> thm
val MUTUAL_INDUCT_THEN = fn : thm -> thm_tactic -> tactic
val list_Axiom =
  |- !x f. ?!fn1. (fn1 [] = x) /\
                  (!h t. fn1 (CONS h t) = f (fn1 t) h t) : thm
val prod_Axiom = |- !f. ?!g. !x y. g (x,y) = f x y : thm
val sum_Axiom = |- !f g. ?!h. (!x. h (INL x) = f x) /\
                              (!x. h (INR x) = g x) : thm
```

4 Definitions of Mutually Recursive Types

Mutually recursive types, with possible nesting of the recursion, are defined using either the `DefineMutualTypesFunc` or `StringDefineMutualTypesFunc` functors. This is best exhibited through an example. Consider the following BNF grammar:

$$\begin{array}{lcl} atexp & = & var \mid \mathbf{let}\ dec\ \mathbf{in}\ exp \\ exp & = & atexp \mid exp\ \ atexp \mid match \\ match & = & rule\ list \\ rule & = & pat\ => \ exp \\ dec & = & valbind \mid \mathbf{local}\ dec\ \mathbf{in}\ dec \mid dec\ ;\ dec \\ valbind & = & \mathbf{bind}\ (pat\ \mathbf{to}\ exp)\ list \mid \mathbf{rec}\ valbind \\ pat & = & \mathbf{wild\_pat} \mid var \end{array}$$

Figure 1 shows the need for mutual recursion by the presence of cycles.

If we represent the types of variables as a type variable `'var`, then these types may be defined as follows.

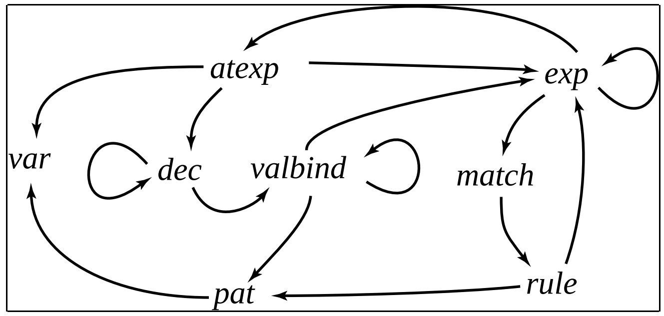

Figure 1: Dependencies among language phrases.

```
structure GramDef =
   DefineMutualTypesFunc
      (val name = "syntax"
       val recursor_thms = [list_Axiom,prod_Axiom]
       val types_spec =

        ‘ atexp = var_exp of ’var
                | let_exp of dec => exp ;

          exp = aexp of atexp
              | app_exp of exp => atexp
              | fn_exp of match ;

          match = match of rule list ;

          rule = rule of pat => exp ;

          dec = val_dec of valbind
              | local_dec of dec => dec
              | seq_dec of dec => dec ;

          valbind = bind of (pat # exp) list
                  | rec_bind of valbind ;

          pat = wild_pat
              | var_pat of ’var ‘ );
```

This closely matches the BNF presented above, and is an improvement over the style of specifying such mutually recursive types in the `nested_rec` library. Using that library requires one to create a structure with specific fields, including a type specification with a recursive record structure. This is illustrated on the next page, where the specification of the above example is given.

```
val var_ty = (==`:'var`==);

local
    structure Ast : NestedRecTypeInputSig =
    struct
    structure DefTypeInfo = DefTypeInfo
    open DefTypeInfo
    val def_type_spec =
    [{type_name = "atexp",
      constructors =
          [{name = "var_exp",
            arg_info = [existing var_ty]},
           {name = "let_exp",
            arg_info = [being_defined "dec",
                        being_defined "exp"]}]},
     {type_name = "exp",
      constructors =
          [{name = "aexp",
            arg_info = [being_defined "atexp"]},
           {name = "app_exp",
            arg_info = [being_defined "exp",
                        being_defined "atexp"]},
           {name = "fn_exp",
            arg_info = [being_defined "match"]}]},
     {type_name = "match",
      constructors =
          [{name = "match",
            arg_info = [type_op{Tyop="list",
                                Args=[being_defined "rule"]}]}]},
     {type_name = "rule",
      constructors =
          [{name = "rule",
            arg_info = [being_defined "pat",
                        being_defined "exp"]}]},
     {type_name = "dec",
      constructors =
          [{name = "val_dec",
            arg_info = [being_defined "valbind"]},
           {name = "local_dec",
            arg_info = [being_defined "dec",
                        being_defined "dec"]},
           {name = "seq_dec",
            arg_info = [being_defined "dec",
                        being_defined "dec"]}]},
```

```
     {type_name = "valbind",
      constructors =
          [{name = "bind",
            arg_info=[type_op
                        {Tyop="list",
                         Args=[type_op
                                 {Tyop="prod",
                                  Args=[being_defined "pat",
                                        being_defined "exp"]}]}]}]},
           {name = "rec_bind",
            arg_info = [being_defined "valbind"]}]},
     {type_name = "pat",
      constructors =
          [{name = "wild_pat",
            arg_info = []},
           {name = "var_pat",
            arg_info = [existing var_ty]}]}];

       val recursor_thms = [list_Axiom,prod_Axiom]

       val New_Ty_Existence_Thm_Name = "syntax_existence_thm"
       val New_Ty_Induct_Thm_Name = "syntax_induction_thm"
       val New_Ty_Uniqueness_Thm_Name = "syntax_uniqueness_thm"
       val Constructors_Distinct_Thm_Name =
                "syntax_constructors_distinct"
       val Constructors_One_One_Thm_Name =
               "syntax_constructors_one_one"
       val Cases_Thm_Name = "syntax_cases"

    end (* struct *)
in
    (* Prove the defining theorems for the type *)
    structure GramDef = NestedRecTypeFunc (Ast);
end;
```

The `mutual` library can condense the above specification due to the introduction of a parser for a mututally recursive types specification language. The language is modeled on that used in the standard HOL type definition package, and is the same except for having multiple type specifications, separated by semicolons. This parser is in fact very similar to the normal HOL90 parser, and could be integrated with it. The parser takes the specification as given in the shorter version above and parses it, creating the longer version seen above, which is then used as an argument in calling the `nested_rec` package.

The `mutual` library does give up some freedom present in `nested_rec`, for choosing the names of the theorems produced. In `nested_rec`, the six theorems are stored in the current theory under names which are specified independently for each theorem. In the `mutual` library tools, only the root is specified by the user (in the above example, as the string `"syntax"`) and the name of each theorem is created in a standard fashion by appending a standard suffix for that theorem, namely "`_exists`," "`_induct`," "`_unique`," "`_distinct`," "`_one_one`," or "`_cases`." This was chosen to ease the use of this tool and improve standardization of naming.

Note that the recursor theorems included with the specification must include the axiomatization theorems for all type operators used to nest types being defined, including new, user-defined type operators as well. It is a common error to leave some out; yet unnecessary ones may confuse the tool.

The `DefineMutualTypesFunc` functor creates a new structure, as well as storing the six resulting theorems in the current theory. The new structure has signature `DefTypeSig`, and contains these theorems as well.

```
signature DefTypeSig =
    sig
      type thm
      val New_Ty_Induct_Thm :thm
      val New_Ty_Uniqueness_Thm :thm
      val New_Ty_Existence_Thm :thm
      val Constructors_Distinct_Thm : thm
      val Constructors_One_One_Thm : thm
      val Cases_Thm : thm
    end;
```

The actual theorems produced by the `mutual` library are not precisely the same as those produced by `nested_rec`. Some of the variable names generated automatically by the `nested_rec` tools were meaningless and hard to work with. Some we retained, like the long names for case functions, but for others, we generated more meaningful names based on the types of the variables, as in the standard recursive types package. In addition, the theorems were restructured and prepared for use by the other facilities of the `mutual` library. For the above example, the existence theorem generated by the `mutual` library is:

```
val New_Ty_Existence_Thm =
  |- !var_exp_case let_exp_case val_dec_case local_dec_case
      seq_dec_case aexp_case app_exp_case fn_exp_case
      match_case wild_pat_case var_pat_case
      atexp_dec_exp_match_pat_rule_valbind_ch44_pat_exp_case
      atexp_dec_exp_match_pat_rule_valbind_NIL_pat_exp_prod_
atexp_dec_exp_match_pat_rule_valbind_case
      atexp_dec_exp_match_pat_rule_valbind_CONS_pat_exp_prod_
atexp_dec_exp_match_pat_rule_valbind_case
      rule_case
      atexp_dec_exp_match_pat_rule_valbind_NIL_rule_case
      atexp_dec_exp_match_pat_rule_valbind_CONS_rule_case
      bind_case rec_bind_case.
```

```
        ?fna fnd fne fnm fnp0 fnp1 fnl0 fnr fnl1 fnv.
          (!x. fna (var_exp x) = var_exp_case x) /\
          (!d e. fna (let_exp d e) =
                 let_exp_case (fnd d) (fne e) d e) /\
          (!v. fnd (val_dec v) = val_dec_case (fnv v) v) /\
          (!d0 d1. fnd (local_dec d0 d1) =
                   local_dec_case (fnd d0) (fnd d1) d0 d1) /\
          (!d0 d1. fnd (seq_dec d0 d1) =
                   seq_dec_case (fnd d0) (fnd d1) d0 d1) /\
          (!a. fne (aexp a) = aexp_case (fna a) a) /\
          (!e a. fne (app_exp e a) =
                 app_exp_case (fne e) (fna a) e a) /\
          (!m. fne (fn_exp m) = fn_exp_case (fnm m) m) /\
          (!l. fnm (match l) = match_case (fnl1 l) l) /\
          (fnp0 wild_pat = wild_pat_case) /\
          (!x. fnp0 (var_pat x) = var_pat_case x) /\
          (!p e.
            fnp1 (p,e) =
            atexp_dec_exp_match_pat_rule_valbind_ch44_pat_exp_case
              (fnp0 p) (fne e) p e) /\
          (fnl0 [] =
           atexp_dec_exp_match_pat_rule_valbind_NIL_pat_exp_prod_
atexp_dec_exp_match_pat_rule_valbind_case) /\
          (!p l.
            fnl0 (CONS p l) =
            atexp_dec_exp_match_pat_rule_valbind_CONS_pat_exp_prod_
atexp_dec_exp_match_pat_rule_valbind_case
              (fnp1 p) (fnl0 l) p l) /\
          (!p e. fnr (rule p e) =
                 rule_case (fnp0 p) (fne e) p e) /\
          (fnl1 [] =
           atexp_dec_exp_match_pat_rule_valbind_NIL_rule_case) /\
          (!r l.
            fnl1 (CONS r l) =
            atexp_dec_exp_match_pat_rule_valbind_CONS_rule_case
              (fnr r) (fnl1 l) r l) /\
          (!l. fnv (bind l) = bind_case (fnl0 l) l) /\
          (!v. fnv (rec_bind v) = rec_bind_case (fnv v) v) : thm
```

Where the above existence theorem has

`?fna fnd fne fnm fnp0 fnp1 fnl0 fnr fnl1 fnv.`

the corresponding theorem generated by the `nested_rec` library has instead

`?y y''''''''' y'''''''' y''''''' y'''''' y''''' y'''' y''' y'' y'.`

with corresponding substitutions throughout.

5 Defining Mutually Recursive Functions

Once the mutually recursive types are defined, we can now define a cooperating set of mutually recursive functions on them. `define_mutual_functions` is used for this, as in the following example. This example defines functions to return the variables in a phrase of the language, except for those in a given set s.

```
val vars_thm = define_mutual_functions
{name = "vars_thm",
 rec_axiom = syntax_exists,
 fixities = NONE,
 def =
(--`(atexpV (var_exp (v:'var)) s = (v IN s => {} | {v})) /\
    (atexpV (let_exp d e) s = (decV d s) UNION (expV e s))
     /\
    (expV (aexp a) s = atexpV a s) /\
    (expV (app_exp e a) s = (expV e s) UNION (atexpV a s)) /\
    (expV (fn_exp m) s = matchV m s)
     /\
    (matchV (match rs) s = matchVs rs s)
     /\
    (matchVs (NIL) s = {}) /\
    (matchVs (CONS r mrst) s = (ruleV r s) UNION (matchVs mrst s))
     /\
    (ruleV (rule p e) s = (patV p s) UNION (expV e s))
     /\
    (decV (val_dec b) s = valbindV b s) /\
    (decV (local_dec d1 d2) s = (decV d1 s) UNION (decV d2 s)) /\
    (decV (seq_dec d1 d2) s = (decV d1 s) UNION (decV d2 s))
     /\
    (valbindV (bind bs) s = valbindVs bs s) /\
    (valbindV (rec_bind vb) s = (valbindV vb s))
     /\
    (valbindVs NIL s = {}) /\
    (valbindVs (CONS bhd brst) s = (valbindVp bhd s) UNION
                                   (valbindVs brst s))
     /\
    (valbindVp (p,e) s = (patV p s) UNION (expV e s))
     /\
    (patV wild_pat s = {}) /\
    (patV (var_pat v) s = (v IN s => {} | {v}))`--)};
```

This creates the following definition:

```
val vars_thm =
  |- (!v s. atexpV (var_exp v) s = ((v IN s) => {} | {v})) /\
     (!d e s. atexpV (let_exp d e) s = decV d s UNION expV e s) /\
     (!a s. expV (aexp a) s = atexpV a s) /\
     (!e a s. expV (app_exp e a) s = expV e s UNION atexpV a s) /\
     (!m s. expV (fn_exp m) s = matchV m s) /\
     (!rs s. matchV (match rs) s = matchVs rs s) /\
     (!s. matchVs [] s = {}) /\
     (!r mrst s. matchVs (CONS r mrst) s =
                 ruleV r s UNION matchVs mrst s) /\
     (!p e s. ruleV (rule p e) s = patV p s UNION expV e s) /\
     (!b s. decV (val_dec b) s = valbindV b s) /\
     (!d1 d2 s. decV (local_dec d1 d2) s =
                decV d1 s UNION decV d2 s) /\
     (!d1 d2 s. decV (seq_dec d1 d2) s =
                decV d1 s UNION decV d2 s) /\
     (!bs s. valbindV (bind bs) s = valbindVs bs s) /\
     (!vb s. valbindV (rec_bind vb) s = valbindV vb s) /\
     (!s. valbindVs [] s = {}) /\
     (!bhd brst s. valbindVs (CONS bhd brst) s =
                   valbindVp bhd s UNION valbindVs brst s) /\
     (!p e s. valbindVp (p,e) s = patV p s UNION expV e s) /\
     (!s. patV wild_pat s = {}) /\
     (!v s. patV (var_pat v) s = ((v IN s) => {} | {v})) : thm
```

This theorem matches the specification, including the names of the variables used. This is not the case for the `nested_rec` library. Also note the additional argument s to each function. Any number of arguments may be added, but the first argument must be one of the recursive types. It is possible to define functions on only one or some of the types defined in a mutual set; not all need be present in the function definition. However, note that if *any* of the constructors of a type are present, they must *all* be present, unless the last pattern for the type is the variable "`allelse`".

The nested_rec version of `define_mutual_functions` supports only one argument. Nevertheless, we can still define the same functions by moving the extra arguments to be lambda abstractions on the right hand side. However, the resulting theorem is different in its structure and names used, as illustrated below:

```
val vars_thm =
  |- (!x1. atexpV (var_exp x1) = (\s. (x1 IN s) => {} | {x1})) /\
     (!x1 x2. atexpV (let_exp x1 x2) =
              (\s. decV x1 s UNION expV x2 s)) /\
     (!x1. expV (aexp x1) = (\s. atexpV x1 s)) /\
     (!x1 x2. expV (app_exp x1 x2) =
              (\s. expV x1 s UNION atexpV x2 s)) /\
```

```
    (!x1. expV (fn_exp x1) = (\s. matchV x1 s)) /\
    (!x1. matchV (match x1) = (\s. matchVs x1 s)) /\
    (matchVs [] = (\s. {})) /\
    (!x1 x2. matchVs (CONS x1 x2) =
             (\s. ruleV x1 s UNION matchVs x2 s)) /\
    (!x1 x2. ruleV (rule x1 x2) =
             (\s. patV x1 s UNION expV x2 s)) /\
    (!x1. decV (val_dec x1) = (\s. valbindV x1 s)) /\
    (!x1 x2. decV (local_dec x1 x2) =
             (\s. decV x1 s UNION decV x2 s)) /\
    (!x1 x2. decV (seq_dec x1 x2) =
             (\s. decV x1 s UNION decV x2 s)) /\
    (!x1. valbindV (bind x1) = (\s. valbindVs x1 s)) /\
    (!x1. valbindV (rec_bind x1) = (\s. valbindV x1 s)) /\
    (valbindVs [] = (\s. {})) /\
    (!x1 x2. valbindVs (CONS x1 x2) =
             (\s. valbindVp x1 s UNION valbindVs x2 s)) /\
    (!x1 x2. valbindVp (x1,x2) =
             (\s. patV x1 s UNION expV x2 s)) /\
    (patV wild_pat = (\s. {})) /\
    (!x1. patV (var_pat x1) = (\s. (x1 IN s) => {} | {x1}))
  : thm
```

This structure obliges one to use beta reduction when using the definition theorem, rather than simple rewriting.

6 Proofs by Mutual Structural Induction

The third and final part of the `mutual` library is the support for proofs of mutual structural induction, through `MUTUAL_INDUCT_TAC`. This is a revised version of the `INDUCT_TAC` written by Tom Melham in the standard recursive types package, expanded for mutually recursive types. There is much care taken in the original version to break the current goal into a practical and convenient set of subgoals according to the induction principle, and we have tried to preserve this quality.

The ML function `MUTUAL_INDUCT_TAC` has type

```
thm -> (thm -> tactic) -> tactic
```

and can be used to generate a structural induction tactic for a set of concrete types definable using the functors of Section 4. The first argument is an induction theorem of the form created by these functors. The second argument is a theorem continuation that determines what is to be done with the induction hypotheses when the resulting tactic is applied to a goal.

If ind_th is an induction theorem for a set of mutually recursive concrete types $op_1, \ldots, op_n$, where this includes all auxiliary types arising through the nesting of types in the definition, and if each concrete type op_i has m_i constructors $C^i_1, \ldots, C^i_{m_i}$, and F is a theorem continuation, then the tactic

$$\texttt{MUTUAL\_INDUCT\_THEN}\ ind\_th\ F$$

will reduce a goal of the form

$$\begin{array}{l}(\Gamma,\ \texttt{(--`}\ (\forall x_1 : op_1.\ t_1[x_1])\ \wedge \\ \qquad\qquad\qquad \vdots \\ \qquad\qquad (\forall x_n : op_n.\ t_n[x_n])\ \texttt{`--)}\ )\end{array}$$

to a collection of (possibly) $\sum_{i=1}^{n} m_i$ induction subgoals (this count may not be precise for various reasons). The goal may list the conjuncts in any order; they need not be in the precise same order as the corresponding clauses listed in the induction theorem *ind_th*. In fact, some of the goal clauses may be missing entirely, in which case the tactic will presume that they are $(\forall x_i : op_i.\ \texttt{T})$.

As an example, consider proving that for the variable-collecting functions defined earlier, none of them collect any variables in the exclusion set s.

```
g `(!a s (x:'var). x IN atexpV a s ==> ~(x IN s)) /\
   (!e s (x:'var). x IN expV e s ==> ~(x IN s)) /\
   (!m s (x:'var). x IN matchV m s ==> ~(x IN s)) /\
   (!rs s (x:'var). x IN matchVs rs s ==> ~(x IN s)) /\
   (!r s (x:'var). x IN ruleV r s ==> ~(x IN s)) /\
   (!d s (x:'var). x IN decV d s ==> ~(x IN s)) /\
   (!v s (x:'var). x IN valbindV v s ==> ~(x IN s)) /\
   (!l s (x:'var). x IN valbindVs l s ==> ~(x IN s)) /\
   (!pr s (x:'var). x IN valbindVp pr s ==> ~(x IN s)) /\
   (!p s (x:'var). x IN patV p s ==> ~(x IN s))`;
```

These clauses are listed in an order similar to the definition, which is convenient. These can be simultaneously broken into cases by mutual structural induction with the following tactic:

```
- e(MUTUAL_INDUCT_THEN syntax_induct ASSUME_TAC);
OK..
19 subgoals:
val it =
  (--`!s x. x IN valbindV (rec_bind v) s ==> ~(x IN s)`--)
  ____________________________
      (--`!s x. x IN valbindV v s ==> ~(x IN s)`--)

  (--`!s x. x IN valbindV (bind l) s ==> ~(x IN s)`--)
  ____________________________
      (--`!s x. x IN valbindVs l s ==> ~(x IN s)`--)

  (--`!s x. x IN matchVs (CONS r rs) s ==> ~(x IN s)`--)
  ____________________________
      (--`!s x. x IN ruleV r s ==> ~(x IN s)`--)
      (--`!s x. x IN matchVs rs s ==> ~(x IN s)`--)
```

```
(--`!s x. x IN matchVs [] s ==> ~(x IN s)`--)


(--`!s x. x IN ruleV (rule p e) s ==> ~(x IN s)`--)
____________________________
    (--`!s x. x IN patV p s ==> ~(x IN s)`--)
    (--`!s x. x IN expV e s ==> ~(x IN s)`--)

(--`!s x. x IN valbindVs (CONS pr l) s ==> ~(x IN s)`--)
____________________________
    (--`!s x. x IN valbindVp pr s ==> ~(x IN s)`--)
    (--`!s x. x IN valbindVs l s ==> ~(x IN s)`--)

(--`!s x. x IN valbindVs [] s ==> ~(x IN s)`--)


(--`!s x. x IN valbindVp (p,e) s ==> ~(x IN s)`--)
____________________________
    (--`!s x. x IN patV p s ==> ~(x IN s)`--)
    (--`!s x. x IN expV e s ==> ~(x IN s)`--)

(--`!x s x'. x' IN patV (var_pat x) s ==> ~(x' IN s)`--)


(--`!s x. x IN patV wild_pat s ==> ~(x IN s)`--)


(--`!s x. x IN matchV (match rs) s ==> ~(x IN s)`--)
____________________________
    (--`!s x. x IN matchVs rs s ==> ~(x IN s)`--)

(--`!s x. x IN expV (fn_exp m) s ==> ~(x IN s)`--)
____________________________
    (--`!s x. x IN matchV m s ==> ~(x IN s)`--)

(--`!s x. x IN expV (app_exp e a) s ==> ~(x IN s)`--)
____________________________
    (--`!s x. x IN expV e s ==> ~(x IN s)`--)
    (--`!s x. x IN atexpV a s ==> ~(x IN s)`--)

(--`!s x. x IN expV (aexp a) s ==> ~(x IN s)`--)
____________________________
    (--`!s x. x IN atexpV a s ==> ~(x IN s)`--)
```

```
  (--`!s x. x IN decV (seq_dec d d') s ==> ~(x IN s)`--)
  ----------------------------
      (--`!s x. x IN decV d s ==> ~(x IN s)`--)
      (--`!s x. x IN decV d' s ==> ~(x IN s)`--)

  (--`!s x. x IN decV (local_dec d d') s ==> ~(x IN s)`--)
  ----------------------------
      (--`!s x. x IN decV d s ==> ~(x IN s)`--)
      (--`!s x. x IN decV d' s ==> ~(x IN s)`--)

  (--`!s x. x IN decV (val_dec v) s ==> ~(x IN s)`--)
  ----------------------------
      (--`!s x. x IN valbindV v s ==> ~(x IN s)`--)

  (--`!s x. x IN atexpV (let_exp d e) s ==> ~(x IN s)`--)
  ----------------------------
      (--`!s x. x IN decV d s ==> ~(x IN s)`--)
      (--`!s x. x IN expV e s ==> ~(x IN s)`--)

  (--`!x s x'. x' IN atexpV (var_exp x) s ==> ~(x' IN s)`--)

   : goalstack
```

In fact, the original goal can be entirely proven by the tactic

```
e(MUTUAL_INDUCT_THEN syntax_induct ASSUME_TAC
   THEN REWRITE_TAC[vars_thm]
   THEN REPEAT GEN_TAC
   THEN ((REWRITE_TAC[theorem "set" "IN_UNION"]
          THEN REWRITE_TAC[theorem "set" "NOT_IN_EMPTY"]
          THEN STRIP_TAC
          THEN RES_TAC
          THEN NO_TAC
        )
        ORELSE
        ( COND_CASES_TAC
          THEN REWRITE_TAC[theorem "set" "IN_INSERT",
                           theorem "set" "NOT_IN_EMPTY"]
          THEN DISCH_TAC
          THEN ASM_REWRITE_TAC[]
        ))
 );
```

In the `nested_rec` library there was no analogous tactic provided. The only thing we could find was an `info-hol` posting by Myra VanInwegen, dated March 19, 1996, where she wrote:

```
We didn't include such a tactic with the package, but obviously,
one is needed to prove properties of mutually recursive types.
This is what I use:

(* for now, the things proven must be in the same order as in the
   conclusion of the induction theorem *)
fun mutual_induct induct_thm (asms, gl) =
    let val props_list = map
            (fn tm => mk_abs (dest_forall tm)) (strip_conj gl)
        val speced_ind = BETA_RULE (SPECL props_list induct_thm)
    in
        MP_IMP_TAC speced_ind (asms, gl)
    end

The only problem with it is, as I note in the comment, that
the properties have to be in the same order as those in the
conclusion of the induction theorem. The result of applying
this function is one subgoal that is a big conjunction, with
each conjunct being a case in the induction.
```

Using the `mutual_induct` function, we can prove a similar result as before. The goal must be reordered, and the tactic must make use of `BETA_TAC`. The resulting tactic is slightly larger than the previous one. To compare these two tactics, where `MUTUAL_INDUCT_THEN` presents the user with

```
(--`!s x. x IN atexpV (let_exp d e) s ==> ~(x IN s)`--)
____________________________
    (--`!s x. x IN decV d s ==> ~(x IN s)`--)
    (--`!s x. x IN expV e s ==> ~(x IN s)`--)
```

`mutual_induct` followed by `REPEAT CONJ_TAC` presents

```
(--`!y y''''''''.
     (!s x. x IN decV y s ==> ~(x IN s)) ==>
     (!s x. x IN expV y'''''''' s ==> ~(x IN s)) ==>
     (!s x. x IN atexpV (let_exp y y'''''''') s ==> ~(x IN s))`--)
```

These `y''''''''` variables appear to be an artifact of the implementation of the `nested_rec` library.

7 Summary and Conclusions

We have defined a new library within HOL, `mutual`, to support the creation and use of mutually recursive types with nesting. This is essentially equivalent to the functionality of the `nested_rec` library, but adds facilities to ease its use in practical ways.

The input specifications are shorter and clearer, close to the BNF form, and similar to the syntax required for the non-mutual recursive type definition package. Functions can be defined on these types with more arguments. Properties may be proved by mutual structural induction, supported by a general-purpose function for these tactics.

The `mutual` library software is currently available for HOL90 versions 7 and 10, through the Web page at

`http://www.cis.upenn.edu/~homeier/holsw.html`.

For all these tools, feedback is welcome and encouraged, as we would like to polish them for general use. Please notify the author if this library is adapted to another environment, so it can be posted here as well.

Caution: this software should be considered only of beta quality, and may contain errors. It is being released now in order to support researchers for whom this level of quality is acceptable, and who may be able to help in testing and improving this software.

This exercise is perhaps best appreciated as an investigation into the relative importance of ease-of-use. This is not a question with a precise answer, but depends on people's preferences. Thus this paper is only an entry in the ongoing discussion.

DEDICATION: This paper is dedicated to David F. Martin, Professor and Founding Member of Computer Science at UCLA, who passed away December 22, 1996. Without his encouragement and involvement, all of my future career would not be.

Soli Deo Gloria.

References

1. Michael J. C. Gordon, Thomas F. Melham: *Introduction to HOL, A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, Cambridge (1993)
2. Myra VanInwegen: *The Machine-Assisted Proof of Programming Language Properties.* Ph.D. Thesis, University of Pennsylvania, Computer and Information Science Tech Report MS-CIS-96-31, December 1996
3. Myra VanInwegen, Elsa Gunter: HOL-ML, in *Higher Order Logic Theorem Proving and Its Applications: 6th International Workshop*, Vancouver, B.C., Canada, August 1993, eds. Jeffrey J. Joyce, Carl-Johan H. Seger. Lecture Notes in Computer Science **780** (1994) 61–74