Effective Support for Mutually Recursive Types

by

Peter Vincent Homeier

Department of Computer and Information Science
University of Pennsylvania
homeier@saul.cis.upenn.edu

Problem Statement

- ***Modeling systems sometimes requries mutually recursive types***

  E.g., the syntax of programming languages is often mutually recursive

- ***Melham's type package is excellent in function and easy to use, but only defines one type***

  Easy to specify the new type with its constructors

  Package automatically constructs the definitions required

  . . . proves various theorems for using the type

  . . . provides tool to define new recursive functions on type

  . . . provides tactic for proving properties of recursive functions

- ***Existing mutrec and nested-rec libraries enable definitions of mutually recursive types***

  nested-rec is more expressive; allows new recursive types to be specified using ***list***, ***prod***, ***sum***, and other type operators

  But these libraries are not as easy to use as Melham's type package

- ***New mutual library provides functionality of nested-rec, with ease of Melham's package***

  Provides tools of Melham's package, but for mutually recursive types

  ☞ What is the proper priority of "ease of use" vs. functionality?

■
Peter Vincent Homeier
UNIVERSITY OF PENNSYLVANIA — DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

Example -- language syntax

atexp = *var* | **let** *dec* **in** *exp*
exp = *atexp* | *exp atexp* | *match*
match = *rule list*
rule = *pat => exp*
dec = *valbind* | **local** *dec* **in** *dec* | *dec* ; *dec*
valbind = **bind** (*pat* **to** *exp*) *list* | **rec** *valbind*
pat = **wild_pat** | *var*

- ***Source code to create this example using "mutual" library:***

```
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 ` );
```


■
Peter Vincent Homeier
UNIVERSITY OF PENNSYLVANIA — DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

Source using “nested-rec” library

```
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",
```


■
Peter Vincent Homeier
UNIVERSITY OF PENNSYLVANIA — DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

```
            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;
```

- ***Condensation is due to <u>parser</u> for a mututally recursive types specification language (essentially a domain-specific language)***

  Uses exact same syntax as Melham's standard package, but allows multiple type specifications, separated by semicolons

■
Peter Vincent Homeier
UNIVERSITY OF PENNSYLVANIA — DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

Definition of mutually recursive functions

- ***Using define_mutual_functions from "mutual" library:***

```
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}))`--)};
```


■
Peter Vincent Homeier
UNIVERSITY OF PENNSYLVANIA — DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

Resulting Definition Theorem

- ***Theorem produced by define_mutual_functions:***

```
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
```

- ***Additional variables permitted now, after the first (as ‘s’ above)***

  The “nested-rec” library required definitions of lambda-forms.

  The resulting definitions could not be used by simple rewriting, but required beta-conversion as well.

- ***Variable names in theorem now match the original definition***

  With the “nested-rec” library, the variable names were arbitrary.

■
Peter Vincent Homeier
UNIVERSITY OF PENNSYLVANIA — DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

Proofs by Mutual Structural Induction

- ***Mutual structural induction tactic provided:***

  `MUTUAL_INDUCT_TAC` has the type

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

  For a set of mututally recursive types op_1 though op_n,

  `MUTUAL_INDUCT_TAC` *ind_th F* reduces a goal of the form

$$(\Gamma,\ (\text{--`}\ (\forall x_1 : op_1.\ t_1[x_1]) \wedge \ldots (\forall x_n : op_n.\ t_n[x_n])\ \text{`--})\ )$$

  to a collection of subgoals, one for each constructor of each type op_i, where the induction hypotheses for each case are processed by the theorem-continuation *F* (typically `ASSUME_TAC`).

  For convenience, the clauses in the goal above may occur in any order; they need NOT be in the same order as the corresponding clauses in the induction theorem *ind_th*.

  Some goal clauses may be missing entirely, in which case they are presumed to be

$$(\forall x_i : op_i.\ \mathsf{T})$$

- ***The "nested-rec" library provided no support for these proofs; Myra VanInwegen informally supplied a less convenient tactic***

■
Peter Vincent Homeier
UNIVERSITY OF PENNSYLVANIA — DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

Mutual Structural Induction Example

- ***Mutual goal:***

```
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))`;
```

Then executing

```
e(MUTUAL_INDUCT_THEN syntax_induct ASSUME_TAC);
```

produces 19 subgoals, among which are:

```
...

(--`!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 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)`--)
```


■
Peter Vincent Homeier
UNIVERSITY OF PENNSYLVANIA — DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

Summary

- ***"mutual" library supports the convenient definition and use of nested, mutually recursive concrete datatypes in the HOL logic.***

- ***Three areas of advantage over existing libraries:***
  - — Clear and concise specifications of mutually recursive types
  - — More general definitions of functions on types
  - — Easier proofs of properties about functions and types

- ***Two versions available, with identical functionality:***
  - — for HOL90, version 7
  - — for HOL90, version 10

- ***Source code is available for download:***
  http://www.cis.upenn.edu/~homeier/holsw.html

- ***Question:***
  how important is "ease of use" versus functionality?

■
Peter Vincent Homeier
UNIVERSITY OF PENNSYLVANIA — DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE