Recursive Data Types in HOL

Roger Bishop Jones

ICL Defence Systems

1. INTRODUCTION
This document describes a means of constructing recursively defined data types in higher order logic (HOL), and documents special ML procedures developed to facilitate this process.

By "recursive data type" we mean, in the first instance, a collection of data types constructed from a set of (non-recursive) data types by (recursive) iteration of labelled record type, and disjoint union type constructions. Provision of recursive data types is therefore intricately bound up with support of labelled record types and disjoint unions. Non-recursive disjoint unions, labelled records and enumeration types are special cases of the constructs provided.

Infinite (or cyclic) data structures are not supported.

2. DESCRIBING RECURSIVE DATA TYPES IN ML
First we define an ML type for describing structured data types. Values of this type are used as parameters to the functions which we provide for defining structured data types in HOL.

Recursive data types normally come in groups, defined by mutually recursive equations. In our system each type in a group is either a product type or a union type, and is described by a set of field descriptions.

Each field either of a union or of a product, has associated with it a field type. A field type may take one of three forms. It may be void, it may be the name of another type within this group of recursive types, or it may be (for the purposes of the construction in hand) primitive.

\[
\text{ML type field_type = V | T of string | P of type};;
\]
A field description associates a string with a field type. In the case of a product this string labels
the component of the product and forms a part of the name of the projection function from the
product type yielding this component. In the case of a union type the name is used for the con-
structor or injection for the component.
Field names for products must be unique only within a single product type. Field names for
unions must be unique within a theory.

```ml
typeabbr field_desc = string # field_type;;
```

These field descriptions are collected together and designated as product types or union types to
give a structured type.

```ml
type struct_type = Prod of field_desc list |
                 Union of field_desc list;;
```

These types must be given names and collected together into a complete collection in which every
type mentioned is defined. Such a collection has type rec_types.

```ml
typeabbr rec_types = (string # struct_type) list;;
```

3. THE REPRESENTATION OF VALUES WITH RECURSIVE DATA TYPES.

Our first task is to determine a HOL data type which will serve as representatives for objects with
recursive types. We confine ourselves to finite data structures obtained from certain "primitive"
(non-recursive) types by finite iteration of labelled product and disjoint union constructions.

Each such data item may be regarded as a tree with labelled arcs. The leaves of the tree are the
primitive data values from which the compound value was constructed. Where a disjoint union
constructor is used there will be just one branch descending from the relevant node. This will be
labelled with the appropriate constructor.

Where a product construction is used there will be one branch for each component labelled by the
projection name.

Each primitive on the tree may be located if the sequence of branch labels descending from it to
the root is known, and this sequence may be thought of as its address. The information in the
data structure may therefore be alternatively thought of as a (partial) mapping from addresses to
primitive values, where an address is simply a list of arc labels.

Using iteration of the sum type constructor to give disjoint unions, and disjoint unions, of void
components as enumerated types, we can represent the values in our recursive domains as partial
functions from addresses (consisting of lists of the enumeration type formed from all the con-
structors) to values (consisting of the disjoint union of all the primitive types).
let is_P f = (case f of P p . true) ? false ;;
let is_T f = (case f of T p . true) ? false ;;
let P_type = fun P t . t ;;
let T_name = fun T name . name ;;
let body = fun Prod b . b | Union b . b ;;
let names = map fst ;;
let cnstr_names = names o body ;;

let list_it fle = itlist f e l ;;
letrec max = fun [] . 0 |
           (h . t) . (h > tmax => h | tmax where tmax = max t ) ;;
letrec up_to from to = if from > to then [] else from . (up_to (from+1) to) ;;
let range = up_to 1 ;;
let map_int fn = map f (range n) ;;

let struc_Ps = (map P_type) o (filter is_P) o (map snd) o body ;;
let struc_Ts = (map T_name) o (filter is_T) o (map snd) o body ;;
let rec_Ps : rec_types → type list
       = make_set o (list_it append []) o (map (struc_Ps o snd)) ;;
let rec_Ts : rec_types → string list
       = make_set o (list_it append []) o (map (struc_Ts o snd)) ;;

let max_cnstr_width l = max (map length (map (cnstr_names o snd) l )) ;;
let c_void l = " :void " ; ;
4. THINGS THAT OUGHT TO BE IN 041

4.1. Constructing Partial Functions from Lists

Before adding more to this we define some primitive polymorphic functions which allow a useful range of operations on all structured data types defined in the above way.

For constructing objects of schema types and of disjoint union types we need a function which will take a list of (*dom # *cod) pairs and construct from them a partial function.

```
ML
list_rec_def('list_mk_pfun',"((list_mk_pfun:(*dom # *cod) list→(*dom,*cod)pfun) [] = pf_empty)
∧ (∀(a:*dom # *cod)(l:(*dom # *cod)list).
 list_mk_pfun (CONS a l) = ((list_mk_pfun l) pf_add (FST a)) (SND a))");;
```

We also require a function which will form the union of two partial functions.

```
ML
new_infix_definition('pf_merge','pf_merge (lf:(*dom,*cod)pfun) (rf:(*dom,*cod)pfun) =
 (mk_pfun:(*dom→(void+*cod))→(*dom,*cod)pfun)
 ....rg,*dom. If pdef arg =>
     pf_val (lf apf arg) |
     (rf pdef arg => pf_val (rf apf arg) | pf_u)");;
```

And a function to merge a list of partial functions.

```
ML
list_rec_def('pf_list_merge',"(pf_list_merge ([]:(*dom,*cod)pfun list) = (pf_empty:(*dom,*cod)pfun))
∧ (pf_list_merge (CONS (a:(*dom,*cod)pfun) l) = a pf_merge (pf_list_merge l))");;
```
Map defined in HOL.

\[
\begin{align*}
\text{ML} \quad \text{list_rec_def('rmap', "} \\
& \quad ((rmap:*dom\ list\rightarrow(*dom\rightarrow*cod)\rightarrow*cod\ list)\ [\ ]\ f\ =\ [\ ]) \\
& \quad \land\ (rmap\ \text{CONS}\ a\ l\ f\ =\ \text{CONS}\ (f\ a)\ (rmap\ l\ f))\));; \\
\end{align*}
\]

\[
\begin{align*}
\text{ML} \quad \text{new_definition('map', "map\ (f:*dom\rightarrow*cod)\ l\ =\ rmap\ l\ f")};;
\end{align*}
\]

4.2. Disjoint Union Injections

We must construct injections from the primitive types to the codomain of the representing function type before we can generate the predicates.

\[
\begin{align*}
\text{ML} \quad \text{letrec new_disj_type type_list =} \\
& \quad \text{if null type_list} \\
& \quad \quad \text{then failwith 'new_disj_type - empty type_list'} \\
& \quad \quad \text{else if null (tl type_list)} \\
& \quad \quad \quad \text{then hd type_list} \\
& \quad \quad \text{else ".:\(\text{hd type_list})\+\text{(new_disj_type (tl type_list))}\);;}
\end{align*}
\]

\[
\begin{align*}
\text{ML} \quad \text{letrec inj_fun_r type_list typ =} \\
& \quad \text{if null type_list} \\
& \quad \quad \text{then failwith 'inj_fun_r - empty type_list'} \\
& \quad \quad \text{else let tl_list = tl type_list} \\
& \quad \quad \quad \text{in if hd type_list = typ} \\
& \quad \quad \quad \quad \text{then let var = "v:\ typ"} \\
& \quad \quad \quad \quad \quad \text{in if null (tl_list)} \\
& \quad \quad \quad \quad \quad \quad \text{then var} \\
& \quad \quad \quad \quad \quad \quad \quad \text{else "(INL:\(\text{typ})\rightarrow\text{(hd type_list)+\text{(new_disj_type tl_list)})}\ "\text{var}"
\end{align*}
\]

\[
\begin{align*}
\text{ML} \quad \text{let mk_injection type_list typ =} \\
& \quad \text{let term = inj_fun_r (tl_list) typ} \\
& \quad \quad \text{in "\(\lambda v:\ typ. \ "\text{term}\);;}
\end{align*}
\]
4.3. Domain Values

We also need to be able to construct the values of the 'enumerated type' which forms the domain of the partial function in the representing type.

First we provide a procedure for constructing the domain type.

```ml
letrec new_enum_type n =
  if n = 0
  then failwith 'new_enum_type - n=0'
  else if n = 1
  then "void"
  else "void+ˆ(new_enum_type (n-1));
```

The following function constructs the $m$th component of an $n$-valued enumerated type.

```ml
letrec dom_val n m =
  if n = 0 or m = 0 or n < m
  then failwith 'dom_val - n=0 or m=0 or n<m'
  else if m = 1
  then let val = "μx:void.T"
     in if n = 1
        then val
        else "(INL:void→ˆ(type_of val)) val"
  else let term = dom_val (n-1) (m-1)
     in "(INR:ˆ(type_of term)→(void+ˆ(type_of term))) term";;
```

dom_vals constructs a list of the first $m$ values of a $n$-valued enumerated type.

```ml
letrec dom_vals n m = (m = 0) => [] | (dom_vals n (m-1)) @ [dom_val n m];;
```
4.4. Projections from Tuples

The following function constructs a projection for the \( m \)th component of some \( n \)-tuple.

\[
\text{let tuple_project term n m =}
\begin{align*}
\text{letrec tpr term n m =} & \\
\text{if } & \text{n = 0 or m = 0 or n < m} \\
\text{then failwith 'tuple_select - n=0 or m=0 or n<m'} & \\
\text{else if } & \text{m = 1} \\
\text{then let } & \text{val = term} \\
\text{in } & \text{if } \text{n = 1} \\
\text{then (val,false)} & \\
\text{else (val,true)} & \\
\text{else let } & \text{(t,f) = tpr term (n-1) (m-1)} \\
\text{in } & \text{"SND 't",f} \\
\text{in let } & \text{t,f} = tpr term n m \\
in & \text{f => "FST 't" | t;;}
\end{align*}
\]

tuple_projections constructs a list of the first \( m \) projections from an \( n \)-tuple \( \text{term} \).

\[
\text{letrec tuple_projections term n m =}
\begin{align*}
\text{if (m = 0)} & \\
\text{then []} & \\
\text{else (tuple_projections term n (m-1)) @ [tuple_project term n m];;}
\end{align*}
\]

The following function constructs a type consisting of an \( n \)-fold boolean product.

\[
\text{letrec bool_tuple_type n = n=1 => "::bool" | ":bool#(bool_tuple_type (n-1))";;}
\]

Next we construct a function which projects the \( m \)th component of an \( n \)-tuple of booleans.

\[
\text{let bool_project n m =}
\begin{align*}
\text{let term = "x:(bool_tuple_type n)"} & \\
in & "\lambda\text{term.}(tuple_project term n m)";;
\end{align*}
\]
And a function which returns a full list of boolean projections from an \( n \)-tuple.

```ml
let bool_projections n = 
  let rec bpr n m = (m = 0) => [] | (bpr n (m-1)) @ [bool_project n m]
  in bpr n;;
```

5. CONSTRUCTING THE REPRESENTATION DOMAIN

```ml
let mk_rep_type type_list = 
  let width = max_cnstr_width type_list
  in let dom = new_enum_type width
      and cod = let Ps = rec_Ps type_list
                  in null Ps => "void" | new_disj_type (Ps)
      in let lab_list = "*:dom list"
      in let rep_type = "(*lab_list, *cod)pfun"
      in (rep_type, width);;
```

6. SOME FUNCTIONS DEFINED ON THE REPRESENTATION DOMAIN

6.1. Primitive Construction

A primitive is a partial function which has a value only for the empty list:

```ml
new_definition('mk_primitive', 
"mk_primitive (v:*cod) = list_mk_pfun [[]:*dom list],v]");
```
6.2. **Injections into Disjoint Unions**

An element of a disjoint union type is a partial function which is defined only on (some) lists whose head is the constructor for the component in question. The constructor therefore takes a list element and a partial function over lists of that type, and yields a partial function which has the same value for a list whose head is the constructor as the given partial has on the tail of the list.

```ml
new_definition('inject',
    "inject (d:*dom) (f:(*dom list,*cod)pfun) =
    (mk_pfun:(*dom list → (void+*cod))→(*dom list,*cod)pfun) (*dom list.
    NULL l =>
    pf_u |
    (((HD l) = d) ∧ (f pdef (TL l))) =>
    pf_val (f apf (TL l)) |
    pf_u");;
```

And a function which will deliver the label of an object of disjoint union type:

```ml
new_definition('union_comp',"union_comp (f:(*dom list,*cod)pfun)
    = µy.∃z f pdef (CONS y z)");;
```

6.3. **Constructing Labelled Records**

A labelled record is the union of a set of objects of some disjoint union type.

```ml
new_definition('mk_schema',"mk_schema (v:(*dom # (*dom list,*cod)pfun)list) =
    pf_list_merge ((map λ((x:*dom),(y:(*dom list,*cod)pfun)).inject x y) v)");;
```

6.4. **Extracting Values**

We can also describe fully polymorphic projection which takes an arbitrary label and returns the value of that component if it has one.

```ml
new_definition('proj',"proj (t:*dom) (s:(*dom list,*cod)pfun) =
    mk_pfun λl:*dom list.
    (s pdef (CONS t l)) =>
    pf_val (s apf (CONS t l)) |
    pf_u");;
```
7. GENERATING THE DEFINING PREDICATES

A predicate must be generated for each type. The complex bit of this is generating the predicates for the recursive types. We therefore deal with the predicates for the non-recursive types first.

7.1. Primitive Types

Predicates for primitive types never need recursive definitions, and are therefore the simplest to generate.

The predicates for the primitive types are formed from an injection from the primitive type into the codomain of the representation type by the following function:

```ml
new_definition('mk_is_primitive',
  "(mk_is_primitive:(*primitive_type→*cod)→(*dom list,*cod)pfun→bool) inj rep =
   ∃v:*primitive_type. rep = mk_primitive (inj v)";;
)
```

Using mk_injection we can therefore obtain the list of predicates by mapping mk_is_primitive over the list of injections.

```ml
let primitive_predicate rept injection =
  let [dom;cod] = snd (dest_type rept)
  in "(mk_is_primitive:(type_of injection)→(rept→bool)) "injection";;

let primitive_terms rept type_list typ =
  let inj = mk_injection type_list typ
  in (typ, inj, primitive_predicate rept inj, "µx:typ.T";;

let p_term_assoc type_list rept = map (primitive_terms rept type_list) type_list;;
```

7.2. Disjoint Union Types

Disjoint unions may or may not need recursively defined predicates. We deal with the simple case first.

The predicate for a simple disjoint union is the disjunction of the predicates for each of the component parts.

The predicates from the component parts read informally "there exists some object of the relevant type such that its injection is the object in hand". To construct this predicate we therefore need to know the predicate for the component, and the injection function for the component.

```ml
let u_comp_pred (comp_pred, inj) =
  "∃c.(comp_pred c) ∧ (v = `inj c)";;
```

To form the complete predicate we therefore need a list of component predicate injection pairs:
ML
let u_pred pair_list = "λv.(list_mk_disj (map u_comp_pred pair_list))";;

7.3. Product Types

To get the brain into gear we show a schematic:

\[λs.∃c' c'' . is_t1 c ∧ is_t2 c' ∧ is_t3 c'' \\
∧ s = mk_schema [d1,c; d2,c'; d3,c'']\]

For the product types we must simultaneously obtain a set of values satisfying the predicates for each component and the combine these to form the product. The combination can be done with the *mk_schema* operation.

Since we need a complete collection of such objects each must be given a separate name. This will be done using variants of c. The existential quantifiers need to be at the (almost) outermost level, so we must form the predicates first, and apply mk_schema to them, before closing the formula.

First a procedure for generating a list of variants of a variable.

ML
letrec variants v n =
    if n = 0
    then []
    else let vars = variants v (n-1)
    in (variant vars v).vars;;

ML
let prod_preds = (list_mk_conj o (map mk_comb) o combine);;
let prod_pairs l =
    let pair_list = ((map mk_pair) o combine) l
    in mk_list (type_of (hd pair_list)) pair_list;;
let mk_prod_pred rep_type preds doms vars =  
  let exist_clauses = prod_preds (preds,vars)  
  and s_clAUSE = "(s:ˆrep_type) = mk_schema "(prod_pairs (doms,vars))"  
  in let exist_closure = list_mk_exists (vars, "ˆexist_clauses ∧ "s_clAUSE")  
  in "λs:ˆrep_type."exist_closure";;

let mk_product_predicate rep_type n preds =  
  let l = length preds  
  in let doms = dom_vals n l  
           and vars = variants "c:ˆrep_type" l  
      in mk_prod_pred rep_type preds doms vars;;

7.4. Control Framework
The general scheme for the constructions is as follows:

[1] Process the leaves.
[2] Iteratively process every type which depends only upon types already processed until there are no types left which do not recurse.
[3] Process all the recursing types.

This is a peculiar compromise, since on one extreme we could treat all the types as mutually recursive and omit steps 1 and 2, and on the other, we might be able to divide the set of recursive types into subgroups which do not recurse through each other.

We do not adopt the first extreme because:

(a) It would probably slow down the whole process.
(b) I need to do steps 1 and 2 before I have any hope of understanding how to do step 3.

We do not adopt the latter extreme, because it looks like hard work, only saves mill, and probably doesn’t often happen. Furthermore if it really makes any difference the user can split his definition into parts.

We now consider the control structure which will take us through this process.

Let us suppose that for each type, as it is processed, all the necessary aspects of processing are done together. As types are processed type will be moved off the original list of unprocessed types and onto a list of processed types. Each of these will be association lists, associating the necessary information with name of the type.

7.4.1. Some useful functions
For checking:

[i] Whether a given type can be non-recursively constructed from a given list of types.
[ii] Whether all the types mentioned in a list of types are defined in that type list (for a diagnostic).
ML

let t_defined rec_types type_name = mem type_name (names rec_types);

let rt_defined (r:rec_types) (s:struc_type) = forall (t_defined r) (struc_Ts s);

let undefined_rts rec_types = subtract (rec_Ts rec_types) (names rec_types);

To understand how the main algorithm works it is essential to know the data structures being processed.

From the parameters supplied to new_rectypes, the top level procedure, are constructed a single large data structure. This data structure is a four-tuple of which the four components have the following roles:

(a) The first component contains information of a global nature, i.e. not specific to any single one of the types being defined. Initially this consists of:
   [i] The name of the complex of types.
   [ii] The HOL type which will be used to represent the various types.
   [iii] Useful terms associated with the primitive types which are constructed during initialisation. This is a list of tuples each of which contains:
       a) A primitive type.
       b) A term which is the injection function from the primitive type into the representation domain.
       c) A term which is the predicate indicating whether any member of the representing domain is a representative of a member of this primitive type.

ML
typeabbrev primitive_terms = (type # term # term) list;;

[iv] The number of elements in the enumeration type which is the domain of the representation type.

ML
typeabbrev global_data = string # type # primitive_terms # int;;

(b) The second component contains the details of those types which have not yet been processed, in just the form in which they are supplied, i.e. as a rec_types.

(c) The third component contains the details of types which have been processed, in their original form.

(d) The fourth contains the items of data which resulted from the processing of the types already processed. These form an association list indexed by the type names.
The algorithm consists in processing this four-tuple by various means, each of which may transfer types from the unprocessed list to the processed list. When the processed list is empty the constructions are complete.

This type processing is factored first into four stages.

**Initial processing**

During this phase the data structure described above is set up, showing all types unprocessed.

**Non-recursive types**

During this phase all types are processed which do not participate in recursion.

**Recursive types**

Here the mutually recursive types are processed.

**Final processing**

Wind up actions.

### 7.4.2. Initial Processing

```ml
let check_completeness rec_types = 
  let u_n = undefined_rts rec_types 
  in u_n = [] => () | 
    failwith `check_completeness - the following types are mentioned but not defined: ` 
    ` (* (itlist (∀x.∀y.x`) , `y) (tl u_n) (hd u_n)) ^ ` `;

let do_initiation (name, rec_types) = 
  check_completeness rec_types; 
  let prims = rec_Ps rec_types 
  and (rept, width) = mk_rep_type rec_types 
  in ((name, rept, prims = [] => []) | p_term_assoc (rec_Ps rec_types) rept, width), 
    rec_types, 
    [], 
    []);
```

### 7.4.3. Processing Non-Recursive Types
7.4.3.1. Unions

In order to construct the predicate for a non-recursive union we must assemble a list of predicates and injections for each component of the union.

Non-recursive unions are not processed until after all their components, so all the component predicates will be available. The injections are constructed using \textit{inject}.

The total data structure resulting after constructing the predicate for the union consists of:

(a) The name of the type.
(b) The term which is the predicate.

\[
\text{typeabbrev union_data = string # term};
\]

We now show the procedure for retrieving the predicate for the component.

\[
\text{let void_pred rep_type =}
\]

\[
\text{let [dom; cod] = snd (dest_type rep_type)}
\]

\[
\text{in } \lambda v:rep_type.v=(\text{list_mk_pfun }[[];\text{dom},\lambda x:cod.T])";;
\]

\[
\text{let predicate ((_, rep_type, terms,_),_,_,results) = fun}
\]

\[
\text{Pt . fst (snd (snd (assoc t terms)))} |
\]

\[
\text{Tn . snd (assoc n results)} |
\]

\[
\text{V . void_pred rep_type};;
\]

\[
\text{let predicates g = map (predicate g)};;
\]

\[
\text{let injections ((u,_,l),_,_) t =}
\]

\[
\text{map (\lambda v:term. "(inject }\backslash v:u\rightarrow u") (map (dom_val l) (range (length t)))};;
\]

\[
\text{let union_predicate g t = (u_pred o combine) (predicates g t, injections g t)};;
\]

\[
\text{let do_nonrec_U g (gn,s) = (gn, union_predicate g (map snd s))};;
\]

7.4.3.2. Products

The construction of the predicate for non-recursive product is similar to that for the union. The differences are that the predicates for the components must all be saare conjoined rather than disjoined.
ML
let product_predicate g t =
  let ((_,rep_type,_,n),_,_,_) = g
  and preds = predicates g t
  in mk_product_predicate rep_type n preds;;

let do_nonrec_P g (gn,s) = (gn, product_predicate g (map snd s));;

7.4.3.3. Non-Recursive Types
Processing a single non-recursive type is done by the procedure \texttt{do_nonrectype}.

ML
let do_nonrectype ad (name, st) =
  case st of
    Union u .do_nonrec_U ad (name, u) | Prod p .do_nonrec_P ad (name, p);;

Processing of non-recursive types is done in batches. We strip out of the as-yet-unprocessed
types all those which can be defined using only types already processed. Then we process those
types.

ML
let nonrectypes_step (g,u,d,r) =
  let ok, not_ok = partition ((rt_defined d) o snd) u
  in ok = [] => failwith 'nonrectypes_step - finished' |
          (g,
           not_ok,
           d @ ok,
           r @ (map (do_nonrectype (g,u,d,r)) ok));;

This is repeated until there are no more unprocessed types.

ML
letrec careful_repeat_f message_list f arg =
  (careful_repeat_f message_list f) (f arg) ?? message_list arg;;

let do_nonrectype_preds four_tuple =
  careful_repeat_f ['nonrectypes_step - finished'] nonrectypes_step four_tuple;;

7.4.4. Recursive Types
By contrast with the non-recursive types, all the recursive types are done in a single batch (could
be modified later). In fact the same code is invoked to do the recursive types as to do the non-
recursive types, but in the recursive case the invocation is preceded and followed by extra work.

The predicates for recursive types are constructed in three stages.

1. Predicates are constructed using the recursion variable \( f \) for each of the recursive types (these are called \textit{first_preds}).

2. The non-recursive predicate construction code is invoked to construct the predicates again, using the predicates constructed in step 1 (these predicates are called \textit{second_preds}).

3. The predicates constructed in stage 2 are all bundled together into a tuple, an abstraction on \( f \) and \((m: \text{int})\) is constructed and PRIM_REC is applied. Then projections are taken from the result to obtain the final version of the predicates.

The reader is recommended to study the worked example to understand how this works.

We need a procedure for constructing the predicates. The following procedure takes a projection function, \( pr \) and constructs a function of the form \( \lambda x\. \text{`rep_type.pr (f x)}. \)

```ml
ML
let do_rectype_preds (g,u,d,r) = 
    let mk_first_pred rep_type pr = 
        \x: `rep_type. pr (f x)
    and (_,rep_type,_,_) = g
    and len_u = length u
    in let first_preds = combine (map fst u, 
        map (mk_first_pred rep_type) 
        (bool_projections len_u))
    in let second_preds = map (do_nonrectype (g, [], u @ d, r @ first_preds)) u
    in let clause_tuple = list_mk_pair (map (snd o dest_abs o snd) second_preds)
    and bool_t_t = bool_tuple_type len_u
    and false_tuple = list_mk_pair (map_int (\x. "F") len_u)
            \(x. \exists n: num. x (\text{rec_preds n r})). `bool_projections len_u)
    in let predicates = map (\x. "\(r: `rep_type. `\exists n: num. x (\text{rec_preds n r})\)) (bool_projections len_u)
    in (g, [], u @ d, r @ (combine (map fst second_preds, predicates)));

ML
let make_predicates = do_rectype_preds o do_nonrectype_preds;
```
The Theory 042
Parents -- HOL 041
Constants --
  pf_empty ":(*dom,*cod)pfun"
  list_mk_pfun ":(*dom # *cod)list → (*dom,*cod)pfun"
  pf_list_merge ":((*dom,*cod)pfun)list → (*dom,*cod)pfun"
  rmap ":(*dom)list → ((*dom → *cod) → (*cod)list)"
  map ":(*dom → *cod) → ((*dom)list → (*cod)list)"
  mk_primitive ":*cod → ((*dom)list,*cod)pfun"
  inject ":*dom → (((*dom)list,*cod)pfun → ((*dom)list,*cod)pfun)"
  union_comp ":((*dom)list,*cod)pfun → *dom"
  mk_schema
  ":(*dom # ((*dom)list,*cod)pfun)list → ((*dom)list,*cod)pfun"
  proj ":*dom → (((*dom)list,*cod)pfun → ((*dom)list,*cod)pfun)"
  mk_is_primitive
  ":(*primitive_type → *cod) → (((*dom)list,*cod)pfun → bool)"
Curried Infixes --
  pf_merge ":(*dom,*cod)pfun → ((*dom,*cod)pfun → (*dom,*cod)pfun)"
Definitions --
  pf_empty |- pf_empty = mk_pfun(λx. pf_u)
  list_mk_pfun_DEF
  |- list_mk_pfun =
     LIST_REC_D pf_empty(λa g00012. (g00012 pf_add (FST a))(SND a))
  pf_merge
  |- If pf_merge rf =
     mk_pfun
     (λarg.
      (If pdef arg =>
       pf_val(lf apf arg) | (pf pdef arg => pf_val(rf apf arg) | pf_u)))
  pf_list_merge_DEF
  |- pf_list_merge = LIST_REC_D pf_empty(λa g00013. a pf_merge g00013)
  rmap_DEF
  |- rmap = LIST_REC_D(λf. [])(λa g00014 f. CONS(f a)(g00014 f))
  map |- map f l = rmap l f
  mk_primitive |- mk_primitive v = list_mk_pfun([],v)
  inject
  |- inject d f =
     mk_pfun
     (λl.
      (NULL l =>
       pf_u | ((HD l = d) ∧ pdef (TL l)) => pf_val(f apf (TL l) | pf_u)))
  union_comp |- union_comp f = (μy. ∃z. f pdef (CONS y z))
mk_schema
|- mk_schema v = pf_list_merge(map(UNCURRY(\x y. inject x y))v)

proj
|- proj t s =
  mk_pfun
  (∀l. (s pdef (CONS t l) => pf_val(s apf (CONS t l)) | pf_u))

mk_is_primitive
|- mk_is_primitive inj rep = (∃v. rep = mk_primitive(inj v))

Theorems --
list_mk_pfun
|- (list_mk_pfun[] = pf_empty) ∧
  (∀a l.
    list_mk_pfun(CONS a l) =
    (((list_mk_pfun l) pf_add (FST a))(SND a))

pf_list_merge
|- (pf_list_merge[] = pf_empty) ∧
  (pf_list_merge(CONS a l) = a pf_merge (pf_list_merge l))

rmap |- (rmap([]f = [])) ∧ (rmap(CONS a l)f = CONS(f a)(rmap l f))