Unifying and Antiunifying Type and Term Nets

Roger Bishop Jones

Date: 2006/10/21 16:53:33

Abstract

Theorem proving in ProofPower is heavily based on rewriting which is supported by term nets which partially match the rewriting rules against target terms. To provide a higher level of automation using unification, closer to the power of modern predicate calculus automation present in other implementations of HOL term nets which unify rather than match, and which also produce antiunifiers have been considered here. This is mainly design, and though there is a very crude implementation, this is for evaluation only and would not deliver reasonable performance.

http://www.rbjones.com/rbjpub/pp/doc/t011.pdf

Id: t011.doc,v 1.5 2006/10/21 16:53:33 rbj01 Exp

Contents

References

[1] ds/fmu/ied/dtd001. *Detailed Design for SML Utilities.* K. Blackburn, Lemma 1 Ltd., `http://www.lemma-one.com`.

[2] ds/fmu/ied/imp001. *Implementation for SML Utilities.* K. Blackburn, Lemma 1 Ltd., `http://www.lemma-one.com`.

1 Introduction

SML

```
open_theory "basic_hol";
set_pc "basic_hol";
```

2 LISTS

The dictionary facilities provided below are given with a simple implementation using lists of pairs for a dictionary. The operations over these dictionaries often may be applicable for other kinds of lists. The following signature therefore provides a supplementary selection of operations over lists which are used to implement the dictionary facilities which appear in the next section.

2.1 Signature ListUtilities2

SML
```
signature ListUtilities2 = sig
```

SML
```
val l_app : ('a -> 'a) -> string -> (string * 'a) list -> (string * 'a) list;
val l_app2 : (string * 'a -> 'a) -> string -> (string * 'a) list -> (string * 'a) list;
```

SML
```
end; (* of signature ListUtilities2 *)
```

2.2 Structure ListUtilities2

SML
```
structure ListUtilities2 : ListUtilities2 = struct
```

SML
```
fun l_app f x [] = fail "ListUtilities2" 100111 [fn y => x]
| l_app f x ((hi, hv)::t) = if hi = x then ((hi, f hv)::t) else (hi, hv)::(l_app f x t);

fun l_app2 f x [] = fail "ListUtilities2" 100111 [fn y => x]
| l_app2 f x ((h as (hi, hv))::t) = if hi = x then ((hi, f h)::t) else h::(l_app2 f x t);

```

SML
```
end; (* of structure ListUtilities2 *)
```

3 DICTIONARIES

3.1 Mapping Dictionary

This is an extension to the simple dictionary in dtd/imp001 [1, 2], but could also be implemented using efficient dictionaries or be given a hybrid implementation.

Strictly extending the signature makes no sense since the simple and efficient dictionaries have disjoint signatures. This new signature is intended to be relatively independendent of implementation method, so that there may be more than one structure which implements it.

Having decided that a new signature is necessary, I have nevertheless made it as close to the precedent provided by *SimpleDictionary* as possible.

3.1.1 Signature

SML

```
signature MappingDictionary = sig
```

Description This is a signature for dictionaries which associate values with string keys and which provide facilities for filtering the dictionary and or mapping functions over the dictionary.

See Also *Simple$_D$ictionary*, *EfficientDictionary*.

Errors

100111 Failed to find key ?0.

SML

```
type (''k,'a) M_DICT;
val initial_m_dict : (''k,'a) M_DICT;
val m_lookup : ''k ->(''k,'a) M_DICT -> 'a OPT;
val m_enter : ''k -> 'a -> (''k,'a) M_DICT -> (''k,'a) M_DICT;
val m_list_enter : (''k * 'a) list -> (''k,'a) M_DICT -> (''k,'a) M_DICT;
val m_extend : ''k -> 'a -> (''k,'a) M_DICT -> (''k,'a) M_DICT;
val m_list_extend : (''k * 'a) list -> (''k,'a) M_DICT -> (''k,'a) M_DICT;
val m_delete : ''k -> (''k,'a) M_DICT -> (''k,'a) M_DICT;
```

Description

initial_m_dict is the empty dictionary which associates nothing with any key value.

m_lookup key dict retrieves value, if any, stored in *dict* under key *key*.

m_enter key val dict gives a dictionary which associates *val* with the key *key* and otherwise is the same as the original dictionary *dict*.

m_extend key val dict is similar to *m_enter* except that it will raise an exception if *key* is already associated with a value in *dict*.

list_m_enter and *list_m_extend* enter into or extend by a list of string value pairs.

m_delete key dict gives a dictionary which associates no value with the key *key* and otherwise is the same as the original dictionary *dict*.

SML

```
val m_app : ''k -> ('a -> 'a) -> (''k,'a) M_DICT -> (''k,'a) M_DICT;
val m_mapfilter : ((''k * 'a) -> 'b) -> (''k,'a) M_DICT -> (''k,'b) M_DICT;
val m_mapfilter_list : ((''k * 'a) -> 'b) -> (''k,'a) M_DICT -> 'b list;
val m_list : (''k,'a) M_DICT -> (''k * 'a) list;
val m_combine : ('a OPT * 'b OPT -> 'c OPT)
        -> (''k,'a) M_DICT -> (''k,'b) M_DICT -> (''k,'c) M_DICT;
val m_override : (''k,'a) M_DICT -> (''k,'a) M_DICT -> (''k,'a) M_DICT;
val m_merge : (''k,'a) M_DICT -> (''k,'a) M_DICT -> (''k,'a) M_DICT;
```

Description

m_app key f gives a dictionary which associates with the key *key* the result of applying the function *f* to the value associated with key *key* in dictionary *dict*, and is otherwise the same as *dict*.

m_mapfilter maps a function over the contents of a dictionary. If the function raises exception Fail this is caught and causes the relevant entry to be omitted from the new dictionary.

m_mapfilter_list is the simlar to *m_mapfilter* except that it returns a list rather than a dictionary.

m_list is the same as *m_mapfilter_list* applied to the function *fnx => x*.

m_combine fun dict1 dict2 gives the dictionary obtained by combining using *fun* the entries in each dictionary for every key which occurs in either.

m_override dict1 dict2 gives the dictionary obtained by entering (as *m_enter*) into *dict1* every *key value* pair present in *dict2*.

m_merge dict1 dict2 gives the dictionary obtained by extending (as *m_extend*) into *dict1* every *key value* pair present in *dict2*.

SML

```
end (* of MappingDictionary signature *);
```

3.1.2 Simple Implementation

This is implemented in a similar manner to SimpleDictionary, representing the dictionary as a list of key/value pairs.

I would have actually used *SimpleDictionary* to implement it if I could have, but the visibility of the representation type is necessary for the extra functionality, so this could not be done.

SML

```
structure SimpleMDictionary : MappingDictionary = struct
```

SML

```
type (''k,'a) M_DICT = (''k * 'a) list;
```

The implementation of the functions corresponding to those of *SimpleDictionary* is copied from imp001 [2], the only changes being to the names of the functions.

The additional functions are implemented as follows:

SML

```
fun m_list_enter l = fold (fn ((s, v), d) => m_enter s v d) l;
fun m_list_extend l = fold (fn ((s, v), d) => m_extend s v d) l;
```

SML

```
fun m_app tag f ([]: ("k,'a) M_DICT) : ("k,'a) M_DICT
    = fail "MappingDictionary" 100111 []
|   m_app tag f ((he as (ht,hv))::t)
    =  if ht = tag
       then (ht, f hv)::t
       else (he :: m_app tag f t);
fun mfa f (k,v) = (k, f(k,v));
fun m_mapfilter f (d: ("k,'a) M_DICT) : ("k,'b) M_DICT = mapfilter (mfa f) d;
val m_mapfilter_list : (("k * 'a) -> 'b) -> ("k,'a) M_DICT -> 'b list
    = mapfilter;
fun m_list (dict: ("k,'a) M_DICT) : ("k * 'a) list = dict;

fun m_combine f d1 d2 =
 let fun    aux acc ((s,v)::t) [] =
        let val acc' = case f (Value v, Nil) of
                  Value v' => ((s, v')::acc)
                | Nil => acc
        in aux acc' t []
        end
    |    aux acc [] ((s,v)::t) =
        let val acc' = case f (Nil, Value v) of
                  Value v' => ((s, v')::acc)
                | Nil => acc
        in aux acc' [] t
        end
    |    aux acc ((s1,v1)::t1) ((s2,v2)::t2) =
        if s1 = s2
        then   let val acc' = case f (Value v1, Value v2) of
                  Value v' => ((s1, v')::acc)
                | Nil => acc
            in aux acc' t1 t2
            end
        else    aux acc ((s1,v1)::t1) t2
    |   aux acc [] [] = acc
 in aux [] d1 d2
 end;

fun m_override (dict1 : ("k,'a) M_DICT) (dict2 : ("k,'a) M_DICT) : ("k,'a) M_DICT = (
    fold (uncurry(uncurry m_enter)) dict1 dict2);
```

SML

```
end (* of SimpleMDictionary *);
```

3.2 List Indexed Dictionary

3.2.1 Signature

SML

```
signature ListIndexDictionary = sig
```

Description Holds a set of Standard ML functions concerned with managing families of values indexed by lists of strings.

Uses For use in implementing generic discrimination nets. The main distinctive features are:

- indexed by lists of strings
- muliple values can be saved under each key
- provides facilities for mapping functions and/or filters over the entries

The motivation for this facility is to support saving data indexed by structured entities such as *TYPE*s and *TERM*s is such a way that computation over the entire set of key values can be done efficiently. The kind of computation we have in mind here is unification, though any computation which can be accomplished on a string stream encoding of a structured value would also be possible. The idea is to be able to select from the dictionary and perform some computation on all the values whose index is unifiable with a given *TYPE* or *TERM*, in such a way that common initial segments of the required computations are not repeated for structures which share that initial segment of structure.

The main differences from the *MappingDictionary* signature are therefore, firstly that lists of strings are used as key values, and secondly that mapping functions take these keys one item at a time, and the intermediate values computed can be reused for every key value sharing that initial segment of the list.

See Also *MappingDictionary*.

SML

```
type ("k,'a) LI_DICT;
val initial_li_dict : ("k,'a) LI_DICT;
val li_lookup : "k list -> ("k,'a) LI_DICT -> 'a list;
val li_enter : "k list -> 'a -> ("k,'a) LI_DICT -> ("k,'a) LI_DICT;
val li_enter_list : "k list -> 'a list -> ("k,'a) LI_DICT -> ("k,'a) LI_DICT;
val li_delete : "k list -> ("k,'a) LI_DICT -> ("k,'a) LI_DICT;
val li_replace : "k list -> 'a list -> ("k,'a) LI_DICT -> ("k,'a) LI_DICT;
val li_list : ("k,'a) LI_DICT -> ("k list * 'a list) list;

val li_merge : ("k,'a) LI_DICT -> ("k,'a) LI_DICT -> ("k,'a) LI_DICT;
```

Description

initial_li_dict is the empty list indexed dictionary.

li_lookup taglist dict returns the values held in *dict* under the index *taglist*.

li_enter taglist value dict adds *value* to the front of the list of elements held in *dict* under the index *taglist*.

li_enter_list taglist valuelist dict appends *valuelist* to the front of the list of elements held in *dict* under the index *taglist*.

li_delete taglist dict removes all the values associated with *taglist* in *dict*.

li_replace taglist valuelist dict replaces the list of values associated with *taglist* in *dict* with *valuelist*.

li_list dict returns a list of pairs of tag lists and value lists corresponding to the entire content of the dictionary.

*li_merge dict*1 *dict*2 constructs a dict in which the list of values associated with any taglist is the list of values associate with that taglist in *dict*1 appended to the list of values associated with that taglist in *dict*2.

SML

```
datatype ("k, 'a, 'b) LIDFUN = LidRetain
        |   LidDiscard
        |   LidFun of {linode: "k -> ("k, 'a, 'b) LIDFUN,
                 lileaf   : ('a list) -> 'b};
val li_mapfilter : ("k,'a,'b list) LIDFUN -> ("k,'a) LI_DICT -> ("k,'b) LI_DICT;
val li_mapfilter2 : ("k,'a,'a list) LIDFUN -> ("k,'a) LI_DICT -> ("k,'a) LI_DICT;
val li_mapfilter_list : ("k,'a,'b) LIDFUN -> ("k,'a) LI_DICT -> 'b list;
```

Description These functions are designed to support efficient processing of the entire dictionary using functions which process a stream of tag values. The datatype *LIDFUN* is concocted to support such applications, and the generic trawling/mapping/filtering operations expect a *LIDFUN* as a parameter. To have a non-trivial effect a *LidFun* would be supplied and the tags would be sucessively applied to the *linode* components of a succession of *LIDFUN*s until a leaf is found which is then transformed by the *lileaf* component of the last *LidFun*. This process is effectively repeated for each path in the dictionary, except that paths sharing an initial segment will also share the computation associated with that initial segment (side effects are not intended). The two other kinds of *LIDFUN* allow this process to be curtailed. When a *LidFun linode* returns a *LidDiscard* then the result is as if there were no paths in the original dictionary with that initial path segment. When a *LidFun linode* is given a tag and returns a *LidRetain* then the entire subtree with that initial path segment is to be retained unchanged (the only operator whose type is compatible with the use of *LidRetain* is *li_mapfilter2*, others will raise an exception if it arises).

li_mapfilter lidfun dict yields a new dictionary in which, against any taglist is held the value obtained by submitting the first value in each taglist to the *linode* component of the *lidfun* argument yeilding a new *LIDFUN*, submitting the next string to the *linode* component of that *LIDFUN* and so on until reaching a leaf, at which point the *'alist* stored under that taglist is supplied to the *lileaf* component of the current *LIDFUN*. If at any point an application of a *LIDFUN* raises a *Fail* exception then no values with a *taglist* which has an initial segment corresponding to the tags so far processed will appear in the resulting dictionary. If any other exception is raised it will not be trapped. For a pure filtering operation the *lileaf* component of the *LIDFUN* argument should be the identity function.

li_mapfilter_list lidfun dict is similar to *li_mapfilter* except that the results of applying the *lileaf* components of the *LIDFUN* are returned as a list rather than a dictionary.

Errors

```
100112 LidRetain encountered by {\it li\_mapfilter}
```

SML

```
end (* of ListIndexDictionary signature *);
```

SML

```
structure SimpleLIDictionary : ListIndexDictionary = struct
open SimpleMDictionary;
```

SML

```
end (* of SimpleLIDictionary *);
open SimpleLIDictionary;
```

4 UNIFYING STORES

SML

```
signature UNet = sig
```

Description This is the signature of a structure providing facilities similar to those provided by NetTools except that lookup involves unification and returns all items indexed by styructures which are unifiable with the lookup structure. The results include the unifying substitutions and an anti-unifier of the matching values.

Though in the target applications the indexes are HOL terms this interface is less specific. It is expected that the structure involved is coded as a list of tag values, the type of which is only partly specified here. Sufficient tag structure is specified to pernit unification, with polymorphic slots for application specific structure the details of which do not affect the unification algorithm.

SML

```
type ('a, 'b)UTAG
type ('a, 'b, 'c)UNET;
type ('a, 'b)USUBS;
```

Description Type $('a,'b)UTAG$ is a type of tags, lists of which may be used to represent various kinds of structures when the type parameters are suitably instantiated.

$('a,'b,'c)UNET$ is the type of a unifying net storing values of type $'c$ indexed by lists of tags of type $('a,'b)UTAG$.

$('a,'b)USUBS$ is a value which represents unifying substitutions.

SML

```
val mk_uterm : (TERM * TYPE list * TERM list * TYPE list * TERM list)
    -> UTERM;
val dest_uterm : UTERM
    -> (TERM * TYPE list * TERM list * TYPE list * TERM list);
```

Description The interfaces to unifying term net facilities make use of the type $UTERM$, which stands for $UnifiableTerm$, and left undetermined by the signature to allow optimisation of this type.

The functions *mk_uterm* and *dest_uterm* provide the external methods for assembling and disassembling $UTERM$s. The components are:

1. a term for unification
2. a list of type variables which are to be avoided when creating new type variables
3. a list of term variables which are to be avoided when creating new term variables
4. a list of type variables which can be instantiated during unification
5. a list of term variables which can be instantiated during unification

SML

```
val empty_unet : ('a, 'b, 'c) UNET;
val make_utmnet : (('a,'b)UTAG * 'c) list -> ('a,'b,'c) UNET;
val unet_enter : ('a,'b,'c) UNET -> (('a,'b)UTAG * 'c) -> ('a,'b,'c)UNET;
val list_utmnet_enter : ('a,'b,'c) UNET -> (('a,'b)UTAG * 'c) list -> ('a,'b,'c)UNET;
```

Description *empty_unet* gives an empty unet, the type parameters are the type of the nodes and leaves of the tag types and the type of the values to be stored in the unet. *make_unet* takes a list of index/value pairs and inserts them into an empty utmnet. *unet_enter* enters a single new value into a utmnet, *list_unet_enter* adds a list of new entries.

The indexing term must be supplied together with information controlling unification which consists of:

1. a list of type variables which should be avoided
2. a list of term variables which should be avoided
3. the list of type variables which may be instantiated
4. the list of term variables which may be instantiated

SML

```
val utmnet_content : ('a UTMNET) -> (UTERM * 'a)list;
val utmnet_lookup : ('a UTMNET) -> UTERM
        -> ((UTMSUBST * UTERM * UTMSUBST * 'a)list * UTMSUBST);
val utmnet_map_filter : ('a UTMNET) -> ((UTERM * 'a) -> 'b)
        -> UTERM -> ('b UTMNET);
val utmnet_map : ('a UTMNET) -> ((UTERM * 'a) -> 'b) -> ('b UTMNET);
val utmnet_filter : ('a UTMNET) -> UTERM -> ('a UTMNET);
val utmnet_fold : (((UTERM * 'a) * 'b) -> 'b) -> ('a UTMNET) -> 'b -> 'b;
```

Description *utmnet_content* is the inverse of *make_utmnet*.

utmnet_lookup net uterm will return a list of the values entered into *net* that were indexed by *uterm*s which can be unified with *uterm*.

Each value is returned with the following information:

1. a substitution which may be applied to the search uterm to unify it with the relevant index uterm
2. an index uterm found to be unifiable with the search uterm
3. a substitution which may be applied to the index uterm to unify it with the search uterm
4. the value associated with the index entry

One further substitution is returned, which instantiates the search uterm to the anti-unifier of the returned terms. This is not guaranteed to be the most specific antiunifier, some implementations may decline to antiunify and should then return the null substitution.

If *utmnet_lookup* returns more than one value, then the only ordering on the resulting values specified is that if two entries are made into the net with the same index term, then if the *net_lookup* term matches the index term then the second entered value will be returned before the first in the list of matches.

utmnet_map_filter filters a *UNET* retaining only items indexed by terms which are unifiable with its argument, and applies the supplied function to the index/value pair replacing the value with the result. If the function fails then the index/value pair is discarded.

utmnet_filter is the special cases of *utmnet_map_filter* in which the map is the right projection function.

utmnet_map is the special cases of *utmnet_map_filter* in which the function is applied to the entire net, the only items dropped being those on which the function fails.

utmnet_foldfutmnc folds the function over the values in the termnet with initial value c.

SML

```
end; (* of signature UNet *)
```

SML

```
structure SimpleUNet : UNet = struct

open SimpleLIDictionary;
```

SML

```
datatype ('a,'b)UTAG =
        UtNode of 'a
    |   UtLeaf of 'b
    |   UtVb of string
    |   UtBv of string
    |   UtIv of string
    |   UtEnd;

type ('a,'b,'c)UNET = (('a,'b)UTAG, 'c) LI_DICT;

type ('a,'b)USUBST = (string * ('a,'b)UTAG list) list;
```

SML

```
end (* of structure SimpleNet *)
```

5 UNIFYING TERM NETS

SML

```
signature TermNet = sig
```

Description This is the signature of a structure providing facilities similar to those provided by NetTools except that lookup involves unification and returns all items indexed by terms which are unifiable with the lookup term. The results include the unifying substitutions and an anti-unifier of the matching terms.

SML

```
type UTERM
type 'a UTMNET;
type UTMSUBST;
```

Description $'aUTMNET$ is the type of a unifying term net storing values of type $'a$ indexed by terms.

$UTMSUBST$ is a value which represents unifying substitutions.

SML
```
val mk_uterm : (TERM * TYPE list * TERM list * TYPE list * TERM list)
    -> UTERM;
val dest_uterm : UTERM
    -> (TERM * TYPE list * TERM list * TYPE list * TERM list);
```

Description The interfaces to unifying term net facilities make use of the type *UTERM*, which stands for *UnifiableTerm*, and left undetermined by the signature to allow optimisation of this type.

The functions *mk_uterm* and *dest_uterm* provide the external methods for assembling and disassembling *UTERM*s. The components are:

1. a term for unification
2. a list of type variables which are to be avoided when creating new type variables
3. a list of term variables which are to be avoided when creating new term variables
4. a list of type variables which can be instantiated during unification
5. a list of term variables which can be instantiated during unification

SML
```
val empty_utmnet : 'a UTMNET;
val make_utmnet : (UTERM * 'a) list -> ('a UTMNET);
val utmnet_enter : ('a UTMNET) -> (UTERM * 'a) -> ('a UTMNET);
val list_utmnet_enter : ('a UTMNET) -> (UTERM * 'a) list -> ('a UTMNET);
```

Description *empty_utmnet* gives an empty utmnet, the type parameter is the type of the values to be stored in the utmnet. *make_utmnet* takes a list of index/value pairs and inserts them into an empty utmnet. *utmnet_enter* enters a single new value into a utmnet, *list_utmnet_enter* adds a list of new entries.

The indexing term must be supplied together with information controlling unification which consists of:

1. a list of type variables which should be avoided
2. a list of term variables which should be avoided
3. the list of type variables which may be instantiated
4. the list of term variables which may be instantiated

SML

```
val utmnet_content : ('a UTMNET) -> (UTERM * 'a)list;
val utmnet_lookup : ('a UTMNET) -> UTERM
        -> ((UTMSUBST * UTERM * UTMSUBST * 'a)list * UTMSUBST);
val utmnet_map_filter : ('a UTMNET) -> ((UTERM * 'a) -> 'b)
        -> UTERM -> ('b UTMNET);
val utmnet_map : ('a UTMNET) -> ((UTERM * 'a) -> 'b) -> ('b UTMNET);
val utmnet_filter : ('a UTMNET) -> UTERM -> ('a UTMNET);
val utmnet_fold : (((UTERM * 'a) * 'b) -> 'b) -> ('a UTMNET) -> 'b -> 'b;
```

Description *utmnet_content* is the inverse of *make_utmnet*.

utmnet_lookup net uterm will return a list of the values entered into *net* that were indexed by *uterm*s which can be unified with *uterm*.

Each value is returned with the following information:

1. a substitution which may be applied to the search uterm to unify it with the relevant index uterm
2. an index uterm found to be unifiable with the search uterm
3. a substitution which may be applied to the index uterm to unify it with the search uterm
4. the value associated with the index entry

One further substitution is returned, which instantiates the search uterm to the anti-unifier of the returned terms. This is not guaranteed to be the most specific antiunifier, some implementations may decline to antiunify and should then return the null substitution.

If *utmnet_lookup* returns more than one value, then the only ordering on the resulting values specified is that if two entries are made into the net with the same index term, then if the *net_lookup* term matches the index term then the second entered value will be returned before the first in the list of matches.

utmnet_map_filter filters a *UNET* retaining only items indexed by terms which are unifiable with its argument, and applies the supplied function to the index/value pair replacing the value with the result. If the function fails then the index/value pair is discarded.

utmnet_filter is the special cases of *utmnet_map_filter* in which the map is the right projection function.

utmnet_map is the special cases of *utmnet_map_filter* in which the function is applied to the entire net, the only items dropped being those on which the function fails.

utmnet_foldfutmnc folds the function over the values in the termnet with initial value c.

SML

```
end; (* of signature TermNet *)
```

The structure *CrudeUnifyNet* is an implementation of signature *UnifyNet* which represents a net as a list of pairs, uses the unification algorithm from the resolution structure and returns the search term as antiunifier. (this is to permit experimentation with the functionality of backchaining before getting into the details of efficient unification nets).

SML

```
structure CrudeTermNet : TermNet = struct
```

SML
```
end; (* of structure CrudeTermNet *)
```

SML
```
structure UNet : UNet
```

SML
```
end; (* structure UNet *)
```

6 INDEX