The HOL pred_sets Library

T. F. Melham

University of Cambridge, Computer Laboratory
New Museums Site, Pembroke Street
Cambridge, CB2 3QG, England.

February 1992

Contents

Chapter 1

The pred_sets Library

The pred_sets library contains a theory of predicates regarded as sets. A predicate s:*->bool is considered as a collection or 'set' of elements of type *, and the standard operations on sets such as union, intersection, and set difference are appropriately defined for this representation. The library was originally written in 1989 by Ton Kalker. It was completely rewritten by the present author for HOL version 2.01 in early 1992. The aim of this revision was to make the pred_sets library closely parallel to the much more developed HOL sets library, with the same names for constants and theorems and the same form of definitions for operations on sets. The present document is itself also adapted from the manual for the sets library [1].

There is only one theory in the pred_sets library, namely the theory 'pred_sets'. This document explains the logical basis of this theory and the theorem-proving support provided by library. The latter includes conversions for expanding set specifications and for evaluating various operations on finite sets described by enumeration of their elements. The library also provides parser and pretty-printer support for terms that denote sets.

1.1 Membership and the axioms of set theory

A value x is defined to be an element of a set exactly when the characteristic predicate of the set is true of x. Since sets in the pred_sets library are just represented by their characteristic predicates, this membership relation is straightforward to define as follows:

```
SPECIFICATION   |- !P x. x IN P = P x
```

The infix function constant IN defined here constitutes the basic language for the entire theory of sets in the pred_sets library; all operators and predicates on sets are ultimately defined in terms of this one function.

The definition of IN shown above loosely corresponds to what is usually called the *axiom of specification* for sets (hence the name SPECIFICATION). This axiom states that sets can be constructed from predicates that describe or 'specify' their elements. A value is an element of the constructed set exactly when the predicate is true of that value.

Since sets and predicates are identical in the `pred_sets` library, we can simply say that `x` is in the 'set' `P` exactly when `P x` holds.

The definition of `IN` is one of two fundamental theorems in the `pred_sets` library, from which all others are derived. The second of these fundamental theorems states what is usually called the *axiom of extension* for sets. This is not, of course, literally an *axiom* of the `pred_sets` theory, but rather a theorem derived by proof:

```
EXTENSION   |- !s t. (s = t) = (!x. x IN s = x IN t)
```

`EXTENSION` states that two sets are equal exactly when they have the same elements. This follows directly from the definition of the constant `IN` and the extensionality functions in higher order logic.

Once the theorems `EXTENSION` and `SPECIFICATION` have been proved, they provide a complete basis for all further reasoning about sets and membership. The library theory `pred_sets` is developed entirely on the basis of these two 'axioms' of set theory.

1.2 Generalized set specifications

In addition to the basic constant `IN`, which allows one to regard a predicate as the set of all values that satisfy it, the `pred_sets` library also provides a general way of constructing sets by describing or specifying their elements. Roughly speaking, there are two components to a generalized set specification: an expression `E[x]` and a predicate `P[x]`. For any such expression and predicate, there is a corresponding set `{E[x] | P[x]}`, the set of all values `E[x]` for which `P[x]` holds.

The `pred_sets` library supports generalized set specifications by means of the constant:

```
GSPEC : (** -> (* # bool)) -> * -> bool
```

The function `GSPEC` takes a function `f : ** -> (* # bool)` and constructs the set (i.e. predicate of type `*->bool`) of all values `FST(f x)` for which `SND(f x)` holds, for some value `x` of type `**`. The formal definition of the constant `GSPEC` is given by the following constant specification:

```
GSPECIFICATION   |- !f v. v IN (GSPEC f) = (?x. v,T = f x)
```

This theorem is analogous to the axiom of specification for `IN`. This states that a value `v` is an element of the set specified by `f` exactly when `v` is one of the values of `FST(f x)` for which `SND(f x)` is true.

To see how this supports the notion of generalized set specification described above, let `f` in this definition be the function `\x.E[x],P[x]`. With a little simplification, we would then have:

```
|- !v. v IN (GSPEC \x.E[x],P[x]) = ?x. (v = E[x]) /\ P[x]
```

That is, a value `v` is in the set constructed by `GSPEC` exactly when for some `x` for which `P[x]`, the value `v` is equal to `E[x]`. The constructed set therefore contains all values `E[x]` for which `P[x]` holds.

1.2.1 Parser and pretty-printer support

To facilitate the use of sets constructed by generalized set specification, the `pred_sets` library provides parser and pretty-printer support for set abstractions expressed by the notation "{E | P}". The built-in ML function `define_set_abstraction_syntax` (see the manual [2] for details) is used to introduce this notation when the library is loaded. The call made to this function extends the HOL parser so that a quotation of the form "{E | P}" parses to:

GSPEC (\(x_1,...,x_n).(E,P))

where x_1, ..., x_n are the variables that occur free in both the expression E and the proposition P (i.e. the set $\{x_1, \ldots, x_n\}$ is the intersection of the set of free variables of E and the set of free variables of P). If there are *no* variables free in both E and P, then a parser error is generated. When the `print_set` flag is `true`, the quotation pretty-printer inverts this transformation.

A simple example of this set abstraction notation is shown in the following HOL session, in which it is assumed that the `pred_sets` library has already been loaded. (See section 1.14 for a description of how `pred_sets` is loaded.)

```
#let gtr = new_definition (`gtr`, "gtr N = {n | n > N}");;        1
gtr = |- !N. gtr N = {n | n > N}

#set_flag (`print_set`,false);;
true : bool

#"{n | n > N}";;
"GSPEC(\n. (n,n > N))" : term
```

The term `{n | n > N}` in the definition of `gtr` denotes the set of all natural numbers greater than `N`. It is important to note that the variable `N` is a free variable in this term, since it occurs on only one side of the bar '`|`'. The set abstraction `{n | n > N}` therefore parses to the generalized set specification

```
GSPEC(\n. (n,n > N))
```

This is what gives this set abstraction the (presumably intended) interpretation 'the set of all `n` greater than `N`'. By contrast, the term

```
GSPEC(\(n,N). (n,n > N))
```

denotes the set of all numbers `n` greater than some number `N`—i.e., the set $\{1, 2, 3, \ldots\}$. This is *not* the default interpretation of the parser, which constructs a generalized set specification that binds the variable `n` only. Note that only default interpretations are pretty-printed using the set abstraction notation:

```
#set_flag(`print_set`,true);;                                          2
false : bool

#"GSPEC (\n. (n,n>N))";;
"{n | n > N}" : term

#"GSPEC (\(n,N). (n,n>N))";;
"GSPEC(\(n,N). (n,n > N))" : term
```

That is, a term of the form:

GSPEC (\(x_1,...,x_n).(E,P))

prints as "{E | P}" only if the variables x_1, ..., x_n occur free in both E and P.

In general, the expression E in a set abstraction "{E | P}" need not be just a variable. Consider, for example, the following HOL session:

```
#let S = "{(n,m) | n < m}";;                                           3
S = "{(n,m) | n < m}" : term

#set_flag(`print_set`,false);;
true : bool

#"{(n,m) | n < m}";;
"GSPEC(\(n,m). ((n,m),n < m))" : term
```

Here, a set abstraction is used to construct the set of all pairs of numbers `(n,m)` for which `n` is less than `m`. Note that both variables `n` and `m` are bound in the underlying generalized set specification.

1.2.2 Theorem-proving support

The `pred_sets` library provides proof support for the set abstraction notation in the form of a conversion called `SET_SPEC_CONV`. This conversion implements the axiom of specification for set abstractions. When v is a variable, evaluating:

SET_SPEC_CONV "t IN {v | P}";;

returns the theorem:

```
|- t IN {v | P} = P[t/v]
```

This states that t is an element of the set of all v such that P exactly when $P[t/v]$ holds. Note that, in general, the term t need not be a variable. The following session illustrates this use of SET_SPEC_CONV for membershipin a particular set abstraction:

```
#SET_SPEC_CONV "12 IN {n | n > N}";;                                    1
|- 12 IN {n | n > N} = 12 > N
```

The conversion SET_SPEC_CONV behaves differently when applied to terms of the form "t IN {E | P}" where E is not a variable. Applying the conversion to a term of this kind yields the theorem:

```
|- t IN {E | P} = ?x1... xn. (t = E) /\ P
```

where x_1, ..., x_n are the variables that occur free in both E and P. The expression E cannot in general be eliminated in this case, as it can by the substitution $P[t/v]$ when E is just a variable v.

The following session illustrates the form of the theorem proved by SET_SPEC_CONV for the second type of input term discussed above:

```
#let th1 = SET_SPEC_CONV "p IN {(n,m) | n < m}";;                        1
th1 = |- p IN {(n,m) | n < m} = (?n m. (p = n,m) /\ n < m)

#let th2 = SET_SPEC_CONV "(a,b) IN {(n,m) | n < m}";;
th2 = |- (a,b) IN {(n,m) | n < m} = (?n m. (a,b = n,m) /\ n < m)

#let th3 = SET_SPEC_CONV "a IN {n + m | n < m}";;
th3 = |- a IN {n + m | n < m} = (?n m. (a = n + m) /\ n < m)
```

The right-hand sides of th1 and th2 could, in principle, be further simplified. The value of the expression '(n,m)' is an injective function of the values of n and m, and so by eliminating the existential quantifiers these two theorems could be simplified to:

```
th1    |- p IN {(n,m) | n < m} = (FST p < SND p)

th2    |- (a,b) IN {(n,m) | n < m} = (a < b)
```

But in general the value of E in a set abstraction "{E | P}" will not be an injective function of its free variables, as for example is the case in theorem th3. The conversion SET_SPEC_CONV therefore attempts no further simplification of its result than is described above for the general case.

1.3 The empty and universal sets

The following two constants are defined in the `pred_sets` library: `EMPTY:*->bool`, which denotes the empty set; and `UNIV:*->bool`, which denotes the universe, or set of all values of type `*`.These constants are defined formally as follows:

```
EMPTY_DEF  |- EMPTY = \x. F
UNIV_DEF   |- UNIV  = \x. T
```

The theorems `EMPTY_DEF` and `UNIV_DEF` shown above are named according to the general convention that all definitions in the `pred_sets` library are given names ending in '`_DEF`'.

Note that because of the restriction on free variables discussed above, the set abstractions `"{x | T}"` and `"{x | F}"` cannot be used in these definitions; the more primitive form of set construction given by the above lambda-abstractions must be used instead. But users of the library will never need to appeal to these definitions, since the following theorems about `EMPTY` and `UNIV` are also made available in the theory `pred_sets`:

```
NOT_IN_EMPTY  |- !x. ~x IN EMPTY
IN_UNIV       |- !x. x IN UNIV
```

That is, nothing is an element of `EMPTY` and everything is an element of `UNIV`. These properties follow directly from the definitions and the theorem `SPECIFICATION`. Other pre-proved theorems about the empty and universal sets are also available in the library; see chapter 3 for a complete list.

1.4 Set inclusion

The infix functions `SUBSET` and `PSUBSET` denote the binary relations of set inclusion and proper set inclusion, respectively. These are defined formally in the obvious way:

```
SUBSET_DEF   |- !s t. s SUBSET t = (!x. x IN s ==> x IN t)
PSUBSET_DEF  |- !s t. s PSUBSET t = s SUBSET t /\ ~(s = t)
```

That is, `s` is a subset of `t` if every element of `s` is also an element of `t`; and `s` is a proper subset of `t` if it is a subset of `t` but not equal to `t`.

Various pre-proved theorems about the subset and proper subset relations are supplied by the `pred_sets` library. For example, the fact that `SUBSET` is a partial order is stated by the three built-in theorems shown below.

```
SUBSET_REFL     |- !s. s SUBSET s
SUBSET_TRANS    |- !s t u. s SUBSET t /\ t SUBSET u ==> s SUBSET u
SUBSET_ANTISYM  |- !s t. s SUBSET t /\ t SUBSET s ==> (s = t)
```

Also provided are built-in theorems about the relationship between set inclusion and other constants or operations on sets. For example, there are the following facts about set inclusion and the empty and universal sets:

```
EMPTY_SUBSET        |- !s. {} SUBSET s
SUBSET_UNIV         |- !s. s SUBSET UNIV
NOT_PSUBSET_EMPTY   |- !s. ~s PSUBSET {}
NOT_UNIV_PSUBSET    |- !s. ~UNIV PSUBSET s
```

As these examples illustrate, the names of theorems in the `pred_sets` library are generally constructed from the names of the constants they contain. Furthermore, the ordering of elements in the name of a theorem attempts to reflect the content of the theorem itself.

1.5 Union, intersection, and set difference

The binary operations of union, intersection and set difference are all defined using the set abstraction notation introduced above in section 1.2.1. The formal definitions are:

```
UNION_DEF    |- !s t. s UNION t = {x | x IN s \/ x IN t}
INTER_DEF    |- !s t. s INTER t = {x | x IN s /\ x IN t}
DIFF_DEF     |- !s t. s DIFF t = {x | x IN s /\ ~x IN t}
```

These definitions illustrate the practical utility of the scheme for variable binding in set abstractions discussed above in section 1.2.1. An abstraction "{E | P}" binds only the variables that occur in both E and P, and the variables `s` and `t` in the set abstractions shown above may therefore be made parameters to the sets constructed by them.

Using `SET_EQ_CONV`, it is trivial to derive the following membership conditions for `UNION`, `INTER` and `DIFF` from the definitions given above. As a general rule, theorems stating membership conditions of the kind illustrated by these examples are given names of the form `IN_`⟨*constant*⟩ ending in the name of the operation used to construct the set in question.

```
IN_UNION  |- !s t x. x IN (s UNION t) = x IN s \/ x IN t
IN_INTER  |- !s t x. x IN (s INTER t) = x IN s /\ x IN t
IN_DIFF   |- !s t x. x IN (s DIFF t) = x IN s /\ ~x IN t
```

These theorems, which are saved in the library under the names indicated above, may in practice be used as the defining properties of union, intersection and set difference; users should almost never have to appeal directly to the definitions of these operations. Other built-in theorems about `UNION`, `INTER` and `DIFF` may be found in chapter 3.

1.6 Disjoint sets

Two sets are *disjoint* if they have no elements in common. This concept is formalized in the `pred_sets` library by the constant `DISJOINT`, the definition of which is:

```
DISJOINT_DEF  |- !s t. DISJOINT s t = (s INTER t = {})
```

At present, there are relatively few pre-proved theorems about the `DISJOINT` relation in the library. But see chapter 3 for the few theorems about `DISJOINT` that are in fact available in the `pred_sets` library.

1.7 Insertion and deletion of an element

To aid in the construction of particular sets of values (especially finite sets) the library contains definitions of two constants `INSERT` and `DELETE`. These denote the operations of augmenting a set with a given value and removing a value from a set, respectively. The formal definitions of these operations are:

```
INSERT_DEF  |- !x s. x INSERT s = {y | (y = x) \/ y IN s}
DELETE_DEF  |- !s x. s DELETE x = s DIFF (INSERT x EMPTY)
```

The elements of the set denoted by `x INSERT s` are all the elements of the set `s` together with the value `x`, which may or may not be an element of `s` itself. The set denoted by `s DELETE x` contains all the elements of `s` except the value `x`.

The membership conditions for sets constructed using `INSERT` and `DELETE` are given by the following pre-proved theorems:

```
IN_INSERT  |- !x y s. x IN (y INSERT s) = (x = y) \/ x IN s
IN_DELETE  |- !s x y. x IN (s DELETE y) = x IN s /\ ~(x = y)
```

In addition, the library contains a substantial collection of theorems about the relationship between the operations `INSERT` and `DELETE` and other relations and operations on sets. Chapter 3 gives a complete list of these theorems.

1.7.1 Parser and pretty-printer support

The `pred_sets` library provides special parser and pretty-printer support for finite sets that are constructed by enumeration of their elements. This notation is introduced by a call made when the library is loaded to the built-in ML function `define_finite_set_syntax` (see [2] for details of this function). This has the effect of extending the HOL parser so that a quotation of the form "{t_1,t_2,...,t_n}" parses to the following set built up from `EMPTY` by repeatedly using the function `INSERT`:

INSERT t_1 (INSERT t_2 ... (INSERT t_n EMPTY)...)

Note that the quotation `"{}"` just parses to the constant `EMPTY`. When the `print_set` flag is `true`, the HOL pretty-printer for terms inverts this transformation.

Users should note that care must be taken with regard to the precedence of comma in a context `"{...}"`, as the following session illustrates:

```
#set_flag(`print_set`,false);;                                              1
true : bool

#"{1,2,3,4}";;
"1 INSERT (2 INSERT (3 INSERT (4 INSERT EMPTY)))" : term

#"{(1,2),(3,4)}";;
"(1,2) INSERT ((3,4) INSERT EMPTY)" : term

#"{((1,2),(3,4))}";;
"((1,2),3,4) INSERT EMPTY" : term
```

Different grouping by means of enclosing parentheses has given sets with four elements (each a number), two elements (each of which is a pair), and one element (a pair of pairs) respectively.

1.7.2 Conversions for enumerated finite sets

The `pred_sets` library provides a collection of optimized conversions for computing the results of operations and predicates on finite sets specified by enumeration of their elements. All these conversions, the current implementations of which are somewhat experimental, are designed to work only for finite sets of the form "{t_1,...,t_n}". The sections that follow describe most of these conversions; the remainder are discussed in later sections of this manual.

1.7.2.1 Membership

The most basic conversion for finite sets is a decision procedure for membership called `IN_CONV`. In general, a way of deciding equality of elements is needed in order to determine whether a given value is an element of a particular finite set. The function

```
IN_CONV : conv -> conv
```

must therefore be supplied with a conversion that implements a decision procedure for equality of set elements. It is assumed that this conversion will map equations "e_1 = e_2" between elements of a base type `ty` to the theorem `|-` (e_1 = e_2) = T or to the theorem `|-` (e_1 = e_2) = F, as appropriate.

If `conv` is an equality conversion of the kind described above, then the function returned by `IN_CONV conv` is a conversion that decides membership in finite sets of values of the base type `ty`. In particular, a call:

IN_CONV conv "t IN {t_1,...,t_n}"

returns the theorem

|- t IN {t_1,...,t_n} = T

if the term t is alpha-equivalent to some term t_i or if the supplied conversion `conv` proves |- (t = t_i) = T for some i where $1 \leq i \leq n$. If, on the other hand `conv` proves the theorem |- (t = t_i) = F for all i where $1 \leq i \leq n$, then the result is the theorem

|- t IN {t_1,...,t_n} = F

In all other cases, the call to `IN_CONV` shown above will fail.

The following session shows how `IN_CONV` can be used in practice.

```
#IN_CONV num_EQ_CONV "1 IN {2,1,3}";;                          1
|- 1 IN {2,1,3} = T

#IN_CONV num_EQ_CONV "4 IN {2,1,3}";;
|- 4 IN {2,1,3} = F
```

The built-in conversion `num_EQ_CONV` is used here to decide equality of the natural numbers involved in the membership assertions being proved.

An example in which `IN_CONV` fails is the following:

```
#IN_CONV num_EQ_CONV "x IN {1,2,3}";;                          2
evaluation failed     IN_CONV

#num_EQ_CONV "x = 1";;
evaluation failed     num_EQ_CONV
```

Failure occurs in this case because the term `x` is a variable, and `num_EQ_CONV` therefore cannot determine if it is equal to any of the set elements `1`, `2` or `3`. Note, however, that the supplied conversion is not required to prove anything if the value being tested for membership happens to be syntactically identical to an element of the given set:

```
#IN_CONV NO_CONV "x IN {1,x,3}";;                              3
|- x IN {1,x,3} = T
```

In this case, the supplied conversion, namely `NO_CONV`, always fails; but the call to `IN_CONV` nonetheless succeeds and returns the appropriate result.

1.7.2.2 Union

The pred_sets library contains a conversion

```
UNION_CONV : conv -> conv
```

that can be used to compute the union of two finite sets. The first argument to UNION_CONV (i.e. the conversion argument) is expected to be an equality conversion of the same kind required as an argument by IN_CONV (see section 1.7.2.1). As will be seen below, this conversion is used by UNION_CONV to simplify the set that it computes as the result of taking the union of two finite sets.

Given an equality conversion conv, the function UNION_CONV returns a conversion that computes the union of a finite set "{t_1,...,t_n}" and another set s. The second set s in fact need not be finite. Ignoring, for the moment, the possible simplification done using the supplied conversion conv, a call:

UNION_CONV conv "{t_1,...,t_n} UNION s"

just returns the theorem

|- {t_1,...,t_n} UNION s = t_1 INSERT (... (t_n INSERT s)...)

That is, UNION_CONV computes the required union as a repeated insertion of values into the set s. When s is a finite set of the form "{u_1,...,u_m}", the resulting theorem will have the form shown below.

|- {t_1,...,t_n} UNION {u_1,...,u_m} = {t_1,...,t_n,u_1,...,u_m}

When computing theorems of this form (i.e. when the second set of the union is a finite set "{u_1,...,u_m}") the function UNION_CONV attempts to remove redundant elements in the resulting set using the supplied equality conversion conv. In particular, if conv is able to prove that some element t_i of "{t_1,...,t_n}" is equal to any element u_j of "{u_1,...,u_m}", that is if the conversion conv maps the term "t_i = u_j" to the theorem |- (t_i = u_j) = T, then the resulting theorem will be

|- {t_1,...t_i,...,t_n} UNION {u_1,...,u_j,...,u_m} = {t_1,...,t_n,u_1,...,u_j,...,u_m}

That is, the redundant term t_i will be removed from the initial sequence of elements in the resulting finite set. The function UNION_CONV also checks for and eliminates alpha-equivalent elements.

Some examples of UNION_CONV in use are shown in the following HOL session:

```
#UNION_CONV NO_CONV "{1,2,3} UNION {4,5,6}";;                          4
|- {1,2,3} UNION {4,5,6} = {1,2,3,4,5,6}

#UNION_CONV NO_CONV "{1,2,3} UNION {3,2,SUC 0}";;
|- {1,2,3} UNION {3,2,SUC 0} = {1,3,2,SUC 0}
```

The supplied equality conversion in these examples is `NO_CONV`, and only the elements of the first set `{1,2,3}` that are redundant by virtue of being alpha-equivalent to elements of the second set are eliminated from the resulting set. An example in which the equality conversion is actually used is:

```
#UNION_CONV num_EQ_CONV "{1,2,3} UNION {3,2,SUC 0}";;                  5
|- {1,2,3} UNION {3,2,SUC 0} = {3,2,SUC 0}
```

In this case, `num_EQ_CONV` is used to prove that `1` is equal to `SUC 0`, so that the resulting union is the set `"{3,2,SUC 0}"`, rather than `"{1,3,2,SUC 0}`".

1.7.2.3 Insertion

The conversion `INSERT_CONV` performs the following reduction on finite sets:

reduce "t INSERT {$t_1,\ldots,t_i,\ldots,t_n$}" to "{$t_1,\ldots,t_i,\ldots,t_n$}"

if a supplied equality conversion can prove |- (t = t_i) = T. Since the enumerated set notation "{$t_1,\ldots,t_n$}" is just a parser-supported abbreviation (see section 1.7.1), this is equivalent to reducing the set "{$t,t_1,\ldots,t_i,\ldots,t_n$}" to "{$t_1,\ldots,t_i,\ldots,t_n$}" when the terms t and t_i are provably equal.

More specifically, if for some t_i in {$t_1,\ldots,t_n$}, the terms t and t_i are alpha-equivalent, of if the conversion `conv` maps "t = t_i" to the theorem |- (t = t_i) = T, then the call:

INSERT_CONV conv "t INSERT {$t_1,\ldots,t_n$}";;

will return the theorem:

|- t INSERT {$t_1,\ldots,t_n$} = {$t_1,\ldots,t_n$}

Here is an example of `INSERT_CONV` in use:

```
#INSERT_CONV num_EQ_CONV "(SUC 2) INSERT {0,1,2,3}";;                  1
|- {SUC 2,0,1,2,3} = {0,1,2,3}
```

When applied repeatedly, `INSERT_CONV` can be used to reduce finite sets by eliminating as many redundant occurrences of elements as possible. An easy to program, but slow-running, way of doing this is to use `DEPTH_CONV`:

```
#DEPTH_CONV (INSERT_CONV num_EQ_CONV) "{1,3,x,SUC 1,SUC(SUC 1),2,1,3,x}";;  2
|- {1,3,x,SUC 1,SUC(SUC 1),2,1,3,x} = {2,1,3,x}
```

For a faster alternative to this method, see the reference entry for `INSERT_CONV` in chapter 2.

1.7.2.4 Deletion

The conversion DELETE_CONV reduces terms of the form "{t_1,...,t_n} DELETE t" by deleting all elements provably equal to t from the set {t_1,...,t_n}. Like IN_CONV and INSERT_CONV, the function DELETE_CONV takes a conversion for deciding equality of set elements as an argument. If conv is such a conversion, the call:

```
DELETE_CONV conv "{t1,...,tn} DELETE t";;
```

will return the theorem:

```
|- {t1,...,tn} DELETE t = {ti,...,tj}
```

where the resulting set {t_i,...,t_j} is the set of all values t_k in the original set {t_1,...,t_n} for which conv proves |- (t_k = t) = F, and where for all t_k in {t_1,...,t_n} but not in {t_i,...,t_j}, either t_k is alpha-equivalent to t or conv proves |- (t_k = t) = T. Note that the conversion conv must prove either equality or inequality for every element of the original set that is not simply alpha-equivalent to the deleted value.

The following session shows DELETE_CONV in use:

```
#DELETE_CONV num_EQ_CONV "{0,1,2,3} DELETE (SUC 1)";;                1
|- {0,1,2,3} DELETE (SUC 1) = {0,1,3}
```

1.8 Singleton sets

A *singleton* set is a set that contains precisely one element. In the pred_sets library, the property of being a singleton set is expressed by the definition:

```
SING_DEF   |- !s. SING s = (?x. s = {x})
```

The library contains several built-in theorems about singleton sets. These are sometimes expressed in terms of the predicate SING, as for example in the theorem

```
SING   |- !x. SING{x}
```

But properties of singleton sets are more usually formulated as theorems about sets of the form '{x}'. For example, the built-in theorems about singleton sets include:

```
NOT_SING_EMPTY  |- !x. ~({x} = {})
IN_SING         |- !x y. x IN {y} = (x = y)
EQUAL_SING      |- !x y. ({x} = {y}) = (x = y)
```

A general convention is that theorems about singleton sets are given names that contain the element 'SING', regardless of whether or not they actually contain the predicate SING.

1.9 The CHOICE and REST functions

The pred_sets library contains the definition of a functions CHOICE which can be used to select an arbitrary element from a non-empty set. The function CHOICE is defined formally by the following constant specification:

```
CHOICE_DEF   |- !s. ~(s = {}) ==> (CHOICE s) IN s
```

This theorem alone is the defining property for the constant CHOICE, which is therefore an only partially specified function from sets to values. Note, in particular, that there is no information given by this definition about the result of applying CHOICE to an empty set.

The library also contains a function REST, which is defined in terms of the CHOICE function as follows

```
REST_DEF   |- !s. REST s = s DELETE (CHOICE s)
```

For any non-empty set s, the set REST s comprises all those elements of s except the value selected from s by CHOICE.

The library contains various built-in theorems about the functions CHOICE and REST; for a full list of these theorems, see chapter 3.

1.10 Image of a function on a set

The *image* of a function f:*->** on a set s:*->bool is the set of values f(x) for all x in s. In the pred_sets library, the image of a function on a set is defined in terms of the obvious set abstraction:

```
IMAGE_DEF   |- !f s. IMAGE f s = {f x | x IN s}
```

Using SET_SPEC_CONV, is is trivial to prove from this definition the following membership condition for sets constructed using IMAGE:

```
IN_IMAGE   |- !y s f. y IN (IMAGE f s) = (?x. (y = f x) /\ x IN s)
```

The pred_sets library contains various theorems about IMAGE in addition to this membership theorem. These include, for example, theorems about the image of a function on sets constructed by the operations of union and intersection. For a full list of theorems about IMAGE, see chapter 3.

1.10.1 Theorem-proving support

The pred_sets library contains a conversion for computing the image of a function f on a finite set {$t_1,\ldots,t_n$}. The function

```
IMAGE_CONV : conv -> conv -> conv
```

is parameterized by two conversions. The first conversion is expected to compute the result of applying the function f to each element t_1, ..., t_n. The second parameter is an equality conversion which is used to simplify the resulting image set by removing redundant occurrences of its elements.

The following session shows a simple example of the use of IMAGE_CONV on terms of the form "IMAGE (\x.x+2) {$t_1,\ldots,t_n$}". We first define a conversion that evaluates the result of applying the function (\x.x+2) to a term t.

```
#let AP_CONV = BETA_CONV THENC (TRY_CONV ADD_CONV);;                    1
AP_CONV = - : conv

#AP_CONV "(\n.n+2) 7";;
|- (\n. n + 2)7 = 9
```

This conversion, together with the function IMAGE_CONV, gives a conversion for computing the image of (\x.x+2) on a finite set of numerical values.

```
#IMAGE_CONV AP_CONV NO_CONV "IMAGE (\x.x+2) {1,2,3,4}";;                2
|- IMAGE(\x. x + 2){1,2,3,4} = {3,4,5,6}

#IMAGE_CONV AP_CONV NO_CONV "IMAGE (\x.x+2) {n,1,n}";;
|- IMAGE(\x. x + 2){n,1,n} = {3,n + 2}
```

In this case, the second parameter supplied to IMAGE_CONV is the conversion NO_CONV. This means that no reduction of the resulting image set is done, beyond the elimination of elements that are provably redundant by virtue of being alpha-equivalent to some other element (as in the second example above).

The following session illustrates the use of the second parameter to IMAGE_CONV.

```
#IMAGE_CONV BETA_CONV NO_CONV "IMAGE (\x. SUC x) {1,SUC 0,2,0}";;       3
|- IMAGE(\x. SUC x){1,SUC 0,2,0} = {SUC 1,SUC(SUC 0),SUC 2,SUC 0}

#IMAGE_CONV BETA_CONV num_EQ_CONV "IMAGE (\x. SUC x) {1,SUC 0,2,0}";;
|- IMAGE(\x. SUC x){1,SUC 0,2,0} = {SUC(SUC 0),SUC 2,SUC 0}
```

In the first evaluation, just applying BETA_CONV to the application of (\x. SUC x) to each element has resulted in an image set containing both SUC 1 and SUC(SUC 0). In the second example, num_EQ_CONV is used to prove these values equal, and therefore to simplify the resulting set by eliminating one of them from it. For more detail about IMAGE_CONV, see the reference entry for this conversion in chapter 2.

1.11 Mappings between sets

The pred_sets library contains a few basic definitions and theorems having to do with mappings between sets. A function f:*->** is an *injective* (one-to-one) mapping from a set s:*->bool to a set t:**->bool if it takes distinct elements of the set s to distinct element of the set t:

```
INJ_DEF =
|- !f s t.
    INJ f s t =
    (!x. x IN s ==> (f x) IN t) /\
    (!x y. x IN s /\ y IN s ==> (f x = f y) ==> (x = y))
```

Likewise, a function f:*->** is a *surjective* (onto) mapping from s to t if for every element x of t there is some element y of s for which f y = x:

```
SURJ_DEF =
|- !f s t.
    SURJ f s t =
    (!x. x IN s ==> (f x) IN t) /\
    (!x. x IN t ==> (?y. y IN s /\ (f y = x)))
```

Finally, a function f:*->** is a *bijection* from s to t if it is both injective and surjective:

```
BIJ_DEF = |- !f s t. BIJ f s t = INJ f s t /\ SURJ f s t
```

There are a few pre-proved theorems about the predicates INJ, SURJ, and BIJ available in the library; see chapter 3 for a full list of these theorems.

The library also contains constant specifications for two functions LINV and RINV, which yield left and right inverses to injective and surjective mappings respectively. These functions are defined by:

```
LINV_DEF = |- !f s t. INJ f s t ==> (!x. x IN s ==> (LINV f s(f x) = x))
RINV_DEF = |- !f s t. SURJ f s t ==> (!x. x IN t ==> (f(RINV f s x) = x))
```

There are, at present, no additional built-in theorems about these two functions. Furthermore, the definitions of LINV and RINV shown above should be regarded as only provisional; they may be changed in future versions.

1.12 Finite and infinite sets

The pred_sets library includes the definition of a predicate called FINITE, which is true of finite sets and false of infinite ones. The definition of this constant is shown below.

```
FINITE_DEF
  |- !s.
      FINITE s =
      (!P. P{} /\ (!s'. P s' ==> (!e. P(e INSERT s'))) ==> P s)
```

That is, a set `s` is finite precisely when it is in the smallest class of sets that contains the empty set and is closed under the `INSERT` operation. This inductive definition makes `FINITE` true of just those sets that can be constructed from the empty set by a finite sequence of applications of the `INSERT` operation.

The `pred_sets` library contains various built-in theorems that follow from the definition of `FINITE` given above. Among these are the two fundamental theorems shown below:

```
FINITE_EMPTY   |- FINITE{}
FINITE_INSERT  |- !x s. FINITE(x INSERT s) = FINITE s
```

These state that the empty set is indeed finite and insertion constructs finite sets only from other finite sets. See chapter 3 for other built-in theorems about finite sets.

The above definition of `FINITE` formalizes the notion of a finite set in logic, and it therefore also determines the form of definition for the complementary notion of an infinite set. In the `pred_sets` library, the predicate `INFINITE` is defined as follows:

```
INFINITE_DEF   |- !s. INFINITE s = ~FINITE s
```

There are a few consequences of this definition stored in the `pred_sets` library. The following theorem, for example, states that the image of an injective function on an infinite set is infinite:

```
IMAGE_11_INFINITE
   |- !f. (!x y. (f x = f y) ==> (x = y)) ==>
          (!s. INFINITE s ==> INFINITE(IMAGE f s))
```

Other built-in theorems about `INFINITE` can be found in chapter 3.

1.12.1 Theorem-proving support

There are two ML functions in the `pred_sets` library for reasoning about propositions that involve the finiteness predicate `FINITE`. The first of these is a conversion `FINITE_CONV` which automatically proves that sets of the form "{t_1,...,t_n}" are finite. Evaluating

FINITE_CONV "FINITE {t_1,...,t_n}";;

yields the theorem `|- FINITE {`t_1`,...,`t_n`} = T`.

The second ML function for reasoning about the predicate `FINITE` is an induction tactic called `SET_INDUCT_TAC`. When applied to a goal of the form `"!`s`. FINITE `s` ==> `P`"`, this tactic reduces it to proving that the property of sets expressed by `\`s`.`P holds of the empty set and is preserved by the insertion of an element into an arbitrary finite set. Since every finite set can be built up from the empty set by repeated insertion of values, these subgoals imply that this property holds of all finite sets.

The following session illustrates the use of the tactic `SET_INDUCT_TAC` for proving that the intersection of an arbitrary set `t` with a finite set `s` is finite. We first set up an appropriate goal:

```
#g "!s:*->bool. FINITE s ==> !t. FINITE(s INTER t)";;                1
"!s. FINITE s ==> (!t. FINITE(s INTER t))"

() : void
```

Expanding with `SET_INDUCT_TAC` yields:

```
#expand SET_INDUCT_TAC;;                                              2
OK..
2 subgoals
"!t. FINITE((e INSERT s) INTER t)"
    [ "FINITE s" ]
    [ "!t. FINITE(s INTER t)" ]
    [ "~e IN s" ]

"!t. FINITE({} INTER t)"

() : void
```

The resulting subgoals are easy to prove, given the two basic theorems `FINITE_EMPTY` and `FINITE_INSERT` shown in the previous section. Note that it may be assumed in the step case that the value `e` being inserted into the set `s` is not already an element of `s`.

1.13 Cardinality of finite sets

The *cardinality* of a finite set is the number of elements it contains. In the `pred_sets` library, this is formalized by a constant `CARD` defined by means of the following constant specification:

```
CARD_DEF
  |- (CARD{} = 0) /\
     (!s.
       FINITE s ==>
       (!x. CARD(x INSERT s) = (x IN s => CARD s | SUC(CARD s))))
```

This theorem is the sole defining property of `CARD`. Because the equation in the second clause holds only under the assumption that `s` is finite, this form of definition allows nothing significant to be deduced about the cardinality '`CARD s`' of an *infinite* set `s`.

The built-in theorems about cardinality are all restricted to finite sets only, either implicitly as in the theorem:

```
CARD_SING  |- !x. CARD{x} = 1
```

or explicitly, as in:

```
FINITE_ISO_NUM
  |- !s:*->bool.
       FINITE s ==>
       (?f:num->*.
         (!n m.
           n < (CARD s) /\ m < (CARD s) ==> (f n = f m) ==> (n = m)) /\
         (s = {f n | n < (CARD s)}))
```

This second theorem states that the elements of a finite set can always be put into a one-to-one correspondence with the natural numbers less than the set's cardinality—i.e. the elements of a finite set `s` can be numbered `0`, `1`, ..., `(CARD s)-1`. Other theorems involving the cardinality function `CARD` can be found in chapter 3.

1.14 Using the library

The `pred_sets` library is loaded into a user's HOL session using the function `load_library` (see the HOL manual for a general description of library loading). The first action in the load sequence is to update the internal HOL search paths. A pathname to the library is added to the search path so that theorems may be autoloaded from the library theory `pred_sets`; and the HOL help search path is updated with a pathname to online help files for the ML functions in the library.

After the search paths are updated, the actions taken by the load sequence for `pred_sets` depend on the current state of the HOL session. If the system is in draft mode, the library theory `pred_sets` is added as a new parent to the current theory. If the system is not in draft mode, but the current theory is an ancestor of the `pred_sets` theory in the library (e.g. the user is in a fresh HOL session) then `pred_sets` is made the current theory. In both cases, the ML functions provided by the library are loaded into HOL and all the theorems in the library (including definitions) are set up to be autoloaded on demand. The parser and pretty-printer for the notation described above in sections 1.2.1 and 1.7.1 are then activated, and the ML functions provided by the library for reasoning about sets are loaded. The `pred_sets` library is then fully loaded into the user's HOL session.

1.14.1 Example session

The following session shows how the `pred_sets` library may be loaded using `load_library`. Suppose, beginning in a fresh HOL session, the user wishes to create a theory `foo` whose parents include the theory `pred_sets` in the library. This may be done as follows:

```
#new_theory `foo`;;                                                        1
() : void

#load_library `pred_sets`;;
  ⋮
Library pred_sets loaded.
() : void
```

Loading the library while drafting the theory `foo` makes the library theory `pred_sets` into a parent of `foo`. The same effect could have been achieved (in a fresh session) by first loading the library and then creating `foo`:

```
#load_library `pred_sets`;;                                                1
  ⋮
Library pred_sets loaded.
() : void

#new_theory `foo`;;
() : void
```

The theory `pred_sets` is first made the current theory of the new session. It then automatically becomes a parent of `foo` when this theory is created by `new_theory`.

Now, suppose that `foo` has been created as shown above, and the user does some work in this theory, quits HOL, and in a later session wishes to load the theory `foo`. This must be done by *first* loading the `pred_sets` library and *then* loading the theory `foo`.

```
#load_library `pred_sets`;;                                                1
  ⋮
Library pred_sets loaded.
() : void

#load_theory `foo`;;
Theory foo loaded
() : void
```

This sequence of actions ensures that the system can find the parent theory `pred_sets` when it comes to load `foo`, since loading the library updates the search path.

1.14.2 The load_pred_sets function

The pred_sets library may in many cases simply be loaded into the system as illustrated by the examples given above. There are, however, certain situations in which the library cannot be fully loaded at the time when the load_library is used. This occurs when the system is not in draft mode and the current theory is not an ancestor of the theory pred_sets. In this case, loading the library can (and will) update the search paths. But the theory pred_sets can neither be made into a parent of the current theory nor be made the current theory. This means that autoloading from the library can not at this stage be activated; and the ML code in the library can not be loaded into HOL, since it requires access to some of the theorems in the library.

In the situation described above—when the system is not in draft mode and the current theory is not an ancestor of the theory pred_sets—the library load sequence defines an ML function called load_pred_sets in the current HOL session. If at a future point in the session the pred_sets theory (now accessible via the search path) becomes an ancestor of the current theory, this function can then be used to complete loading of the library. Evaluating load_pred_sets() in such a context loads the ML functions of the pred_sets library into HOL and activates autoloading from its theory files. It also activates the parser and pretty-printer support for set abstractions and finite sets. The function load_pred_sets fails if the theory pred_sets is not an ancestor of the current HOL theory.

Note that the function load_pred_sets becomes available upon loading the pred_sets library only if the library theory pred_sets at the point of loading the library can neither be made into a new parent (i.e. the system is not in draft mode) nor be made the current theory.

Chapter 2

ML Functions in the Library

This chapter provides documentation on all the ML functions that are made available in HOL when the `pred_sets` library is loaded. This documentation is also available online via the `help` facility.

DELETE_CONV

```
DELETE_CONV : conv -> conv
```

Synopsis

Reduce `{x1,...,xn} DELETE x` by deleting `x` from `{x1,...,xn}`.

Description

The function `DELETE_CONV` is a parameterized conversion for reducing finite sets of the form `"{t1,...,tn} DELETE t"`, where the term `t` and the elements of `{t1,...,tn}` are of some base type `ty`. The first argument to `DELETE_CONV` is expected to be a conversion that decides equality between values of the base type `ty`. Given an equation `"e1 = e2"`, where `e1` and `e2` are terms of type `ty`, this conversion should return the theorem `|- (e1 = e2) = T` or the theorem `|- (e1 = e2) = F`, as appropriate.

Given such a conversion `conv`, the function `DELETE_CONV` returns a conversion that maps a term of the form `"{t1,...,tn} DELETE t"` to the theorem

```
|- {t1,...,tn} DELETE t = {ti,...,tj}
```

where `{ti,...,tj}` is the subset of `{t1,...,tn}` for which the supplied equality conversion `conv` proves

```
|- (ti = t) = F, ..., |- (tj = t) = F
```

and for all the elements `tk` in `{t1,...,tn}` but not in `{ti,...,tj}`, either `conv` proves `|- (tk = t) = T` or `tk` is alpha-equivalent to `t`. That is, the reduced set `{ti,...,tj}` comprises all those elements of the original set that are provably not equal to the deleted element `t`.

Example

In the following example, the conversion num_EQ_CONV is supplied as a parameter and used to test equality of the deleted value 2 with the elements of the set.

```
#DELETE_CONV num_EQ_CONV "{2,1,SUC 1,3} DELETE 2";;
|- {2,1,SUC 1,3} DELETE 2 = {1,3}
```

Failure

DELETE_CONV conv fails if applied to a term not of the form "{t1,...,tn} DELETE t". A call DELETE_CONV conv "{t1,...,tn} DELETE t" fails unless for each element ti of the set {t1,...,tn}, the term t is either alpha-equivalent to ti or conv "ti = t" returns |- (ti = t) = T or |- (ti = t) = F.

See also

INSERT_CONV.

FINITE_CONV

```
FINITE_CONV : conv
```

Synopsis

Proves finiteness of sets of the form "{x1,...,xn}".

Description

The conversion FINITE_CONV expects its term argument to be an assertion of the form "FINITE {x1,...,xn}". Given such a term, the conversion returns the theorem

```
|- FINITE {x1,...,xn} = T
```

Example

```
#FINITE_CONV "FINITE {1,2,3}";;
|- FINITE{1,2,3} = T

#FINITE_CONV "FINITE ({}:num->bool)";;
|- FINITE{} = T
```

Failure

Fails if applied to a term not of the form "FINITE {x1,...,xn}".

IMAGE_CONV

```
IMAGE_CONV : conv -> conv -> conv
```

Synopsis

Compute the image of a function on a finite set.

Description

The function `IMAGE_CONV` is a parameterized conversion for computing the image of a function `f:ty1->ty2` on a finite set `"{t1,...,tn}"` of type `ty1->bool`. The first argument to `IMAGE_CONV` is expected to be a conversion that computes the result of applying the function `f` to each element of this set. When applied to a term `"f ti"`, this conversion should return a theorem of the form `|- (f ti) = ri`, where `ri` is the result of applying the function `f` to the element `ti`. This conversion is used by `IMAGE_CONV` to compute a theorem of the form

```
|- IMAGE f {t1,...,tn} = {r1,...,rn}
```

The second argument to `IMAGE_CONV` is used (optionally) to simplify the resulting image set `{r1,...,rn}` by removing redundant occurrences of values. This conversion expected to decide equality of values of the result type `ty2`; given an equation `"e1 = e2"`, where `e1` and `e2` are terms of type `ty2`, the conversion should return either `|- (e1 = e2) = T` or `|- (e1 = e2) = F`, as appropriate.

Given appropriate conversions `conv1` and `conv2`, the function `IMAGE_CONV` returns a conversion that maps a term of the form `"IMAGE f {t1,...,tn}"` to the theorem

```
|- IMAGE f {t1,...,tn} = {rj,...,rk}
```

where `conv1` proves a theorem of the form `|- (f ti) = ri` for each element `ti` of the set `{t1,...,tn}`, and where the set `{rj,...,rk}` is the smallest subset of `{r1,...,rn}` such no two elements are alpha-equivalent and `conv2` does not map `"rl = rm"` to the theorem `|- (rl = rm) = T` for any pair of values `rl` and `rm` in `{rj,...,rk}`. That is, `{rj,...,rk}` is the set obtained by removing multiple occurrences of values from the set `{r1,...,rn}`, where the equality conversion `conv2` (or alpha-equivalence) is used to determine which pairs of terms in `{r1,...,rn}` are equal.

Example

The following is a very simple example in which `REFL` is used to construct the result of applying the function `f` to each element of the set `{1,2,1,4}`, and `NO_CONV` is the supplied 'equality conversion'.

```
#IMAGE_CONV REFL NO_CONV "IMAGE (f:num->num) {1,2,1,4}";;
|- IMAGE f{1,2,1,4} = {f 2,f 1,f 4}
```

The result contains only one occurrence of 'f 1', even though NO_CONV always fails, since IMAGE_CONV simplifies the resulting set by removing elements that are redundant up to alpha-equivalence.

For the next example, we construct a conversion that maps SUC n for any numeral n to the numeral standing for the successor of n.

```
#let SUC_CONV tm =
     let n = int_of_string(fst(dest_const(rand tm))) in
     let sucn = mk_const(string_of_int(n+1), ":num") in
         SYM (num_CONV sucn);;
SUC_CONV = - : conv
```

The result is a conversion that inverts num_CONV:

```
#num_CONV "4";;
|- 4 = SUC 3

#SUC_CONV "SUC 3";;
|- SUC 3 = 4
```

The conversion SUC_CONV can then be used to compute the image of the successor function on a finite set:

```
#IMAGE_CONV SUC_CONV NO_CONV "IMAGE SUC {1,2,1,4}";;
|- IMAGE SUC{1,2,1,4} = {3,2,5}
```

Note that 2 (= SUC 1) appears only once in the resulting set.

Fianlly, here is an example of using IMAGE_CONV to compute the image of a paired addition function on a set of pairs of numbers:

```
#IMAGE_CONV (PAIRED_BETA_CONV THENC ADD_CONV) num_EQ_CONV
            "IMAGE (\(n,m).n+m) {(1,2), (3,4), (0,3), (1,3)}";;
|- IMAGE(\(n,m). n + m){(1,2),(3,4),(0,3),(1,3)} = {7,3,4}
```

Failure

IMAGE_CONV conv1 conv2 fails if applied to a term not of the form "IMAGE f {t1,...,tn}". An application of IMAGE_CONV conv1 conv2 to a term "IMAGE f {t1,...,tn}" fails unless for all ti in the set {t1,...,tn}, evaluating conv1 "f ti" returns |- (f ti) = ri for some ri.

INSERT_CONV

```
INSERT_CONV : conv -> conv
```

Synopsis
Reduce `x INSERT {x1,...,x,...,xn}` to `{x1,...,x,...,xn}`.

Description
The function `INSERT_CONV` is a parameterized conversion for reducing finite sets of the form `"t INSERT {t1,...,tn}"`, where `{t1,...,tn}` is a set of type `ty->bool` and `t` is equal to some element `ti` of this set. The first argument to `INSERT_CONV` is expected to be a conversion that decides equality between values of the base type `ty`. Given an equation `"e1 = e2"`, where `e1` and `e2` are terms of type `ty`, this conversion should return the theorem `|- (e1 = e2) = T` or the theorem `|- (e1 = e2) = F`, as appropriate.

Given such a conversion, the function `INSERT_CONV` returns a conversion that maps a term of the form `"t INSERT {t1,...,tn}"` to the theorem

```
|- t INSERT {t1,...,tn} = {t1,...,tn}
```

if `t` is alpha-equivalent to any `ti` in the set `{t1,...,tn}`, or if the supplied conversion proves `|- (t = ti) = T` for any `ti`.

Example
In the following example, the conversion `num_EQ_CONV` is supplied as a parameter and used to test equality of the inserted value `2` with the remaining elements of the set.

```
#INSERT_CONV num_EQ_CONV "2 INSERT {1,SUC 1,3}";;
|- {2,1,SUC 1,3} = {1,SUC 1,3}
```

In this example, the supplied conversion `num_EQ_CONV` is able to prove that `2` is equal to `SUC 1` and the set is therefore reduced. Note that `"2 INSERT {1,SUC 1,3}"` is just `"{2,1,SUC 1,3}"`.

A call to `INSERT_CONV` fails when the value being inserted is provably not equal to any of the remaining elements:

```
#INSERT_CONV num_EQ_CONV "1 INSERT {2,3}";;
evaluation failed     INSERT_CONV
```

But this failure can, if desired, be caught using `TRY_CONV`.

The behaviour of the supplied conversion is irrelevant when the inserted value is alpha-equivalent to one of the remaining elements:

```
#INSERT_CONV NO_CONV "(y:*) INSERT {x,y,z}";;
|- {y,x,y,z} = {x,y,z}
```

The conversion `NO_CONV` always fails, but `INSERT_CONV` is nontheless able in this case to prove the required result.

Note that `DEPTH_CONV(INSERT_CONV conv)` can be used to remove duplicate elements from a finite set, but the following conversion is faster:

```
#letrec REDUCE_CONV conv tm =
   (SUB_CONV (REDUCE_CONV conv) THENC (TRY_CONV (INSERT_CONV conv))) tm;;
REDUCE_CONV = - : (conv -> conv)

#REDUCE_CONV num_EQ_CONV "{1,2,1,3,2,4,3,5,6}";;
|- {1,2,1,3,2,4,3,5,6} = {1,2,4,3,5,6}
```

Failure

`INSERT_CONV conv` fails if applied to a term not of the form `"t INSERT {t1,...,tn}"`. A call `INSERT_CONV conv "t INSERT {t1,...,tn}"` fails unless `t` is alpha-equivalent to some `ti`, or `conv "t = ti"` returns `|- (t = ti) = T` for some `ti`.

See also

`DELETE_CONV`.

IN_CONV

```
IN_CONV : conv -> conv
```

Synopsis

Decision procedure for membership in finite sets.

Description

The function `IN_CONV` is a parameterized conversion for proving or disproving membership assertions of the general form:

```
"t IN {t1,...,tn}"
```

where `{t1,...,tn}` is a set of type `ty->bool` and `t` is a value of the base type `ty`. The first argument to `IN_CONV` is expected to be a conversion that decides equality between values of the base type `ty`. Given an equation `"e1 = e2"`, where `e1` and `e2` are terms of type `ty`, this conversion should return the theorem `|- (e1 = e2) = T` or the theorem `|- (e1 = e2) = F`, as appropriate.

Given such a conversion, the function `IN_CONV` returns a conversion that maps a term of the form `"t IN {t1,...,tn}"` to the theorem

```
|- t IN {t1,...,tn} = T
```

if `t` is alpha-equivalent to any `ti`, or if the supplied conversion proves `|- (t = ti) = T` for any `ti`. If the supplied conversion proves `|- (t = ti) = F` for every `ti`, then the result is the theorem

```
|- t IN {t1,...,tn} = F
```

In all other cases, `IN_CONV` will fail.

Example

In the following example, the conversion `num_EQ_CONV` is supplied as a parameter and used to test equality of the candidate element `1` with the actual elements of the given set.

```
#IN_CONV num_EQ_CONV "2 IN {0,SUC 1,3}";;
|- 2 IN {0,SUC 1,3} = T
```

The result is `T` because `num_EQ_CONV` is able to prove that `2` is equal to `SUC 1`. An example of a negative result is:

```
#IN_CONV num_EQ_CONV "1 IN {0,2,3}";;
|- 1 IN {0,2,3} = F
```

Finally the behaviour of the supplied conversion is irrelevant when the value to be tested for membership is alpha-equivalent to an actual element:

```
#IN_CONV NO_CONV "1 IN {3,2,1}";;
|- 1 IN {3,2,1} = T
```

The conversion `NO_CONV` always fails, but `IN_CONV` is nontheless able in this case to prove the required result.

Failure

`IN_CONV conv` fails if applied to a term that is not of the form `"t IN {t1,...,tn}"`. A call `IN_CONV conv "t IN {t1,...,tn}"` fails unless the term `t` is alpha-equivalent to some `ti`, or `conv "t = ti"` returns `|- (t = ti) = T` for some `ti`, or `conv "t = ti"` returns `|- (t = ti) = F` for every `ti`.

SET_INDUCT_TAC

```
SET_INDUCT_TAC : tactic
```

Synopsis

Tactic for induction on finite sets.

Description

SET_INDUCT_TAC is an induction tacic for proving properties of finite sets. When applied to a goal of the form

```
!s. FINITE s ==> P[s]
```

SET_INDUCT_TAC reduces this goal to proving that the property \s.P[s] holds of the empty set and is preserved by insertion of an element into an arbitrary finite set. Since every finite set can be built up from the empty set "{}" by repeated insertion of values, these subgoals imply that the property \s.P[s] holds of all finite sets.

The tactic specification of SET_INDUCT_TAC is:

```
              A ?- !s. FINITE s ==> P
=========================================================  SET_INDUCT_TAC
  A |- P[{}/s]
  A u {FINITE s', P[s'/s], ~e IN s'} ?- P[e INSERT s'/s]
```

where e is a variable chosen so as not to appear free in the assumptions A, and s' is a primed variant of s that does not appear free in A (usually, s' is just s).

Failure

SET_INDUCT_TAC (A,g) fails unless g has the form !s. FINITE s ==> P, where the variable s has type ty->bool for some type ty.

SET_SPEC_CONV

```
SET_SPEC_CONV : conv
```

Synopsis

Axiom-scheme of specification for set abstractions.

Description

The conversion `SET_SPEC_CONV` expects its term argument to be an assertion of the form `"t IN {E | P}"`. Given such a term, the conversion returns a theorem that defines the condition under which this membership assertion holds. When `E` is just a variable `v`, the conversion returns:

```
|- t IN {v | P} = P[t/v]
```

and when `E` is not a variable but some other expression, the theorem returned is:

```
|- t IN {E | P} = ?x1...xn. (t = E) /\ P
```

where `x1`, ..., `xn` are the variables that occur free both in the expression `E` and in the proposition `P`.

Example

```
#SET_SPEC_CONV "12 IN {n | n > N}";;
|- 12 IN {n | n > N} = 12 > N

#SET_SPEC_CONV "p IN {(n,m) | n < m}";;
|- p IN {(n,m) | n < m} = (?n m. (p = n,m) /\ n < m)
```

Failure

Fails if applied to a term that is not of the form `"t IN {E | P}"`.

UNION_CONV

```
UNION_CONV : conv -> conv
```

Synopsis

Reduce `{t1,...,tn} UNION s` to `t1 INSERT (... (tn INSERT s))`.

Description

The function `UNION_CONV` is a parameterized conversion for reducing sets of the form `"{t1,...,tn} UNION s"`, where `{t1,...,tn}` and `s` are sets of type `ty->bool`. The first argument to `UNION_CONV` is expected to be a conversion that decides equality between values of the base type `ty`. Given an equation `"e1 = e2"`, where `e1` and `e2` are terms of type `ty`, this conversion should return the theorem `|- (e1 = e2) = T` or the theorem `|- (e1 = e2) = F`, as appropriate.

Given such a conversion, the function UNION_CONV returns a conversion that maps a term of the form "{t1,...,tn} UNION s" to the theorem

```
|- t UNION {t1,...,tn} = ti INSERT ... (tj INSERT s)
```

where {ti,...,tj} is the set of all terms t that occur as elements of {t1,...,tn} for which the conversion IN_CONV conv fails to prove that |- (t IN s) = T (that is, either by proving |- (t IN s) = F instead, or by failing outright).

Example

In the following example, num_EQ_CONV is supplied as a parameter to UNION_CONV and used to test for membership of each element of the first finite set {1,2,3} of the union in the second finite set {SUC 0,3,4}.

```
#UNION_CONV num_EQ_CONV "{1,2,3} UNION {SUC 0,3,4}";;
|- {1,2,3} UNION {SUC 0,3,4} = {2,SUC 0,3,4}
```

The result is {2,SUC 0,3,4}, rather than {1,2,SUC 0,3,4}, because UNION_CONV is able by means of a call to

```
IN_CONV num_EQ_CONV "1 IN {SUC 0,3,4}"
```

to prove that 1 is already an element of the set {SUC 0,3,4}.

The conversion supplied to UNION_CONV need not actually prove equality of elements, if simplification of the resulting set is not desired. For example:

```
#UNION_CONV NO_CONV "{1,2,3} UNION {SUC 0,3,4}";;
|- {1,2,3} UNION {SUC 0,3,4} = {1,2,SUC 0,3,4}
```

In this case, the resulting set is just left unsimplified. Moreover, the second set argument to UNION need not be a finite set:

```
#UNION_CONV NO_CONV "{1,2,3} UNION s";;
|- {1,2,3} UNION s = 1 INSERT (2 INSERT (3 INSERT s))
```

And, of course, in this case the conversion argument to UNION_CONV is irrelevant.

Failure

UNION_CONV conv fails if applied to a term not of the form "{t1,...,tn} UNION s".

See also

IN_CONV.

Chapter 3

Pre-proved Theorems

The sections that follow list all theorems in the `pred_sets` library, including definitions. The theorems are grouped into sections according to subject matter. Some theorems could be classified under more than one subject, but each theorem is listed in only one section. The reader may therefore have to consult more than one section when searching for any particular theorem.

When the `pred_sets` library is loaded, all the theorems listed in this chapter (including definitions) are set up to autoload when their names are mentioned in ML.

3.1 Membership, equality, and set specifications

3.2 The empty and universal sets

3.3 Set inclusion

3.4 Intersection and union

3.5 Set difference

3.6 Disjoint sets

3.7 Insertion and deletion of an element

3.8 The `CHOICE` and `REST` functions

3.9 Image of a function on a set

3.10 Mappings between sets

3.11 Singleton sets

3.12 Finite and infinite sets

3.13 Cardinality of sets

References

[1] T. F. Melham, *The HOL sets library*, University of Cambridge Computer Laboratory, October 1991.

[2] University of Cambridge Computer Laboratory, *The HOL System: DESCRIPTION*, revised edition, 1991.

Index