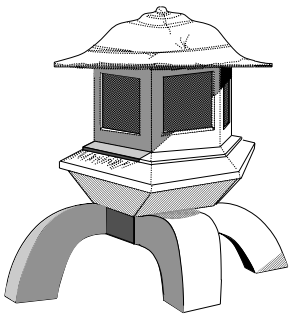
[For HOL-Omega Kananaskis-8] August 7, 2012

The HOL-Omega System
TUTORIAL TEASER



Prologue

This volume contains a short sample of material from the upcoming tutorial on the HOL-Omega system. The tutorial will be one of four documents making up the documentation for HOL-Omega:

(i) *LOGIC*: a formal description of the higher order logic implemented by the HOL-Omega system.

(ii) *TUTORIAL*: a tutorial introduction to HOL-Omega, with case studies.

(iii) *DESCRIPTION*: a detailed user's guide for the HOL-Omega system;

(iv) *REFERENCE*: the reference manual for HOL-Omega.

This document provides a brief and light set of examples of using HOL-Omega, as an introduction, giving a taste of how the system might be used. Like an appetizer to a main meal, it provides just a hint of the sustenance to come.

Getting started

Chapter 1 explains how to get and install HOL-Omega. Then the new, additional concepts and features of the HOL-Omega logic (higher order logic extended with System *F*, kinds, and ranks) are casually demonstrated, in chapter 2.

Chapter 3 briefly discusses some of the examples distributed with HOL-Omega in the `examples/HolOmega` directory.

Acknowledgements

The bulk of HOL-Omega is based on code written by—in alphabetical order—Hasan Amjad, Richard Boulton, Anthony Fox, Mike Gordon, Elsa Gunter, John Harrison, Peter Homeier, Gérard Huet (and others at INRIA), Joe Hurd, Ramana Kumar, Ken Friis Larsen, Tom Melham, Robin Milner, Lockwood Morris, Magnus Myreen, Malcolm Newey, Michael Norrish, Larry Paulson, Konrad Slind, Don Syme, Thomas Türk, Chris Wadsworth, and Tjark Weber. Many others have supplied parts of the system, bug fixes, etc.

Current edition

The current edition of all four volumes (*LOGIC*, *TUTORIAL*, *DESCRIPTION* and *REFERENCE*) has been prepared by Michael Norrish and Konrad Slind, and extended for HOL-Omega by Peter Homeier. Further contributions to these volumes came from: Hasan Amjad, who developed a model checking library and wrote sections describing its use; Jens Brandt, who developed and documented a library for the rational numbers; Anthony Fox, who formalized and documented new word theories and the associated libraries; Mike Gordon, who documented the libraries for BDDs and SAT; Peter Homeier, who implemented and documented the quotient library; Joe Hurd, who added material on first order proof search; and Tjark Weber, who wrote libraries for Satisfiability Modulo Theories (SMT) and Quantified Boolean Formulae (QBF).

The material in the third edition constitutes a thorough re-working and extension of previous editions. The semantics by Andy Pitts (in *LOGIC*) had remained essentially unaltered until HOL-Omega, reflecting the fact that, although the HOL system has undergone continual development and improvement, the HOL logic had been unchanged since the first edition (1988) until the introduction of HOL-Omega in 2009.

Second edition

The second edition of *REFERENCE* was a joint effort by the Cambridge HOL group.

First edition

The three volumes *TUTORIAL*, *DESCRIPTION* and *REFERENCE* were produced at the Cambridge Research Center of SRI International with the support of DSTO Australia.

The HOL documentation project was managed by Mike Gordon, who also wrote parts of *DESCRIPTION* and *TUTORIAL* using material based on an early paper describing the HOL system[1] and *The ML Handbook*[2]. Other contributers to *DESCRIPTION* incude Avra Cohn, who contributed material on theorems, rules, conversions and tactics, and also composed the index (which was typeset by Juanito Camilleri); Tom Melham, who wrote the sections describing type definitions, the concrete type package and the 'resolution' tactics; and Andy Pitts, who devised the set-theoretic semantics of the HOL logic and wrote the material describing it.

The original document design used LaTeX macros supplied by Elsa Gunter, Tom Melham and Larry Paulson. The typesetting of all three volumes was managed by Tom Melham. The cover design is by Arnold Smith, who used a photograph of a 'snow watching lantern' taken by Avra Cohn (in whose garden the original object resides). John Van Tassel composed the LaTeX picture of the lantern.

Many people other than those listed above have contributed to the HOL-Omega documentation effort, either by providing material, or by sending lists of errors in the first edition. Thanks to everyone who helped, and thanks to DSTO and SRI for their generous support.

[1] M.J.C. Gordon, 'HOL: a Proof Generating System for Higher Order Logic', in: *VLSI Specification, Verification and Synthesis*, edited by G. Birtwistle and P.A. Subrahmanyam, (Kluwer Academic Publishers, 1988), pp. 73–128.

[2] *The ML Handbook*, unpublished report from Inria by Guy Cousineau, Mike Gordon, Gérard Huet, Robin Milner, Larry Paulson and Chris Wadsworth.

Contents

1 Getting and Installing HOL-Omega **9**
1.1 Getting HOL-Omega 9
1.2 The `hol-info` mailing list 9
1.3 Installing HOL-Omega 10
1.3.1 Overriding `smart-configure` 12

2 HOL-Omega Appetizers **15**
2.1 Collections 15
2.1.1 Object-oriented collections 17
2.1.2 Fold operation 18
2.1.3 Map operation 23
2.1.4 Abstract collections 27

3 Epilogue **33**

Chapter 1

Getting and Installing HOL-Omega

This chapter describes how to get the HOL-Omega system and how to install it. It is generally assumed that some sort of Unix system is being used, but the instructions that follow should apply *mutatis mutandis* to other platforms. Unix is not a pre-requisite for using the system. HOL-Omega may be run on PCs running Windows operating systems from Windows NT onwards (i.e., Windows 2000, XP and Vista are also supported), as well as Macintoshes running Mac OS X.

1.1 Getting HOL-Omega

HOL-Omega is part of the HOL system. The HOL system has several branches; branch `HOL-Omega` contains the source of the HOL-Omega theorem prover. The naming scheme for HOL-Omega releases is ⟨*name*⟩-⟨*number*⟩; the release described here is Kananaskis-8.

HOL development uses the Git version control system. Git is freely available from `http://git-scm.com/`, and is well described in the book *ProGit* by Scott Chacon.

A fresh copy of the current developer version of HOL-Omega may be checked out into a fresh subdirectory called `hol-omega` by the following Git command:

```
git clone -b HOL-Omega git://github.com/mn200/HOL.git hol-omega
```

As a developer, to check out a fresh copy of HOL-Omega that one could edit and write back to the github repository, one would set up a SSH key with github and then do

```
git clone -b HOL-Omega git@github.com:mn200/HOL.git hol-omega
```

To set up a SSH key, see `https://help.github.com/articles/generating-ssh-keys`.

To become an HOL developer and have write access to the github repository, please contact one of the administrators listed at `http://sourceforge.net/projects/hol/`.

1.2 The `hol-info` mailing list

The `hol-info` mailing list serves as a forum for discussing HOL-Omega and disseminating news about it. If you wish to be on this list (which is recommended for all users of HOL-Omega), visit `http://lists.sourceforge.net/lists/listinfo/hol-info`. This web-page can also be used to unsubscribe from the mailing list.

1.3 Installing HOL-Omega

It is assumed that the HOL-Omega sources have been obtained and the `tar` file unpacked into a directory `hol-omega`.[1] The contents of this directory are likely to change over time, but it should contain the following:

Principal Files on the HOL-Omega Distribution Directory

File name	*Description*	*File type*
`README`	Description of directory `hol-omega`	Text
`COPYRIGHT`	A copyright notice	Text
`INSTALL`	Installation instructions	Text
`tools`	Source code for building the system	Directory
`bin`	Directory for HOL-Omega executables	Directory
`sigobj`	Directory for ML object files	Directory
`src`	ML sources of HOL-Omega	Directory
`help`	Help files for HOL-Omega system	Directory
`examples`	Example source files	Directory

The session in the box below shows a typical distribution directory. The HOL-Omega distribution has been placed in the directory `/Users/palantir/hol-omega/` on a Macintosh running OS X for a user `palantir`.

All sessions in this documentation will be displayed in boxes with a number in the top right hand corner. This number indicates whether the session is a new one (when the number will be *1*) or the continuation of a session started in an earlier box. Consecutively numbered boxes are assumed to be part of a single continuous session. The Unix prompt for the sessions is `$`, so lines beginning with this prompt were typed by the user. After entering the HOL-Omega system (see below), the user is prompted with - for an expression or command of the HOL-Omega meta-language ML; lines beginning with this are thus ML expressions or declarations. Lines not beginning with `$` or - are system output. Occasionally, system output will be replaced with a line containing ... when it is of minimal interest. The meta-language ML is introduced in Chapter **??**.

1

```
$ pwd
/Users/palantir/hol-omega
$ ls -F
COPYRIGHT   bin/          examples/   sigobj/       tools-poly/
INSTALL     cleanall*     help/       src/
Manual/     developers/   icon.gif*   std.prelude
README      doc/          merging     tools/
```

[1] You may choose another name if you want; it is not important.

Now you will need to rebuild HOL-Omega from the sources.[2]

Before beginning you must have a current version of Moscow ML or Poly/ML[3]. In the case of Moscow ML, you must have version 2.01. Moscow ML is available on the web from `http://www.dina.kvl.dk/~sestoft/mosml.html`. Poly/ML is available from `http://polyml.org`. When you have your ML system installed, and are in the root directory of the distribution, the next step is to run `smart-configure`. With Moscow ML, this looks like:

2

```
$ mosml < tools/smart-configure.sml
Moscow ML version 2.01 (January 2004)
Enter ‘quit();’ to quit.
- [opening file "tools/smart-configure-mosml.sml"]

HOL-Omega smart configuration.

Determining configuration parameters: OS mosmldir holdir
OS:                 macosx
mosmldir:           /Users/palantir/mosml/bin
holdir:             /Users/palantir/hol-omega
dynlib_available:   true

Configuration will begin with above values.  If they are wrong
press Control-C.
```

If you are using Poly/ML, then write

```
poly < tools/smart-configure.sml
```

instead.

Assuming you don't interrupt the configuration process, this will build the `Holmake` and `build` programs, and move them into the `hol-omega/bin` directory. If something goes wrong at this stage, consult Section 1.3.1 below.

The next step is to run the `build` program. This should result in a great deal of output as all of the system code is compiled and the theories built. Eventually, a HOL-Omega system[4] is produced in the `bin/` directory.

3

```
$ bin/build
  ...
  ...
Uploading files to /Users/palantir/hol-omega/sigobj

Hol built successfully.
$
```

[2]It is possible that pre-built systems may soon be available from the web-page mentioned above.

[3]Poly/ML cannot be used with HOL-Omega on Windows.

[4]Four HOL-Omega executables are produced: hol, hol.noquote, hol.bare and hol.bare.noquote. The first of these will be used for most examples in the *TUTORIAL*.

1.3.1 Overriding `smart-configure`

If `smart-configure` is unable to guess correct values for the various parameters (`holdir`, `OS` etc.) then you can create a file called to provide correct values. With Moscow ML, this should be `config-override` in the root directory of the HOL-Omega distribution. With Poly/ML, this should be `poly-includes.ML` in the `tools-poly` directory. In this file, specify the correct value for the appropriate parameter by providing an ML binding for it. All variables except `dynlib_available` must be given a string as a possible value, while `dynlib_available` must be either `true` or `false`. So, one might write

```
val OS = "unix";                                                    4
val holdir = "/local/scratch/myholdir";
val dynlib_available = false;
```

The `config-override` file need only provide values for those variables that need overriding.

With this file in place, the `smart-configure` program will use the values specified there rather than those it attempts to calculate itself. The value given for the `OS` variable must be one of `"unix"`, `"linux"`, `"solaris"`, `"macosx"` or `"winNT"`.[5]

In extreme circumstances it is possible to edit the file `tools/configure.sml` yourself to set configuration variables directly. (If you are using Poly/ML, you must edit `tools-poly/configure.sml` instead.) At the top of this file various incomplete SML declarations are present, but commented out. You will need to uncomment this section (remove the `(*` and `*)` markers), and provide sensible values. All strings must be enclosed in double quotes.

The `holdir` value must be the name of the top-level directory listed in the first session above. The `OS` value should be one of the strings specified in the accompanying comment.

When working with Moscow ML, the `mosmldir` value must be the name of the directory containing the Moscow ML binaries (`mosmlc`, `mosml`, `mosmllex` etc). When working with Poly/ML, the `poly` string must be the path to the `poly` executable that begins an interactive ML session. The `polymllibdir` must be a path to a directory that contains the file `libpolymain.a`.

Subsequent values (`CC` and `GNUMAKE`) are needed for "optional" components of the system. The first gives a string suitable for invoking the system's C compiler, and the second specifies a make program.

After editing, `tools/configure.sml` the lines above will look something like:

[5]The string `"winNT"` is used for Microsoft Windows operating systems that are at least as recent as Windows NT. This includes Windows 2000, XP and Vista.

5
```
$ more configure.sml
  ...
val mosmldir:string = "/home/palantir/mosml";
val holdir :string  = "/home/palantir/hol-omega";
val OS :string      = "linux";
                        (* Operating system; choices are:
                             "linux", "solaris", "unix", "macosx",
                             "winNT"   *)

val CC:string       = "gcc";      (* C compiler                       *)
val GNUMAKE:string  = "make";     (* for bdd library and SMV          *)
val DEPDIR:string   = ".HOLMK";   (* where Holmake dependencies kept  *)
  ...
$
```

Now, at either this level (in the `tools` or `tools-poly` directory) or at the level above, the script `configure.sml` must be piped into the ML interpreter (i.e., `mosml` or `poly`). For example,

6
```
$ mosml < tools/configure.sml
Moscow ML version 2.01 (January 2004)
Enter ‘quit();’ to quit.
- > val mosmldir = "/home/palantir/mosml" : string
  val holdir = "/home/palantir/hol-omega" : string
  val OS = "linux" : string
- > val CC = "gcc" : string
  ...
Beginning configuration.
Removing old quotation filter from bin/
Making tools/mllex/mllex.exe
Making bin/Holmake.
  ...
Making bin/build.
Making hol-mode.el (for Emacs/XEmacs)
Generating bin/hol.
Generating bin/hol.noquote.
  ...
Number of states = 170
Number of distinct rows = 90
Approx. memory size of trans. table = 23040 bytes
Analysing filter.sml
Compiling filter.sml
Compiling quote-filter.sml
  ...
Analysing selftest.sml
Quote-filter built
Setting up the muddy library Makefile.

Finished configuration!
$
```

Chapter 2

HOL-Omega Appetizers

This chapter will introduce the HOL-Omega logic, with the idea of motivating it by a series of examples. These examples are only discussed superficially, to showcase the new ideas, and not all details are pursued. A more complete description of the HOL-Omega extensions is provided in the next chapter in the tutorial. But these are presented as appetizers, to lightly show how the new features might be used to good effect.

2.1 Collections

To begin, HOL is blessed with a number of different types in the logic that represent different varieties of collections, like lists, sets, and bags. These are polymorphic types, written e.g. α `list`, where α is the type of the elements of the list. All these collections are similar, in that they all have an empty collection, they all have a way to insert a new element into a collection, they all have a way to measure the size of a collection, etc.

Suppose one wanted to represent the notion of a collection as an abstraction of the normal notions of a set or a list. In HOL there is no natural way to do this, but in HOL-Omega one could use a type operator variable to stand for the various collection types, and then create a record of some of the normal functions used on collections, as follows.

7
```
- new_theory "appetizers";
<<HOL message: Created theory "appetizers">>
> val it = () : unit
> set_trace "Unicode" 0;
val it = () : unit

- Hol_datatype ‘collection_ops =
     <| empty  : ’x ’col;
        insert : ’x -> ’x ’col -> ’x ’col;
        length : ’x ’col -> num |>‘;
<<HOL message: Defined type: "collection_ops">>
> val it = () : unit
```

Here we have used the type variable `’col` as a variable to stand for the type operator we are talking about, whether `list`, `option`, or some other type. In HOL, type variables can only stand for entire types, like `num list`, but not type operators like just `list`. But here, `’col` is being used as a function on types, that takes a type `’x`, the type of the

elements of the collection, and returns a type `'x 'col`, the type of collections of such elements. Such type operator variables are one of the new features of HOL-Omega.

Both `'col` and `'x` are free type variables in this definition, so the type being defined takes two arguments, e.g., `('col, 'x)collection_ops`. The order of the two arguments is by alphabetical order.

Now we can describe lists as collections:

8

```
- val list_ops = Define
     ‘list_ops = <|empty := []:'a list; insert := CONS; length := LENGTH|>‘;
Definition has been stored under "list_ops_def"
> val list_ops =
    |- list_ops = <|empty := []; insert := CONS; length := LENGTH|> : thm

- type_of ‘‘list_ops‘‘;
<<HOL message: inventing new type variable names: 'a>>
> val it =
    ‘‘:(list, 'a) collection_ops‘‘
     : hol_type
```

The type of this collection is `(list, 'a) collection_ops`. The first argument is the type `list`, here being used without any type argument of its own. This is meaningful in HOL-Omega, although it may look weird to HOL users who are used to always seeing `list` with an argument, like `num list` or `'a list`. But here `list` is itself an argument, albeit a type operator alone, replacing `'col` in the definition of `collection_ops` above.

Here are sets described as collections:

9

```
- val set_ops = Define
     ‘set_ops = <|empty := {}:'a set; insert := $INSERT; length := CARD|>‘;
Definition has been stored under "set_ops_def"
> val set_ops = |- set_ops = <|empty := {}; insert := $INSERT; length := CARD|>
     : thm

- type_of ‘‘set_ops‘‘;
<<HOL message: inventing new type variable names: 'b>>
> val it =
    ‘‘:(\'a. 'a -> bool, 'b) collection_ops‘‘
     : hol_type
```

Note that the first argument to this set collection type is `\'a. 'a -> bool`. This is an *abstraction type*, similar to the normal lambda abstraction in terms, but this abstraction is within the type language of HOL-Omega. The scope of the lambda binding of `'a` in the type above is up to but not including the comma, which ends the first type argument.

But, you may ask, why does this type abstraction `\'a. 'a -> bool` appear in this collection type? The reason is that the type of sets in HOL, `'a set`, is actually a type abbreviation, not a real type. It is a feature of the parser and prettyprinter, not the actual logic as such. The abbreviation `'a set` stands for the real type `'a -> bool`. The

HOL-Omega system figures out the appropriate type to substitute for the type argument `'col` to create the type `'a -> bool`, and the substitution is [`'col` ↦ `\'a. 'a -> bool`]. The type resulting from the substitution is `'a (\'a. 'a -> bool)` (in postfix notation), which is equivalent to `'a -> bool` through type beta-reduction.

HOL contains not only lists and sets, but also bags, which are sometimes called multisets. Bags are like sets which can include multiple copies of its elements, whereas sets can only contain a single copy of each. Here are bags described as collections:

10

```
- load "bagLib";
...
- val bag_ops = Define
     `bag_ops = <| empty := {||}:'a bag; insert := BAG_INSERT;
                   length := BAG_CARD|>`;
Definition has been stored under "bag_ops_def"
> val bag_ops =
    |- bag_ops = <|empty := {||}; insert := BAG_INSERT; length := BAG_CARD|> :
  thm

- type_of ``bag_ops``;
<<HOL message: inventing new type variable names: 'b>>
> val it =
    ``:(\'a. 'a -> num, 'b) collection_ops``
     : hol_type
```

Similar to sets, `'a bag` is a type abbreviation for `'a -> num`. In this case, HOL-Omega figures out that the correct type to substitute for `'col` is `\'a. 'a -> num`.

So we can represent lists, sets, and bags as collections using this record type with fields for these three common operations.

2.1.1 Object-oriented collections

In fact we can go further, and try to model collections in an object-oriented way, combining together the data values stored in the collection with the operations used to manipulate them.

11

```
- Hol_datatype `collection =
     <| this : 'x 'col;
        ops  : ('col,'x) collection_ops |>`;
```

Now we can define an operation to insert an element into a collection, without having to know what particular kind of collection it is.

12
```
- val insert_def =
  Define ‘insert x (c:(’col,’x)collection) =
             <| this := c.ops.insert x c.this;
                ops  := c.ops |>‘;
Definition has been stored under "insert_def"
> val insert_def =
    |- !x c. insert x c = <|this := c.ops.insert x c.this; ops := c.ops|>
     : thm
```

Similarly, we can define an operation to measure the size of a collection.

13
```
- val length_def =
  Define ‘length (c:(’col,’x)collection) = c.ops.length c.this‘;
Definition has been stored under "length_def"
> val length_def =
    |- !c. length c = c.ops.length c.this
     : thm
```

So we can use the same functions, `insert` and `length`, to manipulate any lists, sets, or bags, with the appropriate results for each type of collection.

2.1.2 Fold operation

But what if we want to add a “fold” operator, like the `FOLDR` function on lists:

14
```
- type_of ‘‘FOLDR‘‘;
<<HOL message: inventing new type variable names: ’a, ’b>>
> val it =
    ‘‘:(’a -> ’b -> ’b) -> ’b -> ’a list -> ’b‘‘
     : hol_type

- listTheory.FOLDR;
> val it =
    |- (!f e. FOLDR f e [] = e) /\
       !f e x l. FOLDR f e (x::l) = f x (FOLDR f e l)
     : thm
```

We might add a new field `fold` to our new record of collection operations as follows.

15
```
- Hol_datatype ‘collection_ops =
     <| empty  : ’x ’col;
        insert : ’x -> ’x ’col -> ’x ’col;
        length : ’x ’col -> num;
        fold   : (’x -> ’y -> ’y) -> ’y -> ’x ’col -> ’y |>‘;
<<HOL message: Defined type: "collection_ops">>
> val it = () : unit
```

Then we can construct a record of this type using `FOLDR`.

16
```
- val list_ops = Define
     ‘list_ops = <| empty := []:’a list; insert := CONS; length := LENGTH;
                    fold := FOLDR|>‘;
<<HOL message: inventing new type variable names: ’b>>
Definition has been stored under "list_ops_def"
> val list_ops =
    |- list_ops =
       <|empty := []; insert := CONS; length := LENGTH; fold := FOLDR|> : thm

- type_of ‘‘list_ops‘‘;
<<HOL message: inventing new type variable names: ’a, ’b>>
> val it =
    ‘‘:(list, ’a, ’b) collection_ops‘‘
     : hol_type
```

Wait, this is not what we wanted. There is a third type argument in `collection_ops` now, `’b`. This new type argument appears there because there are now three free type variables in the definition of `collection_ops`, `’col`, `’x`, and `’y`. The third argument `’y` is the type of the value computed and returned by `fold`.

But having the `’y` type variable free in this way *fails to be fully general*, as any particular instance of `fold` can produce only one type of result. No matter its arguments, no different type of result can be produced.

To see this problem more clearly, suppose we follow this development further, using this definition of `collection_ops`, and upon it defining the collection type and the fold operation on collections.

17
```
- Hol_datatype ‘collection =
     <| this : ’x ’col;
        ops  : (’col,’x,’y) collection_ops |>‘;
<<HOL message: Defined type: "collection">>
> val it = () : unit

- val fold_def = Define ‘fold f e c = c.ops.fold f e c.this‘;
<<HOL message: inventing new type variable names: ’a, ’b, ’c>>
Definition has been stored under "fold_def"
> val fold_def =
    |- !f e c. fold f e c = c.ops.fold f e c.this
     : thm
```

Now let’s make an example collection.

18
```
- val ex1 = ‘‘<| this := [1;8;27]; ops := list_ops |>‘‘;
<<HOL message: inventing new type variable names: ’a>>
> val ex1 = ‘‘<|this := [1; 8; 27]; ops := list_ops|>‘‘ : term
```

But when we try to do a fold on this example, we see a type error.

19

```
- ‘‘fold (\x y. x+y) 0 ^ex1‘‘;

Type inference failure: unable to infer a type for the application of

(fold (\(x :num) (y :num). x + y) (0 :num) :
   (list, num, num) collection -> num)

on line 16, characters 2-19

which has type

:(list, num, num) collection -> num

to

<|this := [(1 :num); (8 :num); (27 :num)];
  ops := (list_ops :(list, num, 'a) collection_ops)|>

between beginning of frag 1 and end of frag 1

which has type

:(list, num, 'a) collection

unification failure message: unify failed
! Uncaught exception:
! HOL_ERR
```

This example failed type-checking because the type of the result that the collection was able to provide (`'a`) was not the same as the type of the value that the actual fold function, `\x y.x+y`, was trying to return (`num`).

We could try to patch this up by manually instantiating this example.

20

```
- val ex1a = inst [‘‘:'a‘‘ |-> ‘‘:num‘‘] ex1;
> val ex1a = ‘‘<|this := [1; 8; 27]; ops := list_ops|>‘‘ : term
- ‘‘fold (\x y. x+y) 0 ^ex1a‘‘;
> val it =
    ‘‘fold (\x y. x + y) 0 <|this := [1; 8; 27]; ops := list_ops|>‘‘
     : term
```

This does work and the term passes type-checking. But what if we try another example that returns a result of a different type?

21
```
- ‘‘fold (\x y. EVEN x /\ y) T ^ex1a‘‘;

Type inference failure: unable to infer a type for the application of

(fold (\(x :num) (y :bool). EVEN x /\ y) T :
   (list, num, bool) collection -> bool)

on line 21, characters 2-27

which has type

:(list, num, bool) collection -> bool

to

<|this := [(1 :num); (8 :num); (27 :num)];
  ops := (list_ops :(list, num, num) collection_ops)|>

between beginning of frag 1 and end of frag 1

which has type

:(list, num, num) collection

unification failure message: unify failed
! Uncaught exception:
! HOL_ERR
```

The type of the result that the collection was able to provide (`num`) was not the same as the type of the value that the fold function was trying to return (`bool`).

The point here is that the above version of fold is simply not general enough for normal use. What we really want is the following version.

22
```
- Hol_datatype ‘collection_ops =
     <| empty  : 'x 'col;
        insert : 'x -> 'x 'col -> 'x 'col;
        length : 'x 'col -> num;
        fold   : !'y. ('x -> 'y -> 'y) -> 'y -> 'x 'col -> 'y |>‘;
<<HOL message: Defined type: "collection_ops">>
> val it = () : unit
```

In this new defintion of `collection_ops`, the type of the `fold` field begins with "`!'y.`". This indicates a *universal type*; the idea comes from a logic called System F. The `!'y.` universally quantifies `'y` over the body `('x -> 'y -> 'y) -> 'y -> 'x 'col -> 'y`. The quantification binds the occurrences of `'y` within the universal type, so that `'y` does not become a free type variable outside the binding, and thus not a free type variable of the `collection_ops` type. Then this version of the `collection_ops` type is created with just its normal two arguments `'col` and `'x`, not `'y`.

To create an example of this new type of fold operation, we need to provide a term whose type is the above universal type. Such a term is `\:'b. FOLDR`.

23
```
- val list_ops = Define
   ‘list_ops = <|empty := []:'a list; insert := CONS; length := LENGTH;
                 fold := \:'b. FOLDR|>‘;
Definition has been stored under "list_ops_def"
> val list_ops =
    |- list_ops =
       <|empty := []; insert := CONS; length := LENGTH;
         fold := (\:'b. FOLDR)|> : thm

- type_of ‘‘list_ops‘‘;
<<HOL message: inventing new type variable names: 'a>>
> val it =
    ‘‘:(list, 'a) collection_ops‘‘
     : hol_type
```

The term `\:'b. FOLDR` is a *type abstraction term*. It abstracts a term, here `FOLDR`, not by a term variable, but by a type variable `'b`. This is a new variety of term not present in HOL, but added in HOL-Omega. The type of such a term is a universal type. Where the type of `FOLDR` is `('a -> 'b -> 'b) -> 'b -> 'a 'col -> 'b`, the type of `\:'b. FOLDR` is instead `!'b. ('a -> 'b -> 'b) -> 'b -> 'a 'col -> 'b`.

The use of a universal type and a type abstraction term here provides the generality we were looking for, so that `fold` can be used to return results of any desired type.

24
```
- Hol_datatype ‘collection =
     <| this : 'x 'col;
        ops  : ('col,'x) collection_ops |>‘;
<<HOL message: Defined type: "collection">>
> val it = () : unit

- val fold_def =
  Define ‘fold f (e:'b) (c:('col,'a)collection) = c.ops.fold f e c.this‘;
Definition has been stored under "fold_def"
> val fold_def =
    |- !f e c. fold f e c = c.ops.fold f e c.this
     : thm
```

If we turn on the printing of the types of terms, we can see in more detail the types involved in the `fold` operation.

25
```
- show_types := true;
> val it = () : unit
- fold_def;
> val it =
    |- !(f :'a -> 'b -> 'b) (e :'b) (c :('col :ty => ty, 'a) collection).
         fold f e c = c.ops.fold [:'b:] f e c.this
     : thm
```

Now in the definition of `fold`, we see `[:'b:]`. This indicates an application of the term `c.ops.fold` to the type `'b` as a type argument. It is like an application of a term to a term argument, except the argument is a type, not a term. In such a *type application term*, the operator has to have a universal type; in this case, the type of `c.ops.fold` is `!'b. ('a -> 'b -> 'b) -> 'b -> 'a 'col -> 'b`. The result of the type application is to substitute the type argument for the bound type variable throughout the term. In this case, the result has type `('a -> 'b -> 'b) -> 'b -> 'a 'col -> 'b`. It is therefore ready to take as its next arguments the terms `f`, `e`, and `c.this`.

The type arguments to terms are important for the logic, but in practice they tend to make terms harder to read, so by default their printing is turned off. Also, in many cases the user need not mention them when writing terms; the parser's type inference will try to deduce where they are needed, and then exactly which type argument should be inserted there. That is how the `[:'b:]` type argument was inserted into the definition of `fold` above.

This version of the fold operation can be used easily to construct folds returning different types, without any manual instantiations.

26
```
- show_types := false;
> val it = () : unit
- val ex1 = ``<| this := [2;3;5;7]; ops := list_ops |>``;
> val ex1 = ``<|this := [2; 3; 5; 7]; ops := list_ops|>`` : term

- ``fold (\x y. x+y) 0 ^ex1``;
> val it =
    ``fold (\x y. x + y) 0 <|this := [2; 3; 5; 7]; ops := list_ops|>``
     : term

- ``fold (\x y. EVEN x /\ y) T ^ex1``;
> val it =
    ``fold (\x y. EVEN x /\ y) T <|this := [2; 3; 5; 7]; ops := list_ops|>``
     : term
```

2.1.3 Map operation

This seems to be working well. Let's try another extension, adding a "map" operation to the group of operations on collections. The basic idea of a map operation is to apply a function to every element of a collection, and from all of the results form a new collection. For lists, HOL contains the `MAP` function predefined, and there are similar functions for sets and bags.

27
```
- type_of ``MAP``;
<<HOL message: inventing new type variable names: 'a, 'b>>
> val it =
    ``:('a -> 'b) -> 'a list -> 'b list``
     : hol_type
```

28
```
- listTheory.MAP;
> val it =
    |- (!f. MAP f [] = []) /\ !f h t. MAP f (h::t) = f h::MAP f t
     : thm
```

Suppose we try to extend the set of operations with an entry for `map`, using a universally quantified type in the same style as we did for `fold`.

29
```
- Hol_datatype ‘collection_ops =
     <| length : ’x ’col -> num;
        empty  : ’x ’col;
        insert : ’x -> ’x ’col -> ’x ’col;
        fold   : !’y. (’x -> ’y -> ’y) -> ’y -> ’x ’col -> ’y;
        map    : !’y. (’x -> ’y) -> ’x ’col -> ’y ’col |>‘;
<<HOL message: Defined type: "collection_ops">>
> val it = () : unit
```

To fashion an example of this `map` operation, we need to provide a term whose type is the universal type `!’y. (’x -> ’y) -> ’x ’col -> ’y ’col`, such as `\:’b. MAP`.

30
```
- val list_ops = Define
   ‘list_ops = <|empty := []:’a list; insert := CONS; length := LENGTH;
                 fold := \:’b. FOLDR; map := \:’b. MAP |>‘;
Definition has been stored under "list_ops_def"
> val list_ops =
    |- list_ops =
       <|empty := []; insert := CONS; length := LENGTH;
         fold := (\:’b. FOLDR); map := (\:’b. MAP)|>
     : thm

- type_of ‘‘list_ops‘‘;
<<HOL message: inventing new type variable names: ’a>>
> val it =
    ‘‘:(list, ’a) collection_ops‘‘
     : hol_type
```

Next we can recreate the type of collections, using this expanded record of operations.

31
```
- Hol_datatype ‘collection =
     <| this : ’x ’col;
        ops  : (’col,’x) collection_ops |>‘;
<<HOL message: Defined type: "collection">>
> val it = () : unit
```

Now we define the “map” operation that takes a function and a collection and creates a new collection from the results.

32
```
- val map_def =
  Define ‘map (f:’a -> ’b) c =
            <| this := c.ops.map f c.this;
               ops  := c.ops |> ‘;
```

Unfortunately, this definition runs into difficulties with the typing.

33
```
Exception raised at Preterm.typecheck:
on line 113, characters 15-30:

Type inference failure: unable to infer a type for the application of

 _ record fupdatethis
  (K
     ((c :('col :ty => ty, 'a) collection).ops.map [:'b:] (f :'a -> 'b)
        c.this) :'b 'col -> 'b 'col)

between line 112, character 12 and line 113, character 30

which has type

:('col :ty => ty, 'b) collection -> ('col, 'b) collection

to

<|ops := (c :('col :ty => ty, 'a) collection).ops|>

on line 113, characters 15-30

which has type

:('col :ty => ty, 'a) collection

unification failure message: unify failed

! Uncaught exception:
! HOL_ERR
```

The details of the above error message are not important. The real problem here is that the type of the new collection created is `('col,'b)collection`, while the type of the original collection is `('col,'a)collection`. The new collection being formed has its `this` field given a value of the new collection type, but the `ops` field is given a record of operations on the old collection type, not the new one.

This problem can be resolved by using one more universal type for the `ops` field itself.

34
```
- Hol_datatype `collection =
     <| this : 'x 'col;
        ops  : !'x. ('col,'x) collection_ops |>`;
<<HOL message: Defined type: "collection">>
> val it = () : unit
```

Now the `map` function can be defined as we desire, with no type problems.

35
```
- val map_def =
  Define ‘map (f:’a -> ’b) c =
            <| this := c.ops.map f c.this;
               ops  := c.ops |> ‘;
Definition has been stored under "map_def"
> val map_def =
    |- !f c. map f c = <|this := c.ops.map f c.this; ops := c.ops|>
     : thm
```

To check on the types involved, let's turn on the display of types.

36
```
- show_types := true;
> val it = () : unit

- map_def;
> val it =
    |- !(f :’a -> ’b) (c :(’col :ty => ty, ’a) collection).
         map f c =
         <|this := (c.ops [:’a:]).map [:’b:] f c.this; ops := c.ops|>
     : thm
```

Here we can see not only the type argument `[:’b:]` inserted for `map`, as was done before for `fold`, but also the operations record itself `c.ops` is given the type argument `[:’a:]`. The parser's type inference was able to deduce the necessary type arguments from the actual user input and insert them in the appropriate places.

As a final example in this section, let's consider an operation that takes two collections, which may use different underlying data structures, and combines their elements into a single collection. We can do this without expanding the definition of `collection_ops`, but just using the operations that are already present.

37
```
- val union_def =
  Define ‘union (c1: (’col1,’a)collection) (c2: (’col2,’a)collection) =
            <| this := fold c2.ops.insert c2.this c1 : ’a ’col2;
               ops  := c2.ops |>‘;
Definition has been stored under "union_def"
> val union_def =
    |- !(c1 :(’col1 :ty => ty, ’a) collection)
          (c2 :(’col2 :ty => ty, ’a) collection).
         union c1 c2 =
         <|this := fold (c2.ops [:’a:]).insert c2.this c1; ops := c2.ops|>
     : thm

- type_of ‘‘union‘‘;
<<HOL message: inventing new type variable names: ’a, ’b, ’c>>
> val it =
    ‘‘:(’a :ty => ty, ’b) collection ->
       (’c :ty => ty, ’b) collection -> (’c, ’b) collection‘‘
     : hol_type
```

So the use of universal types provides the needed type polymorphism, which could not have been accomplished using simply the traditional higher order logic type system.

Much of the advantage of HOL-Omega comes because of the new universal types. The free type variables in classic HOL types could be thought of as being implicitly univerally quantified, as they can be substituted by any other type to form a type instance. But in HOL-Omega, the $\forall$ quantification can be found *within* a type, as in $(\forall\alpha.\alpha \to \alpha) \to$ `bool`. This use of the $\forall$ in the left hand side of a function type $(\to)$ is key to much of the new functionality of HOL-Omega.

2.1.4 Abstract collections

We have seen how one could create a very nice version of collections, modeled in an object-oriented way, so that the operations that obtain the size of a collection, fold over a collection, etc., are invoked the same whether the actual internal data structure is a list, set, or bag. But what that internal data structure is, is still apparent from the type of the collection.

38

```
- val ex1 = ‘‘<| this := [2;3;5;7]; ops := list_ops |>‘‘;
> val ex1 = ‘‘<|this := [2; 3; 5; 7]; ops := list_ops|>‘‘ : term
- type_of ex1;
> val it =
    ‘‘:(list, num) collection‘‘
     : hol_type

- val ex2 = ‘‘<| this := {2;3;5;7}; ops := set_ops |>‘‘;
> val ex2 = ‘‘<|this := {2; 3; 5; 7}; ops := set_ops|>‘‘ : term
- type_of ex2;
> val it =
    ‘‘:(\’b. ’b -> bool, num) collection‘‘
     : hol_type
```

The internal data structure is visible as `list` in example `ex1` and as `\’b. ’b -> bool` in example `ex2`.

That internal data structure can be represented by a HOL-Omega type operator variable, and that is how a general routine could be written to handle arguments built using any collection structure, as was done above.

But suppose one wanted to completely hide the actual data structure used, abstracting that information away from the external use of the collection, considering it a detail of the implementation. This could be very useful in modularizing a proof, where certain parts of the proof know about the particular implementation data structure, but above a certain layer that information is hidden, and the rest of the proof cannot know or rely on that choice, but instead must work the same irrespective of what data structure is used. This makes it possible, at a later time, to change the implementation data structure to another structure, perhaps better suited to the task at hand, and to have that change

not affect any of the proof work done above the layer where that choice was abstracted, like the edge of a module where internal implementation details cannot leak across the module boundary. This kind of information hiding is very helpful in creating large software systems that are still maintainable and modifiable, and the same ideas apply for large proofs as well.

To accomplish this information hiding, HOL-Omega provides a new variety of type called an *existential type*.

39
```
- ``:?'col. ('col, 'a) collection``;
> val it =
    ``:?'col :ty => ty. ('col, 'a) collection``
     : hol_type
- type_vars it;
> val it = [``:'a``] : hol_type list
```

Existential types are written in the type language, similar to universal types, but using an existential type operator. In the example above, the existential notation binds the type variable `'col` across the body of the type, `('col,'a)collection`, so that the free type variables of the type contain just the type variable `'a`, not `'col`.

Terms of existential type are called *packages*. They can be constructed as a special form using the `pack` keyword, as follows.

40
```
- val list_pack = ``pack (:list, <|this := [2;3;2]; ops := list_ops|>)``;
> val list_pack =
    ``pack (:list,<|this := [2; 3; 2]; ops := list_ops|>)``
     : term
- type_of list_pack;
> val it =
    ``:?'x :ty => ty. ('x, num) collection``
     : hol_type
```

The keyword `pack` is followed by a pair where the first element is a type, preceeded by a colon, and the second element is a term. The term, which normally involves the type mentioned, is packaged up so that the type mentioned is hidden, being replaced by a type variable, which becomes the bound type variable of the existential type of the resulting package.

In the case above, the fact that `list_pack` actually contains a list has been removed from the package's type, where `list` has been replaced by the type variable `'x`.

There is the possibility of ambiguity in the types when creating such a package. Given a pair of a type and a term as above, there many be multiple ways that a resulting existential type may be formed. In such cases, the ambiguity can be resolved by using a type annotation on the package. For example, in the session below two different packages are created from exactly the same ingredients, except that one of them has a type annotation. Note that the resulting packages have different existential types; they are therefore different packages.

41
```
- val list_pack2 =
    ‘‘pack (:list, <| this := [[2];[3;5];[7]]; ops := list_ops |> )‘‘;
> val list_pack2 =
    ‘‘pack (:list,<|this := [[2]; [3; 5]; [7]]; ops := list_ops|>)‘‘
     : term
- type_of list_pack2;
> val it =
    ‘‘:?’x :ty => ty. (’x, num ’x) collection‘‘
     : hol_type

- val list_pack3 =
    ‘‘pack (:list, <| this := [[2];[3;5];[7]]; ops := list_ops |> )
       : ?’x. (’x,num list) collection‘‘;
> val list_pack3 =
    ‘‘pack (:list,<|this := [[2]; [3; 5]; [7]]; ops := list_ops|>)‘‘
     : term
- type_of list_pack3;
> val it =
    ‘‘:?’x :ty => ty. (’x, num list) collection‘‘
     : hol_type
```

We can construct packages of any kind of collection, and if the collections contain elements of the same type, then the packages themselves have the same type.

42
```
- val set_pack =
    ‘‘pack (:\’a.’a set, <| this := {2;3;2}; ops := set_ops |> )‘‘;
> val set_pack =
    ‘‘pack (:\’a. ’a -> bool,<|this := {2; 3; 2}; ops := set_ops|>)‘‘
     : term
- type_of set_pack;
> val it =
    ‘‘:?’x :ty => ty. (’x, num) collection‘‘
     : hol_type

- val bag_pack =
    ‘‘pack (:\’a.’a bag, <| this := {|2;3;2|}; ops := bag_ops |> )‘‘;
> val bag_pack =
    ‘‘pack (:\’a. ’a -> num,<|this := {|2; 3; 2|}; ops := bag_ops|>)‘‘
     : term
- type_of bag_pack;
> val it =
    ‘‘:?’x :ty => ty. (’x, num) collection‘‘
     : hol_type
```

Since all these packages have the same type, it is easy to write routines to take them as arguments. The new feature needed is an extension of the `let ... in` form to deconstruct a package into a pair of a type variable and a term, where the type variable represents the actual type that was hidden, and where the term represents the body of the package, but where the hidden type is again represented by the type variable.

43
```
- val lengthp_def =
  Define ‘lengthp (p: ?’col. (’col,’a)collection) =
            let (:’col, c) = p in
              c.ops.length c.this ‘;
Definition has been stored under "lengthp_def"
> val lengthp_def =
    |- !p. lengthp p = (let (:’col :ty => ty,c) = p in c.ops.length c.this)
     : thm
```

In the `let` form above, the package `p` (of type `?’col.(’col,’a)collection`) is unpacked into the pair of the type variable `’col` and the term variable `c`, where `c` has the type `(’col,’a)collection`. The scopes of both `’col` and `c` include the body of the `let...in` form. But the scope of `’col` also includes the term variable `c`, so that the `’col` that appears in the type of `c`, `(’col,’a)collection`, is that `’col` that was just bound. Both `’col` and `c` have no meaning outside the `let`, so in particular it is meaningless to have the body of the `let` return a value of a type involving `’col`. Such an escape of `’col` from its scope is prevented by the strong typing of the HOL-Omega logic.

Suppose we try to violate this rule, by defining an operation that returns the internal data structure of a collection. Such a definition produces the following error message:

44
```
- val this_def =
  Define ‘this (p: ?’col. (’col,’a)collection) =
            let (:’col, c) = p in
              c.this ‘;
Exception raised at Preterm.typecheck:
roughly on line 85, characters 14-19:

Type inference failure: unable to infer a type for the application of

(UNPACK :(!(’x :ty => ty). (’x, ’a) collection -> ’a (’col :ty => ty))
         -> (?(’x :ty => ty). (’x, ’a) collection) -> ’a ’col)

roughly on line 84, characters 16-29

to

\:’col :ty => ty. (\(c :(’col, ’a) collection). c.this)

roughly on line 85, characters 14-19

which has type

:!’col :ty => ty. (’col, ’a) collection -> ’a ’col

unification failure message: unify failed

! Uncaught exception:
! HOL_ERR
```

But as long as we don't violate the rules, we are fine, and can define operations to return packages newly constructed out of parts of other packages. Here is an example of an operation that takes a package as an argument, inserts an element, and then returns the result as a new package.

45

```
- val insertp_def =
  Define ‘insertp (e:’a) (p: ?’col. (’col,’a)collection) =
            let (:’col, c) = p in
              pack (:’col,
                    <| this := c.ops.insert e c.this : ’a ’col;
                       ops  := c.ops |>) ‘;
Definition has been stored under "insertp_def"
> val insertp_def =
    |- !e p.
         insertp e p =
         (let (:’col :ty => ty,c) = p
          in
            pack
              (:’col :ty => ty,
                <|this := c.ops.insert e c.this; ops := c.ops|>))
     : thm
```

We can define operations to do folds and maps on collections using the operators `fold` and `map` defined before, but where the new operations now work on packages, where the data structure is hidden internally.

46

```
- val foldp_def =
  Define ‘foldp (f:’a -> ’b -> ’b) (e:’b) (p: ?’col. (’col,’a)collection) =
            let (:’col, c) = p in
              fold f e c ‘;
Definition has been stored under "foldp_def"
> val foldp_def =
    |- !f e p. foldp f e p = (let (:’col :ty => ty,c) = p in fold f e c)
     : thm

- val mapp_def =
  Define ‘mapp (f:’a -> ’b) (p: ?’col. (’col,’a)collection) =
            let (:’col, c) = p in
              pack (:’col, map f c) ‘;
Definition has been stored under "mapp_def"
> val mapp_def =
    |- !f p.
         mapp f p =
         (let (:’col :ty => ty,c) = p in pack (:’col :ty => ty,map f c))
     : thm
```

In fact, we can actually build a single operation that takes any two collection pacakges and combines their elements, even if they happen to have different underlying data

structures, like lists and bags. The result here is calculated to have the same underlying data structure as the second argument.

47
```
- val unionp_def =
  Define ‘unionp (p1: ?’col. (’col,’a)collection)
                 (p2: ?’col. (’col,’a)collection) =
            let (:’col1, c1) = p1 in
            let (:’col2, c2) = p2 in
                pack (:’col2, union c1 c2) ‘;
Definition has been stored under "unionp_def"
> val unionp_def =
    |- !p1 p2.
         unionp p1 p2 =
         (let (:’col1 :ty => ty,c1) = p1 in
          let (:’col2 :ty => ty,c2) = p2
          in
            pack (:’col2 :ty => ty,union c1 c2))
     : thm
```

Using packages in this way makes it easier to modularize a large proof, by providing a way to hide the information about which particular types are being used at a lower level in the overall proof. This information hiding has major advantages for proof maintenance and modification over time.

Chapter 3

Epilogue

This small sample of the use of the HOL-Omega theorem prover is only intended to give a taste of what kinds of things are possible. A more complete tutorial is expected to be released soon. When it is, there will be an announcement on the `hol-info` mailing list, `hol-info@lists.sourceforge.net`.

In addition to the examples covered in this text, the full tutorial contains a more complete description of the elements of the HOL-Omega logic, and provides a variety of instructive worked examples. The code for these examples is currently present in the `./examples/HolOmega/` directory. There the following examples (among others) are to be found:

`functorScript.sml` This example shows how a simple version of category theory can be nicely realized as a shallow embedding within the new logic. Both functors and natural transformations are defined, and examples of each are demonstrated. This is similar to a development for HOL2P originally written by Norbert Völker.

`aopScript.sml` Building on the functor theory above, this shows several examples taken from *The Algebra of Programming*, by Richard Bird and Oege de Moor. These include homomorphisms, initial algebras, catamorphisms, and the banana split theorem. This development was originally written by Norbert Völker for HOL2P.

`monadScript.sml` Also building on the functor theory above, this defines the concept of a monad in three different ways, and proves the three are equivalent. Multiple examples of monads are presented, and also how one can convert a monad from one of the styles of definition to another style.

`type_specScript.sml` This file contains examples of creating new types using the new definitial principle for type specification which has been added to the HOL-Omega theorem prover. In particular, this is used to create a new type by specifying it as the initial algebra of a signature. The example used is taken from a 1993 paper by Tom Melham, “The HOL Logic Extended with Quantification over Type Variables.”

`packageScript.sml` This example shows more completely how packages and existential types may be created and used to hide the information about data types. Many of the examples are taken from and related to chapter 24 of the book ”Types and Programmng Languages” by Benjamin C. Pierce, MIT Press, 2002.

`interim` This directory contains an extensive, worked example of a generalized version of category theory, created by Jeremy Dawson of the Australian National University. This generalizes the notions of functor and natural transformation from those in `functorScript.sml`, to allow for a much richer realization of category theory. For example, multiple categories, each with their own composition and identity operations, may have functors defined between them. The development of category theory is continued through the definition of adjoints, and introduces an innovative extension of monads. This example extensively exercises the kind structure of HOL-Omega, to manage the types relating different categories and the operations among them.

Some of these examples will be described at length in the upcoming Tutorial. Until then, the reader is encouraged to try out these examples on their own.