Technical Report

UCAM-CL-TR-481
ISSN 1476-2986

Number 481

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Combining the Hol98 proof assistant with the BuDDy BDD package

Mike Gordon, Ken Friis Larsen

December 1999

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986
DOI *https://doi.org/10.48456/tr-481*

Combining the Hol98 Proof Assistant with the BuDDy BDD package

Mike Gordon* and Ken Friis Larsen†

December 15, 1999

Abstract

Theorem provers descended from LCF allow their users to write complex proof tools with high assurance that false theorems will not be proved. This report describes an experimental system that extends the LCF approach to enable combinations of deduction and BDD-based symbolic calculation to be programmed with a similar assurance. The deduction is supplied by the Hol98 system and the BDD algorithms by Jørn Lind-Nielsen's BuDDy package.

The main idea is to provide LCF-style support to a set of inference rules for judgements $\rho\ t \mapsto b$, where ρ is an order-inducing map from HOL variables to BDD variables, t is a HOL term and b is a BDD. A single oracle rule allows a HOL theorem $\vdash t$ to be deduced from $\rho\ t \mapsto$ `TRUE`.

This report is intended to serve as documentation for the Hol98 library `HolBddLib`. It is partly an exposition of standard results, partly tutorial and partly an account of research in combining deduction and symbolic state enumeration.

*Univerity of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, United Kingdom, Email: mjcg@cl.cam.ac.uk

†IT University of Copenhagen, Glentevej 67, Denmark, Email: kfl@itu.dk

Contents

1 Background and introduction

Theorem proving and model checking are complementary. Theorem proving can be applied to expressive formalisms (such as set theory and higher order logic) that are capable of modelling complex systems like complete processors. However, theorem proving systems require skilled manual guidance to verify most properties of practical interest. Model checking is automatic, but can only be applied to relatively small problems (e.g. fragments of processors). It can also provide counter-examples of great use in debugging.

The ideal would be to be able to automatically verify properties of complete systems (and find counter-examples when the verification of properties fail). This is not likely to be practical in the foreseeable future, so various compromises are being explored, for example

1. adding a layer of theorem proving on top of existing model checkers, to enable large problems to be deductively decomposed into smaller pieces that can be checked automatically [9, 1];

2. adding checking algorithms to theorem provers so that subgoals can be verified automatically [14] and counter-examples found.

These two approaches differ mainly in the starting point: (1) starts from a model checker and (2) starts from a theorem prover. The goal is the same: combine the best of model checking and theorem proving.

This paper describes some experiments in adding simple model checking infrastructure to the Hol98 theorem prover (and so falls under 2 above).

The HOL theorem prover is based on Milner's LCF approach [7] in which there is an abstract type of theorems whose primitive operations are the axioms and rules of inference of a logic (higher order logic in the case of HOL). Theorem proving tools, such as decision procedures, proof search strategies, simplifiers etc., are implemented by composing together the primitive inference rules using programs in the ML programming language (Moscow ML in the case of Hol98).[1] The ML type discipline ensures that theorem-values can only be created via sequences of primitive inferences. There is a long-standing controversy concerning whether this LCF-approach can achieve good enough

[1] In this report, 'HOL' refers to the HOL system(s) generically and 'Hol98' to the a particular version.

efficiency. In a surprising number of cases adequate efficiency can be achieved [3] as is illustrated by existing tools in HOL. However, programming decision procedures and theorem provers in an LCF-style is more demanding than just implementing them as algorithms. Thus, even if a pure LCF-style solution is possible, it might be more cost-effective to use a simpler non-LCF approach. For this (and other) reasons, some modern LCF-style provers (e.g. HOL and Isabelle [12]) allow 'oracles' to create theorems. Such oracles can either be ML programs that operate directly on the datastructures representing terms, or they might be external tools implemented in other languages (e.g. C).

As part of the Prosper project[2] there is currently considerable research into making HOL the basis of a tool integration platform. Part of this work has resulted in the definition and implementation of a *plug-in interface* that enables external tools[3] (e.g. oracles) to be 'plugged-in' to HOL. This enables the easy implementation of the kind of linking of theorem proving and model checking done in pioneering studies with PVS [14].

The work described here differs from the plug-in approach in that it aims to provide general infrastructure enabling users to implement their own bespoke BDD-based verification algorithms and then to tightly integrate them with existing HOL tools like the simplifier. Currently, a plug-in is loosely coupled: it supports a "black box" style of integration in which HOL lobs a problem at an external tool down the plug-in interface and the tool then lobs back a result via the interface. Boyer and Moore [4] have argued that decision procedures need to be tightly integrated with other tools (e.g. simplifiers and normalisers). The same point has been reiterated by the designers of PVS [13], which is built around a powerful suite of cooperating decision procedures.

Whilst we do not claim that tight-integration is always the best way to connect external tools, we think that the work described here provides evidence

[2] `http://www.dcs.gla.ac.uk/prosper`

[3] At the time of write there exist prototype plug-ins for:
Clam (`http://www.cl.cam.ac.uk/Research/HVG/Clam.HOL/`),
Acl2 (`http://www.cs.utexas.edu/users/moore/acl2/`),
Gandalf (`http://www.cs.chalmers.se/~tammet/gandalf/`),
Prover (`http://www.prover.com`),
SVC (`http://sprout.Stanford.EDU/SVC/`),
SMV (`http://www.cs.cmu.edu/~modelcheck/smv.html`),
Mona (`http://www.brics.dk/mona/`).

of its benefits for *some* applications.

Eventually, it is hoped that our experience will be combined with that gained from the various Prosper plug-ins to enable a second generation tool integration platform to be implemented that supports a spectrum spanning both loose and tight integration of external tools.

Many verification algorithms, including symbolic model checking [10], represent Boolean formulae as reduced ordered binary decision diagrams (ROBDDs or BDDs for short) [5]. As part of a project on formal verification of Verilog programs, Ken Larsen interfaced Jørn Lind-Nielsen's BuDDy BDD package[4], which is implemented in C, to Moscow ML[5]. The BuDDy package provides state-of-the-art implementations of standard BDD algorithms [2].

2 MuDDy: The ML interface to BuDDy

The Moscow ML interface to BuDDy provides ML functions for constructing and manipulating BDDs. The storage management of ML and BuDDy is linked: for example, whenever a BDD is garbage collected by ML its reference count in BuDDy is decreased and hence it may be subsequently garbage collected by the BDD package. The combination of ML and BuDDy is called MuDDy [6]. It makes BuDDy available in Moscow ML via three structures:

- `bdd` defines an ML type `bdd` representing nodes in BuDDy's BDD space, and operations for creating and manipulating ML values representing BDDs;
- `fdd` provides support for blocks of BDD variables used to encode values representing elements of finite domains;
- `bvec` provides support for Boolean vectors.

The current Hol98+BuDDy system only uses `bdd` and so the documentation of `fdd` and `bvec` provided here is minimal (see Sections 2.9 and 2.10 below).

[4] `http://www.itu.dk/research/buddy/`

[5] `http://www.dina.kvl.dk/~sestoft/mosml.html`

[6] `http://www.itu.dk/research/muddy/`

2.1 Initialisation and termination of sessions

The BuDDy package must be initialised before any BDD operations are done. Initialisation is done with the ML function

```
init : int * int -> unit
```

Evaluating `init` m n initialises BuDDy with m nodes in the nodetable and a cachesize of n. The library `HolBddLib` (Section 4) initialises the nodetable to 1000000 and cachesize to be 10000. The following is a quotation from the BuDDy documentation [8].

> Good initial values are
>
> | **Example** | **nodenum** | **cachesize** |
> |---|---|---|
> | Small test examples | 1000 | 100 |
> | Small examples | 10000 | 1000 |
> | Medium sized examples | 100000 | 10000 |
> | Large examples | 1000000 | 10000 |
>
> Too few nodes will only result in reduced performance and this increases the number of garbage collections needed. If the package needs more nodes, then it will automatically increase the size of the node table.

The function

```
done : unit -> unit
```

frees all memory used by BuDDy and resets the package to its initial state. The function

```
isRunning : unit -> bool
```

tests whether BuDDy is running (i.e. `init` has been called and `done` has not been called). It is useful for checking if initialialisation is needed – see Section 4.

The functions `init` and `done` should only be called once in a session.

Statistical information from BuDDy is available using the function `stats`

```
stats : unit -> {produced     : int,
                 nodenum      : int,
                 maxnodenum   : int,
                 freenodes    : int,
                 minfreenodes : int,
                 varnum       : int,
                 cachesize    : int,
                 gbcnum       : int}
```

The meaning of the values of the various named fields in the record returned by evaluating `stats()` are

Field name	Meaning
`produced`	total number of new nodes ever produced
`nodenum`	currently allocated number of BDD nodes
`maxnodenum`	user defined maximum number of BDD nodes
`freenodes`	number of currently free BDD nodes
`minfreenodes`	minimum number of nodes left after a BDD garbage collection
`varnum`	number of defined BDD variables
`cachesize`	number of cache entries
`gbcnum`	number of BDD garbage collections done

2.2 BDDs representing `true` and `false`

The atomic BDDs representing the two truthvalues are bound to the ML identifiers `TRUE` and `FALSE`, both of type `bdd`.

Functions for mapping from ML Booleans to BDDs and vice versa are, respectively

```
fromBool : bool -> bdd
toBool   : bdd  -> bool
```

The function `toBool` returns `true` on TRUE and `false` on FALSE. It raises the exception `Domain` on non-atomic BDDs.

```
equal : bdd -> bdd -> bool
```

tests the equality of two BDDs. Thus `TRUE` is `equal` to `fromBool(true)` and `FALSE` is `equal` to `fromBool(false)`.

2.3 Variables

In BuDDy, BDD variables are encoded as integers (type `int` in ML) and the BDD variable ordering is the numerical ordering. Thus to build a BDD to represent a HOL term with a particular variable ordering it is necessary to map HOL variables to integers so that the numerical order corresponds to the desired variable order.

The number of variables in use must be declared using

```
setVarnum : int -> unit
```

Evaluating `setVarnum` n declares that the n variables 0, 1, ... , $n-1$ are available for use. The number of variables can be increased dynamically during a session by calling `setVarnum` with a larger number. The number of variables cannot be decreased dynamically. The function

```
getVarnum : unit -> int
```

returns the number of variables in use (i.e. the argument of the last application of `setVarnum`).

2.3.1 Creating new variable nodes

The function

```
ithvar : int -> bdd
```

maps an ML integer to a BDD that consists of just the variable corresponding to the integer and

```
nithvar : int -> bdd
```

maps an integer to BDD representing the negation of the variable.

Note that evaluating `ithvar` n or `nithvar` n will raise an exception if n has not been declared as in use, i.e. if `setVarnum` m has not been previously evaluated for some m greater than n.

2.3.2 Sets of variables and quantification

BuDDy provides operations on BDDs for quantifying with respect to sets of variables. bdd provides a type `varSet` to represent such sets with, respectively, a constructor and two destructors:

```
makeset : int list -> varSet
scanset : varSet   -> int vector
fromSet : varSet   -> bdd
```

The destructor `scanset` returns a vector of the variables in the set and the destructor `fromSet` returns a BDD representing the conjunction of the variables in the set.

The function `forall` universally and `exist` existentially quantifies a BDD with respect to a set of variables:

```
forall : varSet -> bdd -> bdd
exist  : varSet -> bdd -> bdd
```

2.3.3 Sets of pairs of variables and substitution

The structure bdd provides a type `pairSet` representing sets of pairs of variables, with constructor

```
makepairSet : (int * int)list -> pairSet
```

Evaluating `makepairSet[(`x_1`,`x'_1`), ... , (`x_n`,`x'_n`)]` creates a set of pairs specifying that x'_i be substituted for x_i (for $1 \leq i \leq n$). Such a substitution is performed on a BDD with the function

```
replace : bdd -> pairSet -> bdd
```

A variable can be replaced with a BDD using

```
compose : bdd -> bdd -> int -> bdd
```

Evaluating `compose` b_1 b_2 n substitutes b_2 for the variable n in b_1.

2.4 Boolean operations on BDDs

The structure bdd introduces a type bddop corresponding to Boolean operations on BDDs. The ML function

```
apply : bdd -> bdd -> bddop -> bdd
```

applies a BDD operation to BDD values.

BuDDy provides functions for calculating in a single step the result of performing a Boolean operation and then quantifying the result with respect to several variables.

```
appall : bdd -> bdd -> bddop -> varSet -> bdd
appex  : bdd -> bdd -> bddop -> varSet -> bdd
```

The function appall universally quantifies the result of the Boolean operation and appex existentially quantifies it.

MuDDy provides ten operations of type bddop and for each of these an ML infix of type bdd * bdd -> bdd.

bddop	bdd * bdd -> bdd	Result of applying to (b_1, b_2)
And	AND	$b_1 \wedge b_2$
Nand	NAND	$\neg(b_1 \wedge b_2)$
Or	OR	$b_1 \vee b_2$
Nor	NOR	$\neg(b_1 \vee b_2)$
Biimp	BIIMP	$b_1 = b_2$
Xor	XOR	$\neg(b_1 = b_2)$
Imp	IMP	$b_1 \Rightarrow b_2$
Invimp	INVIMP	$b_2 \Rightarrow b_1$
Lessth	LESSTH	$\neg b_1 \wedge b_2$
Diff	DIFF	$b_1 \wedge \neg b_2$

The ML infixes are pre-defined using apply, for example

```
fun AND(b1,b2)  = apply b1 b2 And
```

There is no type of unary operators analogous to the binary operator type bddop, but MuDDy does provide a negation operator

```
NOT : bdd -> bdd
```

This negates a BDD using the negation program in BuDDy.

2.5 Inspecting BDD nodes

The integer labelling a BDD node and the BDDs corresponding to the high (i.e. `true`) and low (i.e. `false`) nodes are obtained, respectively, with

```
var  : bdd -> int
high : bdd -> bdd
low  : bdd -> bdd
```

Thus if b is the BDD of "*if* x *then* t_1 *else* t_2" then `var` b will return the number representing variable x, `high` b will return the BDD of t_1 and `low` b will return the BDD of t_2.

Note that `var`, `high` and `low` raise an exception if applied to `TRUE` or `FALSE`.

The entire BuDDy node table of a BDD can be coppied into ML using

```
nodetable : bdd -> int * (int * int * int)vector
```

The integer returned as the first component of the pair is a pointer (starting from 0) into the second component, a vector of node descriptors. This pointer points to the root node. Each node descriptor is a triple of integers (v, l, h), where v is the node label (i.e. a number representing a variable), l points to the low (`false`) node in the vector and h points to the high (`true`) node. The first two nodes in the vector are special: they represent `true` and `false`, respectively, and arbitrarily have the structure $(0, 0, 0)$.

The number of nodes in a BDD is computed by the function

```
nodecount : bdd -> int
```

This could be defined by

```
fun nodecount bdd = Vector.length(snd(nodetable bdd));
```

However, `nodecount` defined this way is likely to run out of space on large BDDs (since it involves copying the argument BDD from BuDDy's representation into an ML vector). Thus the ML function provided by MuDDy invokes BuDDy's `nodecount` function directly and so is space-efficient.

2.6 Printing BDDs

BuDDy provides two ways of printing BDDs: (i) as the set of paths from the root node to the *true* node and (ii) to the format used by the `dot` graph drawing program[7].

The functions for printing BDDs are;

```
printset    : bdd -> unit
printdot    : bdd -> unit
fnprintset  : string -> bdd -> unit
fnprintdot  : string -> bdd -> unit
```

`printset` and `printdot` print to standard output, whilst `fnprintset` and `fnprintdot` print to a file with the supplied name.

`printset` and `fnprintset` print out a sequence of paths, each one having the form

$< m_0 : n_0, \ldots, m_l : n_l >$

where the $n_0, \ldots, n_l$ after the colon (`:`) are `0` or `1` and indicate that the next node in the path is reached by following the low (`false`) or high (`true`) pointer, respectively.

For example, evaluating

```
printset (AND(ithvar 0, OR(ithvar 1, NOT(ithvar 2))))
```

results in

```
<0:1, 1:0, 2:0><0:1, 1:1>
```

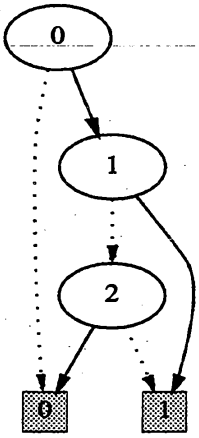
which is best understood by looking at the diagram of the BDD drawn by `dot` that appears below.

To illustrate printing to `dot` format, the same BDD can be printed to a file `ex` by evaluating

```
fnprintdot "ex" (AND(ithvar 0, OR(ithvar 1, NOT(ithvar 2))))
```

executing `dot -Tps ex > ex.ps` (in Unix) results in the following Postscript diagram of a BDD

[7] `http://www.research.att.com/sw/tools/graphviz/`



2.7 Miscellaneous BDD operations

The structure bdd provides a miscellaneous selection of BDD operations from BuDDy.

2.7.1 `restrict`

BDDs can be restricted by instantiating variables to `true` or `false`. Such an instantiation is represented by an ML value of type `restriction`, which has a constructor

```
makeRestriction : (int * bool)list -> restriction
```

Evaluating `makeRestriction[`(v_1,t_1)`, ... ,` (v_n,t_n)`]` creates a restriction specifying that each v_i be instantiated to the truth-value t_i (for $1 \leq i \leq n$). The function

```
restrict : bdd -> restriction -> bdd
```

instantiates the variables in a BDD as specified in the supplied restriction.

2.7.2 `simplify`

The ML function

```
simplify : bdd -> bdd -> bdd
```

simplifies its second argument under the assumption that the first argument is true. Thus evaluating `simplify` b_1 b_2 results in a BDD b_2', hopefully simpler than b_2, such that $b_1 \Rightarrow (b_2 = b_2')$ or, equivalently, $b_1 \wedge b_2 = b_1 \wedge b_2'$. More precisely, the relationship between b_1, b_2 and b_2' is that the BDD `IMP(`b_1`,BIIMP(`b_2`,`b_2'`))` is the BDD `TRUE` (or, equivalently, that `AND(`b_1`,`b_2`)` and `AND(`b_1`,`b_2'`)` are `equal`, i.e. the same BDD).

For more details see Henrik Reif Andersen's lecture notes on BDDs [2], where the algorithm underlying `simplify` is described and attributed to a paper by Courdert, Berthet and Madre [6].

2.7.3 `satcount`

The number of assignments *to all variables in use in the current session* that satisfy a BDD (i.e. make it true) is given by the ML function

```
satcount : bdd -> real
```

The answer is exact until the result is too big to be represented as a Moscow ML integer. Real numbers are used so that results can be returned when this happens.

2.7.4 `support`

The function

```
support : bdd -> varSet
```

gives the variables that a BDD depends on.

An application is to define a function that counts the number of valuations of a BDD using `satcount`.

```
statecount : bdd -> real
```

The definition of `statecount` is

```
fun statecount bdd =
 let val sat    = satcount bdd
     val total  = Real.fromInt(getVarnum())
     val sup    = scanset(support bdd)
     val numsup = Real.fromInt(Vector.length sup)
     val free   = total - numsup
 in
  if equal bdd TRUE
   then 0.0
   else sat / Math.pow(2.0, free)
 end
```

If a BDD is representing a set of states, then `statecount` gives the number of states in the set (hence the name).

2.8 Dynamic variable reordering

BuDDy provides functions for dynamic variable reordering using a variety of methods. See the BuDDy documentation [8] for further details. The dynamic reordering functions provided in ML via MuDDy are in the structure `bdd`.

2.9 The MuDDy structure `fdd`

The structure `fdd` provides functions for manipulating values of finite domains. Functions are provided to allocate blocks of BDD variables to represent integer values instead of only Booleans.

Encoding is done with the least significant bits first in the BDD ordering. For example, if variables v_0, v_1, v_2, v_3 are used to encode 12, then the encoding would yield $v_0 = 0$, $v_1 = 0$, $v_2 = 1$ and $v_3 = 1$.

See the BuDDy documentation [8] for further details. See the ML structure `fdd` for the BuDDy facilities provides in ML via MuDDy.

2.10 The MuDDy structure `bvec`

The structure `bvec` provides tools for encoding integers as arrays of BDDs, where each BDD represents one bit of an expression.

See the BuDDy documentation [8] for further details. See the ML structure `bvec` for the BuDDy facilities provides in ML via MuDDy.

2.11 Technical Details

The heart of the MuDDy package is mostly stub code that mirrors the BuDDy API and takes care of translating C values into SML values and vice versa.

The most tricky part is to make the Moscow ML garbage collector cooperate with the BuDDy garbage collector (we don't want either collector to try to collect the other's garbage). The cooperation is done by using the *finalized values* facility of the Moscow ML runtime system. That is, whenever a `bdd` value is returned from the BuDDy library, MuDDy register it as an external root (via `bdd_addref`) and wraps it into a finalized value.

A finalized value, in the Moscow ML runtime system, is a pair where the first component is the *destructor* (a function pointer) and the second component is the *data* (typicaly a pointer). When the Moscow ML collector collect a finalized value it apply the destructor on the data. In the case of the MuDDy package the destructor is `bdd_delref` and the data is the node-index returned by BuDDy.

3 Formal link from HOL terms to BDDs

In pure HOL, theorems are an abstract datatype, with the initial pre-defined values corresponding to axioms and the operations corresponding to rules of inference. Thus $\vdash t$ is implemented as t being computable using inference rules starting from axioms.

Theorems are one kind of judgement. Consider now another kind of judgement:

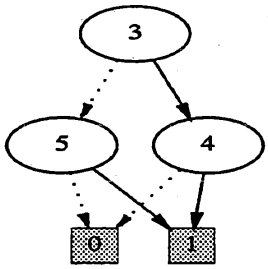
$$\rho\ t \mapsto b$$

This means that the HOL Boolean term t is represented by the BuDDy BDD b with respect to a mapping ρ from HOL variables to ML integers (representing BDD variables).

A mapping from HOL variables to BuDDy variables is called a *variable map*. The notation $\{a \mapsto 3, b \mapsto 4, c \mapsto 5\}$ denotes a variable map that maps a to 3, b to 4 and c to 5.

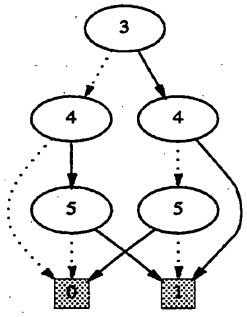
Two examples of judgements $\rho\ t \mapsto b$ with the same t, but different ρs are

$\{a \mapsto 3, b \mapsto 4, c \mapsto 5\} \quad (a \wedge b) \vee (\neg a \wedge c) \ \mapsto$



and

$\{a \mapsto 5, b \mapsto 4, c \mapsto 3\} \quad (a \wedge b) \vee (\neg a \wedge c) \ \mapsto$



3.1 BDD rules for quantifed Boolean formulae

Rules for these BDD judgements that relate HOL logical variables to BuDDy variable nodes are

$$\texttt{ithvar}\ \frac{\rho(v) = n}{\rho\ v\ \mapsto\ \texttt{ithvar}\ n} \qquad \texttt{nithvar}\ \frac{\rho(v) = n}{\rho\ \neg v\ \mapsto\ \texttt{nithvar}\ n}$$

Rules that relate HOL truthvalues and BuDDy atomic BDD nodes are

$$\texttt{TRUE}\ \frac{}{\rho\ \texttt{T} \mapsto \texttt{TRUE}} \qquad \texttt{FALSE}\ \frac{}{\rho\ \texttt{F} \mapsto \texttt{FALSE}}$$

Rules that relate HOL and BuDDy Boolean operations are

$$\texttt{NOT}\ \frac{\rho\ t \mapsto b}{\rho\ \neg t\ \mapsto\ \texttt{NOT}\ b}$$

$$\texttt{AND}\ \frac{\rho\ t_1 \mapsto b_1 \qquad \rho\ t_2 \mapsto b_2}{\rho\ t_1 \wedge t_2\ \mapsto\ b_1\ \texttt{AND}\ b_2} \qquad \texttt{OR}\ \frac{\rho\ t_1 \mapsto b_1 \qquad \rho\ t_2 \mapsto b_2}{\rho\ t_1 \vee t_2\ \mapsto\ b_1\ \texttt{OR}\ b_2}$$

$$\texttt{NAND}\ \frac{\rho\ t_1 \mapsto b_1 \qquad \rho\ t_2 \mapsto b_2}{\rho\ \neg(t_1 \wedge t_2)\ \mapsto\ b_1\ \texttt{NAND}\ b_2} \qquad \texttt{NOR}\ \frac{\rho\ t_1 \mapsto b_1 \qquad \rho\ t_2 \mapsto b_2}{\rho\ \neg(t_1 \vee t_2)\ \mapsto\ b_1\ \texttt{NOR}\ b_2}$$

$$\texttt{IMP}\ \frac{\rho\ t_1 \mapsto b_1 \qquad \rho\ t_2 \mapsto b_2}{\rho\ t_1 \Rightarrow t_2\ \mapsto\ b_1\ \texttt{IMP}\ b_2} \qquad \texttt{INVIMP}\ \frac{\rho\ t_1 \mapsto b_1 \qquad \rho\ t_2 \mapsto b_2}{\rho\ t_2 \Rightarrow t_1\ \mapsto\ b_1\ \texttt{INVIMP}\ b_2}$$

$$\texttt{BIIMP}\ \frac{\rho\ t_1 \mapsto b_1 \qquad \rho\ t_2 \mapsto b_2}{\rho\ t_1 = t_2 \ \mapsto\ b_1\ \texttt{BIIMP}\ b_2} \qquad \texttt{XOR}\ \frac{\rho\ t_1 \mapsto b_1 \qquad \rho\ t_2 \mapsto b_2}{\rho\ \neg(t_1 = t_2) \ \mapsto\ b_1\ \texttt{XOR}\ b_2}$$

$$\texttt{LESSTH}\ \frac{\rho\ t_1 \mapsto b_1 \qquad \rho\ t_2 \mapsto b_2}{\rho\ \neg t_1 \wedge t_2 \ \mapsto\ b_1\ \texttt{LESSTH}\ b_2} \qquad \texttt{DIFF}\ \frac{\rho\ t_1 \mapsto b_1 \qquad \rho\ t_2 \mapsto b_2}{\rho\ t_1 \wedge \neg t_2 \ \mapsto\ b_1\ \texttt{DIFF}\ b_2}$$

The rule for the universal quantifier is

$$\texttt{forall}\ \frac{\rho\ t \mapsto b \qquad \rho(u_1) = n_1 \qquad \cdots \qquad \rho(u_p) = n_p}{\rho\ \forall u_1\ \cdots\ u_p.\ t \ \mapsto\ \texttt{forall}\ (\texttt{makeset}[n_1,\ldots,n_p])\ b}$$

The rule for the existential quantifier is

$$\texttt{exist}\ \frac{\rho\ t \mapsto b \qquad \rho(u_1) = n_1 \qquad \cdots \qquad \rho(u_p) = n_p}{\rho\ \exists u_1\ \cdots\ u_p.\ t \ \mapsto\ \texttt{exist}\ (\texttt{makeset}[n_1,\ldots,n_p])\ b}$$

If t is a quantified Boolean formula[8] and ρ maps all the variables in t to distinct numbers, then it is clear that the rules above enable a (necessarily unique) BDD b to be deduced such that $\rho\ t \ \mapsto\ b$.

The Hol98+BuDDy system maintains an extendable list of BDD judgements called *the BDD map* (see Section 4.1) and provides a function `termToBdd` that takes a term t and, using the judgements in the BDD map and the above rules, tries to compute a BDD b such that $\rho\ t \mapsto b$, using a variable map ρ that is either explicitly supplied by the user or derived from the order in which variables are encountered..

3.2 Deriving HOL theorems via a BuDDy oracle

The *only* rule for deriving HOL theorems from BuDDy is

$$\texttt{bddOracle}\ \frac{\rho\ t \mapsto \texttt{TRUE}}{\vdash t}$$

A rule for transferring BDD representations across HOL equalities is

$$\texttt{addEquation}\ \frac{\vdash t_1 = t_2 \qquad \rho\ t_2 \mapsto b}{\rho\ t_1 \mapsto b}$$

Hol98+BuDDy is a fully-expansive system whose inference rules are the union of the HOL logic for theorem judgements $\vdash t$ and BDD rules such as those above for BDD judgements $\rho\ t \mapsto b$

[8] A quantified Boolean formula (QBF) is a formula built out of Boolean variables and constants using Boolean operations and quantification over Boolean variables.

The term t in any judgement $\rho\ t \mapsto b$ that can be derived will have HOL type `bool` and will only contain free variables of type `bool`. However t need not be a quantified Boolean formula, because it might contain non-Boolean subterms (e.g. quantifications over non-Boolean variables).

3.3 Example proof using BDD rules

Model checking consists of algorithmically checking that all executions of a model of a system satisfy a property. Checkable properties are often expressed in temporal logic and models are specified in application-specific languages. The standard semantics of temporal logic defines the truth-value of temporal formulae with respect to sequences of states – called traces – that represent successive states of a system. Thus a temporal formula can be considered to be a predicate on sequences of states, i.e. a formula of the form $\Phi(\sigma)$, where Φ is the property and σ a sequence of states (trace). A model defines a set of sequences of states corresponding to possible executions of a system. Thus a model can also be represented as a predicate on traces, $\mathcal{M}(\sigma)$ say. To check that property Φ holds of all executions of model $\mathcal{M}$ it is sufficient to check the truth of the formula $\forall\sigma.\ \mathcal{M}(\sigma) \Rightarrow \Phi(\sigma)$.

In this section we outline a special case in which the property Φ is `AG` P of CTL [10] (i.e. P true at all points in the trace) and $\mathcal{M}$ is given by a transition system $(\mathcal{R}, \mathcal{B})$, where $\mathcal{B}$:*state*->`bool` defines an initial set of states and $\mathcal{R}$:*state*#*state*->`bool` is a next-state relation (transition relation). The type *state* is a Cartesian product of `bool`.

Assume terms $\mathcal{B}(s)$ and $\mathcal{R}(s, s')$ are given, define

$$\begin{array}{lcl} \mathcal{M}_{(\mathcal{R},\mathcal{B})}(\sigma) & = & \mathcal{B}(\sigma(0)) \ \wedge\ \forall n.\ \mathcal{R}(\sigma(n), \sigma(n{+}1)) \\ (\texttt{AG}\ P)(\sigma) & = & \forall n.\ P(\sigma(n)) \end{array}$$

Then proving $\forall\sigma.\ \mathcal{M}_{(\mathcal{R},\mathcal{B})}(\sigma) \Rightarrow (\texttt{AG}\ P)(\sigma)$ is an example of a model checking problem: P holds globally (`AG`) of all executions of $\mathcal{M}_{(\mathcal{R},\mathcal{B})}$. We sketch how the problem can be solved using Hol98+BuDDy .

First, inductively define $\mathcal{S}_n$ to be a predicate that is true of a state s if and only if s is reachable from a state satisfying $\mathcal{B}$ in n or fewer $\mathcal{R}$-steps.

$$\begin{array}{lcl} \mathcal{S}_0(s) & = & \mathcal{B}(s) \\ \mathcal{S}_{n+1}(s) & = & \mathcal{S}_n(s) \ \vee\ (\exists u.\ \mathcal{S}_n(u) \ \wedge\ \mathcal{R}(u, s)) \end{array}$$

The formula $\exists n.\ \mathcal{S}_n(s)$ true if s is reachable from $\mathcal{B}$ via some number of

$\mathcal{R}$-steps.

Now clearly if $\mathcal{M}_{(\mathcal{R},\mathcal{B})}(\sigma)$ then the nth state in the trace σ is reachable in n $\mathcal{R}$-steps, so $\vdash \forall\sigma.\ \mathcal{M}_{(\mathcal{R},\mathcal{B})}(\sigma) \Rightarrow \forall n.\ \mathcal{S}_n(\sigma(n))$.

Now suppose we can show that

$$\forall s\ n.\ \mathcal{S}_n(s) \Rightarrow P(s)$$

then it would follow by specialising s to $\sigma(n)$ that

$$\forall\sigma\ n.\ \mathcal{S}_n(\sigma(n)) \Rightarrow P(\sigma(n))$$

and hence

$$\forall\sigma.\ \mathcal{M}_{(\mathcal{R},\mathcal{B})}(\sigma) \Rightarrow \forall n.\ P(\sigma(n))$$

which is

$$\forall\sigma.\ \mathcal{M}_{(\mathcal{R},\mathcal{B})}(\sigma) \Rightarrow (\mathtt{AG}\ P)(\sigma)$$

This shows that the model checking problem above can be reduced to the state exploration problem $\forall s\ n.\ \mathcal{S}_n(s) \Rightarrow P(s)$. Such a reduction is easily done automatically by deduction in HOL.

To solve the state exploration problem, note the following obvious fixed-point lemma (obvious because the sets corresponding to $\mathcal{S}_i$ increase as i increases).

$$(\mathcal{S}_i(s) = \mathcal{S}_{i+1}(s)) \Rightarrow ((\exists n.\ \mathcal{S}_n(s)) = \mathcal{S}_i(s))$$

The outline deduction below shows how a mixture of BuDDy BDD calculation and HOL proof can be used to deduce $\forall\sigma.\ \mathcal{M}_{(\mathcal{R},\mathcal{B})}(\sigma) \Rightarrow (\mathtt{AG}\ P)(\sigma)$.

Judgement	Justification
$\rho\ \mathcal{S}_0(s) \mapsto b_0$	`termToBdd`
$\vdots$	$\vdots$
$\rho\ \mathcal{S}_{20}(s) \mapsto b_{20}$	`termToBdd`
$\rho\ \mathcal{S}_{21}(s) \mapsto b_{21}$	`termToBdd`
$\rho\ (\mathcal{S}_{20}(s) = \mathcal{S}_{21}(s)) \mapsto b_{20}$ `BIIMP` b_{21}	`BIIMP`
$\vdash \mathcal{S}_{20}(s) = \mathcal{S}_{21}(s)$	`bddOracle` (assuming b_{20} `BIIMP` b_{21} is `TRUE`)
$\vdash (\mathcal{S}_{20}(s) = \mathcal{S}_{21}(s)) \Rightarrow (\exists n.\ \mathcal{S}_n(s)) = \mathcal{S}_{20}(s)$	Fixed-point lemma ($i = 20$)
$\vdash (\exists n.\ \mathcal{S}_n(s)) = \mathcal{S}_{20}(s)$	Modus Ponens
$\rho\ (\exists n.\ \mathcal{S}_n(s)) \mapsto b_{20}$	`addEquation`
$\rho\ P(s) \mapsto b_P$	`termToBdd`
$\rho\ (\exists n.\ \mathcal{S}_n(s)) \Rightarrow P(s) \mapsto b_{20}$ `IMP` b_P	`IMP`
$\vdash (\exists n.\ \mathcal{S}_n(s)) \Rightarrow P(s)$	`bddOracle` (assuming b_{20} `IMP` b_P is `TRUE`)
$\vdash \forall n.\ \mathcal{S}_n(s) \Rightarrow P(s)$	Deduction
$\vdash \forall \sigma.\ \mathcal{M}_{(\mathcal{R},\mathcal{B})}(\sigma) \Rightarrow (\mathtt{AG}\ P)(\sigma)$	Argument given above

Such combinations of deduction and BDD calculation illustrate how symbolic model checking can be implemented in Hol98+BuDDy .

3.4 Miscellaneous BDD rules

The semantics of the BuDDy operations described in Section 2.7 can be succinctly specified using BDD judgements.

In the rule `restrict` below c_1, ..., c_n are Boolean constants (i.e. `T` or `F`), u_1, ..., u_n are variables, $t[u_1, \ldots, u_p]$ denotes a term containing u_1, ..., u_n and $t[c_1, \ldots, c_p]$ denotes the term obtained by substituting c_i for u_i $(1 \leq i \leq p)$.

`restrict`

$$\frac{\rho\ t[u_1,\ldots,u_p] \mapsto b \qquad \rho\ u_1 \mapsto n_1 \qquad \cdots \qquad \rho\ u_p \mapsto n_p}{\rho\ t[c_1,\ldots,c_p] \ \mapsto\ \texttt{restrict(makeRestriction[}(n_1,c_1),\ldots,(n_p,c_p)\texttt{])}\ b}$$

The rule `simplify` below is experimental. Note that this rule is currently *not* available in Hol98+BuDDy as an oracle (the only oracle is `bddOracle`). The ML function `simplify`, as described in Section 2.7, is available for constructing BDDs, but it is not clear how it can be used in conjunction with HOL (adding the rule `simplify` as an additional oracle would be a possibility, but this requires further thought).

`simplify`

$$\frac{\rho\ t_1 \mapsto b_1 \qquad \rho\ t_2 \mapsto b_2 \qquad \rho\ t \mapsto \texttt{simplify}\ b_1\ b_2}{\vdash t_1 \Rightarrow (t_2 = t)}$$

4 Introduction to `HolBddLib`

The library `HolBddLib` enables Boolean HOL terms to be easily represented as BDDs and then BDD calculations to be mixed with HOL deduction. The implementation uses the generalised 'LCF-like' or 'fully expansive' approach described above in Section 3.

`HolBddLib` currently contains two structures: `HolBdd` which has general tools for connecting HOL and BuDDy, and `StateEnum` (which uses `HolBdd`) which defines some simple symbolic state enumeration programs.

Loading `HolBddLib` first loads `HolBdd` (which loads the BuDDy structures `bdd`, `fdd` and `bvec`), next `StateEnum` is loaded, and finally BuDDy is initialised with a nodesize of 1000000 and cachesize of 10000. If you want to perform your own BuDDy initialisation with different values, then instead of loading `HolBddLib`, load `StateEnum` and then call `bdd.init`.

The rest of this section describes and documents the tools in `HolBdd`.

Section 5 presents a simple use of the `HolBdd` tools, together with some timing data.

Section 6 is a tutorial introduction to using `HolBdd` for simple symbolic state enumeration. It is illustrated with running examples.

Section 7 documents the modest state enumeration tools that are pre-defined in `StateEnum`.

4.1 Representing HOL terms as BDDs: `termToBdd`

The functions

```
termToBdd     : term -> bdd
pureTermToBdd : term -> bdd
```

try to represent a HOL term as a BDD using a variable ordering held in an extendable datastructure[9], called the variable map, that maps HOL variables to integers. At the start of a session using `HolBddLib` the user can explicitly declare a variable ordering (using the function `initHolBdd` described below). Without such a declaration, the system orders variables in the order in which they are encountered: a side-effect of a call to `termToBdd` is to add any previously unseen variables to the variable map (the exact order is implementation dependent, but is normally left to right).

The difference between `pureTermToBdd` and `termToBdd` is that the latter makes use of a dynamically extendable global table that maps HOL terms to BDDs. This table is called the *BDD map* and is described later. Evaluating `TermToBdd` t will attempt to construct a BDD of t using any BDDs of subterms of t that are stored in the table. This enables hierarchical specification to be efficiently managed: the BDDs of components are precomputed and stored in the table, then if t represents a combination of instances of these components, its BDD is easily computed from their BDDs.

If t is a pure quantified Boolean formula, then it may be more efficient to compute its BDD directly by `pureTermToBdd` t, which does not try to find precomputed BDDs of subterms of t in the table.

A simplified ML pseudo-code description of the algorithm used by `termToBdd` is given below. The algorithm used by `pureTermToBdd` is the same as that for `termToBdd`, but without the first two conditionals.

In the pseudo-code, if *op* is a HOL logical operator (e.g. $\wedge$) then $\overline{op}$ is the corresponding BDD operator from the table on page 11 (e.g. `And`). The equality operator $=$ only corresponds to `Biimp` when its arguments are Boolean (see 8th, 10th and 12th conditional in pseudo-code below). The equality of pairs of Booleans is reduced to the conjunction of the equality of the components (see 9th conditional).

[9] Actually a binary map `http://www.dina.kvl.dk/~sestoft/mosmllib/Binarymap.html`.

```
fun termToBdd t =
 if t is in the table
  then return corresponding BDD else
 if t matches a term in table
  then return corresponding BDD instance else
 if t is a new variable
  then add variable to the variable map and then
       return ithvar n for appropriate n else
 if t is variable number n in variable map
  then ithvar n else
 if t = (λv. t1)t2
  then compose (termToBdd t1) (termToBdd t2) (termToBdd v) else
 if t = T
  then TRUE else
 if t = F
  then FALSE else
 if t = ¬t1
  then NOT(termToBdd t1) else
 if t = ((t1,t2) = (t3,t4))
  then AND(termToBdd (t1 = t3),termToBdd (t2 = t4)) else
 if t = t1 op t2
  then apply (termToBdd t1) (termToBdd t2) op̄ else
 if t = (t1 → t2 | t3)
  then let val (b1,b2,b3) = (termToBdd t1,termToBdd t2,termToBdd t3)
       in OR(AND(b1,b2),AND(NOT(b1),b3)) end else
 if t = ∃x1 ... xn. t1 op t2
  then appex (termToBdd t1) (termToBdd t2) op̄ {x1,...,xn} else
 if t = ∃x1 ... xn. t1
  then exist {x1,...,xn} (termToBdd t1) else
 if t = ∀x1 ... xn. t1 op t2
  then appall (termToBdd t1) (termToBdd t2) op̄ {x1,...,xn} else
 if t = ∀x1 ... xn. t1
  then forall {x1,...,xn} (termToBdd t1)  else
  raise holToBddError
```

The phrase '*return corresponding BDD instance*' used in the pseudo-code is implemented by a fairly complex and somewhat heuristic mixture of HOL deduction and BuDDy operations. For example, `replace` can only replace distinct variables with distinct variables. Suppose we have in the BDD map an entry for `Foo(u,v,w,x,y,z)` and we want to compute the BDD of `Foo(a,b,p,q,p,(x/\y))` (note that `p` is repeated). The current implementation of `termToBdd` uses a special HOL conversion (`BDD_CONV`) to rewrite this to

```
?y' z. ((y' = p) /\ (z = x /\ y)) /\ Foo (a, b, p, q, y', z)
```

and then computes the BDD of this. Note that in the rewritten term `Foo` is applied to distinct variables. Setting the ML reference `BDD_CONV_flag` to `true` will cause theorems generated by `BDD_CONV` to be printed.

It is expected that in the future the definition of `termToBdd` will be further tuned to better exploit combinations of `replace` and `compose`, and possibly other BDD-building algorithms available in BuDDy, but for the time being performance seems adequate.

Note that `termToBdd` and `pureTermToBdd` may have the side-effect of extending the variable map.

The function `bddToTerm` is total and creates a nested conditional corresponding to a BDD.

```
bddToTerm : bdd -> term
```

The BDD map can be manipulated explicitly by the user or by verification programs. The table uses the ML structure `Polyhash` to hash terms to BDDs that represent them. It also uses the HOL structure `Net` to index terms so that it can be efficiently determined whether a given term is an instance of a term that already has an associated BDD. This is used in the 2nd conditional of the `termToBdd` pseudo-code. The ML data structure `bdd_map` is a hash table paired with a term net:

```
type bdd_map = (term, robdd)hash_table * (term)net
```

During a session a reference to a pair consisting of the variable map and the BDD map is maintained. This pair is called the *BDD table* and is initialised with the function

```
initHolBdd : (string)list -> unit
```

Evaluating `initHolBdd`$[x_0, x_1, \ldots, x_n]$ initializes the variable map with the variables ordered as listed (i.e. $x_i \mapsto i$ for $0 \leq i \leq n$). It also initializes the BDD map to have an empty hash table and term net.

A new entry is added to the BDD table using the function

```
addEquation : thm -> (term * bdd)
```

Evaluating `addEquation`($\vdash t_1 = t_2$) applies `termToBdd` to t_2 to compute a BDD, b_2 say, and then puts $\{t_1 \mapsto b_2\}$ into the BDD map (i.e. hashes t_1 to b_2 and enters t_1 in the net). Note that the call of `termToBddd` may extend the variable map. An exception is raised by `addEquation` if it is not applied to an equation or if `termToBdd` fails.

When `termToBdd` t is evaluated, the BDD hashed to t is returned, if one exists. If not, then a list of terms with pre-computed BDDs and that can be instantiated to t is obtained from the net. A simple heuristic[10] is used that aims to select from the list the term whose BDD requires least work to instantiate to t; the resulting instantiated BDD is then returned.

The BDD a term hashes to can be removed from the BDD table using the function

```
deleteBdd : term -> bdd
```

Removing terms from the BDD table may enable the BuDDy garbage collector to reclaim space. Note that not all side effects resulting from adding a term to the BDD table are undone by `deleteBdd` – in particular, any extensions to the global variable ordering made when the term was added will persist.

4.2 Inspecting the variable and BDD maps

The following functions return, respectively, the current variable map and BDD map.

```
showVarMap : unit -> (string * int)list
showBddMap : unit -> (term * bdd)list
```

The current variable order is returned by

```
showVarOrd :  unit -> string list
```

which is just defined by

```
fun showVarOrd() =
 map fst (sort (fn(_,m)=>fn(_,n)=>m<n) (showVarMap()));
```

[10]Initial experiments show that the heuristic used works well, but further experiments and performance analysis might lead to a better implementation (see Section 7.4).

4.3 Various BDD-based checking functions

Tautology checkers

```
tautCheck     : term -> (bool)option
pureTautcheck : term -> (bool)option
```

can be defined by

```
fun tautCheck tm =
 SOME(toBool(termToBdd tm))     handle _ => NONE
and pureTautCheck tm =
 SOME(toBool(pureTermToBdd tm)) handle _ => NONE
```

Evaluating `tautCheck` t or `pureTautCheck` t returns `SOME true` if the BDD of t is a tautology (i.e. is `TRUE`), returns `SOME false` if the BDD of t is a contradiction (i.e. is `FALSE`) and returns `NONE` otherwise (i.e. if the construction of the BDD of t fails, or the BDD is neither a tautology or contradiction).

A term can be tested for equivalence to `T` or `F`:

```
isT : term -> bool
isF : term -> bool
```

These are defined by

```
fun isT tm =
 case tautCheck tm of SOME true => true | _ => false;
```

and

```
fun isF tm =
 case tautCheck tm of SOME false => true | _ => false;
```

The functions `isT` and `isF` do not raise an exception if the BDD of the argument can't be computed: they just return `false`. The equivalence of two terms can be tested by

```
eqCheck : term * term -> bool
```

The definition is simply:

```
fun eqCheck(t1,t2) = isT(mk_eq(t1,t2));
```

Note that these function do not create HOL theorems, merely return ML Booleans.

4.4 Finding models and refutations

The functions

```
findModel      : term -> term
findRefutation : term -> term
```

try to find a conjunction of the variables or negated variables occuring in a term that makes it true (`findModel`) or false (`findRefutation`). Thus, for a suitable term *tm*, it should be the case that

```
|- ^(findModel tm) ==> tm
|- ^(findRefutation tm) ==> ~tm
```

For example, applied to `(x /\ y) \/ (~x /\ z)`, the function `findModel` returns `~x /\ z` and `findRefutation` returns `~z /\ ~x`. Exceptions are raised on tautologies and contradictions.

4.5 The oracle `bddOracle`

The oracle function

```
bddOracle : term -> thm
```

returns the theorem $\vdash t$, if `termToBdd` t successfully evaluates to the BDD representing T.

This function is the only way that HOL theorems can be created via BuDDy.

Theorems created using `bddOracle` are tagged with `"BDD"` and the Hol98 tagging mechanism propagates this tag to any theorems deduced from results

of `bddOracle`, so that the provenance of theorems is explicit (i.e. whether they are proved using only the rules of higher order logic, or proved from theorems created using BuDDy).

To use BuDDy to prove t_1 and t_2 equivalent, it is sufficient to evaluate `bddOracle` $(t_1 = t_2)$. The following function does this

```
bddEqOracle : term * term -> thm
```

The definition is simply

```
fun bddEqOracle(t1,t2) = bddOracle(mk_eq(t1,t2));
```

A special case is proving that a term is equal to the HOL representation of its BDD:

```
bddRepOracle : term -> thm
```

The definition is simply

```
fun bddRepOracle t = bddEqOracle(t, bddToTerm(termToBdd t));
```

4.6 Printing BDDs with variable names

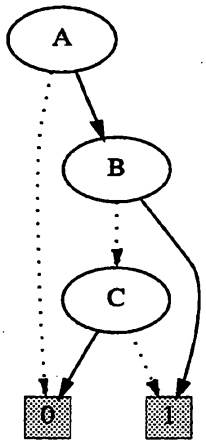
The BDD state contains, via the variable map, the encoding of variables as numbers. The function `showTerm` enables the BDDs of HOL terms to be displayed with nodes labelled with variables.

```
showTerm : term -> unit
```

Evaluating `showTerm` t has the side-effect of writing a file scratchBDD.ps in the current directory containing a diagram of the BDD of t drawn with the `dot` program. For example

```
showTerm ``A /\ (B \/ ~C)``
```

writes a file `scratchBDD.ps` containing the following BDD diagram



`showTerm` is a special case of a more general function

```
termToBddDiagram :
  string -> string -> term -> (bdd * (string * int)list)
```

Evaluating `termToBddDiagram` *filename label t* writes a file called *filename*`.ps` containing a diagram of the BDD of t labelled with the string *label*. The BDD of t is returned, together with the variable map used to convert the BuDDy node labels to HOL variable names.

5 Example: tautology checking

As a simple example, consider disjunctive normal form tautologies created by the following program

```
fun DNF n = let fun sum_of_prod_terms 0 =
                    let val v = mk_var("v"^(int_to_string 0), bool)
                    in
                     [[v],[mk_neg v]]
                    end
              | sum_of_prod_terms n =
                    let val v = mk_var("v"^(int_to_string n), bool)
                        val dnf = sum_of_prod_terms(n-1)
                    in
                     map (cons v) dnf) @ (map (cons(mk_neg v)) dnf
                    end
            in
             list_mk_disj(map list_mk_conj (sum_of_prod_terms n))
            end
```

For example, `DNF 1` is $(v1 \wedge v0) \vee (v1 \wedge \neg v0) \vee (\neg v1 \wedge v0) \vee (\neg v1 \wedge \neg v0)$. Note that `DNF` n is a term containing $2^{n+1} \times (n+1)$ distinct literals (variables or negated variables).

The table shows the runtime and garbage collection time (GC) of creating `DNF` n, a HOL term, and then the additional times to check that it is a tautology using `TAUT` (Hol98's built-in non BDD-based tautology checker), `tautCheck` and `pureTautCheck`.

Times are also given for a bespoke ML program `bddDNF` that builds the BDD of the DNF directly using `bdd` operations (i.e. it doesn't go via HOL terms). Note that once the BDD is built it will just be `TRUE`, so there is no significant additional time needed to check that it's a tautology!

The definition of `bddDNF` is

```
fun bddDNF n =
 (if not(getVarnum() > n) then setVarnum(n+1) else ();
  let
   fun sum_of_prod_bdds 0 = [[ithvar 0], [nithvar 0]]
    |  sum_of_prod_bdds n =
        let val v   = ithvar n
            val vn  = nithvar n
            val dnf = sum_of_prod_bdds(n-1)
        in
         map (cons v) dnf @ map (cons vn) dnf
        end
  in
   foldl OR FALSE (map (foldl AND TRUE) (sum_of_prod_bdds n))
  end)
```

The times in the following table are rounded to the nearest second.[11]

[11] 333MHz Intel Pentium II (Deschutes) processor with 387780 KB of memory, running Linux.

n	DNF		TAUT		tautCheck		pureTautCheck		bddDNF	
	Run	GC	Run	GC	Run	GC	Run	GC	Run	GC
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	2	0	0	0	0	0	0	0
5	0	0	13	0	0	0	0	0	0	0
6	0	0	89	2	0	0	0	0	0	0
7	0	0	668	9	1	0	0	0	0	0
8	0	0	5573	58	2	0	0	0	0	0
9	1	0			9	1	0	0	0	0
10	2	0			34	7	1	0	0	0
11	4	1			147	48	2	0	0	0
12	9	2			703	320	5	1	1	0
13	21	7			3733	2235	12	3	2	1
14	57	27			23142	17217	32	12	4	2
15	175	111			156574	133044	88	46	10	6
16	590	453					280	183	25	16

As n increases the work done by `tautCheck` becomes increasingly dominated by lookups in the BDD table and hence `pureTautCheck`, which omits this lookup, is faster. The function `bddDNF` doesn't build any HOL terms and just builds BDDs directly[12].

The table above illustrate why building large terms and then converting them to BDDs is to be avoided. As long as BDDs are built hierarchically out of small components, with the hierarchy represented in the BDD map, then terms in HOL representing large BDDs in BuDDy can be built incrementally inside HOL. Fortunately, complex designs are often built hierarchically out of components!

If large BDDs are to be built non-incrementally, then they should be constructed in BuDDy space by directly calling the `bdd` functions, as in done by `bddDNF`. If this is done then the resulting BDD can be associated with a term in the BDD table, however care must be taken that the BDD is a valid representation of the term. This kind of purely external creation of BDDs is not explored here, though `HolBddLib` provides the necesssary infrastructure to support it. In what follows, all entries in the BDD table are created using `addEquation`, which ensures the table's soundness.

[12]Note that the resultant BDD is just TRUE. The point is the time taken to build the BDD, not the size of the result!

6 Example: computing reachable states

In this section a complete ML program to compute the set of reachable states of a transition system is described. This is intended to be a tutorial illustration of how model checkers can be written in Hol98+BuDDy .

The state enumeration tools provided by `HolBddLib` are documented in Section 7. The programs in this section are designed to illustrate ideas and are simplified.

The theory `HolBddTheory` contains HOL definitions and theorems that provide a formal basis for representing transition systems.

6.1 A partial description of `HolBddTheory`

Definitions and theorems are shown in a format similar to that used in `HolBddTheory.sig`. The statement of a definition or theorem is followed by a brief explanation.

```
[Next_def]
Definition
|- !R B state. Next R B state =
                ?state_. B state_ /\ R (state_,state)
```

This is the definition of the constant `Next`. The ML name of the definition, shown in square brackets, is `Next_def`. The definition states that `Next R B state` is true if `state` is reachable in one `R`-step from a state `state_` for which `B` is true. Sets of states are represented by predicates and transition relations by predicates on pairs of states. Thus `Next R B` represents the image of the set represented by `B` under relation `R`.

```
[ReachIn_def]
Definition
|- (!R B. ReachIn R B 0 = B) /\
   (!R B n. ReachIn R B (SUC n) = Next R (ReachIn R B n))
```

`ReachIn R B n` is defined by primitive recursion to be the set of states reachable in exactly `n` `R`-steps.

```
[ReachIn_rec]
Theorem
|- (!R B state. ReachIn R B 0 state = B state) /\
   (!R B n state.
      ReachIn R B (SUC n) state =
      ?state_. ReachIn R B n state_ /\ R (state_,state))
```

This theorem, which is used below, is just the result of unfolding the definition of `Next` in `ReachIn_def`.

```
[ReachBy_def]
Definition
|- !R B n state.
     ReachBy R B n state = ?m. m <= n /\ ReachIn R B m state
```

`ReachBy R B n` is defined to be the set of states reachable in `n` or fewer R-steps.

```
[ReachBy_rec]
Theorem
|- (!R B state. ReachBy R B 0 state = B state) /\
   (!R B n state.
      ReachBy R B (SUC n) state =
      ReachBy R B n state \/
      ?state_. ReachBy R B n state_ /\ R (state_,state))
```

This theorem could have been taken as a primitive-recursive definition of `ReachBy`. It shows how to compute `ReachBy n R B` for successive values of `n`.

```
[Reachable_def]
Definition
|- !R B state. Reachable R B state = ?n. ReachIn R B n state
```

`Reachable R B` is defined to be the set of states reachable in some finite number of R-steps. It is thus the set of reachable states.

```
[ReachBy_fixedpoint]
Theorem
|- !R B n.
     (ReachBy R B n = ReachBy R B (SUC n)) ==>
     (Reachable R B = ReachBy R B n)
```

The theorem `ReachBy_fixedpoint` shows that the set of all reachable states has been reached as soon as the set reachable in `n` steps is the same as the set reached in `SUC n` steps. Thus `Reachable R B` can be computed by iteratively computing `ReachByRB1`, `ReachByRB2`, ... until the sets stop increasing.

6.2 Programming symbolic state enumeration

The 2-bit binary counter from McMillan's book [10, Page 29,3.1] will be used as a first running example. The material shown in boxes with numbers in their top right hand corners constitute a continuous session. Sometimes the output from Hol98 is edited, for example, printed terms may be reformatted to fit in the boxes and ML types may be deleted. Voluminous output that is not illuminating is usually deleted. Any output shown is indicated with > at its start.

The first step is to load and open `HolBddLib`.

1

```
- load "HolBddLib"; open HolBddLib;
```

The counter is defined by giving its transition relation as a predicate `Count` on pairs of states

2

```
- val Count_def =
   bossLib.Define
    `Count((v0,v1),(v0',v1')) =
      (v0' = ~v0) /\ (v1' = ~(v0 = v1))`;
Definition stored under "Count_def".
> val Count_def =
    |- !v0 v1 v0' v1'.
         Count ((v0,v1),v0',v1') =
          (v0' = ~v0) /\ (v1' = ~(v0 = v1))
```

and specifying the set of initial states with a predicate `Zero` on states

```
- val Zero_def = bossLib.Define `Zero(v0,v1) = ~v0 /\ ~v1`;
Definition stored under "Zero_def"
> val Zero_def = |- !v0 v1. Zero (v0,v1) = ~v0 /\ ~v1
```
3

In general, a state transition system is specified by a pair of theorems, `(Rth,Bth)` say, of the form

$$(\vdash \mathcal{R}((v_1,\ldots,v_p),(v'_1,\ldots,v'_p)) = t_{\mathcal{R}},\ \vdash \mathcal{B}(v_1,\ldots,v_p) = t_{\mathcal{B}})$$

defining a transition relation $\mathcal{R}$ and set of initial states $\mathcal{B}$.

These theorems could be definitions, or equations derived by proof. In the example $\mathcal{R}$ is `Count` and $\mathcal{B}$ is `Zero`.

The ML programs described here are simpler than the ones in the system (e.g. less error trapping, more rigid requirements on the form of the supplied theorems, no output during calculation).

We are going to write an ML function

```
ComputeReachable : thm * thm -> thm
```

that takes a pair `(Rth,Bth)` and returns a theorem of the form

$$\texttt{|- Reachable R B } (v_1,\ldots,v_p) \texttt{ = ReachBy R B i } (v_1,\ldots,v_p)$$

`ComputeReachable` works by computing the BDDs of `ReachBy R B` n for successive values of n until a fixed point ($n =$ `i`) is reached and then using the theorem `ReachBy_fixedpoint`.

To prepare for computing the reachable states, the theorem `ReachBy_rec` needs to be instantiated using the specifications `Rth` and `Bth`. The function `MakeIterRthms` defined below does this instantiation.

4

```
- fun MakeIterThms iter_th (Rth,Bth) =
   let
    val (R,_)  = dest_comb(lhs(concl(SPEC_ALL Rth)))
    val (B,s)  = dest_comb(lhs(concl(SPEC_ALL Bth)))
    val ntm    = ``n:num``
    val th_0   = ISPECL[R,B,s](CONJUNCT1 iter_th)
    val th_SUC = GEN ntm
                  (Ho_rewrite.REWRITE_RULE
                    [pairTheory.EXISTS_PROD]
                    (ISPECL[R,B,ntm,s](CONJUNCT2 iter_th)))
   in
    (th_0,th_SUC)
   end;
> val MakeIterThms = fn : thm -> thm * thm -> thm * thm
```

The effect of MakeIterThms is illustrated by

5

```
- val (ReachIn_0,ReachIn_SUC) =
       MakeIterThms HolBddTheory.ReachIn_rec (Count_def,Zero_def);
  val ReachIn_SUC =
    |- !n.
         ReachIn Count Zero (SUC n) (v0,v1) =
         ?p_1 p_2. ReachIn Count Zero n (p_1,p_2)
                   /\
                   Count ((p_1,p_2),v0,v1)

- val (ReachBy_0,ReachBy_SUC) =
       MakeIterThms HolBddTheory.ReachBy_rec (Count_def,Zero_def);
> val ReachBy_0 =
    |- ReachBy Count Zero 0 (v0,v1) = Zero (v0,v1)
  val ReachBy_SUC =
    |- !n.
         ReachBy Count Zero (SUC n) (v0,v1) =
         ReachBy Count Zero n (v0,v1) \/
         ?p_1 p_2. ReachBy Count Zero n (p_1,p_2)
                   /\
                   Count ((p_1,p_2),v0,v1)
```

The main work done by ComputeReachable is an iteration to find an i such that ReachBy R B 1 equals ReachBy R B (SUC i). The iteration can be

done manually to illustrate the process. First, compute the BDDs of the initial set of states and the transition relation.

6

```
- addEquation Zero_def;
> val it = (‘Zero (v0,v1)‘, <bdd>) : term * bdd
- addEquation Count_def;
> val it = (‘Count ((v0,v1),v0',v1')‘, <bdd>) : term * bdd
```

Next, compute the BDD of `ReachBy Count Zero 0 (v0,v1)`.

7

```
- addEquation ReachBy_0;
> val it = (‘ReachBy Count Zero 0 (v0,v1)‘, <bdd>)
```

Compute the BDD of `ReachBy Count Zero (SUC 0) (v0,v1)` by specialising `n` in `ReachBy_SUC` to 0.

8

```
- SPEC ‘‘0‘‘ ReachBy_SUC;
> val it =
    |- ReachBy Count Zero (SUC 0) (v0,v1) =
       ReachBy Count Zero 0 (v0,v1) \/
       ?p_1 p_2. ReachBy Count Zero 0 (p_1,p_2)
                 /\
                 Count ((p_1,p_2),v0,v1)
- addEquation it;
> val it = (‘ReachBy Count Zero (SUC 0) (v0,v1)‘, <bdd>)
```

Check whether the fixed point has been reached:

9

```
- isT ‘‘ReachBy Count Zero 0 (v0,v1) =
        ReachBy Count Zero (SUC 0) (v0,v1)‘‘;
> val it = false : bool
```

The fixed point has not been reached.

Compute the BDD of `ReachBy Count Zero (SUC(SUC 0)) (v0,v1)` by specialising `n` in `ReachBy_SUC` to `SUC 0`.

10

```
- SPEC ``SUC 0`` ReachBy_SUC;
> val it =
    |- ReachBy Count Zero (SUC(SUC 0)) (v0,v1) =
       ReachBy Count Zero (SUC 0) (v0,v1) \/
       ?p_1 p_2.
         ReachBy Count Zero (SUC 0) (p_1,p_2)
         /\
         Count ((p_1,p_2),v0,v1)
- addEquation it;
> val it =
    (`ReachBy Count Zero (SUC(SUC 0)) (v0,v1)`, <bdd>)
```

Check whether fixed point reached:

11

```
- isT ``ReachBy Count Zero (SUC 0) (v0,v1) =
         ReachBy Count Zero (SUC(SUC 0)) (v0,v1)``;
> val it = false : bool
```

The fixed point has not been reached.
Compute the BDD of `ReachBy Count Zero (SUC(SUC(SUC 0))) (v0,v1)`.

12

```
-  SPEC ``SUC(SUC 0)`` ReachBy_SUC;
> val it =
    |- ReachBy Count Zero (SUC(SUC(SUC 0))) (v0,v1) =
       ReachBy Count Zero (SUC(SUC 0)) (v0,v1) \/
       ?p_1 p_2.
         ReachBy Count Zero (SUC(SUC 0)) (p_1,p_2)
         /\
         Count ((p_1,p_2),v0,v1)
- addEquation it;
> val it =
    (`ReachBy Count Zero (SUC(SUC(SUC 0))) (v0,v1)`, <bdd>)
```

Check for fixed point:

13

```
- isT ``ReachBy Count Zero (SUC(SUC 0)) (v0,v1) =
         ReachBy Count Zero (SUC(SUC(SUC 0))) (v0,v1)``;
> val it = false : bool
```

The fixed point has still not been reached, so iterate.

14

```
- SPEC ``(SUC(SUC(SUC 0)))`` ReachBy_SUC;
> val it =
    |- ReachBy Count Zero (SUC(SUC(SUC(SUC 0)))) (v0,v1) =
       ReachBy Count Zero (SUC(SUC(SUC 0))) (v0,v1) \/
       ?p_1 p_2.
         ReachBy Count Zero (SUC(SUC(SUC 0))) (p_1,p_2)
         /\
         Count ((p_1,p_2),v0,v1)
- addEquation it;
> val it =
    (`ReachBy Count Zero (SUC(SUC(SUC(SUC 0)))) (v0,v1)`, <bdd>)
```

Check for fixed point:

15

```
- isT ``ReachBy Count Zero (SUC(SUC(SUC 0))) (v0,v1) =
         ReachBy Count Zero (SUC(SUC(SUC(SUC 0)))) (v0,v1)``;
> val it = true : bool
```

The fixed point has now been reached and a justifying theorem can be derived using `bddOracle`.

16

```
- bddOracle
   ``ReachBy Count Zero (SUC(SUC(SUC 0))) (v0,v1) =
      ReachBy Count Zero (SUC(SUC(SUC(SUC 0)))) (v0,v1)``;
> val it =
    |- ReachBy Count Zero (SUC(SUC(SUC 0))) (v0,v1) =
       ReachBy Count Zero (SUC(SUC(SUC(SUC 0)))) (v0,v1)
```

Rather than interactively computing the fixed point, as laboriously done above, the ML procedure `iterate` defined below can be used to do the iterations.

17

```
- fun iterate suc_thm ntm rtm =
   let val th   = SPEC ntm suc_thm
       val rtm' = lhs(concl th)
   in addEquation th;
      if eqCheck(rtm,rtm')
       then bddEqOracle(rtm,rtm')
       else iterate suc_thm ``SUC ^ntm`` rtm'
   end;
> val iterate = fn : thm -> term -> term -> thm
```

Invoking `iterate` computes the fixed point and returns the justifying theorem.

18

```
iterate ReachBy_SUC ``0`` ``ReachBy Count Zero 0 (v0,v1)``;
> val it =
    |- ReachBy Count Zero (SUC(SUC(SUC 0))) (v0,v1) =
       ReachBy Count Zero (SUC(SUC(SUC(SUC 0)))) (v0,v1)
```

Recall that `ReachBy_fixedpoint` is an implication between predicate equations.

```
[ReachBy_fixedpoint]
Theorem
|- !n R B.
     (ReachBy R B n = ReachBy R B (SUC n))
     ==>
     (Reachable R B = ReachBy R B n)
```

To use `ReachBy_fixedpoint` on the theorem just computed, the variables `v0` and `v1` can be cancelled by extensionality. The rule `HolBddLib.PGEN_EXT` does this,

19

```
- PGEN_EXT it;
> val it =
    |- ReachBy Count Zero (SUC(SUC(SUC 0))) =
       ReachBy Count Zero (SUC(SUC(SUC(SUC 0))))
```

Applying `MATCH_MP` to `ReachBy_fixedpoint` and the theorem just proved gives an equation for `Reachable Count Init`.

```
- val Reach_thm = MATCH_MP HolBddTheory.ReachBy_fixedpoint it;
> val Reach_thm =
    |- Reachable Count Zero =
       ReachBy Count Zero (SUC(SUC(SUC 0)))
```
20

The function ComputeReachable can now be defined

```
- fun ComputeReachable(Rth,Bth) =
   let val (ReachBy_0,ReachBy_SUC) =
         MakeIterThms HolBddTheory.ReachBy_rec (Rth,Bth)
       val reach0_tm = lhs(concl ReachBy_0)
       val vars = rand reach0_tm
   in
    addEquation Rth;
    addEquation Bth;
    addEquation ReachBy_0;
    let val fix_thm = iterate ReachBy_SUC ``0`` reach0_tm
    in
     MATCH_MP HolBddTheory.ReachBy_fixedpoint (PGEN_EXT fix_thm)
    end
   end;
> val ComputeReachable = fn : thm * thm -> thm
```
21

Using this function the fixed point is easily computed.

```
- ComputeReachable(Count_def,Zero_def);
> val it =
    |- Reachable Count Zero = ReachBy Count Zero (SUC(SUC(SUC 0)))
```
22

The definition of ComputeReachable can be improved to use numerals, rather than SUC(···(SUC 0)···). Observe that:

```
- CONV_RULE(TOP_DEPTH_CONV reduceLib.SUC_CONV) it;
> val it = |- Reachable Count Zero = ReachBy Count Zero 3
```
23

In addition, BDDs that are no longer needed can be deleted. The function delBdd applies deleteBdd if a flag deleteBdd_flag is set.

24

```
- val deleteBdd_flag = ref true;
> val deleteBdd_flag = ref true : bool ref
- fun delBdd tm =
   if !deleteBdd_flag then (deleteBdd tm;()) else ();
> val delBdd = term -> unit
```

A version of `iterate` with these two optimisations is given below. Note that `iterate` now takes an ML integer as argument, rather than a term representing a HOL numeral. The conversion from integers to terms is done using `intToTerm`. So that progress can be monitored, `iterate` prints out a dot each time it is called.

25

```
- fun iterate suc_thm n rtm =
   let val ntm  = intToTerm n
       val th   = CONV_RULE
                    (TOP_DEPTH_CONV reduceLib.SUC_CONV)
                    (SPEC ntm suc_thm)
       val rtm' = lhs(concl th)
   in addEquation th;
      print ".";
      if eqCheck(rtm,rtm')
       then bddEqOracle(rtm,rtm')
       else (delBdd rtm; iterate suc_thm (n+1) rtm')
   end;
> val iterate = fn : thm -> int -> term -> thm
- iterate ReachBy_SUC 0 ``ReachBy Count Zero 0 (v0,v1)``;
....> val it =
    |- ReachBy Count Zero 3 (v0,v1) =
       ReachBy Count Zero 4 (v0,v1)
```

The definition of `ComputeReachable` needs to be adjusted to work with the new version of `iterate`.

26

```
- fun ComputeReachable(Rth,Bth) =
   let val (ReachBy_0,ReachBy_SUC) =
             MakeIterThms HolBddTheory.ReachBy_rec (Rth,Bth)
       val reach0_tm = lhs(concl ReachBy_0)
   in
    addEquation Rth;
    addEquation Bth;
    addEquation ReachBy_0;
    let val fix_th1 = iterate ReachBy_SUC 0 reach0_tm
        val fix_th2 = (* evaluate ``SUC n'' to numeral *)
             CONV_RULE
              ((RHS_CONV o RATOR_CONV o RAND_CONV)
               numLib.num_CONV)
              fix_th1
    in
     MATCH_MP HolBddTheory.ReachBy_fixedpoint (PGEN_EXT fix_th2)
    end
   end;
> val ComputeReachable = fn : thm * thm -> thm
- ComputeReachable(Count_def,Zero_def);
> val it = |- Reachable Count Zero = ReachBy Count Zero 3
```

As all states are reachable, the BDD of `ReachBy Count Zero 3` is just `TRUE`.

27

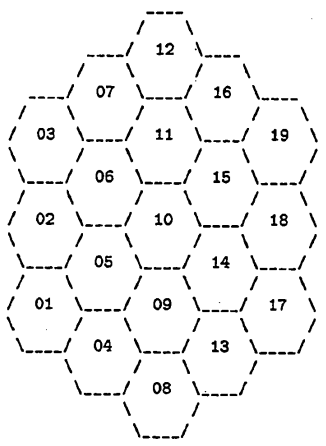
```
- bddToTerm(termToBdd ``ReachBy Count Zero 3 (v0,v1)``);
> val it = ``T`` : term
- bddOracle ``ReachBy Count Zero 3 (v0,v1)``;
> val it = |- ReachBy Count Zero 3 (v0,v1) : thm
```

The example in the next section computes a more interesting set of reachable startes.

6.3 A version of Peg Solitaire

Consider a game like Peg Solitaite played on a board made of hexagons, each with a hole in the middle into which a peg can be inserted:[13]

[13]Do not confuse Peg Solitaire with the card game having the same name!



Initially all the hexagons except the middle one (numbered 10 above) have a peg. A move consists in a peg hopping over an adjacent one into a hexagon whose hole is empty. The peg that is hopped over is removed, leaving a hole. For example, in the initial state 03 could hop over 06 into 10, with 06 being removed. The goal is to devise a sequence of 17 moves that will result in there being only one peg left on the board in the middle hexagon (i.e. at position 10).

Let us investigate this problem using BDDs. The state can be represented by 19 Boolean variables `v01` to `v19`, with a variable being `T` meaning that there is a peg in the corresponding hexagon. The initial state is defined by

28

```
- val HexSolitaireInit_def =
   bossLib.Define
    `HexSolitaireInit
      (v01,v02,v03,v04,v05,v06,v07,v08,v09,v10,
       v11,v12,v13,v14,v15,v16,v17,v18,v19) =
      v01/\v02/\v03/\v04/\v05/\v06/\v07/\v08/\v09
      /\~v10/\
      v11/\v12/\v13/\v14/\v15/\v16/\v17/\v18/\v19`;
```

The transition relation `HexSolitaireTrans(`s`,`s'`)` is quite complex since s and s' are tuples of 19 variables, and there are many possible moves. Rather than write out the definition of the relation explicitly, we'll generate it with an ML program. In the sessions that follow, output is often not shown to save space.

First, functions for constructing Boolean state variables of `v01`, `v02`, ..., `v19` are defined.

29

```
- fun mk_v n =
   if n<10 then mk_var("v0"^(int_to_string n),bool)
           else mk_var("v"^(int_to_string n),bool);
> val mk_v = fn : int -> term
- fun mk_v' n =
   if n<10 then mk_var("v0"^(int_to_string n)^"'",bool)
           else mk_var("v"^(int_to_string n)^"'",bool);
> val mk_v' = fn : int -> term
```

Next, the vectors s and s' of state variables and primed state variables, respectively, are defined.

30

```
- val vl = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19];
- val s  = list_mk_pair(map mk_v vl)
  and s' = list_mk_pair(map mk_v' vl);
> val s =
    ``(v01,v02,v03,v04,v05,v06,v07,v08,v09,v10,
       v11,v12,v13,v14,v15,v16,v17,v18,v19)``
  val s' =
    ``(v01',v02',v03',v04',v05',v06',v07',v08',v09',
       v10',v11',v12',v13',v14',v15',v16',v17',v18',v19')``
```

The function make_move takes a triple of integers (n1,n2,n3) representing a possible move in which a peg at n1 takes a peg at n2 and moves into n3, and returns a transition terms representing the move.

31

```
- fun make_move (n1,n2,n3) =
    let val (_,unchanged) =
          List.partition (fn n => mem n [n1,n2,n3]) vl
    in
     list_mk_conj
      ([mk_v n1,mk_v n2,mk_neg(mk_v n3),
        mk_neg(mk_v' n1),mk_neg(mk_v' n2),mk_v' n3]
       @
       map (fn n => mk_eq(mk_v' n,mk_v n)) unchanged)
    end;
> val make_move = fn : int * int * int -> term
- make_move(03,06,10);
> val it =
    ``v03 /\ v06 /\ ~v10 /\ ~v03' /\ ~v06' /\ v10' /\
      (v01'=v01) /\ (v02'=v02) /\ (v04'=v04) /\ (v05'=v05) /\
      (v07'=v07) /\ (v08'=v08) /\ (v09'=v09) /\ (v11'=v11) /\
      (v12'=v12) /\ (v13'=v13) /\ (v14'=v14) /\ (v15'=v15) /\
      (v16'=v16) /\ (v17'=v17) /\ (v18'=v18) /\ (v19'=v19)``
```

The complete list of moves is

32

```
- val moves =
    [(01,02,03),(01,05,10),(01,04,08),
     (02,06,11),(02,05,09),
     (03,07,12),(03,06,10),(03,02,01),
     (04,05,06),(04,09,14),
     (05,06,07),(05,10,15),(05,09,13),
     (06,11,16),(06,10,14),(06,05,04),
     (07,11,15),(07,06,05),
     (08,04,01),(08,09,10),(08,13,17),
     (09,05,02),(09,10,11),(09,14,18),
     (10,09,08),(10,05,01),(10,06,03),
     (10,11,12),(10,15,19),(10,14,17),
     (11,10,09),(11,06,02),(11,15,18),
     (12,11,10),(12,07,03),(12,16,19),
     (13,09,05),(13,14,15),
     (14,09,04),(14,10,06),(14,15,16),
     (15,14,13),(15,10,05),(15,11,07),
     (16,15,14),(16,11,06),
     (17,13,08),(17,14,10),(17,18,19),
     (18,14,09),(18,15,11),
     (19,18,17),(19,15,10),(19,16,12)];
```

This list was obtained by just inspecting the diagram and manually listing all three position line segments.

The transition relation can now be defined (only the first two of the 54 disjuncts are shown).

33

```
- val HexSolitaireTrans_def =
   bossLib.Define `HexSolitaireTrans(^s,^s') =
    ^(list_mk_disj(map make_move moves))`;
> val HexSolitaireTrans_def =
    |- !v01 v02 v03 v04 v05 v06 v07 v08 v09 v10 v11 v12 v13
       v14 v15 v16 v17 v18 v19 v01' v02' v03' v04' v05' v06'
       v07' v08' v09' v10' v11' v12' v13' v14' v15' v16' v17'
       v18' v19'.
        HexSolitaireTrans
          ((v01,v02,v03,v04,v05,v06,v07,v08,v09,v10,v11,v12,v13,
            v14,v15,v16, v17,v18,v19),
           (v01',v02',v03',v04',v05',v06',v07',v08',v09',v10',
           v11',v12',v13',v14',v15',v16',v17',v18',v19')) =
        v01 /\ v02 /\ ~v03 /\ ~v01' /\ ~v02' /\ v03' /\
        (v04'=v04) /\ (v05'=v05) /\ (v06'=v06) /\ (v07'=v07) /\
        (v08'=v08) /\ (v09'=v09) /\ (v10'=v10) /\ (v11'=v11) /\
        (v12'=v12) /\ (v13'=v13) /\ (v14'=v14) /\ (v15'=v15) /\
        (v16'=v16) /\ (v17'=v17) /\ (v18'=v18) /\ (v19'=v19)
        \/
        v01 /\ v05 /\ ~v10 /\ ~v01' /\ ~v05' /\ v10' /\
        (v02'=v02) /\ (v03'=v03) /\ (v04'=v04) /\ (v06'=v06) /\
        (v07'=v07) /\ (v08'=v08) /\ (v09'=v09) /\ (v11'=v11) /\
        (v12'=v12) /\ (v13'=v13) /\ (v14'=v14) /\ (v15'=v15) /\
        (v16'=v16) /\ (v17'=v17) /\ (v18'=v18) /\ (v19'=v19)
        \/
        ...
```

Now `ComputeReachable` can be called. We use `time` to get the time taken to reach the fixed point (numerous occurences of <$>, which indicate BuDDy garbage collections, have been deleted from the output)

```
- val reach_th =
   time
   ComputeReachable(HexSolitaireTrans_def,HexSolitaireInit_def);
.................
runtime: 694.060s,    gctime: 63.040s,     systime: 0.480s.
> val reach_th =
    |- Reachable HexSolitaireTrans HexSolitaireInit =
       ReachBy HexSolitaireTrans HexSolitaireInit 16
```
34

reach_th is an equation between predicates. It can be easily converted to a Boolean equation, and then the BDD of the set of reachable states computed.

```
- AP_THM reach_th s;
> val it =
    |- Reachable HexSolitaireTrans HexSolitaireInit
         (v01,v02,v03,v04,v05,v06,v07,v08,v09,v10,
          v11,v12,v13,v14,v15,v16,v17,v18,v19) =
       ReachBy HexSolitaireTrans HexSolitaireInit 16
         (v01,v02,v03,v04,v05,v06,v07,v08,v09,v10,
          v11,v12,v13,v14,v15,v16,v17,v18,v19)
- addEquation it;
> val it =
    (``Reachable HexSolitaireTrans HexSolitaireInit
        (v01,v02,v03,v04,v05,v06,v07,v08,v09,v10,
         v11,v12,v13,v14,v15,v16,v17,v18,v19)``, <bdd>)
```
35

Now we can check whether a solution to the puzzle exists. A solution consists in a state in which just one peg is left in the middle of the board (i.e. at position 10). Thus we want to see if

```
(F,F,F,F,F,F,F,F,F,T,F,F,F,F,F,F,F,F,F)
```

is reachable.

```
- isT ``Reachable HexSolitaireTrans HexSolitaireInit
        (F,F,F,F,F,F,F,F,F,T,F,F,F,F,F,F,F,F,F)``;
> val it = false : bool
```
36

Thus there is no solution! Actually, this could already be deduced from the number of dots printed out by `iterate`: since only 17 dots were printed before the fixed point was reached, it is clear that two pegs must be left in all final states.

We did not specify an explicit variable ordering, so variables were automatically ordered in the order in which they were encountered by `termToBdd`. We can obtain this ordering using `showVarOrd`

37

```
- showVarOrd();
> val it =
    ["v01", "v02", "v03", "v01'", "v02'", "v03'", "v04'", "v04",
     "v05'", "v05", "v06'", "v06", "v07'", "v07", "v08'", "v08",
     "v09'", "v09", "v10'", "v10", "v11'", "v11", "v12'", "v12",
     "v13'", "v13", "v14'", "v14", "v15'", "v15", "v16'", "v16",
     "v17'", "v17", "v18'", "v18", "v19'", "v19", "p_1", "p_11",
     "p_12", "p_13", "p_14", "p_15", "p_16", "p_17", "p_18",
     "p_19", "p_110", "p_111", "p_112", "p_113", "p_114",
     "p_115", "p_116", "p_117", "p_2"]
    : string list
```

It turns out that this ordering is fortuitously a good one: a general heuristic is to use an ordering in which state variables are interleaved with next-state (primed) variables, e.g.: `v01` < `v01'` < `v02` < `v02'` < `v03` < `v03'` etc. The order of occurrence of variables in `HexSolitaireTrans_def`, as shown above, is fairly close to this good ordering. If all the variables `v01`, ..., `v19` had all been ordered before any of `v01'`, ..., `v19'`, then the time to compute `reach_th` using `ComputeReachable` would have been nearly ten times longer.

An even better variable ordering could be forced by executing the following before computing the fixed point.

```
fun shuffle (l1,l2) =
 ListPair.foldr (fn(x1,x2,l) => x1 :: x2 :: l) [] (l1,l2);

initHolBdd
 (shuffle
   (map (fst o dest_var o mk_v) vl,
    map (fst o dest_var o mk_v') vl));
```

The effect of this is only to reduce the time to compute the fixed point by about five percent (compared with multiplying it by ten if the really bad ordering were used).

Since the original puzzle is unsolvable, let us try to invent a solvable puzzle by coming up with a new final state that is reachable. To this end we investigate the set of states reached after 16 iterations by computing

```
ReachIn HexSolitaireTrans HexSolitaireInit 16
```

using the theorem `ReachIn_rec`. Since `ReachIn` is not monotonic, it won't reach a fixed point. Let us generalise `iterate` to use a user-supplied ML predicate to terminate iteration.

38

```
- fun gen_iterate stop suc_thm n rtm =
   let val _    = print "."
       val ntm  = intToTerm n
       val th   = CONV_RULE
                    (TOP_DEPTH_CONV reduceLib.SUC_CONV)
                    (SPEC ntm suc_thm)
       val rtm' = lhs(concl th)
       val eqtm = mk_eq(rtm,rtm')
   in addEquation th;
      if stop(n,rtm,rtm')
       then (n,rtm,rtm')
       else (delBdd rtm; gen_iterate stop suc_thm (n+1) rtm')
   end;
> val gen_iterate =
    fn : (int * term * term -> bool)
         -> thm -> int -> term -> int * term * term
```

The stopping criterion is supplied by the user in the form of an ML predicate `stop` whose argument is a triple `(n,rtm,rtm')`, where `rtm` is the n-th iteration and `rtm'` the n+1-th iteration. The triple is returned as the result of `gen_iterate`.

Using `gen_iterate` we define a function to iterate until `stop` holds.

39

```
- fun Iter stop iter_th (Rth,Bth) =
   let val (iter_0,iter_SUC) = MakeIterThms iter_th (Rth,Bth)
       val reach0_tm = lhs(concl iter_0)
   in
    addEquation Rth;
    addEquation Bth;
    addEquation iter_0;
    gen_iterate stop iter_SUC 0 reach0_tm
   end;
> val Iter =
    fn : (int * term * term -> bool)
         -> thm -> thm * thm -> int * term * term
```

Iter, like gen_iterate, returns a triple (n,rtm,rtm'). It can be used to implement ComputeReachable by taking the stopping criterion to be the equality of the BDDs of rtm and rtm'.

40

```
- fun ComputeReachable(Rth,Bth) =
   let val (n,rtm,rtm') =
            Iter (fn(_,rtm,rtm') => isT(mk_eq(rtm, rtm')))
                 HolBddTheory.ReachBy_rec
                 (Rth,Bth)
       val fix_th = CONV_RULE ((RHS_CONV o RATOR_CONV o RAND_CONV)
                               Num_conv.num_CONV)
                               (bddEqOracle(rtm,rtm'))
   in
    MATCH_MP HolBddTheory.ReachBy_fixedpoint (PGEN_EXT fix_th)
   end;
```

Iter can also be used to iterate ReachIn_rec until the set of states becomes empty. In this case the stopping criterion is that the BDD of rtm' is FALSE (i.e. the set of successor states is empty).

```
- val (n,rtm,rtm') =
      Iter (fn(_,_,rtm') => isF rtm') HolBddTheory.ReachIn_rec
      (HexSolitaireTrans_def,HexSolitaireInit_def);
.................
> val n = 16 : int
  val rtm =
    ``ReachIn HexSolitaireTrans HexSolitaireInit 16
       (v01,v02,v03,v04,v05,v06,v07,v08,v09,v10,
        v11,v12,v13,v14,v15,v16,v17,v18,v19)``
  val rtm' =
    ``ReachIn HexSolitaireTrans HexSolitaireInit 17
       (v01,v02,v03,v04,v05,v06,v07,v08,v09,v10,
        v11,v12,v13,v14,v15,v16,v17,v18,v19)``
```
41

Our goal is to look at the set of final states and to choose one as the goal for a solvable puzzle. First we check to see how many states there are.

```
- statecount(termToBdd rtm);
> val it = 30.0 : real
```
42

Printing out the BDD of the set of states as a conditional will result in a large inscrutable nested conditional, so we first rewrite the conditional using COND_NORM

```
- val COND_NORM =
   bossLib.DECIDE
    ``((if b then b1 else b2) = (b /\ b1) \/ (~b /\ b2))
     /\
     ((b /\ (b1 \/ b2)) = ((b /\ b1) \/ (b /\ b2)))
     /\
     (((b1 \/ b2) /\ b) = ((b1 /\ b) \/ (b2 /\ b)))``;
```
43

Let us look at those final reachable states that have one of the two remaining pegs in the middle (i.e. v10=T).

44

```
- val reachable10 =
      REWRITE_RULE
       [COND_NORM]
       (bddRepOracle
         ``ReachIn HexSolitaireTrans HexSolitaireInit 16
            (v01,v02,v03,v04,v05,v06,v07,v08,v09,T,
             v11,v12,v13,v14,v15,v16,v17,v18,v19)``);
> val reachable10 =
    |- ReachIn HexSolitaireTrans HexSolitaireInit 16
         (v01,v02,v03,v04,v05,v06,v07,v08,v09,T,
          v11,v12,v13,v14,v15,v16,v17,v18,v19) =
       ~v01/\v02/\~v03/\~v04/\~v05/\~v06/\~v07/\~v08/\~v09/\
       ~v11/\~v12/\~v13/\~v14/\~v15/\~v16/\~v17/\~v18/\~v19 \/
       ~v01/\~v02/\~v03/\v04/\~v05/\~v06/\~v07/\~v08/\~v09/\
       ~v11/\~v12/\~v13/\~v14/\~v15/\~v16/\~v17/\~v18/\~v19 \/
       ~v01/\~v02/\~v03/\~v04/\~v05/\~v06/\v07/\~v08/\~v09/\
       ~v11/\~v12/\~v13/\~v14/\~v15/\~v16/\~v17/\~v18/\~v19 \/
       ~v01/\~v02/\~v03/\~v04/\~v05/\~v06/\~v07/\~v08/\~v09/\
       ~v11/\~v12/\v13/\~v14/\~v15/\~v16/\~v17/\~v18/\~v19 \/
       ~v01/\~v02/\~v03/\~v04/\~v05/\~v06/\~v07/\~v08/\~v09/\
       ~v11/\~v12/\~v13/\~v14/\~v15/\v16/\~v17/\~v18/\~v19 \/
       ~v01/\~v02/\~v03/\~v04/\~v05/\~v06/\~v07/\~v08/\~v09/\
       ~v11/\~v12/\~v13/\~v14/\~v15/\~v16/\~v17/\v18/\~v19
```

From this we see that the other pegs must be at position 2, 4, 7, 13, 16 or 18.

Let us now define our new solvable game to be the problem of constructing a sequence of moves from the initial state to the state with two pegs remaining at positions 2 and 10.

45

```
- isT ``Reachable HexSolitaireTrans HexSolitaireInit
        (F,T,F,F,F,F,F,F,F,T,F,F,F,F,F,F,F,F)``;
> val it = true : bool
```

We will write an ML function to compute a sequence of states from an initial state to a goal state. Such a sequence is called a trace and defined by the predefined predicate `IsTrace` that satisfies

```
[IsTrace_def]
Theorem
|- (IsTrace R B Q [] = F) /\ (IsTrace R B Q [s] = B s /\ Q s)
   /\
   (IsTrace R B Q (s0::s1::tr) =
    B s0 /\ R (s0,s1) /\ IsTrace R (Eq s1) Q (s1::tr))
```

where the function Eq is just curried equality (Eq is used here instead of $= for purely cosmetic reasons).

```
[Eq_def]
Definition
|- !state0 state. Eq state0 state = state0 = state
```

The meaning of IsTrace is illustrated by

46

```
- REWRITE_CONV
   [HolBddTheory.IsTrace_def,HolBddTheory.Eq_def]
   ``IsTrace R B Q [s0;s1;s2;s3;s4]``;
> val it =
    |- IsTrace R B Q [s0; s1; s2; s3; s4] =
       B s0
       /\
       R(s0,s1) /\ R(s1,s2) /\ R(s2,s3) /\ R(s3,s4)
       /\
       Q s4
```

The solution to which we want to find a path is defined by

47

```
- val HexSolitaireSoln_def =
   bossLib.Define
    `HexSolitaireSoln
      (v01,v02,v03,v04,v05,v06,v07,v08,v09,v10,
       v11,v12,v13,v14,v15,v16,v17,v18,v19) =
      ~v01
      /\v02/\
      ~v03/\~v04/\~v05/\~v06/\~v07/\~v08/\~v09
      /\v10/\
      ~v11/\~v12/\~v13/\~v14/\~v15/\~v16/\~v17/\~v18/\~v19`;
```

In this case there only one state satisfying the predicate, but in other examples there might be more.

The function `findModel` can be used to extract a description of a state satisfying a predicate:

48

```
- addEquation HexSolitaireSoln_def;
- findModel
   ``HexSolitaireSoln
      (v01,v02,v03,v04,v05,v06,v07,v08,v09,v10,
       v11,v12,v13,v14,v15,v16,v17,v18,v19)
     /\
     ReachIn HexSolitaireTrans HexSolitaireInit 16
      (v01,v02,v03,v04,v05,v06,v07,v08,v09,v10,
       v11,v12,v13,v14,v15,v16,v17,v18,v19)``;
> val it =
    ``~v01/\v02/\~v03/\~v04/\~v05/\~v06/\~v07/\~v08/\~v09/\
      v10/\~v11/\~v12/\~v13/\~v14/\~v15/\~v16/\~v17/\~v18/\~v19``
```

To get an actual state vector from this term is easy:

49

```
- rhs(concl(REWRITE_CONV[ASSUME it]s));
> val it = ``(F,T,F,F,F,F,F,F,T,F,F,F,F,F,F,F,F)`` : term
```

Call this `s16`. Next we want to find `s15` such that

```
ReachIn HexSolitaireTrans HexSolitaireInit 15 s15
/\
HexSolitaireTrans(s15,s16)
```

50

```
- addEquation HexSolitaireTrans_def;
- findModel
   ``ReachIn HexSolitaireTrans HexSolitaireInit 15
      (v01,v02,v03,v04,v05,v06,v07,v08,v09,v10,
       v11,v12,v13,v14,v15,v16,v17,v18,v19)
     /\
    HexSolitaireTrans
     ((v01,v02,v03,v04,v05,v06,v07,v08,v09,v10,
       v11,v12,v13,v14,v15,v16,v17,v18,v19),
      (F,T,F,F,F,F,F,F,T,F,F,F,F,F,F,F,F))``;
> val it =
    ``~v01/\v02/\~v03/\~v04/\~v05/\~v06/\~v07/\~v08/\~v09/\
      ~v10/\~v11/\~v12/\~v13/\~v14/\v15/\~v16/\~v17/\~v18/\
      v19``
- val s15 = rhs(concl(REWRITE_CONV[ASSUME it]s));
> val s15 = ``(F,T,F,F,F,F,F,F,F,F,F,F,F,T,F,F,F,T)``
```

This can be repeated to get `s14` (omitting some output):

51

```
- findModel
   ``ReachIn HexSolitaireTrans HexSolitaireInit 14
      (v01,v02,v03,v04,v05,v06,v07,v08,v09,v10,
       v11,v12,v13,v14,v15,v16,v17,v18,v19)
     /\
    HexSolitaireTrans
     ((v01,v02,v03,v04,v05,v06,v07,v08,v09,v10,
       v11,v12,v13,v14,v15,v16,v17,v18,v19),
      (F,T,F,F,F,F,F,F,F,F,F,F,F,T,F,F,F,T))``;
- val s14 = rhs(concl(REWRITE_CONV[ASSUME it]s));
> val s14 = ``(F,T,F,F,F,F,F,F,F,F,F,F,F,T,F,T,T,F)`` : term
```

An ML function `BackIter` to compute the whole trace is easy to write:

52

```
- fun BackIter (Rth,Bth) state 0 = [state]
  | BackIter (Rth,Bth) state n =
      let val (R,_)  = dest_comb(lhs(concl(SPEC_ALL Rth)))
          val (B,s)  = dest_comb(lhs(concl(SPEC_ALL Bth)))
          val mtm    = findModel
                         ``ReachIn ^R ^B ^(intToTerm(n-1)) ^s
                           /\
                           ^R(^s,^state)``
          val state_ = rhs(concl(REWRITE_CONV[ASSUME mtm]s))
      in
       BackIter (Rth,Bth) state_ (n-1) @ [state]
      end;
```

A sequence of states can then be computed

53

```
- val trace =
   BackIter (HexSolitaireTrans_def,HexSolitaireInit_def)
            ``(F,T,F,F,F,F,F,F,F,T,F,F,F,F,F,F,F,F)``
            16;
> val trace =
    [``(T,T,T,T,T,T,T,T,T,F,T,T,T,T,T,T,T,T,T)``,
     ``(T,T,T,T,T,T,T,T,T,T,T,T,T,T,F,T,T,T,F)``,
     ``(T,T,T,T,F,T,T,T,T,F,T,T,T,T,T,T,T,T,F)``,
     ``(T,T,T,T,F,T,T,F,F,T,T,T,T,T,T,T,T,T,F)``,
     ``(F,T,T,F,F,T,T,T,F,T,T,T,T,T,T,T,T,T,F)``,
     ``(F,T,T,F,F,T,T,T,T,F,F,T,T,T,T,T,T,T,F)``,
     ``(F,T,F,F,F,F,T,T,T,T,F,T,T,T,T,T,T,T,F)``,
     ``(F,T,F,F,F,T,T,T,T,F,F,T,T,F,T,T,T,T,F)``,
     ``(F,T,T,F,F,T,F,T,T,F,F,F,T,F,T,T,T,T,F)``,
     ``(F,T,F,F,F,F,F,T,T,T,F,F,T,F,T,T,T,T,F)``,
     ``(F,T,F,F,T,F,F,T,T,F,F,F,T,F,F,T,T,T,F)``,
     ``(F,T,F,F,T,F,F,F,F,T,F,F,T,F,F,T,T,T,F)``,
     ``(F,T,F,F,F,F,F,F,F,F,F,F,T,F,T,T,T,T,F)``,
     ``(F,T,F,F,F,F,F,F,F,F,F,F,F,T,F,F,T,T,F)``,
     ``(F,T,F,F,F,F,F,F,F,F,F,F,F,F,T,F,T,T,F)``,
     ``(F,T,F,F,F,F,F,F,F,F,F,F,F,F,F,T,F,F,F,T)``,
     ``(F,T,F,F,F,F,F,F,T,F,F,F,F,F,F,F,F,F)``]
```

This is the result of an ML computation. A theorem certifying that it is a trace is easily proved:

54

```
- bddRepOracle
   (fst
     (addEquation
      (REWRITE_CONV
        [HolBddTheory.IsTrace_def,HolBddTheory.Eq_def]
        ``IsTrace
           HexSolitaireTrans
           HexSolitaireInit
           HexSolitaireSoln
           ^(mk_list(trace,type_of(hd trace)))``)));
> val it =
    |- IsTrace HexSolitaireTrans HexSolitaireInit HexSolitaireSoln
         [(T,T,T,T,T,T,T,T,T,F,T,T,T,T,T,T,T,T,T);
          (T,T,T,T,T,T,T,T,T,T,T,T,T,T,F,T,T,T,F);
          (T,T,T,T,F,T,T,T,T,T,F,T,T,T,T,T,T,T,F);
          (T,T,T,T,F,T,T,F,F,T,T,T,T,T,T,T,T,T,F);
          (F,T,T,F,F,T,T,T,F,T,T,T,T,T,T,T,T,T,F);
          (F,T,T,F,F,T,T,T,T,F,F,T,T,T,T,T,T,T,F);
          (F,T,F,F,F,F,T,T,T,T,F,T,T,T,T,T,T,T,F);
          (F,T,F,F,F,T,T,T,T,F,F,T,T,F,T,T,T,T,F);
          (F,T,T,F,F,T,F,T,T,F,F,F,T,F,T,T,T,T,F);
          (F,T,F,F,F,F,F,T,T,T,F,F,T,F,T,T,T,T,F);
          (F,T,F,F,T,F,F,T,T,F,F,F,T,F,F,T,T,T,F);
          (F,T,F,F,T,F,F,F,F,T,F,F,T,F,F,T,T,T,F);
          (F,T,F,F,F,F,F,F,F,F,F,F,T,F,T,T,T,T,F);
          (F,T,F,F,F,F,F,F,F,F,F,F,T,T,F,F,T,T,F);
          (F,T,F,F,F,F,F,F,F,F,F,F,F,F,T,F,T,T,F);
          (F,T,F,F,F,F,F,F,F,F,F,F,F,F,T,F,F,F,T);
          (F,T,F,F,F,F,F,F,T,F,F,F,F,F,F,F,F,F)] = T
```

Normally, it would be overkill to formally verify that an ML-computed trace is actually a trace, but it is possible.

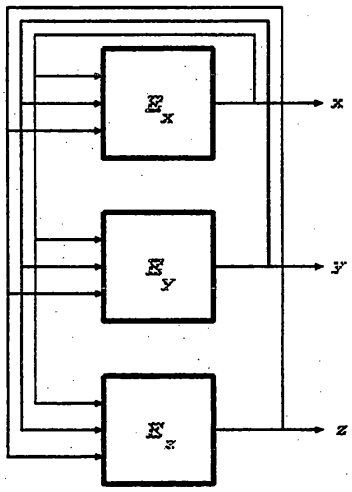
The standard game of Peg Solitaire can also be solved using the methods just described. It takes several hours to compute a solution by iterating **ReachIn**. See Section 7.4 for further discussion.

6.4 Efficient image computation

The computation of `ReachBy R B (SUC n)` using

```
[ReachBy_rec]
Theorem
|- (!R B state. ReachBy R B 0 state = B state) /\
   (!R B n state.
      ReachBy R B (SUC n) state =
      ReachBy R B n state \/
      ?state_. ReachBy R B n state_ /\ R (state_,state))
```

is sometimes called computing the *forward image* of `ReachBy R B n` under `R`. This computation can be optimised using a standard method called *early quantification* [10, page 45] or *disjunctive partitioning*[14]. The idea will be illustrated using three machines running asynchronously in parallel:



This can be modelled by a transition relation $\mathcal{R}$ of the form

$$\begin{aligned}&\mathcal{R}(x,y,z),(x',y',z')) = \\ &\quad (x' = E_x(x,y,z) \ \wedge \ y' = y \ \wedge \ z' = z) \vee \\ &\quad (x' = x \ \wedge \ y' = E_y(x,y,z) \ \wedge \ z' = z) \vee \\ &\quad (x' = x \ \wedge \ y' = y \ \wedge \ z' = E_z(x,y,z))\end{aligned}$$

Assume a predicate $\mathcal{B}$ specifies the set of initial states. Let $\mathcal{S}(p,q,r)$ abbreviate ReachBy n $\mathcal{R}$ $\mathcal{B}(p,q,r)$ then:

[14] In the theorem proving literature, early quantification is called 'miniscoping' and can reduce proof length more than exponentially [11].

$$
\begin{aligned}
&\exists p\ q\ r.\ \mathsf{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}(p,q,r)\ \wedge\ \mathcal{R}((p,q,r),(x,y,z)) \\
&= \exists p\ q\ r.\ \mathcal{S}(p,q,r)\ \wedge\ \mathcal{R}((p,q,r),(x,y,z)) \\
&= \exists p\ q\ r.\ \mathcal{S}(p,q,r)\ \wedge\ ((x = E_x(p,q,r)\ \wedge\ y = q\ \wedge\ z = r)\ \vee \\
&\qquad\qquad\qquad\qquad (x = p\ \wedge\ y = E_y(p,q,r)\ \wedge\ z = r)\ \vee \\
&\qquad\qquad\qquad\qquad (x = p\ \wedge\ y = q\ \wedge\ z = E_z(p,q,r))) \\
&= (\exists p\ q\ r.\ \mathcal{S}(p,q,r)\ \wedge\ x = E_x(p,q,r)\ \wedge\ y = q\ \wedge\ z = r)\ \vee \\
&\quad (\exists p\ q\ r.\ \mathcal{S}(p,q,r)\ \wedge\ x = p\ \wedge\ y = E_y(p,q,r)\ \wedge\ z = r)\ \vee \\
&\quad (\exists p\ q\ r.\ \mathcal{S}(p,q,r)\ \wedge\ x = p\ \wedge\ y = q\ \wedge\ z = E_z(p,q,r)) \\
&= ((\exists p.\ \mathcal{S}(p,y,z) \wedge x{=}E_x(p,y,z))\ \wedge\ (\exists q.\ y{=}q)\ \wedge\ (\exists r.\ z{=}r))\ \vee \\
&\quad ((\exists p.\ x{=}p)\ \wedge\ (\exists q.\ \mathcal{S}(x,q,z) \wedge y{=}E_y(x,q,z))\ \wedge\ (\exists r.\ z{=}r))\ \vee \\
&\quad ((\exists p.\ x{=}p)\ \wedge\ (\exists q.\ y{=}q)\ \wedge\ (\exists r.\ \mathcal{S}(x,y,r) \wedge z{=}E_z(x,y,r))) \\
&= (\exists p.\ \mathcal{S}(p,y,z)\ \wedge\ x = E_x(p,y,z))\ \vee \\
&\quad (\exists q.\ \mathcal{S}(x,q,z)\ \wedge\ y = E_y(x,q,z))\ \vee \\
&\quad (\exists r.\ \mathcal{S}(x,y,r)\ \wedge\ z = E_z(x,y,r))
\end{aligned}
$$

Thus the BDD of $\exists p\ q\ r.\ \mathsf{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}(p,q,r)\ \wedge\ \mathcal{R}((p,q,r),(x,y,z))$ can be computed without ever computing the BDD of $\mathcal{R}((p,q,r),(x,y,z))$.

In Hol98, early quantification can be done using off-the-shelf tools from `simpLib`. To illustrate this, we first compute the state transition equations for HexSolitaire.

55

```
- val (ReachBy_0,ReachBy_SUC) =
       MakeIterThms
        HolBddTheory.ReachBy_rec
        (HexSolitaireTrans_def,HexSolitaireInit_def);
> val ReachBy_0 =
    |- ReachBy HexSolitaireTrans HexSolitaireInit 0
         (v01,v02,v03,v04,v05,v06,v07,v08,v09,v10,
          v11,v12,v13,v14,v15,v16,v17,v18,v19) =
       HexSolitaireInit
         (v01,v02,v03,v04,v05,v06,v07,v08,v09,v10,
          v11,v12,v13,v14,v15,v16,v17,v18,v19)
  val ReachBy_SUC =
    |- !n.
       ReachBy HexSolitaireTrans HexSolitaireInit (SUC n)
        (v01,v02,v03,v04,v05,v06,v07,v08,v09,v10,
         v11,v12,v13,v14,v15,v16,v17,v18,v19) =
       ReachBy HexSolitaireTrans HexSolitaireInit n
        (v01,v02,v03,v04,v05,v06,v07,v08,v09,v10,
         v11,v12,v13,v14,v15,v16,v17,v18,v19) \/
       ?p_1 p_11 p_12 p_13 p_14 p_15 p_16 p_17 p_18 p_19
        p_110 p_111 p_112 p_113 p_114 p_115 p_116 p_117 p_2.
        ReachBy HexSolitaireTrans HexSolitaireInit n
         (p_1,p_11,p_12,p_13,p_14,p_15,p_16,p_17,p_18,p_19,
          p_110,p_111,p_112,p_113,p_114,p_115,p_116,p_117,p_2)
        /\
        HexSolitaireTrans
         ((p_1,p_11,p_12,p_13,p_14,p_15,p_16,p_17,p_18,p_19,
           p_110,p_111,p_112,p_113,p_114,p_115,p_116,p_117,p_2),
          (v01,v02,v03,v04,v05,v06,v07,v08,v09,v10,
           v11,v12,v13,v14,v15,v16,v17,v18,v19))
```

The following box shows the result of rewriting ReachBy_SUC with the definition of HexSolitaireTrans (only 2 of the 54 disjuncts inside the existential quantification are shown).

56

```
- val ReachBy_SUC_exp =
      REWRITE_RULE [HexSolitaireTrans_def] ReachBy_SUC;
> val ReachBy_SUC_exp =
    |- !n. ReachBy HexSolitaireTrans HexSolitaireInit (SUC n)
            (v01,v02,v03,v04,v05,v06,v07,v08,v09,v10,
             v11,v12,v13,v14,v15,v16,v17,v18,v19) =
            ReachBy HexSolitaireTrans HexSolitaireInit n
             (v01,v02,v03,v04,v05,v06,v07,v08,v09,v10,
              v11,v12,v13,v14,v15,v16,v17,v18,v19)
           \/
           ?p_1 p_11 p_12 p_13 p_14 p_15 p_16 p_17 p_18 p_19 p_110
            p_111 p_112 p_113 p_114 p_115 p_116 p_117 p_2.
            ReachBy HexSolitaireTrans HexSolitaireInit n
             (p_1,p_11,p_12,p_13,p_14,p_15,p_16,p_17,p_18,p_19,
              p_110,p_111,p_112,p_113,p_114,p_115,p_116,p_117,p_2)
            /\
             p_1/\p_11/\~p_12/\~v01/\~v02/\v03/\(v04=p_13)/\
             (v05=p_14)/\(v06=p_15)/\(v07=p_16)/\(v08=p_17)/\
             (v09=p_18)/\(v10=p_19)/\(v11=p_110)/\(v12=p_111)/\
             (v13=p_112)/\(v14=p_113)/\(v15=p_114)/\(v16=p_115)/\
             (v17=p_116)/\(v18=p_117)/\(v19=p_2)
            \/
             p_1/\p_14/\~p_19/\~v01/\~v05/\v10/\(v02=p_11)/\
             (v03=p_12)/\(v04=p_13)/\(v06=p_15)/\(v07=p_16)/\
             (v08=p_17)/\(v09=p_18)/\(v11=p_110)/\(v12=p_111)/\
             (v13=p_112)/\(v14=p_113)/\(v15=p_114)/\(v16=p_115)/\
             (v17=p_116)/\(v18=p_117)/\(v19=p_2)
            \/
            ...
```

The early quantification simplification will eliminate all the existential quantifiers in the right hand side completely! This is illustrated below, where only 3 of the 55 disjuncts are shown.

57

```
val ReachBy_SUC_simp =
 simpLib.SIMP_RULE
  HOLSimps.hol_ss
  [AND_OR,EXISTS_OR_THM]
  ReachBy_SUC_exp;
> val ReachBy_SUC_simp =
    |- !n. ReachBy HexSolitaireTrans HexSolitaireInit (SUC n)
            (v01,v02,v03,v04,v05,v06,v07,v08,v09,v10,v11,
             v12,v13,v14,v15,v16,v17,v18,v19) =
           ReachBy HexSolitaireTrans HexSolitaireInit n
            (v01,v02,v03,v04,v05,v06,v07,v08,v09,v10,
             v11,v12,v13,v14,v15,v16,v17,v18,v19)
           \/
           ReachBy HexSolitaireTrans HexSolitaireInit n
            (T,T,F,v04,v05,v06,v07,v08,v09,v10,
             v11,v12,v13,v14,v15,v16,v17,v18,v19) /\
            ~v01 /\ ~v02 /\ v03
           \/
           ReachBy HexSolitaireTrans HexSolitaireInit n
            (T,v02,v03,v04,T,v06,v07,v08,v09,F,
             v11,v12,v13,v14,v15,v16,v17,v18,v19) /\
            ~v01 /\ ~v05 /\ v10
           \/
           ...
```

The difference in using `ReachBy_SUC_exp` and `ReachBy_SUC_simp` to compute the BDD of `ReachBy HexSolitaireTrans HexSolitaireInit (SUC 0)` is dramatic (<$> indicates a BuDDy garbage collection).

```
58
- addEquation ReachBy_0;
  (* Needed for subsequent BDD calculation *)
- time addEquation (SPEC ``0`` ReachBy_SUC_exp);
<$><$><$><$><$><$><$><$><$><$><$><$><$><$><$><$><$><$><$><$>
<$><$><$><$><$><$><$><$><$><$><$><$><$><$><$><$><$><$><$><$>
<$><$><$><$><$><$><$><$><$><$><$><$><$><$><$><$><$><$><$><$>
<$><$><$><$><$><$><$><$><$><$><$><$><$><$><$><$><$><$><$><$>
<$><$><$><$><$><$><$><$><$><$><$><$><$><$><$><$><$><$><$><$>
runtime: 604.320s,    gctime: 86.690s,     systime: 0.570s.
> val it =
    (``ReachBy HexSolitaireTrans HexSolitaireInit (SUC 0)
         (v01,v02,v03,v04,v05,v06,v07,v08,v09,v10,
          v11,v12,v13,v14,v15,v16,v17,v18,v19)``, <bdd>)
- time addEquation (SPEC ``0`` ReachBy_SUC_simp);
runtime: 2.510s,    gctime: 0.420s,     systime: 0.010s.
> val it =
    (``ReachBy HexSolitaireTrans HexSolitaireInit (SUC 0)
         (v01,v02,v03,v04,v05,v06,v07,v08,v09,v10,
          v11,v12,v13,v14,v15,v16,v17,v18,v19)``, <bdd>)
```

Early quantification can also be used to find paths to counter-examples. To apply it, the computation of the trace is reformulated in terms of computing *backward images* of the transition relation.

The backward image of $\mathcal{Q}$ under $\mathcal{R}$ is $\exists s'.\ \mathcal{R}(s, s') \ \wedge \ \mathcal{Q}\ s'$.

`HolBddLib` supports backward images via a defined constant `Prev`

```
[Prev_def]
Definition
|- !R Q state.
    Prev R Q state = ?state'. R(state,state') /\ Q state'
```

A trace s_0, s_1, ..., s_n is computed by working backwards from s_n, as already illustrated. The state s_{i-1} is obtained from s_i by using `findModel` to pick s such that

$$\mathsf{ReachBy}\ \mathcal{R}\ \mathcal{B}\ (i{-}1)\ s\ \wedge\ \mathsf{Pre}\ \mathcal{R}\ (\mathsf{Eq}\ s_i)\ s$$

Note that the BDDs for $\mathsf{ReachBy}\ \mathcal{R}\ \mathcal{B}\ i\ s$ (for $i = 0, 1, \ldots, n$) will already have been computed when searching for s_n.

Note alse that `Prev` $\mathcal{R}$ $\mathcal{Q}$ s can be deductively simplified using early quantification so that its BDD can be computed without having to compute the BDD of the transition relation $\mathcal{R}$.

7 The HolBddLib structure StateEnum

The structure `StateEnum` in `HolBddLib` provides tools for computing sets of reachable states similar to those described earlier. These are a bit more robust and efficient than the pedagogical tools described in Section 6. `StateEnum` also contains some miscellaneous functions, documented in Section 7.3.

7.1 Symbolic state enumeration tools

Each function in this section takes as argument a pair of theorems, `(Rth,Bth)` say, of the form

$$(\vdash \mathcal{R}((v_1,\ldots,v_p),(v'_1,\ldots,v'_p)) = t_{\mathcal{R}},\ \vdash \mathcal{B}(v_1,\ldots,v_p) = t_{\mathcal{B}})$$

defining a transition relation $\mathcal{R}$ and set of initial states $\mathcal{B}$ (the theorems can be closed under universal quantification).
The functions

```
MakeSimpReachInRecThm : thm * thm -> thm
MakeSimpReachByRecThm : thm * thm -> thm
```

return, respectively, theorems of the form

$$\vdash \texttt{(ReachIn } \mathcal{R}\ \mathcal{B}\ \texttt{0 } (v_1,\ldots,v_p) = t_{\mathcal{B}})$$
$$\texttt{/\ (ReachIn } \mathcal{R}\ \mathcal{B}\ \texttt{(SUC n) } (v_1,\ldots,v_p) = t_{simp})$$

$$\vdash \texttt{(ReachBy } \mathcal{R}\ \mathcal{B}\ \texttt{0 } (v_1,\ldots,v_p) = t_{\mathcal{B}})$$
$$\texttt{/\ (ReachBy } \mathcal{R}\ \mathcal{B}\ \texttt{(SUC n) } (v_1,\ldots,v_p) = t_{simp})$$

where t_{simp} is computed by early quantification.
The functions

```
ComputeReachable
  : thm * thm -> {ReachThm : thm, iterations : int}

ComputeSimpReachable
  : thm * thm -> {ReachThm : thm, SimpTransThm : thm, iterations : int}
```

compute theorems of the form

$$\vdash \texttt{Reachable}\ \mathcal{R}\ \mathcal{B}\ (v_1,\ldots,v_p) = \texttt{ReachBy}\ \mathcal{R}\ \mathcal{B}\ i\ (v_1,\ldots,v_p)$$

where i is the number of iterations to a fixed point. The difference between the two functions is that `ComputeReachable` doesn't use early quantification to simplify the next-state calculation, but `ComputeSimpReachable` does.

7.2 Computing traces

```
val FindRefutationTrace : thm * thm * thm -> thm list
```

This takes a triple `(Rth,Bth,Qth)`, where `Rth` and `Bth` define a transition relation and set of initial states, respectively, as above. The third argument `Qth` defines a predicate, $\mathcal{Q}$ say, by

$$\vdash \mathcal{Q}(v_1,\ldots,v_p) = t_{\mathcal{Q}}$$

that is intended to hold of all reachable states.

The function `FindRefutationTrace` seaches for a counter-example to show that in fact there is a reachable state, s_0 say, for which $\mathcal{Q}$ fails. An ML list of theorems is returned:

$$[\vdash \mathcal{B}\ s_m, \vdash \texttt{Next}\ \mathcal{R}\ (\texttt{Eq}\ s_m)\ s_{m-1}, \ \ldots\ , \vdash \texttt{Next}\ \mathcal{R}\ (\texttt{Eq}\ s_1)\ s_0, \vdash \neg(\mathcal{Q}\ s_0)\ ]$$

The sequence $s_m,\ldots,s_0$ is a sequence of states, of minimal length, from an initial state to a reachable state refuting $\mathcal{Q}$.

`FindRefutationTrace` tries to use early quantification to simplify both the forward (`Next`) and backward image (`Prev`) calculations.

Note that applying

```
map (simpLib.SIMP_RULE
       HOLSimps.hol_ss
       [HolBddTheory.Next_def,HolBddTheory.Eq_def])
```

to the list of theorem above results in

$$[\vdash \mathcal{B}\ s_m, \vdash \mathcal{R}(s_m, s_{m-1}),\ \ldots\ , \vdash \mathcal{R}(s_1, s_0), \vdash \neg(\mathcal{Q}\ s_0)\ ]$$

7.3 Miscellaneous ML functions

The functions listed here are miscellaneous ML and HOL tools that have been mentioned earlier. Various other functions are provided by `HolBddLib` (e.g. for encoding HOL enumerated types as products of `bool`). These will be documented in future versions of this report.

```
intToTerm : int -> term
```

Converts an ML integer to a HOL numeral.

```
val PGEN_EXT : thm -> thm
```

Implements a version of extensionality:

$$\frac{\vdash P\ =\ Q}{\vdash \forall v_1\ \cdots\ v_n.\ P(v1,\ldots,v_n)\ =\ Q(v1,\ldots,v_n)}$$

7.4 Efficiency issues

On small examples the fairly naive implementation described here (and available with the first release of `HolBddLib`) seems to work well. However, larger examples reveal that there are some efficiency problems. For example, using `ComputeReachable` the computation of the sets of reachable states of the standard game of Peg Solitaire grinds to a halt around depth 16. However, a pure MuDDy solution that directly computes the BDDs without going via HOL formulae can calculate the BDDs of all depths in a few hours.

It is hoped that by comparing the BDD operations performed by the pure MuDDy solution with those invoked via `ComputeReachable` it will be possible to improve to tools in `StateEnum` to match the pure MuDDy performance.

Acknowledgements

The research reported here was supported by EPSRC grant GR/K57343 *Checking Equivalence Between Synthesised Logic and Non-Synthesisable Behavioural Prototypes* and ESPRIT Framework IV LTR 26241 project Prosper (*Proof and Specification Assisted Design Environments*).

Data from Atanas Parashkevov and Bill Roscoe on the BDD and state space sizes arising from Peg Solitaire was useful for evaluating and testing `HolBddLib`.

Paul Jackson, Jesper Møller and Konrad Slind provided detailed comments and suggestions on a first draft of this report.

References

[1] Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Lifted-fl: A pragmatic implementation of combined model checking and theorem proving. In *Theorem Proving in Higher Order Logics (TPHOLs99)*, July 1999.

[2] Henrik Reif Andersen. An Introduction to Binary Decision Diagrams, October 1997. Lecture notes for 49285 Advanced Algorithms E97. Available from: `http://www.it.dtu.dk/~hra`.

[3] R. J. Boulton. *Efficiency in a Fully-Expansive Theorem Prover.* PhD thesis, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, U.K., May 1994. Technical Report 337.

[4] R. S. Boyer and J Moore. Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic. In *Machine Intelligence 11*, pages 83–124. Oxford University Press, 1988.

[5] Randall E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[6] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution.

In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, number 407 in Lecture Notes in Computer Science, pages 365–373. Springer-Verlag, 1989.

[7] M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[8] Jørn Lind-Nielsen's BuDDy package is documented at `http://www.itu.dk/research/buddy/`.

[9] K. L. McMillan. A methodology for hardware verification using compositional model checking. Technical report, Cadence Berkeley Labs, April 1999. Available at `http://www-cad.eecs.berkeley.edu/~kenmcmil/`.

[10] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[11] Larry Paulson, private communication.

[12] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[13] See web page `http://www.csl.sri.com/pvs.html`.

[14] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, June 1995. Springer-Verlag.