Logic via Foundational Algorithms

James Hook and Tim Sheard

October 20, 2014

1 Finite Sets

With the advent of efficient SAT-solvers, it is possible to solve very large SAT-problems with hundreds of variables and thousands of clauses. Interesting questions include: How does one create such problems? How is one assured that the SAT-problems created faithfully encode the problem of interest? In general, how does one create SAT-problems at a higher level of abstraction than manipulating individual propositions?

One needs a mechanism for aggregating propositions, and high-level operators for operating on those aggregations to create new propositions. This is the role of the FiniteSet module.

1.1 Finite Sets as Mappings

Every finite domain can be mapped to a contiguous finite prefix of the Natural numbers. For example, given the data declaration for primary colors:

```
data Color = Red | Blue | Green deriving (Enum,Show,Read,Typeable)
```

One might map Color as follows: `[(Red,0),(Blue,1),(Green,2)]`. Here we have indicated the mapping as an association list, mapping `Red` to `0`, `Blue` to `1`, and `Green` to 2. A finite set over a finite domain `D` can then be represented as a characteristic function with type `D -> Bool`. Because the finite domain over which a set is drawn is a finite mapping from `D` to Natural, we can represent the characteristic function as a finite map `Natural -> Bool` by composing the two mappings.

```
Primary Color Domain
[(Red,0),(Blue,1),(Green,2)]

Set           Mapping
{Blue}        [(0,False),(1,True),(2,False)]
{Red}         [(0,True),(1,False),(2,False)]
{Red,Green}   [(0,True),(1,False),(2,True)]
```

Experience with such mappings show that the vast majority of such sets map most of the Natural numbers to `False`. Thus, we can compress the mapping by storing only those pairs that map to `True`. For example:

```
Primary Color Domain
[(Red,0),(Blue,1),(Green,2)]

Set             Mapping                              Compressed Mapping
{} =            [(0,False),(1,False),(2,False)]      []
{Red,Green}     [(0,True),(1,False),(2,True)]        [(0,True),(2,True)]
```

A product of two finite sets can be represented in a similar manner. We represent a binary product as a binary tuple, a triple as a ternary tuple, a quadruple as 4-tuple, etc. An n-tuple is represented as a list. The tupling occurs both with the finite domains, and with the domain of the characteristic function.

```
 Primary Color Domain          Boolean Domain
[[(Red,0),(Blue,1),(Green,2)], [(False,0),(True,1)]]

Set                           Compressed Mapping
{} =                          []
{(Red,True),(Green,False)}    [([0,1],True),([2,0],True)]
{(Blue,True)}                 [([1,1],True)]
```

It is a good metaphor to think of a set of n-tuples as an n-dimensional matrix stored using a sparse array representation. The convention of storing only the mappings to True in the characteristic function, really pays off for storing higher dimensional data in many cases. Finally, in order to use a single representation for the characteristic function for all tuple sizes, we exploit the property that a product of finite domains can be mapped to a single Natural number. For example, the product of a 3-point domain and a 2-point domain can be numbered as a single Natural by row-major ordering as:

```
The set {(Red,True),(Green,False)}       Row-major order
as a 2-dimensional sparse array.         of array elements

     0   1                                    0   1
   +---+---+                              +---+---+
0  |   | T |                           0  | 0 | 1 |
   +---+---+                              +---+---+
1  |   |   |                           1  | 2 | 3 |
   +---+---+                              +---+---+
2  | T |   |                           2  | 4 | 5 |
   +---+---+                              +---+---+
```

Thus, sets of `Color` × `Bool` can be represented as follows

```
(Primary Color Domain,Boolean Domain)
[[(Red,0),(Blue,1),(Green,2)], [(False,0),(True,1)]]

Set                           Compressed Mapping            Row-major Mapping
{} =                          []                            []
{(Red,True),(Green,False)}    [([0,1],True),([2,0],True)]   [(1,True),(4,True)]
{(Blue,True)}                 [([1,1],True)]                [(3,True)]
```

A last detail, rather than use an association list, we use a balanced-tree representation (`Data.Map.Map`) of the characteristic function, which provides logarithmic-time access to the boolean value given a row-major index. Thus, A finite set is represented as a (possibly n-ary) domain and a mapping from Natural to Bool.

We can represent sets of elements from base types. We encode this as:

```
data Base = Int | String | Char | Double | Bool | Enum String | Tuple [Base]
  deriving Eq
```

Base types are basically scalar types, enumerations and tuples. An enumeration, like `Color` is reprsented as `(Enum "Color")`, and a tuple as a list of `Base`. In the FiniteSet module a FiniteSet is declared by the following Haskell definitions.

```
data Dimension =                -- Invariant: (Dim n b xs) => length xs == n.
     Dim Int Base [Literal]

data FiniteSet a =
     FA [Dimension]             -- The n dimensions.
        (Data.Map.Map Int a)    -- Mapping of Row-major index to value.
```

A FiniteSet stores a metaphorical k-dimensional array as a 1-dimensional partial sparse array. It combines the k indices to a single index (using row-major order), and uses this as the key in a Data.Map. The array is partial and sparse, because only the known indexes are stored. By specializing the type parameter `a` to a boolean type we obtain an efficient representation of Sets of n-ary tuples. Of course there is nothing to stop us from creating FiniteSets where that parameter is not a boolean type, or to define types other than Haskell's `Bool` to use as a Boolean type (See Section 1.2).

The module provides a number of mechanisms for creating both `Dimension`s and `FiniteSet`s. We can create Dimensions a number of ways.

```
-- State the size, and we generate a mapping [0 .. n-1]
dim :: Int -> Dimension
dim n = Dim n Int [show x | x <- [0 .. n-1]]

-- Exhibit a list of strings that describes the mapping
dimS:: [String] -> Dimension
dimS xs = Dim (length xs) String xs

--Turn a list of anything that can be turned into a string into a dimension.
dimL :: (Show a) => [a] -> Dimension
dimL xs = Dim (length xs) String (map show xs)

-- Create a multi-Dimensional mapping where all the individual
-- mappings are over a 0-based range of Int.
expand :: [Int] -> [Dimension]
expand xs = map dim xs
```

```
-- Any element of a type in the Enum class describes a dimension
-- over the finite set described by the full enumeration.
dimE :: (Show a,Enum a) => String -> a -> Dimension
dimE name x = Dim (length xs) (Enum name) (map (LCon name . show) xs)
  where first = (toEnum 0)
        theyHaveTheSameType = [first,x]
        xs = enumFrom first
```

Here are some examples of `Dimension` creation, which were cut and pasted from a Ghci interactive session. Lines with the prompt `*FiniteSet>` are input by the user, and the other lines are the `Dimension`s printed by the evaluation mechanism of Haskell.

```
*FiniteSet> dim 4
Int#[0,1,2,3]

*FiniteSet> dimS ["a","cde","tom"]
String#["a","cde","tom"]

*FiniteSet> dimL [True,False]
String#["True","False"]

*FiniteSet> dimE Color Blue
Color#[Red,Blue,Green]

*FiniteSet> expand [2,3]
[Int#[0,1],Int#[0,1,2]]
```

To create a finite set, the module provides two modes. The first mode lets one choose the value associated with a particular list of indices by supplying a function. The function is given the full tuple of indices, rather than the row-major single index.

```
-- Use a function to choose if a tuple is present, and if so, what value is stored.
partial:: [Dimension] -> ([Int] -> Maybe a) -> FiniteSet a
partial dims pred =
   FA dims
      (fromList [ (i,j)
                | i <- flatInts dims
                , Just j <- [pred (kIndex dims i)] ])

-- Supply a value for every tuple.
universe :: [Dimension] -> ([Int] -> a) -> FiniteSet a
universe dims initf = partial dims (Just . initf)
```

Here are some example uses of FiniteSet creation, cut and pasted from a Ghci interaction session that illustrate the functions `partial` and `universe`.

```
*FiniteSet> partial (expand [2,2]) (\ [x,y]-> if even x then (Just True) else Nothing)
(Int#2,Int#2)
```

```
{(0,0) (0,1)}

*FiniteSet> universe (expand [2,2]) (\ [x,y]-> x+y)
(Int#2,Int#2)
{(0,0)=0 (0,1)=1 (1,0)=1 (1,1)=2}
```

Note that the first call associates a `Bool`, and the second associates an `Int`, with every tuple. The system prints finite sets of Bool, as a characteristic function, listing only those indicies where the True value is associated. It prints finite sets of Int by printing the Int along with the index.

The second mode for finite set creation lets the user enumerate the elements in the set by supplying a list.

```
-- supply a list of tuples, and make a set from them.
-- every tuple is associated with the same value 'v'
fromIndexList:: t -> [Dimension] -> [[Int]] -> FiniteSet t
fromIndexList v dim xss | any (badElements dim) xss =
   error ("In 'fromIndexList' one of the elements\n"++show xss++"\nis not appropriate for dimension "+
fromIndexList v dim xss = FA dim (fromList (map f xss))
   where f xs = (flatIndex dim xs,v)


-- Uses Finite class magic to create Sets from lists of objects from a Finite type.
fromFiniteList ::
  Finite a => b -> [Dimension] -> [a] -> FiniteSet b
fromFiniteList true ds xs = FA ds (fromList (map f xs))
  where f t = (flatIndex ds (zipWith litToIndex ds (toLit t)),true)
```

The function `fromIndexList` allows the programmer to supply the a list of `Int` representing the indices for each tuple.

```
*FiniteSet> fromIndexList 'z' (expand [3,2]) [[0,1],[2,0],[1,1]]
(Int#3,Int#2)
{(0,1)=z (1,1)=z (2,0)=z}
```

The `fromIndexList` function requires the user to supply appropriate tuples as a list of `Int` indices. This is can be tedious if the dimensions are not `Int` based. The function `fromFiniteList` uses the Haskell class system to turn lists of single elements or lists of tuples into these indices. Lists of any type, t, where t is an instances of the `Finite` class will do. The types `Int`, `Bool`, `Integer`, `String`, and `Char` are instances of this class, as well as binary and ternary tuples of these types. Users can add their own instances if they desire.

```
*FiniteSet> fromFiniteList True [dimE "Bool" True,dimE "Color" Red] [(True,Blue),(False,Red)]
(Bool#2,Color#3)
{(False,Red) (True,Blue)}
```

This depends upon the type `Color` being an instance of the `Finite` class Adding a new enumeration type to the Finite class is trivial as the default method definitions work for all types that are in the `Show`,`Enum`,`Read`, and `Typeable` classes.

```
data Color = Red | Blue | Green deriving (Enum,Read,Show,Typeable)
instance Finite Color where
```

1.2 Relations as FiniteSet of Generalized Booleans

A `(FiniteSet a)` associates an `a` object with every tuple. If the type `a` is the Haskell type `Bool` a natural interpretation is that tuples associated with `True` are present in the set, and those associated with `False` are absent from the set. *Presence* is an all or nothing situation.

In many cases *presence* is conditional. This motivates the class definition `Boolean`, and allows the user to associate each tuple with something more dynamic than the `Bool` type. A perfect candidate is the `(Prop a)` type of the Prop module which is used to represent the propositional calculus.

The strategy is to write operations on `(FiniteSet b)` where the type of associated values is an instance of the `Boolean` class.

```
class Show b => Boolean b where
  true :: b
  false :: b
  isTrue :: b -> Bool
  isFalse :: b -> Bool
  conj:: b -> b -> b     -- conjunction
  disj:: b -> b -> b     -- disjunction
  neg:: b -> b           -- negation
  imply:: b -> b -> b    -- implication
```

The operations of the class allow us to combine `FiniteSet`s to get richer sets, where the association may denote conditional *presence* in the set. The operations of the class are just a generalization of the traditional operations on the type `Bool`. This strategy now supports our original goal of building aggregate structures of propositional formula, equipped with a rich algebra for combining the propositional formula inside the aggregates (see Section 1.3).

Of course it is now easy to make both the `Bool` type and the `Prop` type instance of the `Boolean` class.

```
instance Boolean Bool where
  true = True
  false = False
  isTrue x = x
  isFalse = not
  conj = (&&)
  disj = (||)
  neg = not
  imply x y = not x || y

instance (PPLetter n,Ord n) => Boolean (Prop n) where
  true = TruthP
  false = AbsurdP
  isTrue TruthP = True
  isTrue x = False
  isFalse AbsurdP = True
  isFalse x = False
  conj = andB
```

```
  disj = orB
  neg = notB
  imply = implyB
```

The functions `andB`, `orB`, `notB`, and `implyB` are versions of `AndP`, `OrP`, `NotP` and `ImpliesP` that know about the special properties of these functions on TruthP and AbsurdP. See the `FiniteSet` module implementation for exact details.

Given an overloaded expression of type `Boolean b => (FiniteSet b)` we can choose the `Bool` instance to denote the all or nothing behavior, or the `Prop` instance to denote the conditional *presence* behavior. The conditionality of the elements will depend upon the value of the propositional `LetterP` components of the value associated with every tuple.

Because the propositional interpretation of the `FiniteSets` is the one we most often use, we make special instances of the construction functions, and give the name `Relation` to the type `(Prop Int)`.

```
type Relation = FiniteSet (Prop Int)

many::[Dimension] -> [[Int]] -> Relation
many ds xs = fromIndexList true ds xs

manyD:: Finite a => [Dimension] -> [a] -> Relation
manyD ds xs = fromFiniteList true ds xs

manyL::[Dimension] -> [Literal] -> Relation
manyL ds xs = fromLitList true ds xs
```

Here we construct a few `Relations` using these specialized constructors.

```
*FiniteSet> let units = many [dim 5] [[i] | i <- [0..4]]
*FiniteSet> units
(Int#5)
{(0)=T (1)=T (2)=T (3)=T (4)=T}

*FiniteSet> let pairs = manyD [dim 5,dimE True] [(3::Int,True),(4,False),(0,True),(1,False)]
*FiniteSet> pairs
(Int#5,Bool#2)
{(0,True)=T (1,False)=T (3,True)=T (4,False)=T}
```

The last important means of construction, is to construct `FiniteSets` where the tuples are associated with propositional letters. This is the role of the function `enum`. The strategy is to associate every tuple with a unique propositional letter, and to interpret that letter with the `presence` of its tuple. Experience dictates that whether or not a tuple is associated with a propositional letter depends upon the details of the problem being solved. So the `enum` function allows the programmer to supply this information by passing `enum` a function.

```
enumPoly:: ([Int] -> a -> Maybe (a, t)) -> [Dimension] -> a -> (a, FiniteSet t)
enumPoly pred dims seed = finish (walk seed (flatInts dims))
  where finish (last,zs) = (last,FA dims (fromList zs))
        walk next [] = (next,[])
        walk next (i:is) = case pred (kIndex dims i) next of
                             Nothing -> walk next is
                             Just(new,j) -> (final,(i,j):ys)
                                where (final,ys) = walk new is

enum:: ([Int] -> Int -> Maybe (Int, t)) -> [Dimension] -> Int -> (Int, FiniteSet t)
enum = enumPoly
```

The function `enum` is a state transformer, taking an `Int` (the current letter) and producing a new `Int` (for the next letter) and a `Prop` formula.

```
*FiniteSet> let (n',square) = enum f [dim 2,dim 3] 1 where f xs n = Just(n+1,LetterP n)
*FiniteSet> square
(Int#2,Int#3)
{(0,0)=p1 (0,1)=p2 (0,2)=p3 (1,0)=p4 (1,1)=p5 (1,2)=p6}

*FiniteSet> n'
7
```

Note how the function `enum` returns a pair `(n',square)` where `n'` is the next letter (in case one wants to continue using the same sequence of letters in another `FiniteSet`) and a `FiniteSet` numbered with distinct `LetterP`.

1.3 Operations Over Aggregates - The Relational Algebra

The overloaded type `Boolean b => FiniteSet (Prop b)` provides a convenient implementation for aggregating propositional formulas. All we need is some operations over such aggregates. The module provides three types of operations: Point-wise operators (which operate on the boolean values `b` one at a time), the traditional set based operations: `complement`, `union`, `intersection`, and `difference`; and the traditional relational operations over sets of tuples: `select`, `project`, and `join`.

```
-- Pointwise operations on FiniteSets
unary::  (Boolean b1, Boolean b2) =>
         ([Int] -> b1 -> b2) -> FiniteSet b1 -> FiniteSet b2
binary:: (Boolean a, Boolean b, Boolean c) =>
         (a -> b -> c) -> FiniteSet a -> FiniteSet b -> FiniteSet c

-- Traditional Set based operations
complement :: Boolean b => FiniteSet b -> FiniteSet b
intersect  :: Boolean b => FiniteSet b -> FiniteSet b -> FiniteSet b
union      :: Boolean b => FiniteSet b -> FiniteSet b -> FiniteSet b
difference :: Boolean b => FiniteSet b -> FiniteSet b -> FiniteSet b

-- Traditional Relational operations
```

```
project    :: Boolean a => [Int] -> FiniteSet a -> FiniteSet a
select     :: Boolean t => ([Int] -> Bool) -> FiniteSet t -> FiniteSet t
join       :: Boolean a => Int -> FiniteSet a -> FiniteSet a -> FiniteSet a
```

In the case where the overloaded type parameter `b` is the all or nothing instance `Bool`, these operations implement exactly the semantics their names suggest. In the `Prop Int` instance they support a powerful means of specifying conditional sets, where the conditionality of the input sets is accurately carried over into the result.

We illustrate the all or nothing case by a series of Ghci interactions using the above operations over the set `p` declared as follows. Note the use of `True` as the second argument to `fromFiniteList` fixes the type of `p` as `(FiniteSet Bool)`. We interpret a tuple `(p,c)`, as `p` is the parent of `c`.

```
people = ["Anita","Barbara","Caleb","Frank","George","Margareet","Tim","Walter"]

tuples = [ ("Frank","Tim"),("Tim" , "Caleb"),("Walter","Frank"),
           ("Anita","Tim"),("Margareet","Barbara"),("Barbara","Caleb")]

pd = dimS people
p = fromFiniteList True [pd,pd] tuples
```

Each interaction (except the first, where we just print `p`) begins by a local introduction of a new variable by some operation over `p`, then by printing the new variable.

```
*FiniteSet> p
(String#8,String#8)
{("Anita","Tim") ("Barbara","Caleb") ("Frank","Tim")
 ("Margareet","Barbara") ("Tim","Caleb") ("Walter","Frank")}

*FiniteSet> let children = project [1] p
*FiniteSet> children
(String#8)
{("Barbara") ("Caleb") ("Frank") ("Tim")}

*FiniteSet> let parents = project [0] p
*FiniteSet> parents
(String#8)
{("Anita") ("Barbara") ("Frank") ("Margareet") ("Tim") ("Walter")}

*FiniteSet> let both = intersect children parents
*FiniteSet> both
(String#8)
{("Barbara") ("Frank") ("Tim")}

*FiniteSet> let threeGen = join 1 p (project [1,0] p)
*FiniteSet> threeGen
```

```
(String#8,String#8,String#8)
{("Barbara","Caleb","Margareet") ("Frank","Tim","Walter")
 ("Tim","Caleb","Anita") ("Tim","Caleb","Frank")}

*FiniteSet> let grandparents = project [2] threeGen
*FiniteSet> grandparents
(String#8)
{("Anita") ("Frank") ("Margareet") ("Walter")}
```

To illustrate the conditional interpretation we define a new set `q` which has type `(FiniteSet (Prop Int))` where every tuple is labeled by a unique proposition letter. We use the `enum` function to do this.

```
(_,q) = enum f [pd,pd] 1
  where f xs n = if elem xs indexes
                   then Just(n+1,LetterP n) else Nothing

        indexes = (map (kIndex [pd,pd] . toIndex [pd,pd]) tuples)
```

Note how `q` associates proposition letters to each tuple when we print it.

```
*FiniteSet> q
(String#8,String#8)
{("Anita","Tim")=p1 ("Barbara","Caleb")=p2 ("Frank","Tim")=p3
 ("Margareet","Barbara")=p4 ("Tim","Caleb")=p5
 ("Walter","Frank")=p6}
```

When we operate on sets like this, the conditionality of the inputs flows into the output. For example, joining 0 columns, is a simple way to implement the cross product operation. When we take the product of `q2` with itself, we get the following. We elide some of the tuples (indicated by the "...") for brevity.

```
*FiniteSet> join 0 q q
(String#8,String#8,String#8,String#8)
{("Anita","Tim","Anita","Tim")=p1
 ("Anita","Tim","Barbara","Caleb")=p1 /\ p2
 ("Anita","Tim","Frank","Tim")=p1 /\ p3
 ("Anita","Tim","Margareet","Barbara")=p1 /\ p4
...
```

Note how the tuple `(Anita,Tim,Barbara,Caleb)` is conditioned by the proposition `p1 /\ p2`. The tuple `(Anita,Tim,Barbara,Caleb)` will appear in the product, only if the tuple `(Anita,Tim)=p1` appears in `q` and the tuple `(Barbara,Caleb)=p2` appears in `q`. The other operations propagate conditionality similarly.

The pointwise `unary` and `binary` operations act as generic maps applying functions to the Boolean values associated with each tuple individually. For example to conjoin every associated value with the propositional variable `p99` we could write.

```
*FiniteSet> unary (\ xs p -> conj (LetterP 99) p) q
(String#8,String#8)
{("Anita","Tim")=p1 /\ p99 ("Barbara","Caleb")=p2 /\ p99
 ("Frank","Tim")=p3 /\ p99 ("Margareet","Barbara")=p4 /\ p99
 ("Tim","Caleb")=p5 /\ p99 ("Walter","Frank")=p6 /\ p99}
```

1.4 Propositions from Sets

.

Finite sets aggregate many individual propositions, but a SAT-solver finds a satisfying solution for only a single proposition. The last step in the strategy for specifying SAT-problems at a high level of abstraction, is to extract properties from finite sets. Fortunately there are a number of natural ways to extract a single proposition from a set. We have implemented four general kinds of extraction functions. First, propositions about the size of a set. Second, asking if one set is a subset of another. Third, for a set of tuples representing a relation, asking if any column(s) functionally determines another columns(s). Fourth, a simple kind of universal quantification.

```
-- Cardinality questions
some :: (Boolean b) => FiniteSet b -> b
none :: (Boolean b) => FiniteSet b -> b
full :: Boolean b => FiniteSet b -> b
one :: (Boolean t) => FiniteSet t -> t

-- Subset
subset:: (Boolean t) => FiniteSet t -> FiniteSet t -> t

-- Functional dependency between columns of a Relation
funDep:: (Boolean b) => [Int] -> [Int] -> FiniteSet b -> b

-- Quantification
forall:: (Boolean b) => FiniteSet t -> (FiniteSet t -> b) -> b
```

We will illustrate these operations on sets using the parent-child `FiniteSet`s `p` and `q` from the running example above, as well as the set `six` defined below.

```
counter xs n = Just(n+1,LetterP n)
(_,six) = enum counter [dim 3, dim 2] 1
```

The set `six` associates a unique variable for each of its six tuples.

```
*FiniteSet> six
(Int#3,Int#2)
{(0,0)=p1 (0,1)=p2 (1,0)=p3 (1,1)=p4 (2,0)=p5 (2,1)=p6}
```

One. The predicate `one` asks if the set has exactly one element. For that to be true, exactly one of the variables `p1` to `p6` must be true. The `one` predicate generates a clause for each variable. In the generated clause, that variable is true, and all the others are negated.

```
*FiniteSet> one six
(p1 /\ ~p2 /\ ~p3 /\ ~p4 /\ ~p5 /\ ~p6)
\/
(~p1 /\ p2 /\ ~p3 /\ ~p4 /\ ~p5 /\ ~p6)
\/
(~p1 /\ ~p2 /\ p3 /\ ~p4 /\ ~p5 /\ ~p6)
\/
(~p1 /\ ~p2 /\ ~p3 /\ p4 /\ ~p5 /\ ~p6)
\/
(~p1 /\ ~p2 /\ ~p3 /\ ~p4 /\ p5 /\ ~p6)
\/
(~p1 /\ ~p2 /\ ~p3 /\ ~p4 /\ ~p5 /\ p6)
```

The size of the proposition generated grows quickly as the number of variables grows. For a set with many variables the `one` predicate may be too large to solve effectively.

None. The `none` predicate asks if the set is empty. It generates a clause which negates the proposition associated with each tuple in the set.

```
*FiniteSet> none six
~p1 /\ ~p2 /\ ~p3 /\ ~p4 /\ ~p5 /\ ~p6
```

Under our *presence* interpretation, none of the tuples are *present*, so the relation must be empty. The size of the proposition generated is equal to the sum of the sizes of all propositions in the set.

Some. The `some` predicate asks if the set has at least one tuple. It generates a clause which is a disjunction of the propositions associated with each tuple in the set.

```
*FiniteSet> some six
p1 \/ p2 \/ p3 \/ p4 \/ p5 \/ p6
```

Full. The `full` predicate asks if the set is the universe. I.e. if every possible tuple for the given domain is present in the set. It generates a clause which is a conjunction of the propositions associated with each tuple in the set. It also conjoins the "missing" tuples not stored explicitly in the characteristic set. Many times the proposition can be determined to be `AbsurdP` because a tuple is not explicitly stored.

```
*FiniteSet> full six
p1 /\ p2 /\ p3 /\ p4 /\ p5 /\ p6
```

Subset. A set `y1` is a subset of a set `y2` iff element *present* in `y1` is also *present* in `y2`. In terms of the propositions associated with each tuple, the proposition of each tuple in `y1` must imply the proposition associated with the same tuple in `y2`. This is illustrated by the Ghci interaction below.

```
(ii,y1) = enum counter [dim 4] 1
(_,y2) = enum counter [dim 4] ii
```

```
*FiniteSet> y1
(Int#4)
{(0)=p1 (1)=p2 (2)=p3 (3)=p4}

*FiniteSet> y2
(Int#4)
{(0)=p5 (1)=p6 (2)=p7 (3)=p8}

*FiniteSet> subset y1 y2
(p1 => p5) /\ (p2 => p6) /\ (p3 => p7) /\ (p4 => p8)
```

Functional Dependency. A very useful predicate over a set of tuples is whether one column functionally determines another column. A column i determines a column j, if for all pairs of tuples, when the values in the i position are the same in both, then in both those tuples, the values in the j position are also the same. For example, given the parent-child relation `p`:

```
*FiniteSet> p
(String#8,String#8)
{("Anita","Tim") ("Barbara","Caleb") ("Frank","Tim")
 ("Margareet","Barbara") ("Tim","Caleb") ("Walter","Frank")}

*FiniteSet> funDep [0][1] p
True

*FiniteSet> funDep [1][0] p
False
```

Column 0 (the parent) determines column 1 (the child), because every parent has exactly one child. But the column 1 (the child) does not determine column 0 (the parent), because two children, Caleb and Tim, have two parents.

1.5 Examples

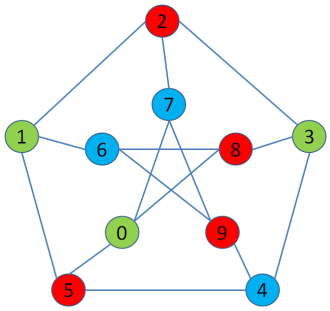
In this section we apply the tools we have learned to code several problems as SAT-problems. The idea is to define a few conditional relations, and a list of predicates that must hold between the relations. We then generate a SAT-problem from the list. The generation phase consists of extracting a proposition from each predicate in the list, transforming each proposition into conjunctive normal form (CNF), conjoining them all together, and passing the resulting formula in CNF to the *MiniSat* SAT-solver. If a solution exists, the solution is in terms of a satisfying assignment mapping every propositional letter to either `True` or `False`. When this assignment is applied to the original relations, a solution to the problem can be read from the result.

1.5.1 The Map Coloring Problem

The map coloring problem is to assign colors to countries on a map. The constraint that must be met is that no two adjacent countries should be drawn

with the same color. The key to solving the problem is to imagine the map as a graph, where the countries are the vertices, and the adjacency information is represented by the edges. For a planar graph (where the graph can be drawn in two dimensions and no edges cross) 4 colors is always sufficient. But for non-planar graphs the minimum number of colors depends upon the graph geometry.

In this example we will choose a particular graph called the Peterson Graph, and develop our instruction by experimenting with that graph. We will then develop a general solution.




```
numC = 4
graph::[(Int,Int)]
graph = [(1,2),(2,3),(3,4),(4,5),
         (5,1),(1,6),(2,7),(3,8),
         (4,9),(5,0),(6,8),(7,9),
         (8,0),(9,6),(0,7)]

colors = take numC
           ["Red","Blue","Green",
            "Yellow","Purple","Orange",
            "Violet","Indigo"]

edges  = manyD [dim 10,dim 10] graph
color = manyD [dimS colors] colors
```

Figure 1: The Peterson Graph is represented by the edge list stored in `graph`. The number of colors, `numC`, is 4. The variable `colors` selects the first `numC` out of all possible colors. The `Relation`, `edges`, represents the graph as a `FiniteSet`, and the `Relation`, `color`, represents the set of colors we will try and color the graph with.

In Figure 1 is a drawing of the Peterson Graph and a few lines of code that represents the graph as a pair of finite sets. Our strategy is to try a different numbers of colors, so we have created a long list of colors, and then we select some shorter prefix for each try. For our first try we select 4.

The key to the solution is to represent the association between a Vertex and a Color as a relation. We initialize such a relation where every possible relationship is represented by a unique propositional letter.

```
(_,coloring) = enum f [dim 10,dimS colors] 1
     where f [i,j] n = Just(n+1,LetterP n)
```

A listing of the tuples in the relation are listed for reference.

```
*FiniteSet> coloring
(Int#10,String#4)
{(0,"Red")=p1 (0,"Blue")=p2 (0,"Green")=p3 (0,"Yellow")=p4
 (1,"Red")=p5 (1,"Blue")=p6 (1,"Green")=p7 (1,"Yellow")=p8
 (2,"Red")=p9 (2,"Blue")=p10 (2,"Green")=p11 (2,"Yellow")=p12
```

```
(3,"Red")=p13 (3,"Blue")=p14 (3,"Green")=p15 (3,"Yellow")=p16
(4,"Red")=p17 (4,"Blue")=p18 (4,"Green")=p19 (4,"Yellow")=p20
(5,"Red")=p21 (5,"Blue")=p22 (5,"Green")=p23 (5,"Yellow")=p24
(6,"Red")=p25 (6,"Blue")=p26 (6,"Green")=p27 (6,"Yellow")=p28
(7,"Red")=p29 (7,"Blue")=p30 (7,"Green")=p31 (7,"Yellow")=p32
(8,"Red")=p33 (8,"Blue")=p34 (8,"Green")=p35 (8,"Yellow")=p36
(9,"Red")=p37 (9,"Blue")=p38 (9,"Green")=p39 (9,"Yellow")=p40}
```

The key constraint we want to hold is: No two adjacent vertexes have the same color. Our strategy is to construct a ternary relation, `same`, of two Vertices and one Color. A triple will be *present* in this relation only if the two vertices are adjacent, and they both have the same color. We will require that this ternary relation is empty. A concise way of specifying the relation `same` is a prolog-like program:

```
same(x,y,c1) :-  coloring(x,c1),edge(x,y),coloring(y,c2),c1=c2
```

This has a straight-forward translation into the relational algebra, where we have indicated as a comment the "shape" of the tuples in the set computed by the expression on each line.

```
same = project [2,0,1] (                    -- (x,y,c1)
       select (\ [y,c2,x,c1] -> c1==c2) (   -- (y,c1,x,c1)
       join 1                               -- (y,c2,x,c1)
              coloring                      -- (y,c2)
              (project [2,0,1]              -- (y,x,c1)
                       (join 1              -- (x,c1,y)
                             coloring       -- (x,c1)
                             edges))))      -- (x,y)
```

We list a few tuples from the original `coloring` relation and a few tuples from the `same` relation. Note how the tuple `(0,7,Red)` is *present* only if the tuples `(0,Red)=p1` and `(7,Red)=p29` are *present* in the `coloring` relation. This is indicated by the proposition `p1 /\ p29` associated with the tuple `(0,7,Red)`.

```
*FiniteSet> coloring
(0,Red)=p1
(0,Blue)=p2
(0,Green)=p3
(0,Yellow)=p4
(7,Red)=p29
(7,Blue)=p30
(7,Green)=p31
(7,Yellow)=p32
```

```
*FiniteSet> same
(0,7,Red)=p1 /\ p29
(0,7,Blue)=p2 /\ p30
(0,7,Green)=p3 /\ p31
```

The next step is to describe a list of constraints that must hold. The obvious constraint is that the `same` relation is empty, since that relations holds pairs of adjacent vertices with the same color. If this is the only constraint we use, we get a satisfying solution that assigns all 40 proposition letters from `coloring`

to false. This means the `coloring` relation itself is empty as none of its tuples are *present*. It is not enough that the `same` relation is empty, we want a color assigned to *every* edge. The set of all edges is just the zeroth projection of the `coloring` relation. We want *every* edge to be assigned some color, so we want the set of edges to be the `full` relation. This leads to the following.

```
--  Extracted propositions
colorProp = [ none same
            , full (project [0] coloring)]
```

Finally we conjoin the list of propositions, and then turn that constraint into conjunctive normal form.

```
colorCnf = cnf (andL colorProp)
```

```
*FiniteSet> colorCnf
 [[p1,p2,p3,p4],[p5,p6,p7,p8],[p9,p10,p11,p12],[p13,p14,p15,p16],[p17,p18,p19,p20],[p21,p22
 ,p23,p24],[p25,p26,p27,p28],[p29,p30,p31,p32],[p33,p34,p35,p36],[p37,p38,p39,p40],[~p1,~p2
 1],[~p1,~p29],[~p1,~p33],[~p2,~p22],[~p2,~p30],[~p2,~p34],[~p3,~p23],[~p3,~p31],[~p3,~p35]
 ,[~p4,~p24],[~p4,~p32],[~p4,~p36],[~p5,~p9],[~p5,~p21],[~p5,~p25],[~p6,~p10],[~p6,~p22],[~
 p6,~p26],[~p7,~p11],[~p7,~p23],[~p7,~p27],[~p8,~p12],[~p8,~p24],[~p8,~p28],[~p9,~p13],[~p9
 ,~p29],[~p10,~p14],[~p10,~p30],[~p11,~p15],[~p11,~p31],[~p12,~p16],[~p12,~p32],[~p13,~p17]
 ,[~p13,~p33],[~p14,~p18],[~p14,~p34],[~p15,~p19],[~p15,~p35],[~p16,~p20],[~p16,~p36],[~p17
 ,~p21],[~p17,~p37],[~p18,~p22],[~p18,~p38],[~p19,~p23],[~p19,~p39],[~p20,~p24],[~p20,~p40]
 ,[~p25,~p33],[~p25,~p37],[~p26,~p34],[~p26,~p38],[~p27,~p35],[~p27,~p39],[~p28,~p36],[~p28
,~p40],[~p29,~p37],[~p30,~p38],[~p31,~p39],[~p32,~p40]]
```

We then write that formula in CNF to a file in Dimacs CNF format appropriate for the MiniSat solver.

```
colorFile = putClauses "color.cnf" colorCnf
```

The function `putClauses` creates the file `color.cnf`. The first few lines of which are listed here.

```
p cnf 40 70
1 2 3 4 0
5 6 7 8 0
9 10 11 12 0
13 14 15 16 0
17 18 19 20 0
21 22 23 24 0
25 26 27 28 0
29 30 31 32 0
33 34 35 36 0
37 38 39 40 0
-1 -21 0
-1 -29 0
...
```

The first line states that the file is in Dimacs CNF form with 40 variables and 70 clauses. Each clause is implicitly a disjunction, one clause per line. The last clause listed is an encoding of `~p1 \/ ~p29`. The trailing `0` is a sentinel and ends every clause. Negated literals are represented by negative numbers. There are 70 lines in all. We can solve the problem by calling MiniSat from a shell command line. Entering the correct directory we call MiniSat with parameters to point it to the input and name a file where the output should be stored. MiniSat writes some information to the standard output. A short listing follows. I typed the commands following the prompt (`$`), the rest was produced by MiniSat.

```
$ cd /cygdrive/d/work/sheard/Courses/LogicAndProgLang/web/hw/FiniteSets

$ MiniSat color.cnf color.sol
================================[MINISAT]================================
| Conflicts |     ORIGINAL     |              LEARNT              | Progress |
|           | Clauses Literals |   Limit Clauses Literals  Lit/Cl |          |
==========================================================================
|         0 |      70      160 |      23       0        0     nan |  0.000 % |
==========================================================================
restarts              : 1
conflicts             : 0              (nan /sec)
decisions             : 21             (inf /sec)
propagations          : 40             (inf /sec)
conflict literals     : 0              ( nan % deleted)
Memory used           : 86.81 MB
CPU time              : 0 s

SATISFIABLE

$ more color.sol
SAT
-1 -2 3 -4 -5 -6 7 -8 9 -10 -11 -12 -13 -14 15 -16 -17 18 -19 -20 21 -22 -23 -24 -25 26 -2
7 -28 -29 30 -31 -32 33 -34 -35 -36 37 -38 -39 -40 0
```

The file `color.sol` contains the satisfying assignment. It assigns the variables 3, 7, 9, 15, 18, 21, 26, 30, 33, and 37 to true, and the other variables to false. When this assignment is applied to the `coloring` relation, only a few tuples are `present`. These tuples indicate one solution to coloring the Peterson graph.

```
(0,Green)=T
(1,Green)=T
(2,Red)=T
(3,Green)=T
(4,Blue)=T
(5,Red)=T
(6,Blue)=T
(7,Blue)=T
```

```
(8,Red)=T
(9,Red)=T
```

The picture of the graph in Figure 1 is dislayed using these colors. Although we initialized the number of colors to 4, we found a solution with 3 colors. Is it possible to color the graph with just 2 colors? This is an easy experiment to carry out. Change the Haskell variable `numC` to 2 and repeat all the steps. With fewer colors the problem is smaller, the `coloring` relation has only 20 (10×2) propositional letters, and the CNF formula has only 40 clauses. But there is no solution. Here is the transcript of running MiniSat on this problem.

```
================================[MiniSat]================================
| Conflicts |     ORIGINAL     |              LEARNT              | Progress |
|           | Clauses Literals |   Limit Clauses Literals  Lit/Cl |          |
=========================================================================
|         0 |      40       80 |      13       0        0     nan |  0.000 % |
=========================================================================
restarts              : 1
conflicts             : 2              (133 /sec)
decisions             : 1              (67 /sec)
propagations          : 30             (2000 /sec)
conflict literals     : 1              (0.00 % deleted)
Memory used           : 62.31 MB
CPU time              : 0.015 s

UNSATISFIABLE
```

1.5.2 The 8 Queens Problem

The traditional 8-queens problem asks is there any way to place 8 queens on an 8×8 board such that no queen places another queen in check? To solve this problem we will first experiment with a smaller problem of placing 4 queens on a 4×4 board. Here is a picture of one solution.

```
+---+---+---+---+
|   |   | Q |   |
+---+---+---+---+
| Q |   |   |   |
+---+---+---+---+
|   |   |   | Q |
+---+---+---+---+
|   | Q |   |   |
+---+---+---+---+
```

An obvious start to a solution would be to generate a 4×4 relation. If a tuple `(i,j)` is present in the relation, then the i^{th} row and j^{th} column has a queen. Using our expertise in creating finite sets this is easy.

```
qsize = 4
(_,queens) = enum f (expand [qsize,qsize]) 1
     where f [i,j] n = Just(n+1,LetterP n)
```

Queens place in check any piece in the same row, or the same column, or on the same diagonal. A partial constraint states that there must be exactly 1 queen in every row, and 1 queen in every column. Having one queen in row 0 could be expressed as: `(one (select ( [i,j]->i==0) queens))`. A sample interaction shows the result of this strategy.

```
*FiniteSet> (select (\ [i,j]->i==0) queens)
(0,0)=p1
(0,1)=p2
(0,2)=p3
(0,3)=p4

*FiniteSet> one (select (\ [i,j]->i==0) queens)
(p1 /\ ~p2 /\ ~p3 /\ ~p4)
\/
(~p1 /\ p2 /\ ~p3 /\ ~p4)
\/
(~p1 /\ ~p2 /\ p3 /\ ~p4) \/ (~p1 /\ ~p2 /\ ~p3 /\ p4)
```

We would need eight such constraints, and as the problem grew from 4 to 8 queens we would need 16 such constraints. Another consideration is the poor behavior of the `one` constraint, it grows quite quickly. A better solution is to use functional dependency. A single queen in each row and each column means that for a relation with tuples `(i,j)`, `i` determines `j`, and `j` determines i.

```
fdCol r = funDep [0] [1] r

fdRow r = funDep [1] [0] r
```

Thus we are left with the diagonal constraints. How can we express this? A position `(a,b)` on a grid is on the same diagonal as another position `(i,j)`, if there exists a `n` such that both the row and column position differ by the same `n`. We can write a Haskell program to express this.

```
diag a b [i,j] n = (a==i+n && b==j+n)||(a==i-n && b==j-n)||
                   (a==i+n && b==j-n)||(a==i-n && b==j+n)
```

Armed with such a predicate, given a position, say `(2,2)`, we can select the subset of positions on the board, on a diagonal from `(2,2)` as follows.

```
*FiniteSet> select (\ xs -> any (diag 2 2 xs) [1.. n] ) queens
(Int#4,Int#4)
{(0,0)=p1 (1,1)=p6 (1,3)=p8 (3,1)=p14 (3,3)=p16}
```

Provided that `n` is at least as large as the maximum dimension of the binary relation. Here is a generic function that works for any 2-D relation.

```
diagonal a b (r@(FA [n,m] tab)) = select alongDiag r
  where alongDiag xs = any (diag a b xs) [1.. (max (dimSize m) (dimSize n))-1]
```

The diagonal constraint can be expressed as follows. For every tuple, if it contains a queen, then the subset of the queens relation on the diagonal of that tuple must be empty.

```
noneOnDiag r = full(unary f r)
  where f [a,b] p = imply p (none(diagonal a b r))
```

Note that the `full` constraint specifies that this must hold for every possible tuple in the relation `r`. Fortunately, the `imply` makes it vacuously true when a queen isn't *present* in a tuple `(a,b)`. A partial listing of the set `noneOnDiag queens` is listed below.

```
*FiniteSet> noneOnDiag queens
(p1 => (~p6 /\ ~p11 /\ ~p16)) /\ (p2 => (~p5 /\ ~p7 /\ ~p12)) /\
(p3 => (~p6 /\ ~p8 /\ ~p9)) /\ (p4 => (~p7 /\ ~p10 /\ ~p13)) /\
(p5 => (~p2 /\ ~p10 /\ ~p15)) /\
(p6 => (~p1 /\ ~p3 /\ ~p9 /\ ~p11 /\ ~p16)) /\
(p7 => (~p2 /\ ~p4 /\ ~p10 /\ ~p12 /\ ~p13)) /\
(p8 => (~p3 /\ ~p11 /\ ~p14)) /\ (p9 => (~p3 /\ ~p6 /\ ~p14)) /\
(p10 => (~p4 /\ ~p5 /\ ~p7 /\ ~p13 /\ ~p15)) /\
(p11 => (~p1 /\ ~p6 /\ ~p8 /\ ~p14 /\ ~p16)) /\
(p12 => (~p2 /\ ~p7 /\ ~p15)) /\ (p13 => (~p4 /\ ~p7 /\ ~p10)) /\
(p14 => (~p8 /\ ~p9 /\ ~p11)) /\ (p15 => (~p5 /\ ~p10 /\ ~p12)) /\
(p16 => (~p1 /\ ~p6 /\ ~p11))
```

If a tuple is associated with `p1`, and `p1` is true, then the propositional variables associated with the tuples on its diagonal must all be false.

One last constraint is needed, the constraints we have specified are all satisfied by the empty set, so we need some statement that at least one of them has a non-vacuous solution. An easy such statement is that every row has at least 1 queen (the functional dependency will ensure it has exactly one queen).

```
eachRow r = full(project [0] r)

queenProp = [ eachRow queens,
              fdCol queens,fdRow queens,noneOnDiag queens]
```

Following the same strategy as before, we create and then write the conjunctive normal form to a file in DIMACS CNF format.

```
queenCNF = cnf(andL queenProp)
queenFile = putClauses "queen.cnf" queenCNF
```

Running MiniSat at the shell command line we see the following

```
$ MiniSat queen.cnf queen.sol
================================[MiniSat]================================
| Conflicts |     ORIGINAL     |              LEARNT              | Progress |
|           | Clauses Literals |   Limit Clauses Literals  Lit/Cl |          |
=========================================================================
```

```
|         0 |     108      224 |      36       0        0     nan |  0.000 % |
=========================================================================
restarts              : 1
conflicts             : 3              (200 /sec)
decisions             : 12             (800 /sec)
propagations          : 40             (2667 /sec)
conflict literals     : 7              (0.00 % deleted)
Memory used           : 61.31 MB
CPU time              : 0.015 s

SATISFIABLE

sheard@fox2 /cygdrive/d/work/sheard/Courses/Logic/dev
$ more queen.sol
SAT
-1 -2 3 -4 5 -6 -7 -8 -9 -10 -11 12 -13 14 -15 -16 0
```

When this solution is applied to the `queens` relation we obtain the solution

```
*FiniteSet> instantiate queen1Sol queens
(0,2)=T
(1,0)=T
(2,3)=T
(3,1)=T
```

To solve the 8 queens problem just change the Haskell variable `qsize` to 8. There are now 64 propositional variables and the formulae are considerably larger: 1016 clauses (rather than 108), but a solution is still found.

```
(0,2)=T
(1,5)=T
(2,3)=T
(3,1)=T
(4,7)=T
(5,4)=T
(6,6)=T
(7,0)=T
```

```
+---+---+---+---+---+---+---+---+
|   |   | Q |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   |   |   |   | Q |   |   |
+---+---+---+---+---+---+---+---+
|   |   |   | Q |   |   |   |   |
+---+---+---+---+---+---+---+---+
|   | Q |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   | Q |
+---+---+---+---+---+---+---+---+
|   |   |   |   | Q |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   | Q |   |
+---+---+---+---+---+---+---+---+
| Q |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
```

1.5.3 Embedding MiniSat in Haskell

As we can see from the two examples above, once we have constructed the list of constraints the steps necessary to use MiniSat to find a solution are mechanical. Thankfully Haskell is a great scripting language and it is easy to automate these steps. The module `MiniSat` automates a few commonly used actions. It exports three functions and one variable `minisat`.

```
minisat:: String
putClauses:: (Ord a, Show a) => FilePath -> [[Prop a]] -> IO ()
solveWithMiniSat:: FilePath -> FilePath -> [Prop Int] -> IO (Maybe [Int])
cycleMiniSat:: a -> ([Int] -> IO a) -> FilePath -> FilePath -> [Prop Int] -> IO a
```

It is important that you alter the value of the variable `minisat` to contain the path of where MiniSat is installed on your machine.

We have seen the use of `putClauses` to write a DIMACS CNF format file. A function call `(solveWithMiniSat cnf sol ps)` does the following.

- Turns a list of propositions, `ps`, (each encoding a single constraint) into conjunctive normal form.
- Writes the clauses to the file `cnf`.
- Calls MiniSat on the files `cnf` and `sol`.
- Reads the potential solution created by MiniSat from the file `sol`.
- Returns the result.

If MiniSat fails to find a solution it returns `Nothing`, otherwise it returns `(Just xs)` where xs is a satisfying solution. The solution uses negative numbers, `-i`, to encode negated literals `(NotP (LetterP i))`, and positive numbers, `j`, to encode propositional letters `(LetterP j)`.

Many problems have more than one solution and we can use `cycleMiniSat` to browse through them. A function call `(cycleMiniSat noSolution action cnf sol ps)` does the following.

- Repeatedly calls `solveWithMiniSat`.
- If the formula is unstaisfiable, return `noSolution`.
- For each satisfying solution, call `action`, and pause.
- If the user types in a line that starts with the charcter 'n', the cycle and look for the next solution, otherwise quit and return the current solution.

For example, we write the following, where the action prints the solution, uses it to instantiate `queens`, prints the instantiated `queens`.

```
zzz = cycleMiniSat queens action "queen1.cnf" "queen1.sol" queenProp
  where action xs = do { print xs
                       ; let ans = (instantiate xs queens)
                       ; print ans
                       ; return ans }
```

The short transcript below, runs the 4-queens problem. At this size there are only two distinquishable solutions (each one a reflection about the center axis of the other). When we cycle, after the second solution is found, the next cycle discovers that the extended problem (looking for more solutions) is unsatisfiable.

```
*FiniteSet> zzz
==================================[MiniSat]==================================
| Conflicts |     ORIGINAL     |              LEARNT              | Progress |
|           | Clauses Literals |   Limit Clauses Literals  Lit/Cl |          |
==============================================================================
|         0 |     108      224 |      36       0        0     nan |  0.000 % |
==============================================================================
restarts              : 1
conflicts             : 3              (200 /sec)
decisions             : 12             (800 /sec)
propagations          : 40             (2667 /sec)
conflict literals     : 7              (0.00 % deleted)
Memory used           : 62.31 MB
CPU time              : 0.015 s

SATISFIABLE
Solution 1
[-1,-2,3,-4,5,-6,-7,-8,-9,-10,-11,12,-13,14,-15,-16]
(0,2)=T
(1,0)=T
(2,3)=T
(3,1)=T
n
==================================[MiniSat]==================================
| Conflicts |     ORIGINAL     |              LEARNT              | Progress |
|           | Clauses Literals |   Limit Clauses Literals  Lit/Cl |          |
==============================================================================
|         0 |     109      240 |      36       0        0     nan |  0.000 % |
==============================================================================
restarts              : 1
conflicts             : 6              (400 /sec)
decisions             : 13             (867 /sec)
propagations          : 64             (4267 /sec)
conflict literals     : 16             (11.11 % deleted)
Memory used           : 62.31 MB
CPU time              : 0.015 s

SATISFIABLE
Solution 2
```

```
[-1,2,-3,-4,-5,-6,-7,8,9,-10,-11,-12,-13,-14,15,-16]
(0,1)=T
(1,3)=T
(2,0)=T
(3,2)=T
n
================================[MiniSat]================================
| Conflicts |     ORIGINAL     |              LEARNT              | Progress |
|           | Clauses Literals |   Limit Clauses Literals  Lit/Cl |          |
==========================================================================
|         0 |     110      256 |      36       0        0     nan |  0.000 % |
==========================================================================
restarts              : 1
conflicts             : 9              (600 /sec)
decisions             : 13             (867 /sec)
propagations          : 82             (5467 /sec)
conflict literals     : 18             (10.00 % deleted)
Memory used           : 62.31 MB
CPU time              : 0.015 s

UNSATISFIABLE
Solution 3
[-1,2,-3,-4,-5,-6,-7,8,9,-10,-11,-12,-13,-14,15,-16]
(0,1)=T
(1,3)=T
(2,0)=T
(3,2)=T
```

2 Acknowledgements

Many of the ideas in the FiniteSet module are based upon the paper, *Kodkod: A Relational Model Finder*, by Emina Torlak and Daniel Jackson.