

GENERATING MINIMAL MODELS FOR GEOMETRIC THEORIES

A Major Qualifying Project Report
submitted to the Faculty of

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of
the requirements for the degree of
Bachelor of Science

by

.....
MICHAEL FICARRA

on

4th October, 2010

.....
DANIEL DOUGHERTY
professor, project advisor

Abstract

This paper describes a method, referred to as the chase, for generating jointly minimal models for a geometric theory. A minimal model for a theory is a model for which there exists a homomorphism to any other model that can satisfy the theory. These models are useful in solutions to problems in many practical applications, including but not limited to firewall configuration examination, protocol analysis, and access control evaluation. Also described is a Haskell implementation of the chase and its development process and design decisions.

Table of Contents

1	Introduction	1
1.1	Goals	1
1.2	The Chase	1
2	Technical Background	2
2.1	Models	2
2.2	First-order Logic	2
2.3	Variable Binding	3
2.4	Environment	3
2.5	Satisfiability	3
2.6	Entailment	4
2.7	Homomorphisms	4
2.8	Minimal Models	5
2.9	Positive Existential Form	5
2.10	Geometric Logic	5
3	The Chase	6
3.1	Algorithm	6
3.2	Examples	6
4	Haskell Chase Implementation	8
4.1	Operation	8
4.2	Options	10
4.2.1	I/O	10
4.2.2	Tracing	10

4.3	Future Considerations	10
5	An Extended Application: Cryptographic Protocol Analysis	11
A	Table of Syntax	12
B	Chase code	13
	References	15

1 Introduction

Introductory text...

1.1 Goals

1.2 The Chase

2 Technical Background

2.1 Models

A *model* \mathbb{M} is a construct that consists of:

- a set, referenced as $|\mathbb{M}|$, called the *universe* or *domain* of \mathbb{M}
- a set of pairings of a *predicate* and a non-negative integral arity
- for each predicate R with arity k , a relation $R_k^{\mathbb{M}} \subseteq |\mathbb{M}|$

It is important to distinguish the predicate, which is just a symbol, from the relation that it refers to when paired with its arity. The relation itself is a set of tuples of elements from the universe.

2.2 First-order Logic

First-order logic, also called *predicate logic*, is a formal logic system. For our purposes, this logic system will not contain any constant symbols or function symbols, which are commonly included in first-order and propositional logic. We will see that these can be removed without loss of expressiveness in **reference to GL section**.

A first-order logic formula is defined inductively by

- if R is a relation symbol of arity k and each of $x_0 \dots x_{k-1} \in \vec{x}$ is a variable, then $R[\vec{x}]$ is a formula, specifically an *atomic formula*
- if x and y are variables, then $x = y$ is a formula
- \top and \perp are formulæ
- if α is a formula, then $(\neg\alpha)$ is a formula
- if α and β are formulæ, then $(\alpha \wedge \beta)$ is a formula
- if α and β are formulæ, then $(\alpha \vee \beta)$ is a formula
- if α and β are formulæ, then $(\alpha \rightarrow \beta)$ is a formula
- if α is a formula and x is a variable, then $(\forall x : \alpha)$ is a formula
- if α is a formula and x is a variable, then $(\exists x : \alpha)$ is a formula

A shorthand notation may sometimes be used which omits either the left or right side of an implication and denotes $(\top \rightarrow \sigma)$ and $(\sigma \rightarrow \perp)$ respectively. If α is a formula and \vec{x} is a set of variables of size k , then $(\forall \vec{x} : \alpha)$ is $(\forall x_0 \dots \forall x_{k-1} : \alpha)$. If α is a formula and \vec{x} is a set of variables of size k , then $(\exists \vec{x} : \alpha)$ is $(\exists x_0 \dots \exists x_{k-1} : \alpha)$.

2.3 Variable Binding

Given a function $free$ that returns the set of free variables in a formula, the set of free variables in a formula is defined inductively as follows

- any variable occurring in an atomic formula is a free variable
- the set of free variables in \top and \perp is \emptyset
- the set of free variables in $x = y$ is $\{x, y\}$
- the set of free variables in $\neg\alpha$ is $free(\alpha)$
- the set of free variables in $\alpha \wedge \beta$ is $free(\alpha) \cup free(\beta)$
- the set of free variables in $\alpha \vee \beta$ is $free(\alpha) \cup free(\beta)$
- the set of free variables in $\alpha \rightarrow \beta$ is $free(\alpha) \cup free(\beta)$
- the set of free variables in $\forall x : \alpha$ is $free(\alpha) - \{x\}$
- the set of free variables in $\exists x : \alpha$ is $free(\alpha) - \{x\}$

A formula α is a *sentence* if $free(\alpha) = \emptyset$.

2.4 Environment

An *environment* λ for a model \mathbb{M} is a function from a variable v to a domain element e where $e \in |\mathbb{M}|$. The syntax $\lambda_{[v \mapsto a]}$ denotes the environment $\lambda'(x)$ that returns a when $x = v$ and returns $\lambda(x)$ otherwise.

2.5 Satisfiability

A model \mathbb{M} is said to satisfy a formula σ in an environment λ , denoted $\mathbb{M} \models_{\lambda} \sigma$ and read “under λ , σ is true in \mathbb{M} ”, when

- σ is a relation symbol R and $R[\lambda(a_0), \dots, \lambda(a_n)] \in \mathbb{M}$ where a is a set of variables
- σ is of the form $\neg\alpha$ and $\mathbb{M} \not\models_{\lambda} \alpha$
- σ is of the form $\alpha \wedge \beta$ and both $\mathbb{M} \models_{\lambda} \alpha$ and $\mathbb{M} \models_{\lambda} \beta$
- σ is of the form $\alpha \vee \beta$ and either $\mathbb{M} \models_{\lambda} \alpha$ or $\mathbb{M} \models_{\lambda} \beta$
- σ is of the form $\alpha \rightarrow \beta$ and either $\mathbb{M} \not\models_{\lambda} \alpha$ or $\mathbb{M} \models_{\lambda} \beta$

- σ is of the form $\forall x : \alpha$ and for every $x' \in |\mathbb{M}|$, $\mathbb{M} \models_{\lambda[x \mapsto x']} \alpha$
- σ is of the form $\exists x : \alpha$ and for at least one $x' \in |\mathbb{M}|$, $\mathbb{M} \models_{\lambda[x \mapsto x']} \alpha$

The notation $\mathbb{M} \models \sigma$ (no environment specification) means that, under any environment l , $\mathbb{M} \models_l \sigma$.

A model \mathbb{M} satisfies a set of formulæ Σ for an environment λ if for every σ such that $\sigma \in \Sigma$, $\mathbb{M} \models_{\lambda} \sigma$. This is denoted as $\mathbb{M} \models_{\lambda} \Sigma$ and read "M is a model of Σ ".

2.6 Entailment

Given an environment λ , a set of formulæ Σ is said to *entail* a formula σ ($\Sigma \models_{\lambda} \sigma$) if the set of all models satisfied by Σ under λ is a subset of the set of all models satisfied by σ under λ .

The notation used for satisfiability and entailment is very similar, in that the operator used (\models) is the same, but they can be distinguished by the type of left operand.

2.7 Homomorphisms

A *homomorphism* from \mathbb{A} to \mathbb{B} is a function $h : |\mathbb{A}| \rightarrow |\mathbb{B}|$ such that, for each relation symbol R and tuple $\langle a_0, \dots, a_n \rangle$ where $a \subseteq |\mathbb{A}|$, $\langle a_0, \dots, a_n \rangle \in R^{\mathbb{A}}$ implies $\langle h(a_0), \dots, h(a_n) \rangle \in R^{\mathbb{B}}$.

A homomorphism h is also a *strong homomorphism* if, for each relation symbol R and tuple $\langle a_0, \dots, a_n \rangle$ where $a \subseteq |\mathbb{A}|$, $\langle a_0, \dots, a_n \rangle \in R^{\mathbb{A}}$ if and only if $\langle h(a_0), \dots, h(a_n) \rangle \in R^{\mathbb{B}}$.

The notation $\mathbb{M} \preceq \mathbb{N}$ means that there exists a homomorphism $h : \mathbb{M} \rightarrow \mathbb{N}$. The identity function is a homomorphism from any model \mathbb{M} to itself. Homomorphisms have the property that $\mathbb{A} \preceq \mathbb{B} \wedge \mathbb{B} \preceq \mathbb{C}$ implies $\mathbb{A} \preceq \mathbb{C}$.

However, $\mathbb{M} \preceq \mathbb{N} \wedge \mathbb{N} \preceq \mathbb{M}$ does not imply that $\mathbb{M} = \mathbb{N}$, but instead that \mathbb{M} and \mathbb{N} are *homomorphically equivalent*. For example, fix two models \mathbb{M} and \mathbb{N} that are equivalent except that \mathbb{N} has one more domain element than \mathbb{M} . Both $\mathbb{M} \preceq \mathbb{N}$ and $\mathbb{N} \preceq \mathbb{M}$ are true, yet $\mathbb{M} \neq \mathbb{N}$. Homomorphic Equivalence between a model \mathbb{M} and a model \mathbb{N} is denoted $\mathbb{M} \simeq \mathbb{N}$.

Given models \mathbb{M} and \mathbb{N} where $\mathbb{M} \preceq \mathbb{N}$ and a formula in positive-existential form σ , if $\mathbb{M} \models \sigma$ then $\mathbb{N} \models \sigma$.

A homomorphism $h : \mathbb{A} \rightarrow \mathbb{B}$ is also an *isomorphism* when h is 1:1 and onto and the inverse function $h^{-1} : \mathbb{B} \rightarrow \mathbb{A}$ is a homomorphism.

2.8 Minimal Models

Minimal models, also called *universal* models, are models for a theory with the special property that there exists a homomorphism from the minimal model to any other model that satisfies the theory. Intuitively, minimal models have no unnecessary entities or relations and thus display the least amount of constraint necessary to satisfy the theory for which they are minimal.

More than one minimal model may exist for a given theory, and not every theory must have a minimal model. **give examples.**

A set of models \mathcal{M} is said to be *jointly minimal* for a set of formulæ Σ when every model \mathbb{N} such that $\mathbb{N} \models \Sigma$ has a homomorphism from a model $\mathbb{M} \in \mathcal{M}$ to \mathbb{N} .

2.9 Positive Existential Form

Formulæ in *positive existential form* are constructed using only conjunctions (\wedge), disjunctions (\vee), existential quantifications (\exists), tautologies (\top), contradictions (\perp), equalities, and relations.

Negation of a relation R with arity k can be implemented by assuming another relation R' with arity k , adding two formulæ to the theory of the form $R \wedge R' \rightarrow \perp$ and $\top \rightarrow R \vee R'$, and using R' where $\neg R$ would be used.

2.10 Geometric Logic

Geometric logic formulæ are implicitly universally quantified implications between positive existential formulæ. More specifically, a geometric logic formula is of the form

$$\forall (free(F_L) \cup free(F_R)) : F_L \rightarrow F_R$$

where *free* is the function that returns the set of all free variables for a given formula and both F_L and F_R are first-order logic formulæ in positive existential form.

A set of geometric logic formulæ is called a *geometric theory*.

It is convention to treat a positive existential formula σ as $\top \rightarrow \sigma$ when expecting a geometric logic formula. It is also convention to treat a negated positive existential formula $\neg\sigma$ as $\sigma \rightarrow \perp$.

Examples of geometric logic formulæ

<i>reflexivity</i>	$\top \rightarrow R[x, x]$
<i>symmetry</i>	$R[x, y] \rightarrow R[y, x]$
<i>transitivity</i>	$R[x, y] \wedge R[y, z] \rightarrow R[x, z]$

3 The Chase

The *chase* is a function that, when given a geometric theory, will generate a set of jointly minimal models for that theory. More specifically, if \mathcal{U} is the set of all models obtained from an execution of the chase over a geometric theory T , for any model \mathbb{M} such that $\mathbb{M} \models T$, there is a homomorphism from some model $\mathbb{U} \in \mathcal{U}$ to \mathbb{M} .

Geometric logic formulæ are used by the chase because they have the useful property where adding any relations or domain members to a model that satisfies a geometric logic formula will never cause the formula to no longer be satisfied. This is particularly helpful when trying to create a model that satisfies all formulæ in a geometric theory.

There are three types of runs of the chase:

- a non-empty result in finite time
- an empty result in finite time
- an infinite run, with possible return dependent on implementation

3.1 Algorithm

3.2 Examples

Define Σ as the following geometric theory

$$\top \rightarrow \exists y, z : R[y, z] \tag{1}$$

$$R[x, w] \rightarrow (\exists y : Q[x, y]) \vee (\exists z : P[x, z]) \tag{2}$$

$$Q[u, v] \rightarrow (\exists z : R[u, z]) \vee (\exists z : R[z, w]) \tag{3}$$

$$P[u, v] \rightarrow \perp \tag{4}$$

The following three chase runs show the different types of results depending on which disjunct the algorithm attempts to satisfy when a disjunction is encountered

A non-empty result in finite time:

$$\begin{aligned} \emptyset &\mapsto \{ \quad a, b \quad \mid \quad R[a, b] \quad \} \\ &\mapsto \{ \quad a, b, c \quad \mid \quad R[a, b], Q[a, c] \quad \} \end{aligned}$$

An empty result in finite time:

$$\begin{aligned}
\emptyset &\mapsto \{ \quad a, b \quad \mid \quad R[a, b] \quad \} \\
&\mapsto \{ \quad a, b, c \quad \mid \quad R[a, b], P[a, c] \quad \} \\
&\mapsto \{ \quad a, b, c \quad \mid \quad R[a, b], P[a, c], \perp \quad \} \\
&\mapsto \varepsilon
\end{aligned}$$

An infinite run:

$$\begin{aligned}
\emptyset &\mapsto \{ \quad a, b \quad \mid \quad R[a, b] \quad \} \\
&\mapsto \{ \quad a, b, c \quad \mid \quad R[a, b], Q[a, c] \quad \} \\
&\mapsto \{ \quad a, b, c, d \quad \mid \quad R[a, b], Q[a, c], R[d, c] \quad \} \\
&\mapsto \{ \quad a, b, c, d, e \quad \mid \quad R[a, b], Q[a, c], R[d, c], Q[d, e] \quad \} \\
&\mapsto \{ \quad a, b, c, d, e, f \quad \mid \quad R[a, b], Q[a, c], R[d, c], Q[d, e], R[f, e] \quad \} \\
&\mapsto \dots
\end{aligned}$$

4 Haskell Chase Implementation

The goal of the implementation of the chase is to deterministically find all possible outcomes of the chase. It does this by forking and taking all paths when encountering a disjunct rather than nondeterministically choosing one disjunct to satisfy.

The results from the attempts to satisfy each disjunct are returned as a list. The returned list will not contain an entry for runs that return no model, and will merge lists returned from runs that themselves encountered a disjunct. The lazy evaluation of Haskell allows a user to access members of the returned list even though some chase runs have not returned a value.

Appendix B contains the chase-running portions of the implementation.

4.1 Operation

The first step of the chase implementation is to verify that each formula of the given theory is a geometric logic formula. If a formula φ is not a geometric logic formula, the chase tries to coerce it into one using the following algorithm:

```
switch  $\varphi$  do
  case  $\neg\alpha$ 
    if  $\alpha$  is in positive existential form then
       $\varphi$  is replaced with  $\alpha \rightarrow \perp$ 
    else error
  otherwise
    if  $\varphi$  is in positive existential form then
       $\varphi$  is replaced with  $\top \rightarrow \varphi$ 
    else error
```

After the input verification and coercion step, the *chase* function sorts the input formulæ by the number of disjuncts on the right side of the implication. This step will cause each fork of the algorithm to finish in less time, as they are likely to halt before forking yet again.

Once the input formulæ are sorted, the *chase* function begins processing a *pending* list, which is initially populated with a single model that has an empty domain and no facts. In the special case where *chase* is run on an empty list, an empty list of models is returned.

For each *pending* model, each formula is evaluated to see if it holds in the model for all environments. If an environment is found that does not satisfy the model, the model and environment in which the formula did not hold is passed to the *satisfy* function, along with the formula that needs to be satisfied. The list of models returned from *satisfy*

is merged into the *pending* list, and the result of running *chase* on the new *pending* list is returned. If, however, the model holds for all formulæ in the theory and all possible associated environments, it is concatenated with the result of running the chase on the rest of the models in the *pending* list.

The *satisfy* function performs a pattern match on the type of formula given. Assuming *satisfy* is given a model \mathbb{M} , an environment λ , and a formula φ , *satisfy* will behave as outlined in the following algorithm.

Algorithm: $\text{satisfy} :: \text{Model} \rightarrow \text{Environment} \rightarrow \text{Formula} \rightarrow [\text{Model}]$

```

return switch  $\varphi$  do
  case  $\top$  return a list containing  $\mathbb{M}$ 
  case  $\perp$  return an empty list
  case  $x = y$  return a list containing  $\text{quotient}(\mathbb{M})$ 
  case  $\alpha \vee \beta$  return  $\text{satisfy}(\alpha) \cup \text{satisfy}(\beta)$ 
  case  $\alpha \wedge \beta$ 
    create an empty list  $r$ 
    foreach model  $m$  in  $\text{satisfy}(\alpha)$  do
      union  $r$  with  $\text{satisfy}(\beta)$ 
    return  $r$ 
  case  $\alpha \rightarrow \beta$ 
    if  $\mathbb{M}_\lambda \models \alpha$  then return  $\text{satisfy}(\beta)$ 
    else return an empty list
  case  $R[\vec{x}]$ 
    define a new model  $\mathbb{N}$  where  $|\mathbb{N}| = |\mathbb{M}|$ 
    add a new element  $\omega$  to  $|\mathbb{N}|$ 
    forall the  $P_{\mathbb{M}}$  do  $P_{\mathbb{N}} = P_{\mathbb{M}}$ 
    define  $R_{\mathbb{N}}[x_0 \dots x_n]$  as  $R_{\mathbb{M}}[\lambda(x_0) \dots \lambda(x_n)]$ 
    foreach  $v \in \vec{x}$  do
      if  $v \notin \lambda$  then  $\lambda$  becomes  $\lambda_{v \mapsto \omega}$ 
    return a list containing  $\mathbb{N}$ 
  case  $\exists \vec{x} : \alpha$ 
    if  $\vec{x} = \emptyset$  then recurse on  $\alpha$ 
    if  $|\mathbb{M}| \neq \emptyset$  and  $\exists v' \in |\mathbb{M}| : (\lambda' = \lambda_{x_0 \mapsto v'} \text{ and } \mathbb{M} \models_{\lambda'} \alpha)$  then
      return a list containing  $\mathbb{M}$ 
    else
      define a new model  $\mathbb{N}$  where  $|\mathbb{N}| = |\mathbb{M}|$ 
      add an element  $\omega$  to  $|\mathbb{N}|$  such that  $\omega \notin |\mathbb{N}|$ 
      forall the  $R_{\mathbb{M}}$  do  $R_{\mathbb{N}} = R_{\mathbb{M}}$ 
      define  $\kappa = \lambda_{x_0 \mapsto \omega}$ 
      using model  $\mathbb{N}$  and environment  $\kappa$ , return  $\text{satisfy}(\exists \{x_1 \dots x_n\} : \alpha)$ 

```

4.2 Options

4.2.1 I/O

4.2.2 Tracing

4.3 Future Considerations

5 An Extended Application: Cryptographic Protocol Analysis

A Table of Syntax

syntax	definition
f^{-1}	the inverse function of f
$R[a_0, a_1, a_2]$	a <i>relation</i> of: relation symbol R , arity 3, and tuple $\langle a_0, a_1, a_2 \rangle$
\top	a <i>tautological formula</i> ; one that will always hold
\perp	a <i>contradictory formula</i> ; one that will never hold
$\rho = \tau$	given assumed environment λ , $\lambda(\rho) = \lambda(\tau)$
$\neg\alpha$	α does not hold
$\alpha \wedge \beta$	both α and β hold
$\alpha \vee \beta$	either α or β hold
$\alpha \rightarrow \beta$	either α does not hold or β holds
$\forall x : \alpha$	for each element of the domain as x , α holds
$\forall \vec{x} : \alpha$	for each $x_i \in x$, $\forall x_i : \alpha$ holds
$\exists x : \alpha$	for each at least one element of the domain as x , α holds
$\exists \vec{x} : \alpha$	for each $x_i \in x$, $\exists x_i : \alpha$ holds
$\lambda[x \mapsto y]$	the environment λ with variable x mapped to domain member y
$\mathbb{M} \models_l \sigma$	\mathbb{M} is a <i>model of</i> σ under environment l
$\mathbb{M} \models \sigma$	$\mathbb{M} \models_l \sigma$ given any environment l
$\mathbb{M} \models_l \Sigma$	for each $\sigma \in \Sigma$, $\mathbb{M} \models_l \sigma$
$\mathbb{M} \models \Sigma$	for each $\sigma \in \Sigma$, $\mathbb{M} \models \sigma$
$\Sigma \models \sigma$	Σ <i>entails</i> σ
$\mathbb{M} \preceq \mathbb{N}$	there exists a <i>homomorphism</i> $h : \mathbb{M} \rightarrow \mathbb{N} $
$\mathbb{M} \simeq \mathbb{N}$	\mathbb{M} and \mathbb{N} are <i>homomorphically equivalent</i>

B Chase code

TODO: make the long lines short so they fit

```
1 module Chase where
2 import Parser
3 import Helpers
4 import qualified Debug.Trace
5 import Data.List
6
7 trace x = id
8 — trace = Debug.Trace.trace
9
10 verify :: Formula -> Formula
11 — verifies that a formula is in positive existential form and performs some
12 — normalization on implied/constant implications
13 verify formula =
14   let isNotPEF = not.isPEF in
15   case formula of
16     Implication a b ->
17       if isNotPEF a || isNotPEF b then
18         error ("implication_must_be_in_positive_existential_form:" ++ showFormula formula)
19       else formula
20     Not f ->
21       if isNotPEF f then
22         error ("formula_must_be_in_positive_existential_form:" ++ showFormula formula)
23       else (Implication f Contradiction)
24     - ->
25       if isNotPEF formula then
26         error ("formula_must_be_in_positive_existential_form:" ++ showFormula formula)
27       else (Implication Tautology formula)
28
29 order :: [Formula] -> [Formula]
30 —
31 order formulae = sortBy (\a b ->
32   let extractRHS = (\(Implication lhs rhs) -> rhs) in
33   let (rhsA, rhsB) = (extractRHS a, extractRHS b) in
34   let (lenA, lenB) = (numDisjuncts rhsA, numDisjuncts rhsB) in
35   if lenA == lenB then EQ
36   else
37     if lenA < lenB then LT
38     else GT
39 ) formulae
40
41 chase :: [Formula] -> [Model]
42 — a wrapper for the chase' function to hide the model identity and theory
43 — manipulation
44 chase formulae = nub $ chase' (order $ map verify formulae) [([]), []])
45
46 chase' :: [Formula] -> [Model] -> [Model]
47 — runs the chase algorithm on a given theory, manipulating the given list of
48 — models, and returning a list of models that satisfy the theory
49 chase' formulae [] = []
50 chase' formulae pending@(m:rest) =
51   let self = chase' formulae in
52   trace ("running_chase_on_" ++ show pending) $
53   case findFirstFailure m formulae of
54     Just newPending ->
55       trace ("at_least_one_formula_does_not_hold_for_model_" ++ showModel m) $
56       trace ("unioning_" ++ show rest ++ "_with_" ++ intercalate ", " (map showModel newPending)) $
57       self (union rest newPending)
58     Nothing -> — represents no failures
59       trace ("all_formulae_in_theory_hold_for_model_" ++ showModel m) $
60       trace ("moving_model_into_done_list") $
```

```

61         m : self rest
62
63 findFirstFailure :: Model -> [Formula] -> Maybe [Model]
64 —
65 findFirstFailure model [] = Nothing — no failure found
66 findFirstFailure model@(domain,relations) (f:ormulae) =
67     let self = findFirstFailure model in
68     let bindings = allBindings (freeVariables f) domain [] in
69     if holds model (UniversalQuantifier (freeVariables f) f) then self ormlae
70     else Just $ findFirstBindingFailure model f bindings
71
72 findFirstBindingFailure :: Model -> Formula -> [Environment] -> [Model]
73 —
74 findFirstBindingFailure model formula (e:es) =
75     let self = findFirstBindingFailure model formula in
76     if holds' model e formula then self es
77     else
78         trace ("Attempting to satisfy (" ++ showFormula formula ++ ") with env " ++ show e) $
79         satisfy model e formula
80
81 satisfy :: Model -> Environment -> Formula -> [Model]
82 —
83 satisfy model env formula =
84     let (domain,relations) = model in
85     let domainSize = length domain in
86     let self = satisfy model in
87     case formula of
88         Tautology -> [model]
89         Contradiction -> []
90         Or a b -> union (self env a) (self env b)
91         And a b -> concatMap (\m -> satisfy m env b) (self env a)
92         Equality v1 v2 -> case (lookup v1 env,lookup v2 env) of
93             (Just v1, Just v2) -> [quotient model v1 v2]
94             _ -> error("Could not look up one of \" ++ variableName v2 ++ "\" or \" ++ variableName
95             Implication a b -> if holds' model env a then self env b else []
96         Atomic predicate vars ->
97             let newRelationArgs = genNewRelationArgs env vars (fromIntegral (length domain)) in
98             let newRelation = mkRelation predicate (length vars) [newRelationArgs] in
99             let newModel = mkModel (mkDomain domainSize) (mergeRelation newRelation relations) in
100             trace ("Adding new relation:" ++ show newRelation) $
101             [newModel]
102         ExistentialQuantifier [] f -> self env f
103         ExistentialQuantifier (v:vs) f ->
104             let f' = ExistentialQuantifier vs f in
105             let nextDomainElement = fromIntegral $ (length domain) + 1 in
106             if (domain /= []) && (any (\v' -> holds' model (hashSet env v v') f') domain) then
107                 trace (" " ++ showFormula formula ++ " already holds") $
108                 [model]
109             else
110                 trace ("Adding new domain element " ++ show nextDomainElement ++ " for variable " ++
111                 satisfy (mkDomain nextDomainElement,relations) (hashSet env v nextDomainElement) f'
112             -> error ("formula not in positive existential form:" ++ showFormula formula)
113
114 genNewRelationArgs :: Environment -> [Variable] -> DomainElement -> [DomainElement]
115 — for each Variable in the given list of Variables, retrieves the value
116 — assigned to it in the given environment, or the next domain element if it
117 — does not exist
118 genNewRelationArgs env [] domainSize = []
119 genNewRelationArgs env (v:vs) domainSize =
120     let self = genNewRelationArgs env vs in
121     case lookup v env of
122         Just v' -> v' : self domainSize
123         _ -> domainSize+1 : self (domainSize+1)

```

References

- [1] A Cottrell, *Word Processors: Stupid and Inefficient*,
www.ecn.wfu.edu/~cottrell/wp.html