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

8th 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

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 members 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 section 2.10.

TODO: add those $\Uparrow$ 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 $\bot$ 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 $(\sigma \rightarrow \bot)$ respectively. If α is a formula and $\vec{x}$ is a set of variables of size k, then $(\forall \vec{x} : \alpha)$ is $(\forall x_0 \ldots \forall x_{k-1} : \alpha)$. If α is a formula and $\vec{x}$ is a set of variables of size k, then $(\exists \vec{x} : \alpha)$ is $(\exists x_0 \ldots \exists x_{k-1} : \alpha)$.

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 occuring in an atomic formula is a free variable
- the set of free variables in $\top$ and $\bot$ 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 member 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 $R[\lambda(a_0), \ldots, \lambda(a_n)] \in \mathbb{M}$ 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 "$\mathbb{M}$ 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 $\langle a_0, \ldots, a_n \rangle$ where $a \subseteq |\mathbb{A}|$, $\langle a_0, \ldots, a_n \rangle \in R^{\mathbb{A}}$ implies $\langle h(a_0), \ldots, h(a_n) \rangle \in R^{\mathbb{B}}$.

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

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 member 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 ($\bot$), 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 $R \wedge R' \rightarrow \bot$ 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 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 $\sigma \rightarrow \bot$.

Examples of geometric logic formulæ

reflexivity	$\top \rightarrow R[x, x]$
symmetry	$R[x, y] \rightarrow R[y, x]$
transitivity	$R[x, y] \wedge R[y, z] \rightarrow R[x, z]$

3 The Chase

The *chase* is a function that, when given a gemoetric 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 satisified. This is particularly helpful when trying to create a model that satisifies 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

The chase algorithm begins by defining an empty list of models to return $\mathcal{D}$ and a list of pending models $\mathcal{P}$ containing a model with an empty domain and an empty set of facts. A single input is provided to the chase: the geometric theory Σ.

Before anything else, the chase algorithm sorts Σ by the number of disjunctions on the right side of each formula's implication. Formulæ with zero disjunctions will be ordered first, followed by those with one disjunction, and finally those with more than one disjunction. Within each sorting classification, formulæ should remain in their original order.

For every model $\mathbb{M} \in \mathcal{P}$, the chase loops through every formula $\sigma \in \Sigma$. For any σ and environment λ such that $\mathbb{M} \not\models_\lambda \sigma$, $\mathbb{M}$ has domain members and facts added to it **as described in ¡algorithm below¿**. If $\mathbb{M} \models \Sigma$ instead, remove $\mathbb{M}$ from $\mathcal{P}$ and add it to $\mathcal{D}$.

algorithm for how to make a model satisfy a formula

Finally, the chase algorithm returns $\mathcal{D}$, a list of jointly minimal models for its input theory Σ.

3.2 Examples

Define Σ as the following geometric theory.

$$
\begin{aligned}
\top &\rightarrow \exists\ y, z : R[y, z] && (1)\\
R[x, w] &\rightarrow (\exists\ y : Q[x, y]) \vee (\exists\ z : P[x, z]) && (2)\\
Q[u, v] &\rightarrow (\exists\ z : R[u, z]) \vee (\exists\ z : R[z, w]) && (3)\\
P[u, v] &\rightarrow \bot && (4)
\end{aligned}
$$

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.

1. A non-empty result in finite time:

   $$
   \begin{array}{lllll}
   \emptyset & \mapsto & \{ & a, b & | \quad R[a, b] \} \\
   & \mapsto & \{ & a, b, c & | \quad R[a, b], Q[a, c] \}
   \end{array}
   $$

   Since the left side of **(1)** is always satisfied, but its right side is not, domain members a and b and fact $R[a, b]$ are added to the initially empty model to satisfy **(1)**. The left side of **(2)** holds, but the right side does not, so one of the disjuncts $\exists\ y : Q[x, y]$ or $\exists\ z : P[x, z]$ is chosen to be satisfied. Assuming the left operand is chosen, x will already have been assigned to a and a new domain member c and a new fact $Q[a, c]$ will be added to satisfy **(2)**. With the current model, all rules hold under any environment. Therefore, this model is in the set of jointly minimal models for our theory.

   This is worded for a simpleton. Is that okay?

2. An empty result in finite time:

   $$
   \begin{array}{lllll}
   \emptyset & \mapsto & \{ & a, b & | \quad R[a, b] \} \\
   & \mapsto & \{ & a, b, c & | \quad R[a, b], P[a, c] \} \\
   & \mapsto & \{ & a, b, c & | \quad R[a, b], P[a, c], \bot \} \\
   & \mapsto & \varepsilon & &
   \end{array}
   $$

   Again, domain members a and c and fact $R[a, b]$ are added to the initial model to satisfy **(1)**. This time, when attempting to satisfy **(2)**, the right side is chosen and $P[a, c]$ is added to the set of facts. After adding this new fact, rule **(4)** no longer holds; its left side is satisfied, but its right side does not hold for all of the bindings for which it is satisfied. When we attempt to satisfy the right side of **(4)**, it is found to be a contradiction and therefore unsatisfiable. Since this model can never satisfy this theory, the chase fails.

3. An infinite run:

   $$
   \begin{array}{lllll}
   \emptyset & \mapsto & \{ & a, b & | \quad R[a, b] \} \\
   & \mapsto & \{ & a, b, c & | \quad R[a, b], Q[a, c] \} \\
   & \mapsto & \{ & a, b, c, d & | \quad R[a, b], Q[a, c], R[d, c] \} \\
   & \mapsto & \{ & a, b, c, d, e & | \quad R[a, b], Q[a, c], R[d, c], Q[d, e] \} \\
   & \mapsto & \{ & a, b, c, d, e, f & | \quad R[a, b], Q[a, c], R[d, c], Q[d, e], R[f, e] \} \\
   & \mapsto & \ldots & &
   \end{array}
   $$

Like in the example above that returned a non-empty, finite result, the first two steps add domain members a, b, and c and facts $R[a, b]$ and $Q[a, c]$.

it would never run infinitely like this... ask me why

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 make sure that each formula of the given theory can be represented as a geometric logic formula. If a formula φ can not be coerced to a geometric logic formula, the chase tries to coerce it into one using the following algorithm:

switch φ **do**
- **case** $\neg\alpha$
  - **if** α *is in positive existential form* **then**
    - φ is replaced with $\alpha \to \bot$
  - **else** error
- **otherwise**
  - **if** φ *is in positive existential form* **then**
    - φ is replaced with $\top \to \varphi$
  - **else** error

The chase function then sorts the input formulæ by the number of disjunctions on the right side of the implication. it allows branches to terminate without growing to an unnecessarily (and possibly infinitely) large size. This step will cause each branch of the algorithm to finish in less time, as they are likely to halt before branching yet again. The formulæ are not sorted purely by absolute number of disjunctions on the right side, but by whether there are zero, one, or many disjunctions. This is done to avoid unnecessary re-ordering for no gain because formulæ with no disjunctions or only a single disjunct on the right side are more likely to cause a branch to stop growing than one with many disjunctions. Likewise, formulæ with zero disjunctions are more likely to cause a branch to halt than those with one or more disjunctions.

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.

For each *pending* model, each formula is evaluated to see if it holds in the model for all environments. If an envionment 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: satisfy :: Model → Environment → Formula → [Model]
return **switch** φ **do**
- **case** $\top$ return a list containing $\mathbb{M}$
- **case** $\bot$ return an empty list
- **case** $x = y$ return a list containing $quotient(\mathbb{M})$
- **case** $\alpha \vee \beta$ return $satisfy(\alpha) \cup satisfy(\beta)$
- **case** $\alpha \wedge \beta$
  - create an empty list r
  - **foreach** *model* m *in* $satisfy(\alpha)$ **do**
    - union r with $satisfy(\beta)$
  - return r
- **case** $\alpha \rightarrow \beta$
  - **if** $\mathbb{M}_\lambda \models \alpha$ **then** return $satisfy(\beta)$
  - **else** return an empty list
- **case** $R[\vec{x}]$
  - define a new model $\mathbb{N}$ where $|\mathbb{N}| = |\mathbb{M}|$
  - add a new member ω to $|\mathbb{N}|$
  - **forall the** $P_\mathbb{M}$ **do** $P_\mathbb{N} = P_\mathbb{M}$
  - define $R_\mathbb{N}[x_0 \ldots x_n]$ as $R_\mathbb{M}[\lambda(x_0) \ldots \lambda(x_n)]$
  - **foreach** $v \in \vec{x}$ **do**
    - **if** $v \notin \lambda$ **then** λ becomes $\lambda_{v \mapsto \omega}$
  - return a list containing $\mathbb{N}$
- **case** $\exists\ \vec{x} : \alpha$
  - **if** $\vec{x} = \emptyset$ **then** recurse on α
  - **if** $|\mathbb{M}| \neq \emptyset$ *and* $\exists\ v' \in |\mathbb{M}| : (\lambda' = \lambda_{x_0 \mapsto v'}$ *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 a member ω 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 κ, return $satisfy(\exists\ \{x_1 \ldots x_n\} : \alpha)$

4.2 Input Format

Input to the parser must be in a form parsable by the following context-free grammar. Terminals are denoted by a `monospace style` and nonterminals are denoted by an *obliquestyle*. The greek letter ε matches a zero-length list of tokens. Patterns that match non-literal terminals are defined in the table following the grammar.

program : ε
| *exprList optNEWLINE*

exprList : *expr*
| *exprList* NEWLINE *expr*

expr : TAUTOLOGY
| CONTRADICTION
| *expr* OR *expr*
| *expr* AND *expr*
| NOT *expr*
| *expr* -> *expr*
| -> *expr*
| *atomic*
| VARIABLE EQ VARIABLE
| FOR_ALL *argList optCOLON expr*
| THERE_EXISTS *argList optCOLON expr*
| (*expr*)
| [*expr*]

atomic : PREDICATE *index*

index : (*argList*)
| [*argList*]

argList : *arg*
| *argList* , *arg*

arg : VARIABLE

optCOLON : ε
| :

optNEWLINE : ε
| NEWLINE

Input Pattern	Terminal
|	OR
&	AND
!	NOT
=	EQ
[Tt]autology	TAUTOLOGY
[Cc]ontradiction	CONTRADICTION
[\r\n]+	NEWLINE
[a-z][A-Za-z0-9_']*	VARIABLE
[A-Z][A-Za-z0-9_']*	PREDICATE
For[Aa]ll	FOR_ALL
Exists	THERE_EXISTS

Comments are removed at the lexical analysis step and have no effect on the input to the parser. Single-line comments begin with either a hash (#) or double-dash (--). Multiline comments begin with /* and are terminated by */.

4.3 Options

4.3.1 I/O

When no options are given to the executable output by Haskell, it expects input from stdin and outputs models in a human-readable format to stdout. To take input from a file instead, pass the executable the -i or --input option followed by the filename.

To output models to numbered files in a directory, pass the -o or --output option along with an optional directory name. The given directory does not have to exist. If the output directory is omitted, it defaults to "./models".

Using the -o or --output options will change the selection for output format to a machine-readable format. To switch ouput formats at any time, pass the -h or -m flags for human-readable and machine-readable formats respectively.

4.3.2 Tracing

not yet implemented

4.4 Future Considerations

This section details areas of possible improvement/development that were unable to be explored during the timeframe allotted for the project.

4.4.1 Better Data Structures

This project was started with no knowledge of the most desirable implementation language, Haskell. Because of this, some less-than-optimal data structures were used to hold data that should really be in a Data.Map or Data.Set. One such example of this is with the truth table contained in relations. This truth table should probably be implemented as a Data.Map. Also, instead of Domains being a list of DomainMember, it would probably be better if a Domain was a Data.Set.

4.4.2 Broader use of the Maybe Monad

4.4.3 Advanced Rewriters / Simplifiers

talk about purple book and a reverse pullquants

5 An Extended Application: Cryptographic Protocol Analysis

5.1 The Problem

5.2 The Solution: Minimal Models

5.3 Designing An Analagous Theory

5.4 The Results

5.5 Derived Chase Implementation Enhancements

A Table of Syntax

syntax	definition
f^{-1}	the inverse function of f
$R[a_0, a_1, a_2]$	a *relation* of: relation symbol R, arity 3, and tuple $\langle a_0, a_1, a_2 \rangle$
$\top$	a *tautological formula*; one that will always hold
$\bot$	a *contradictory formula*; 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 member of the domain as x, α holds
$\forall \vec{x} : \alpha$	**FIXME:** for each $x_i \in x$, $\forall x_i : \alpha$ holds
$\exists x : \alpha$	for at least one member of the domain as x, α holds
$\exists \vec{x} : \alpha$	**FIXME:** 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 model of* 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 *entails* σ
$\mathbb{M} \preceq \mathbb{N}$	there exists a *homomorphism* $h : \lvert\mathbb{M}\rvert \rightarrow \lvert\mathbb{N}\rvert$
$\mathbb{M} \simeq \mathbb{N}$	$\mathbb{M}$ and $\mathbb{N}$ are *homomorphically equivalent*

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 = case formula of
14    Implication a b -> Implication (pef a) (pef b)
15    Not f -> Implication (pef f) Contradiction
16    _ -> Implication Tautology (pef formula)
17
18 order :: [Formula] -> [Formula]
19 --
20 order formulae = sortBy (\a b ->
21    let extractRHS = (\f -> case f of; (Implication lhs rhs) -> rhs; _ -> f) in
22    let (rhsA,rhsB) = (extractRHS a, extractRHS b) in
23    let (lenA,lenB) = (numDisjuncts rhsA, numDisjuncts rhsB) in
24    let (vA,vB) = (length (variables a), length (variables b)) in
25    if lenA == lenB || lenA > 1 && lenB > 1 then
26       if vA == vB then EQ
27       else
28          if vA < vB then LT
29          else GT
30    else
31       if lenA < lenB then LT
32       else GT
33    ) formulae
34
35 chase :: [Formula] -> [Model]
36 -- a wrapper for the chase' function to hide the model identity and theory
37 -- manipulation
38 chase theory = nub $ chase' (order $ map verify theory) [([],[])]
39
40 chase' :: [Formula] -> [Model] -> [Model]
41 -- runs the chase algorithm on a given theory, manipulating the given list of
42 -- models, and returning a list of models that satisfy the theory
43 chase' _ [] = []
44 chase' theory pending =
45    trace ("running chase on " ++ show pending) $
46    concatMap (branch theory) pending
47
48 branch :: [Formula] -> Model -> [Model]
49 --
50 branch theory model =
51    let reBranch = chase' theory in
52    case findFirstFailure model theory of
53       Just newModels ->
54          trace ("  at least one formula does not hold for model " ++ showModel model) $
55          reBranch newModels
56       Nothing -> -- represents no failures
57          trace ("  all formulae in theory hold for model " ++ showModel model) $
58          trace ("  moving model into done list") $
59          [model]
60
```

```
61  findFirstFailure :: Model -> [Formula] -> Maybe [Model]
62  --
63  findFirstFailure model [] = Nothing -- no failure found
64  findFirstFailure model@(domain,relations) (f:ormulae) =
65     let self = findFirstFailure model in
66     let bindings = allBindings (freeVariables f) domain [] in
67     trace ("  checking formula: (v:" ++ show (length$variables f) ++ ") (fv:" ++ show (length$freeVa
68     if holds model (UniversalQuantifier (freeVariables f) f) then self ormulae
69     else Just $ findFirstBindingFailure model f bindings
70
71  findFirstBindingFailure :: Model -> Formula -> [Environment] -> [Model]
72  --
73  findFirstBindingFailure _ (Implication a Contradiction) _ = []
74  findFirstBindingFailure model formula@(Implication a b) (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 (" ++ show b ++ ") with env " ++ show e) $
79        satisfy model e b
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 \"" ++ show v2 ++ "\" or \"" ++ show v2 ++ "\" in env
95        Atomic predicate vars ->
96           let newRelationArgs = genNewRelationArgs env vars (fromIntegral (length domain)) in
97           let newRelation = mkRelation predicate (length vars) [newRelationArgs] in
98           let newModel = mkModel (mkDomain domainSize) (mergeRelation newRelation relations) in
99           trace ("    adding new relation: " ++ show newRelation) $
100          [newModel]
101       ExistentialQuantifier [] f -> self env f
102       ExistentialQuantifier (v:vs) f ->
103          let f' = ExistentialQuantifier vs f in
104          let nextDomainMember = fromIntegral $ (length domain) + 1 in
105          if (domain /= []) && (any (\d -> holds' model (hashSet env v d) f') domain) then
106             trace ("    " ++ show formula ++ " already holds") $
107             [model]
108          else
109             trace ("    adding new domain element " ++ show nextDomainMember ++ " for variable " ++
110             satisfy (mkDomain nextDomainMember,relations) (hashSet env v nextDomainMember) f'
111       _ -> error ("formula not in positive existential form: " ++ show formula)
112
113 genNewRelationArgs :: Environment -> [Variable] -> DomainMember -> [DomainMember]
114 -- for each Variable in the given list of Variables, retrieves the value
115 -- assigned to it in the given environment, or the next domain element if it
116 -- does not exist
117 genNewRelationArgs env [] domainSize = []
118 genNewRelationArgs env (v:vs) domainSize =
119    let self = genNewRelationArgs env vs in
120    case lookup v env of
121       Just v' -> v' : self domainSize
122       _ -> domainSize+1 : self (domainSize+1)
```

References

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