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

12th 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

This document details work done for a Major Qualifying Project at Worcester Polytechnic Institute by Michael Ficarra in partial fulfillment of the requirements for a Bachelor of Science degree.

1.1 Goals

The two main goals of this Major Qualifying Project are:

1. first, to implement an algorithm known as "the chase" accurately and with a well-defined, usable interface
2. second, to use the chase implementation for a real-world application: generating models used in analysis of a specific protocol

Secondary goals include implementing various optimizations and integrating the chase implementation into a program that can take advantage of the functionality it provides.

1.2 The Chase

The chase is an algorithm used to find jointly minimal models for a set of geometric logic formulæ. Many common real-world problems can be expressed as a set of geometric logic formulæ. When these problems have an unbounded scope of possible solutions, the chase can be used to find the possible solutions that are interesting. This allows researchers to go through only models that will behave differently than most models, rather than testing each on separately.

more introduction stuff here

2 Technical Background

2.1 Vocabulary

A *vocabulary* consists of a set of pairings of *relation symbols*, with a non-negative integral arity.

A relation symbol, often called a *predicate*, can be any unique symbol. It is used in conjunction with an arity to refer to relations in models.

2.2 Models

A *model* $\mathbb{M}$ for a vocabulary $\mathcal{V}$ is a construct that consists of:

- a set, denoted $|\mathbb{M}|$, called the *universe* or *domain* of $\mathbb{M}$
- for each pairing of a predicate R and an arity k in $\mathcal{V}$, 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.

2.3 First-order Logic

First-order logic, also called *predicate logic*, is a formal logic system. 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

For our purposes, this logic system will not contain any constant symbols or function symbols, which are commonly included in first-order logic. A function $f(x, y)$ can be encoded as the geometric logic formula $F(x, y, z1) \wedge F(x, y, z2) -> z1 = z2$. Constant symbols can be encoded as relations with an arity of zero, for example $C()$.

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 $(\exists x_0 \ldots \exists x_{k-1} : \alpha)$.

2.4 Variable Binding

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 $\bot$ 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 $free(\alpha) = \emptyset$.

2.5 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.

2.6 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 the empty environment l, $\mathbb{M} \models_l \sigma$.

A model $\mathbb{M}$ satisfies a set of formulæ Σ under 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".

2.7 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 satisfying 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.

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

2.9 Minimal Models

Minimal models, also called *universal* models, are models for a theory T with the special property that there exists a homomorphism from the minimal model to any other model that satisfies T. 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.

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 $\mathbb{N}$.

More than one minimal model may exist for a given theory. Given a model $\mathbb{M}$ that is minimal for a theory T, any model $\mathbb{N}$ such that $\mathbb{N} \simeq \mathbb{M}$ is also minimal for T.

Not every theory must have a minimal model. A simple example of this is the theory containing a single formula σ where σ contains a disjunction. There exists no single minimal model for the formula $P \vee Q$ because any model that satisfies both P and Q would not have a homomorphism to a model that satisfies the theory with only P or Q in its set of facts. However, the set containing a model $\mathbb{M}$ that contains P in its set of facts and a model $\mathbb{N}$ that contains Q in its set of fact would be jointly minimal.

2.10 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.

The set of models of a sentence σ is closed under homomorphisms if and only if σ is logically equivalent to a positive existential formula φ.

2.11 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	$R(x, y) \wedge R(y, z) \rightarrow R(x, z)$

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

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 set of jointly minimal models is found in finite time
- an empty result is found in finite time
- an infinite run with possible return dependent on implementation

3.1 Algorithm

The chase starts with a model $\mathbb{M}$ that has an empty domain and an empty set of facts. While there is a formula σ in the input theory T such that $\mathbb{M} \not\models \sigma$, perform a *chase correction*.

A *chase correction* performs the following steps:

1. choose formula $\sigma \in T$ such that $\mathbb{M} \not\models \sigma$
2. split this formula into the left and right sides of its implication, α and β
3. if β is a contradiction, halt with failure
4. a binding λ is chosen such that $\mathbb{M} \models_\lambda \alpha$
5. if β contains a disjunction, choose a disjunct and assign it to β
6. add a new domain member to the domain for every free variable not in the domain of λ
7. for every new domain element e and the variable v for which it was added, redefine λ as $\lambda_{v \mapsto e}$
8. for every atomic R, replace each variable v with $\lambda(v)$ and add it to the model

If no formula σ exists such that $\mathbb{M} \not\models \sigma$, halt with result $\mathbb{M}$.

3.2 Examples

Define Σ as the following geometric theory.

$$\begin{aligned}
\top &\rightarrow \exists\, x, y : R(x, y) && (1)\\
R(x, y) &\rightarrow (\exists\, z : Q(x, z)) \vee P && (2)\\
Q(x, y) &\rightarrow (\exists\, z : R(x, z)) \vee (\exists\, z : R(z, y)) && (3)\\
P &\rightarrow \bot && (4)\\
& && (5)
\end{aligned}$$

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:

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

   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 $\exists\, z : Q(x, z)$ or $P(x)$ 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 minimal.

2. An empty result in finite time:

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

   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 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:

$$
\begin{array}{lllll}
\emptyset & \mapsto & \{ \quad a, b & \mid & R(a,b) \quad \} \\
& \mapsto & \{ \quad a, b, c & \mid & R(a,b), Q(a,c) \quad \} \\
& \mapsto & \{ \quad a, \ldots, d & \mid & R(a,b), Q(a,c), R(d,c) \quad \} \\
& \mapsto & \{ \quad a, \ldots, e & \mid & R(a,b), Q(a,c), R(d,c), Q(d,e) \quad \} \\
& \mapsto & \{ \quad a, \ldots, f & \mid & R(a,b), Q(a,c), R(d,c), Q(d,e), R(f,e) \quad \} \\
& \mapsto & \ldots & &
\end{array}
$$

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 $Q(a,c)$. The left side of the implication in **(3)** now holds, but the right side does not. In order to make the right side hold, one of the disjuncts needs to be satisfied. If the right disjunct is chosen, a new domain member d and a new relation $R(d,c)$ will be added. This will cause the left side of the implication in **(2)** to hold for $R(d,c)$, but the right side will no hold for the same binding. $Q(d,e)$ will be added, and this loop will continue indefinitely unless a different disjunct is chosen in **(2)** or **(3)**.

3.3 Joint Minimality Theorem

Theorem 1. *A geometric theory T is satisfiable if and only if there is a chase run of T that returns a non-empty result.* ***do infinite runs imply satisfiability?***

Theorem 2. *Let $\mathcal{M}$ be the set of models returned from a successful run of the chase over a geometric theory T. For any model $\mathbb{N}$ such that $\mathbb{N} \models T$, there is a model $\mathbb{M} \in \mathcal{M}$ such that $\mathbb{M} \preceq \mathbb{N}$.*

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 by applying following rules recursively:

- $\neg\alpha \wedge \neg\beta \mapsto \alpha \vee \beta$
- $\neg\alpha \vee \neg\beta \mapsto \alpha \wedge \beta$
- $\neg\neg\alpha \mapsto \alpha$
- $\neg\alpha \rightarrow \beta \mapsto \alpha \vee \beta$
- $\alpha \rightarrow \neg\beta \mapsto \alpha \wedge \beta$
- $\neg(\forall\ \vec{x} : \neg\alpha) \mapsto \exists\vec{x} : \alpha$

All other constructs are preserved. If this transformed formula is still not in positive existential form, an error is thrown.

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. A secondary sort also occurs within these classifications that orders formulæ by the number of variables to reduce the number of bindings generated.

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 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 and returned.

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 $R_\mathbb{N}(x_0 \dots x_n)$ as $R_\mathbb{M}(\lambda(x_0) \dots \lambda(x_n))$
  - **foreach** $v \in \vec{x}$ **do**
    - **if** $v \notin \lambda$ **then** λ becomes $\lambda_{v \mapsto \omega}$
  - return a list containing $\mathbb{N}$
- **case** $\exists\ \vec{x} : \alpha$
  - **if** $\vec{x} = \emptyset$ **then** $satisfy(\alpha)$
  - **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 $satisfy(\exists\ \{x_1 \dots x_n\} : \alpha)$

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 *oblique style*. 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.

Should these be appendices?

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` *optIndex*

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

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

arg : `VARIABLE`

optIndex : ε
| *index*

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 (--). Multi-line comments begin with /* and are terminated by */.

4.3 Options

Help on the usage of the chase implementation can be found by passing the executable output by Haskell the `--help` or `-?` options.

When no options are given to the executable, 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 output formats at any time, pass the `-h` or `-m` flags for human-readable and machine-readable formats respectively.

4.4 Future Considerations

This section details areas of possible improvement/development.

4.4.1 Better Data Structures

Some less-than-optimal data structures are being 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 holding the

relation information of a model. 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

In several helper functions, the program's execution is halted and an error is output when the function receives certain invalid inputs. These functions should take advantage of the Maybe Monad and return `Maybe a` where they return `a`. One particular example of this is the `pef` function. When `pef` takes a formula as input that can not be converted to positive-existential form, it causes the program to produce an error and exit. Instead, `pef` should return `Maybe Formula` and the places where it is used should handle the error condition however they choose.

4.4.3 Binding Search Approach for Satisfaction Checking

When checking if a model with 30 domain members and 60 facts (a reasonable real-life usage example) holds for a formula with 5 universally quantified variables, 30^5 or $24,300,000$ bindings will have to be generated, and the formula will be checked under each one. But by limiting the checked bindings to only those that can produce facts that exist in the model from the atomics in the formula, only the 60 facts will have to be checked in the worst case, and only 1 in the best. In practical use, this should dramatically reduce the running time of theories that produce finite results.

4.4.4 Avoid Isomorphic Model Generation

By taking different paths to arrive at the same model, the chase often creates equivalent models that satisfy its given theory. These duplicate models are already being filtered from the output. However, another problem exists when the chase returns isomorphic models. Given two models, it is very computationally expensive to determine if there exists an isomorphism between them. If a fast method of determining if two models are isomorphic is found, including an implementation of it would provide more valuable results to the user.

4.4.5 Improve Efficiency

The main goals of this project were to write a correct chase implementation and apply it in a real-world situation. While reasonable and obvious optimizations were made, there is still plenty of room for optimization.

In Harrison's Practical Logic and Automated Reasoning **reference**, Harrison mentions a large number of formula rewriting and simplification algorithms, many of which are already implemented in the Helpers module. Those functions that are written, however, are not currently being used by the chase functions, and there are surely other functions that Harrison mentions that have not yet been written.

One specific example of a function that Harrison mentions is `pullQuants` **reference**, which pulls all quantifiers in a formula to the outside. This results in a formula with no conjunctions or disjunctions of quantified subformulæ. This function is implemented, but a function that does exactly the opposite of this will help speed up satisfaction checking when using the traditional looping method. This will minimize the number of times bindings need to be generated for all permutations of the current model's domain because the number of quantified variables will be reduced.

4.4.6 Tracing

Currently, Debug.Trace is being used to output real-time status for ease of debugging. This output is helpful to both developers of the chase implementation and theories that will be given as input. Unfortunately, all of the output can really slow down a chase run of a simple theory. Currently, the only way to disable tracing is to replace the definition

$$trace = Debug.Trace.trace$$

with

$$trace\ x = id$$

A flag is already being read in from the command line as `-d` and stored in the options record under `optDebug`.

5 An Extended Application: Cryptographic Protocol Analysis

The chase can be used for protocol analysis. A common technique for the analysis of protocols involves identifying the *essentially different* runs of the protocol. These essentially different protocol runs are analogous to minimal models. When a protocol is described using geometric logic, the chase can find such minimal models.

The protocol can then be analysed for characteristics such as the existence of security violations or other unexpected behaviour.

5.1 Background

5.1.1 Strand Spaces

The *strand space formalism* was developed as a method of formally reasoning about cryptographic protocols. Participants in a protocol run are represented by *strands*. A single physical entity can be represented as multiple strands if they play many roles in the protocol. A strand is made up of a sequence of nodes where every node either sends or receives a message.

Adversary strands represent the actions of participants that attempt to use the protocol for a purpose other than the one for which it was originally intended. Adversary strands are not bound by the rules defined by the protocol; they manipulate messages being sent and received by non-adversarial strands.

The capabilities of the adversary strands are given by the Dolev-Yao Threat Model **references**. Adversary strands are able to perform precisely five operations:

pairing the pairing of two terms

unpairing the extraction of a term from a pair

encryption given a key k and a plaintext m, the construction of the ciphertext $\{|m|\}_k$ by encrypting m with k

decryption given a ciphertext $\{|m|\}_k$ and its decryption key k^{-1}, the extraction of the plaintext m

generation the generation of an original term, which is not assumed to be secure

Pairing, encryption, and generation are construction operations. Decryption and unpairing are deconstruction operations.

5.1.2 Cremers' Algorithm

Cremers' Algorithm adds constraints to adversarial actions to allow one to infer the possible shapes of protocol runs.

Cremers gives two major constraints. A protocol is *efficient* if an adversary always takes a message from the earliest point at which it appears. A protocol is *normal* when the adversary always performs zero or more deconstruction operations followed by zero or more construction operations, except in the case of constructing decryption keys.

Cremers proved that these constraints do not limit the capabilities of an adversary **reference**.

An important insight used in Cremers' algorithm, called *chaining*, states that terms in messages received from an adversary strand always originate in a non-adversarial strand.

5.2 The Problem

Protocol researchers want to be able to programmatically reason about cryptographic protocols. A common technique for this is to find a set of essentially different classes of protocol runs. This can be accomplished by finding minimal models of a geometric logic representation of the protocol. This happens to be precisely the problem the chase solves.

5.3 The Solution: Minimal Models

Given a theory $\mathcal{T}$, the jointly minimal models $\mathcal{M}$ that the chase outputs are representative of all models because there always exists a homomorphism from some model $\mathbb{M} \in \mathcal{M}$ to any model that satisfies $\mathcal{T}$. Each model that satisfies $\mathcal{T}$ represents a class of runs of the protocol. The set of all models output by the chase represents every possible run of the protocol. Finding every possible run of the protocol is prohibitive because there are infinitely many. Because the set of models is countably infinite, they can be enumerated, but can not be listed.

5.4 Designing An Analogous Theory

In order to create a geometric theory describing a protocol, the formulæ that define strand spaces, normilisation, efficiency, and chaining must be derived. The formulæ defining the protocol must be combined with this scaffolding to create a theory that can be used to infer the possible runs of the protocol.

The *half-duplex protocol* was chosen to be used as an example. This protocol involves two participants, Alice and Bob. The protocol specifies that the following actions take place:

1. Alice sends Bob a nonce that she generated, encrypted with Bob's public key
2. Bob receives the encrypted nonce
3. Bob replies to Alice with the decrypted nonce
4. Alice receives Bob's message

The geometric logic rules that model this protocol were generated manually by direct translation into geometric logic. Ideally, the process of generating geometric logic formulæ from protocols should be done automatically.

5.5 The Results

The chase was run on the logic representation of the half-duplex protocol. A single model was returned during the execution of the algorithm, which was manually stopped before natural completion. This model, like all models returned by the chase, satisfies the input theory, and belongs to a set of jointly minimal models for the theory. The returned model denotes a run of the protocol which contains no adversary strands and is a correct execution of the protocol.

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' _ [] = trace "__but_it_is_impossible_to_make_the_model_satisfy_the_theory_by_adding_to_it" [
44 chase' theory pending = concatMap (branch theory) pending
45
46 branch :: [Formula] -> Model -> [Model]
47 --
48 branch theory model =
49    let reBranch = chase' theory in
50    trace ("running_chase_on_" ++ show model) $
51    case findFirstFailure model theory of
52       Just newModels ->
53          trace ("__at_least_one_formula_does_not_hold_for_model_" ++ showModel model) $
54          reBranch newModels
55       Nothing -> -- represents no failures
56          trace ("all_formulae_in_theory_hold_for_current_model") $
57          trace ("returning_model_" ++ showModel model) $
58          [model]
59
60 findFirstFailure :: Model -> [Formula] -> Maybe [Model]
```

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

References

[1] Marc Bezem and Thierry Coquand. Automating coherent logic. In Geoff Sutcliffe and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 3835 of *Lecture Notes in Computer Science*, pages 246–260. Springer Berlin / Heidelberg, 2005.

[2] Cas J.F. Cremers. Unbounded verification, falsification, and characterization of security protocols by pattern refinement. In *CCS '08: Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 119–128, New York, 2008. ACM.

[3] Ronald Fagin, Phokion Kolaitis, Rene Miller, and Lucian Popa. Data exchange: Semantics and query answering. In Diego Calvanese, Maurizio Lenzerini, and Rajeev Motwani, editors, *Database Theory ICDT 2003*, volume 2572 of *Lecture Notes in Computer Science*, pages 207–224. Springer Berlin / Heidelberg, 2002.

[4] Ronald Fagin, Phokion G. Kolaitis, and Lucian Popa. Data exchange: getting to the core. *ACM Trans. Database Syst.*, 30(1):174–210, 2005.

[5] Joshua D. Guttman, F. Javier, and F. Javier Thayer Fbrega. Authentication tests and the structure of bundles. *Theoretical Computer Science*, 283:2002, 2002.

[6] John Harrison. *Handbook of Practical Logic and Automated Reasoning.* Cambridge University Press, New York, 2009.

[7] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge University Press, New York, second edition, 2004.

[8] Graham Hutton. *Programming in Haskell.* Cambridge University Press, New York, 2007.