Generating Universal 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

24th October, 2010

Daniel Dougherty
professor, project advisor

Abstract

This paper describes an algorithm, referred to as the chase, that generates jointly universal models for a geometric theory. A universal 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. A Haskell implementation of the chase and its development process and design decisions are documented. The chase implementation is then used to generate universal models for a theory that represents a crptographic protocol. The results will be discussed and analysed.

Table of Contents

1 Introduction

1.1 Goals

The two main goals of this Major Qualifying Project are:

1. to implement an algorithm known as "the chase" accurately and with a well-defined, usable interface, and
2. to use the chase implementation for a real-world application: generating models used in analysis of a specific cryptographic 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 (see 3) is an algorithm used to find jointly universal models (see 2.9) for a set of geometric logic formulæ (see 2.11). 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 represent a large set of models rather than testing each of the infinite number of models separately.

To generate these jointly universal models, the chase begins with a model $\mathbb{M}$ that has an empty domain and no facts. The chase goes through each formula $\sigma \in T$ in the geometric theory T such that $\mathbb{M}$ is not a model of σ and alters $\mathbb{M}$ until $\mathbb{M}$ is a model of all $\sigma \in T$. Geometric formulæ are implications of positive-existential formulæ (see 2.11). Positive-existential formulæ have the useful property in that adding elements/facts to a model that satisfies one will never cause the model to no longer satisfy that formula. Because of this, the chase can keep adding to the model until all $\sigma \in T$ are satisfied. The set of all models generated by this process is a jointly universal set for the input theory T.

The chase is nondeterministic over disjunctions. When a disjunction is encountered, a disjunct is chosen fairly, and the chase continues deterministically until it encounters another disjunction.

1.3 Chase Implementation

We will see in section 4 that, in a Haskell implementation of the chase, all disjuncts of a disjunction can be satisfied by forking the chase and returning a list containing the concatenation of the lists returned by the forks. In this way, we can deterministically calculate a set of jointly universal models by finding all of the models returned by successful runs of a naturally nondeterministic algorithm.

The choice of Haskell for an implementation language is very benefitial because of Haskell's lazy evaluation. When the chase would run infinitely, the program runs infinitely as well, but it will still process and return all of the results of the halting runs of the chase. Haskell is also commonly used by mathemeticians and theoretical computer scientists, the target audience.

1.4 Application

Cryptographic protocol analysis is the particular application that is explored in section 5.1. In this application, protocols are modeled in the strand space formalism. Each role of every participant in a legal run of the protocol is modeled as a strand. The roles of a special participant that does not obey the rules of the protocol are called adversary strands (see 5.1.2). These adversary strands consist of a series of nodes that send/receive messages to/from regular strands while manipulating those messages.

Because the positions and actions of adversary strands are variable, there exists a large number of possible runs of a single protocol. It is prohibitive to test all of these to find if they break assumptions made about the properties of the protocol because of the large number of possibilities.

This protocol can be represented as a geometric theory and given to the chase to find universal models. When universal models are found, they can describe nearly all interesting ways that the adversaries can interact with regular strands. These models can then be analysed in a finite amount of time.

Section 2 can be used as a reference for terms discussed in later sections. Appendices A and B contain a syntax reference and glossary, respectively.

2 Technical Background

2.1 Vocabulary

A *vocabulary* consists of a set of pairings of a *relation symbol* 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 the following:

- 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. We will see in section 2.11 that these are unnecessary and can be replicated using other, allowed constructs.

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 σ, denoted $free(\sigma)$ is defined inductively as follows:

- $free(R(x_0, \ldots, x_n)) = \{x_0, \ldots, x_n\}$
- $free(\top) = \emptyset$
- $free(\bot) = \emptyset$
- $free(x = y) = \{x, y\}$
- $free(\neg\alpha) = free(\alpha)$
- $free(\alpha \wedge \beta) = free(\alpha) \cup free(\beta)$
- $free(\alpha \vee \beta) = free(\alpha) \cup free(\beta)$
- $free(\alpha \rightarrow \beta) = free(\alpha) \cup free(\beta)$
- $free(\forall\ x : \alpha) = free(\alpha) \setminus \{x\}$
- $free(\exists\ x : \alpha) = free(\alpha) \setminus \{x\}$

A formula σ is a *sentence* if $free(\sigma) = \emptyset$.

2.5 Environment

An *environment* λ for a model $\mathbb{M}$ is a function from a set of variables $\vec{v}$ to $|\mathbb{M}|$. 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. In other words, given a model $\mathbb{M}$, a set of formulæ Σ, and a formula σ such that $\Sigma \models \sigma$, whenever $\mathbb{M} \models \Sigma$, $\mathbb{M} \models \sigma$.

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 $h : |\mathbb{A}| \to |\mathbb{B}|$ such that, for each relation symbol R and tuple $\langle a_0, \ldots, a_n \rangle$, $\langle a_0, \ldots, a_n \rangle \in R^{\mathbb{A}}$ implies $\langle h(a_0), \ldots, h(a_n) \rangle \in R^{\mathbb{B}}$. The identity function is a homomorphism from any model $\mathbb{M}$ to itself.

A homomorphism h is also a *strong homomorphism* if, for each relation symbol R and tuple $\langle a_0, \ldots, a_n \rangle$, $\langle a_0, \ldots, a_n \rangle \in R^{\mathbb{A}}$ 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 $h : \mathbb{M} \to \mathbb{N}$. $\preceq$ has 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 $\mathbb{M} = \mathbb{N}$. For example, fix two models $\mathbb{M}$ and $\mathbb{N}$ that are equivalent except that $|\mathbb{N}| = |\mathbb{M}| \cup \omega$ where $\omega \notin |\mathbb{M}|$. Both $\mathbb{M} \preceq \mathbb{N}$ and $\mathbb{N} \preceq \mathbb{M}$ are true, yet $\mathbb{M} \neq \mathbb{N}$.

Given two models $\mathbb{M}$ and $\mathbb{N}$, when $\mathbb{M} \preceq \mathbb{N}$ and $\mathbb{N} \preceq \mathbb{M}$, $\mathbb{M}$ and $\mathbb{N}$ are *homomorphically equivalent*. 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 Universal Models

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

A set of models $\mathcal{M}$ is said to be *jointly universal* for a set of formulæ Σ when, for every model $\mathbb{N}$ such that $\mathbb{N} \models \Sigma$, there exists a homomorphism from a model $\mathbb{M} \in \mathcal{M}$ to $\mathbb{N}$. It follows that any superset of a jointly universal set of models is also jointly universal.

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

Not every theory must have a universal model. A simple example of this is the theory containing a single formula σ where σ contains a disjunction. There exists no single universal model for the formula $P \vee Q$ (P and Q are relations with arity 0) 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 universal.

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.

Theorem 1. *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.*

Proof. This is a well-known classical result in model theory. See [2], section 5.2 for an example.

We will only use one direction of Theorem 1: the fact that if a sentence is in positive-existential form then it is closed under homomorphisms. To keep this paper self-contained, we provide here a proof of this fact.

It suffices to prove the following, more general, claim for positive-existential *formulæ*.

> If σ is a positive-existential formula, $\mathbb{M}$ is a model, $\lambda : [var] \to |\mathbb{M}|$ is an environment such that $\mathbb{M} \models_\lambda \sigma$, and h is a homomorphism $h : |\mathbb{M}| \to |\mathbb{M}'|$, then $\mathbb{M}' \models_{h \circ \lambda} \sigma$.

We prove this by induction over formulæ.

When σ is an atomic formula $R(x_0, \ldots, x_n)$, we know that $R(\lambda(x_0), \ldots, \lambda(x_n)) \in \mathbb{M}$ and by the definition of a homomophism

$$R((h(\lambda(x_0)), \ldots, (h(\lambda(x_n))) \in \mathbb{M}'.$$

That is,

$$R((h \circ \lambda)(x_0), \ldots, (h \circ \lambda)(x_n)) \in \mathbb{M}',$$

as desired.

When σ is $x = y$, **I need help here. I thought I could explain this, but I'm having trouble.**

When σ is $\bot$, there exists no λ such that $\mathbb{M} \models_\lambda \bot$, so the condition of our claim never holds, which causes our claim to always hold.

When σ is $\top$, we know that $\mathbb{M} \models_\lambda \top$ and by the induction hypothesis, $\mathbb{M}' \models_{h \circ \lambda} \top$.

When σ is $\alpha \wedge \beta$, we know that $\mathbb{M} \models_\lambda \alpha$ and by the induction hypothesis $\mathbb{M}' \models_{h \circ \lambda} \alpha$. We also know that $\mathbb{M} \models_\lambda \beta$ and by the induction hypothesis $\mathbb{M}' \models_{h \circ \lambda} \beta$. By the definition of conjunction, because $\mathbb{M}' \models_{h \circ \lambda} \alpha$ and $\mathbb{M}' \models_{h \circ \lambda} \beta$, we know that $\mathbb{M}' \models_{h \circ \lambda} \alpha \wedge \beta$.

When σ is $\alpha \vee \beta$, we know that if $\mathbb{M} \models_\lambda \alpha$, by the induction hypothesis $\mathbb{M}' \models_{h \circ \lambda} \alpha$. We also know that if $\mathbb{M} \models_\lambda \beta$, by the induction hypothesis $\mathbb{M}' \models_{h \circ \lambda} \beta$. By the definition of disjunction, because $\mathbb{M}' \models_{h \circ \lambda} \alpha$ or $\mathbb{M}' \models_{h \circ \lambda} \beta$, we know that $\mathbb{M}' \models_{h \circ \lambda} \alpha \vee \beta$.

When σ is $\exists\, \vec{x} : \alpha$, we know that $\mathbb{M} \models_\lambda \exists\, x : \alpha$, and want to prove that $\mathbb{M}' \models_{h \circ \lambda} \exists\, x : \alpha$. There is a $d \in |\mathbb{M}|$ such that $\mathbb{M} \models_{\lambda[x \mapsto d]} \alpha$. By our induction hypothesis, $\mathbb{M}' \models_{h \circ (\lambda[x \mapsto h(d)])} \alpha$. This is equivalent to $\mathbb{M}' \models_{(h \circ \lambda)[x \mapsto h(d)]} \alpha$.

□

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

$$\forall\, \vec{x} : F_L \to F_R$$

where $\vec{x} = free(F_L) \cup free(F_R)$, $free$ 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 $\top \to \sigma$ when expecting a geometric logic formula. It is also convention to treat a negated positive existential formula $\neg\sigma$ as $\sigma \to \bot$.

Examples of geometric logic formulæ:

reflexivity	$\top \rightarrow R(x, x)$
symmetry	$R(x, y) \rightarrow R(y, x)$
asymmetry	$R(x, y) \wedge R(y, x) \rightarrow \bot$
serial	$\top \rightarrow \exists\ y : R(x, y)$
totality	$\top \rightarrow R(x, y) \vee R(y, x)$
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. Constant symbols can be encoded as relations with an arity of zero, for example $C()$. A function $f(x, y)$ can be encoded as the geometric logic formula $F(x, y, z1) \wedge F(x, y, z2) -> z1 = z2$.

3 The Chase

The *chase* is a function that, when given a geometric theory, will generate a model in the set of jointly universal 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 $\mathbb{M}$. Note that given a model $\mathbb{M}$ returned by the chase there may exist a model $\mathbb{N}$ such that $\mathbb{N} \preceq \mathbb{M}$.

Verifying the universality of a model or the joint universality of a set of models requires checking if a homomorphism exists from the universal model(s) to each of the infinite set of all models that satisfy the theory. It may not at first seem obvious that generating a universal model would be a computable task, but it will be shown that the chase is able to do this, and it will be proven that the models it returns during successful runs are in fact members of a set of jointly universal models.

Geometric logic sentences are used by the chase both because they are natural expressions of many common applications and because they take advantage of the useful properties of the positive-existential sentences of which they are constructed. Recall that, when adding any relations or domain members to a model that satisfies a positive-existential sentence, the model will always satisfy the sentence. This is particularly helpful when trying to create a model that satisfies all sentences in a geometric theory.

3.1 Algorithm

To describe the chase algorithm succinctly, it is convenient to introduce the following notation. Given $\mathbb{M}$ is a model, T is a geometric theory, σ is a sentence in T, and $\lambda : free(\sigma) \to |\mathbb{M}|$ is an environment, the pair (σ, λ) is a *test*, specifically a test of $\mathbb{M}$ based on T. The model $\mathbb{M}$ *passes* this test if and only if $\mathbb{M} \models_\lambda \sigma$.

An existentially-quantified conjunction of atomics (ECA) E is defined inductively by

- if E is an atomic, E is an ECA
- if E is $\exists\, \vec{x} : \alpha$ and α is an ECA, E is an ECA
- if E is $\alpha \wedge \beta$ and both α and β are ECAs, E is an ECA

For simplicity, we describe the chase assuming each sentence of the input theory is of the

form

$$\delta_0 \vee \ldots \vee \delta_n \rightarrow \gamma_0 \vee \ldots \vee \gamma_k\}$$

where each δ_i and γ_j are ECAs. The chase can take this form as input without loss of generality. In other words, a logical equivalent to any legal chase input theory can be expressed in a way that respects these constraints.

Next we define a *chase step*, denoted $\mathbb{M} \xrightarrow{(\sigma,\lambda)} \mathbb{M}'$, with $\mathbb{M}'$ being the result of the following algorithm applied to the model $\mathbb{M}$, the sentence σ, and the environment λ.

Algorithm: chaseStep :: Model ($\mathbb{M}$) $\rightarrow$ Sentence (σ) $\rightarrow$ Environment (λ) $\rightarrow$ Model
let σ be $\alpha \rightarrow \beta$
choose some disjunct $e \equiv \exists\ x_0, \ldots, x_p : c_0(\vec{a_0}) \wedge \ldots \wedge c_n(\vec{a_n})$ of β
add new elements $d_0 \ldots d_p$ to $|\mathbb{M}|$
define θ as $\lambda_{[x_0 \mapsto d_0, \ldots, x_p \mapsto d_p]}$
add each of the facts $\{c_i(\theta\vec{a_j}) | 0 \leq i \leq n\}$ to $\mathbb{M}$
return $\mathbb{M}$

A chase step fails if and only if the right side of the implication is a contradiction.

Now that the notion of a chase step is established, the chase can be defined as follows.

Algorithm: chase :: [Sentence] (T) $\rightarrow$ Model
let $\mathbb{M}$ be a model that has an empty domain and an empty set of facts
while $\mathbb{M} \not\models T$ **do**
  choose a test (σ, λ) for which $\mathbb{M}$ fails
  update $\mathbb{M}$ to be the result of the chase step on $\mathbb{M}$ based on (σ, λ)
return $\mathbb{M}$

There are three types of runs of the chase:

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

3.2 Examples

Define Σ as the following geometric theory.

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

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 universal.

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 not 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 Properties of the Chase

we REALLY need a better section title than this

Fairness A deterministic realization of the chase algorithm (the pairing of the nondeterministic chase algorithm with an evaluation strategy) is *fair* if the scheduler would not allow for a (rule, binding) **(syntax?)** pair to go unevaluated during an infinite run.

I used to think this was an unfavorable fairness description, but I really like it now. Is it good enough to stay?

Lemma 1. *Let T be a geometric theory and $\mathbb{M}$ be a model of T. Let $\mathbb{N}$ be a model and suppose there exists a homomorphism $h : \mathbb{N} \to \mathbb{M}$. If $\mathbb{N}$ fails the test (σ, λ), there exists a chase step $\mathbb{N} \xrightarrow{(\sigma,\lambda)} \mathbb{N}'$ and a homomorphism $h' : \mathbb{N}' \to \mathbb{M}$.*

Proof. Suppose σ is in the form

$$E(\vec{x}) \to \bigvee_i F_i(\vec{x})$$

Since $\mathbb{N} \not\models_\lambda \sigma$, we know $\mathbb{N} \models_\lambda E(\vec{x})$ while $\mathbb{N} \not\models_\lambda \bigvee_i F_i(\vec{x})$. Since h is a homomorphism,

$\mathbb{M} \models_{h\circ\lambda} E(\vec{x})$. Since $M \models_{h\circ\lambda} \sigma$, we have $\mathbb{M} \models_{h\circ\lambda} \bigvee_i F_i(\vec{x})$ for some disjunct i. There exists a chase step $\mathbb{N} \xrightarrow{(\sigma,\lambda)} \mathbb{N}'$ that will choose this disjunct.

[@@ HERE!] **something important was supposed to be here, but I forgot what it was**

In case $F_i(\vec{x})$ is of the form $\bigwedge_j R_{ij}(\vec{x})$ the chase step generates $\mathbb{N}'$ by adding the facts $R_{ij}(\lambda(\vec{x}))$ for each j. To see that $h : \mathbb{N} \to \mathbb{M}$ is also a homomorphism from $\mathbb{N}'$ to $\mathbb{M}$ it suffices to see that each of the new facts added to $\mathbb{N}'$ is preserved, by h, in $\mathbb{M}$. That is, we want to see that for each j, the fact $R_{ij}(h(\lambda(x_0)), \ldots, h(\lambda(x_n))$ holds in $\mathbb{M}$. But this follows from our earlier observation that $\mathbb{M} \models_{h\circ\lambda} F_i(\vec{x})$.

In case F_i is of the form $\exists\vec{y} : \bigwedge_j R_{ij}(\vec{x})$ **(should this be $R_{ij}(\vec{y})$)** ... □

Theorem 2. *Let T be a geometric theory. For any model $\mathbb{M}$ such that $\mathbb{M} \models T$, there exists a run of the chase that returns a model $\mathbb{N}$ such that $\mathbb{N} \models T$ and $\mathbb{N} \preceq \mathbb{M}$.*

Proof. □

3.4 History

In [4] *Data Exchange: Semantics and Query Answering*, Fagin et. al. first introduce a chase algorithm. The version they defined disallows disjunctions and, because of that, is limited in its utility. Input sentences without disjunctions are not as expressive as geometric sentences and have fewer applications. It is, however, a completely deterministic algorithm.

The chase was originally used to solve the problem of data exchange. As Fagin et al. states, "Data exchange is the problem of taking data structured under a source schema and creating an instance of a target schema that reflects the source data as accurately as possible". The solution to the stated problem was to find a universal model. In theories without disjunction, a single universal model exists that has a homomorphism to any other model that satisfies the theory. This absolute universal model can be calculated from a single run of their deterministic chase algorithm.

The definition of the chase algorithm used by Fagin et. al. is similar to the one defined in section 3.1 in all ways except that it does not have to choose a disjunct.

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 as they are found, even though some chase runs have not returned a value, and even if a chase run is infinite.

To be sure that a chase implementation returns every model a chase run could possibly return, it is important that the implementation is *fair*. Recall the definition of fairness from section 3.3.

Though the discussed implementation is efficient, it is not fair. The domain is represented by an ordered type. Any time the algorithm would ask us to choose a binding, each variable is assigned a member of the domain, starting with all variables paired with the representative with the lowest ordering. Successive pairings relate variables with successive domain members. Problems similar to this are the cause of unfairness in the implementation.

Appendix C 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 implementation tries to coerce it into one by applying the following rules recursively:

$$
\begin{aligned}
\neg\alpha \wedge \neg\beta &\mapsto \alpha \vee \beta \\
\neg\alpha \vee \neg\beta &\mapsto \alpha \wedge \beta \\
\neg\neg\alpha &\mapsto \alpha \\
\neg\alpha \rightarrow \beta &\mapsto \alpha \vee \beta \\
\alpha \rightarrow \neg\beta &\mapsto \alpha \wedge \beta \\
\neg(\forall\ \vec{x} : \neg\alpha) &\mapsto \exists\ \vec{x} : \alpha
\end{aligned}
$$

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. `satisfy`

will behave as outlined in the following algorithm.

should probably move this to an appendix

Algorithm: satisfy :: Model ($\mathbb{M}$) $\rightarrow$ Environment (λ) $\rightarrow$ Formula (φ) $\rightarrow$ [Model]

return **switch** φ **do**

- **case** $\top$ return $\{\mathbb{M}\}$
- **case** $\bot$ return $\emptyset$
- **case** $x = y$ `/* TODO: add comment here */`
  - let $a = \lambda(x)$ and $b = \lambda(y)$
  - let $\mathbb{N}$ be a model where $|\mathbb{N}| = |\mathbb{M}| - \{b\}$
  - **foreach** *fact* $R(\vec{v}) \in \mathbb{M}$ **do** add $R(v_{[\vec{b \mapsto a}]})$ as a fact in $\mathbb{N}$
  - return $\mathbb{N}$
- **case** $\alpha \vee \beta$ return $satisfy(\mathbb{M}, \lambda, \alpha) \cup satisfy(\mathbb{M}, \lambda, \beta)$
- **case** $\alpha \wedge \beta$
  - let $r = \emptyset$
  - **forall the** *models* $m \in satisfy(\mathbb{M}, \lambda, \alpha)$ **do**
    - redefine r as $r \cup satisfy(\mathbb{M}, \lambda, \beta)$
  - return r
- **case** $R(\vec{x})$
  - define a model $\mathbb{N}$ where $|\mathbb{N}| = |\mathbb{M}| \cup \omega$ and $\omega \notin |\mathbb{N}|$ to $|\mathbb{N}|$
  - **foreach** $P_{\mathbb{M}}$ **do** $P_{\mathbb{N}} = P_{\mathbb{M}}$
  - **forall the** $v \in \vec{x}$ **do**
    - **if** $v \notin \lambda$ **then** redefine λ as $\lambda_{v \mapsto \omega}$
  - define $R_{\mathbb{N}}$ as $R_{\mathbb{M}}(\lambda(x_0) \ldots \lambda(x_n))$
  - return $\{\mathbb{N}\}$
- **case** $\exists\ \vec{x} : \alpha$
  - **if** $\vec{x} = \emptyset$ **then** return $satisfy(\mathbb{M}, \lambda, \alpha)$
  - **if** $|\mathbb{M}| \neq \emptyset$ *and* $\exists\ v' \in |\mathbb{M}|$ *such that* $\lambda' = \lambda_{x_0 \mapsto v'}$ *and* $\mathbb{M} \models_{\lambda'} \alpha$ **then**
    - return $\{\mathbb{M}\}$
  - **else**
    - define a model $\mathbb{N}$ where $|\mathbb{N}| = |\mathbb{M}| \cup \omega$ and $\omega \notin |\mathbb{N}|$ to $|\mathbb{N}|$
    - **foreach** $R_{\mathbb{M}}$ **do** $R_{\mathbb{N}} = R_{\mathbb{M}}$
    - define $\kappa = \lambda_{x_0 \mapsto \omega}$
    - return $satisfy(\mathbb{N}, \kappa, \exists\ \{x_1 \ldots x_n\} : \alpha)$

4.2 Input Format

Input to the program must be in a form parsable by the context-free grammar seen in Appendix D. 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 in Appendix E. The expected input is essentially a newline-separated list of ASCII representations of geometric formulæ.

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 be implemented as a Data.Map. Environments are currently a list of tuples, but should really be a Data.Map. Instead of Domains being a list of DomainMember, it would 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 `a` is the type they currently return. 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 holds for a formula with 5 universally quantified variables (a reasonable real-life usage example), 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 [7] Harrison's *Practical Logic and Automated Reasoning*, 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 [7] Harrison mentions on pages 141 to 144 is `pullQuants`, 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 Parallelize

The chase implementation should be very easily parallelizable. The program can fork for every call to the `branch` function as it is mapped over a pending list in the `chase'` function. As each fork finds a contradiction, the forks will die. This method could still lead to a large number of threads. In a proper implementation, forks that are waiting on a single child could consume that child's work to minimize this problem. Doing this would prevent long chains of waiting threads.

4.4.7 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`. When this flag is found, a callback function is being invoked in the Main module. Someone who implements this enhancement would need to be able to alter the behaviour of the Chase module's *trace* function from a function within the Main module. Ideally, a better debugging output method will be found and Debug.Trace.trace will no longer need to be used in what is otherwise production-ready code.

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 universal models. When a protocol is described using geometric logic, the chase can find such universal models. The protocol can then be analysed for characteristics such as the existence of security violations or other unexpected behaviour.

5.1 Strand Spaces

The *strand space formalism* was developed as a method for formally reasoning about cryptographic protocols. The formalism distinguishes between two different kinds of participants: regular participants and an adversary. A single participant can be represented as multiple regular strands if they play more than one role in the protocol.

Roles in a protocol run are represented by *strands*, and communicate with each other by sending and receiving messages. A regular rols is represented by a regular strand and must follow the protocol. The adversary is represented by zero or more adversary strands, and can manipulate the messages that regular strands send/receive.

A strand is made up of a non-empty, finite sequence of nodes. Every node either sends or receives a term called its *message*. A *term* is anything can be sent between nodes.

5.1.1 Messaging

Terms are defined inductively as follows:

- any text is a term, specifically a *basic term*
- the ciphertext $\{|\tau_1|\}_{\tau_2}$ is a term if the plaintext τ_1 and the key τ_2 are terms
- the pair (τ_1,τ_2) is a term if τ_1 and τ_2 are terms

A term t is an *ingredient* of another term u if u can be constructed from t by repeatedly pairing with arbitrary terms and encrypting with arbitrary keys. A *component* of a term

t is any term that can be retrieved simply by applying repeated unpairing operations to t and is not a pair itself.

A *nonce* is a uniquely-originating basic term. A term t *originates* on a node n of a strand s if n is a sending node, t is an ingredient of the message of n, and t is not an ingredient of any previous node on s. A term is *uniquely originating* if it originates on only one strand. A term is non-originating if it does not appear in any strand.

If a regular participant generates a random fresh nonce, it will be assumed uniquely-originating because of the extreme unlikelihood of any other participant generating and originating the same value.

Similarly, non-origination is helpful in describing private asymmetric keys that should never be sent as part of a message.

5.1.2 The Adversary

An adversary is included in the strand space formalism to represent a worst-case situation, where an attacker may control every point of communication between regular participants. The actions of the adversary are represented by *adversary strands*. Recall that 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**. The five possible operations that an adversary may perform are derived from both the Dolev-Yao Threat Model and the strand space formalism. These operations are:

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 when that term is not assumed to be secure

Pairing and encryption are *construction operations*. Decryption and unpairing are *deconstruction operations*.

5.2 Paths

An *edge* is a directional relationship between two nodes. The direct predecessor of a node within its strand is called its *parent*. Not every node has a parent. Whenever a node n has a parent p, there is an edge from p to n. A *link* is an edge from a sending node to a receiving node that have the same message. If there is a node n with a link to a node m, n sends m a message and m receives it unaltered. The *path* relation, written $x < y$, is the transitive closure of the edge relation. A node n *precedes* another node m when there is a path from n to m. In this formalism, events are partially ordered.

5.2.1 Normalisation / Simplification

Two simplifying assumptions can be made about the sequence of actions an adversary can perform without limiting their capabilities. These simplifications are called normalisation and efficiency. Guttman and Thayer [6] proved that adding these constraints does not limit the capabilities of an adversary.

A protocol is *efficient* if an adversary always takes a message from the earliest point at which it appears. To be precise, a protocol is efficient if, for every sending node m and receiving adversary node n, if every component of m is also a component of n, then there is no regular node m' such that $m < m' < n$.

A protocol is *normal* if, for any path through adversary strands, the adversary always either performs a generation followed by zero or more construction operations or performs zero or more deconstruction operations followed by zero or more construction operations. This constraint limits redundancy.

An important insight used in Cremers' algorithm [3], which will be called *chaining*, states that terms in messages received from an adversary strand always originate in a non-adversarial strand. In simpler terms, an adversary can not send a message to himself nor receive a message from himself.

5.3 The Problem

Some 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, which are each a subset of all possible runs of the protocol. Together, these classes encompass every possible run of the protocol. This can be accomplished by finding universal models of a geometric logic representation of the protocol. This happens to be precisely the problem the chase solves.

things from Justin go here later

5.4 The Solution: Universal Models

Given a theory T, the jointly universal 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 T. Every model $\mathbb{N}$ that satisfies T describes a possible run of the protocol. It also indirectly represents a larger class of runs, specifically the set of all runs of the protocol that are described by a model $\mathbb{P}$ such that $\mathbb{P} \models T$ and $\mathbb{N} \preceq \mathbb{P}$.

Since the set of all models output of the chase is jointly universal, they represents every possible run of the protocol. Finding every possible run of the protocol would be prohibitive because there are infinitely many. Fortunately, the chase may halt with a finite result, providing a model that is representative of a class of protocol runs. Fortunately, the deterministic chase implementation may output a finite number of models before halting. These models are jointly universal, and therefore represent all runs of the protocol.

5.5 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, playing two roles, and 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

A visual representation of the protocol, as modeled in the strand space formalism, can be seen in Figure 1.

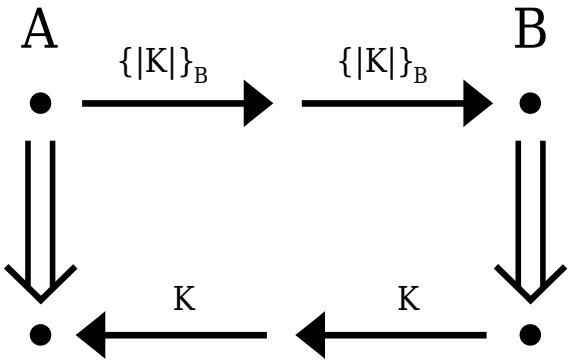

Figure 1: A visual representation of the half-duplex protocol modeled in strand space

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.

5.6 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, visualized in Figure 2, like all models returned by the chase, satisfies the input theory, and belongs to a set of jointly universal 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.

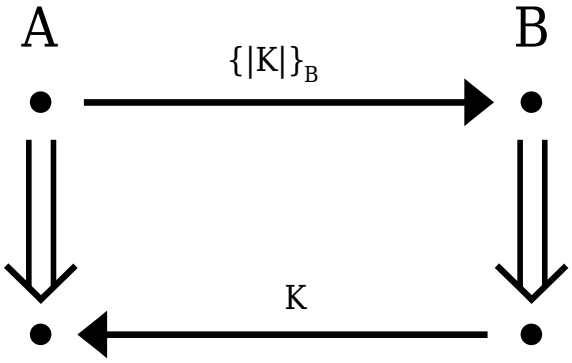

Figure 2: A visual representation of the discovered run of the half-duplex protocol

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 Glossary

adversary a participant in the *strand space formalism*, represented by zero or more *adversary strands*

adversary strand a type of *strand* in the *strand space formalism* that is not bound by the rules defined by the protocol; manipulates *messages* being sent and received by non-adversarial strands

chase a nondeterministic algorithm used to find *jointly universal models* for a set of *geometric logic formulæ*

component a component of a *term* t is any term that can be retrieved simply by applying repeated unpairing operations to t and is not a pair itself

construction operation pairing and encryption operations

deconstruction operation decryption and unpairing operations

efficient a protocol is efficient if, for every sending *node* m and receiving adversary node n, if every *component* of m is also a *component* of n, then there is no regular node m' such that $m < m' < n$.

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

environment an environment for a model $\mathbb{M}$ is a function from a variable v to a domain member e where $e \in |\mathbb{M}|$

first-order logic a formal logic system; also called *predicate logic*

geometric logic formula implicitly universally quantified implication of *positive-existential formulæ*

geometric theory a set of geometric logic formulæ

homomorphic equivalence two models $\mathbb{M}$ and $\mathbb{N}$ are homomorphically equivalent if $\mathbb{M} \preceq \mathbb{N} \wedge \mathbb{N} \preceq \mathbb{M}$

homomorphism a function $h : |\mathbb{A}| \to |\mathbb{B}|$ 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 $\langle h(a_0), \ldots, h(a_n) \rangle \in R^{\mathbb{B}}$

ingredient a *term* t is an ingredient of another term u if u can be constructed from t by repeatedly pairing with arbitrary terms and encrypting with arbitrary keys

isomorphism 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.

jointly universal models a set of models $\mathcal{M}$ is said to be jointly universal 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 $\mathbb{N}$.

message a *term* sent or received by a *node*

universal model a model for a theory T with the special property that there exists a *homomorphism* from the model to any other model that satisfies T; also called *universal model*

model a model $\mathbb{M}$ is a construct that consists of a set, denoted $|\mathbb{M}|$, called the *universe* or *domain* of $\mathbb{M}$ and a relation $R_k^{\mathbb{M}} \subseteq |\mathbb{M}|$ for each pairing of a predicate R and an arity k in a vocabulary $\mathcal{V}$

nonce a uniquely-originating basic term

normal a protocol is *normal* if, for any path through adversary strands, the adversary always either performs a generation followed by zero or more construction operations or performs zero or more deconstruction operations followed by zero or more construction operations

origination a *term* t originates on a *node* n of a *strand* s if n is a sending node, t is an *ingredient* of the *message* of n, and t is not an ingredient of any previous node on s

positive-existential form formula constructed using only conjunctions ($\wedge$), disjunctions ($\vee$), existential quantifications ($\exists$), tautologies ($\top$), contradictions ($\bot$), equalities, and relations

predicate see *relation symbol*

predicate logic see *first-order logic*

relation symbol any unique symbol; also called a *predicate*

sentence a formula α if $\mathit{free}(\alpha) = \emptyset$

strand space formalism a method for formally reasoning about cryptographic protocols

strong homomorphism 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 $\langle h(a_0), \ldots, h(a_n) \rangle \in R^{\mathbb{B}}$.

uniquely originating a term is uniquely originating if it originates on only one strand

universal model see *universal model*

vocabulary a set of pairings of a *relation symbol* with a non-negative integral arity

C 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)
```

D 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` *optIndex*

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

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

arg : `VARIABLE`

optIndex : ε
| *index*

optCOLON : ε
| `:`

optNEWLINE : ε
| `NEWLINE`

E Lexical Rules

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

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] Chen Chung Chang and Jerome Keisler. *Model Theory*. Number 73 in Studies in Logic and the Foundations of Mathematics. North-Holland, 1973. Third edition, 1990.

[3] 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.

[4] 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.

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

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

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

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

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