MQP Title

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

26th September, 2010

.....
Daniel Dougherty
professor, project advisor

Abstract

This paper describes a method, referred to as the chase, for generating 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 firewall configuration examination and access control evaluation. Also described is a Haskell implementation of the chase and its development process and design decisions.

Table of Contents

1 Introduction **1**

1.1 Goals 1

1.2 The Chase 1

2 Technical Background **2**

2.1 Definitions 2

2.1.1 Models 2

2.1.2 First-order Logic 2

2.1.3 Geometric Logic 3

2.1.4 Satisfiability, Entailment 3

2.1.5 Variable Binding, Environments 3

2.2 Homomorphisms 3

2.2.1 Significance 4

2.2.2 Minimal Models 4

2.2.3 Relation to the Chase 4

A Chase code **5**

References **7**

List of Figures

List of Tables

1 Introduction

Introductory text...

1.1 Goals

Morbi ac augue ac nisi euismod venenatis. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris vulputate dolor vitae purus dignissim ullamcorper. Aenean ante orci, posuere eu convallis at, feugiat et elit. Aenean rhoncus eros ut dui pellentesque convallis. Curabitur consectetur pretium varius. Vestibulum interdum convallis eros id pretium. Maecenas aliquam erat varius odio molestie consectetur.

1.2 The Chase

Aliquam vulputate mi non metus lacinia rutrum. In hac habitasse platea dictumst. Quisque magna nisi, lacinia quis molestie in, varius eu diam. Nullam tristique porta ante, malesuada egestas purus viverra nec. Nulla vestibulum pretium massa id mattis. Donec ut velit urna. Suspendisse potenti. Vivamus vitae consectetur quam. Mauris non ante mauris. Nulla id lectus ut velit mollis convallis vel non leo. Integer ac pulvinar nisl. Maecenas posuere fringilla consectetur.

2 Technical Background

2.1 Definitions

In this paper, logic symbols and other possibly ambiguous or uncommon notation will be used extensively, and thusly must be clearly defined.

2.1.1 Models

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

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

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

2.1.2 First-order Logic

First-order logic, also called *predicate logic*, is a formula logic system that is an extension of propositional logic. For our purposes, this logic system will not contain any constant symbols or function symbols, which are commonly included in first-order and propositional logic.

A first-order logic formula is defined inductively by

- if R is a relation symbol of arity k and each of $t_0, \ldots, t_{k-1}$ is a variable, then $R[t_0, \ldots, t_{k-1}]$ is a formula, specifically an *atomic formula*
- if α is a formula then $(\neg\alpha)$ is a formula
- if α and β are formulas then $(\alpha \wedge \beta)$ is a formula
- if α and β are formulas then $(\alpha \vee \beta)$ is a formula
- if α and β are formulas 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 $\vec{x}$ 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 x is a variable then $(\exists x : \alpha)$ is a formula
- if α is a formula and $\vec{x}$ is a set of variables of size k then $(\exists \vec{x} : \alpha)$ is $(\forall x_0 \ldots \forall x_{k-1} : \alpha)$

2.1.3 Geometric Logic

Geometric logic is first-order logic with constraints on the shape of the expression. Geometric logic formulas are implicitly universally quantified first-order logic expressions of the form

$$A_0 \wedge \ldots \wedge A_n \rightarrow E_0 \vee \ldots \vee E_m$$

where $A_0 \ldots A_n$ are atomics, $E_0 \ldots E_m$ are first-order logic expressions of the form $\exists_{x_0 \ldots x_k} A_0 \wedge \ldots \wedge \exists_{x_0 \ldots x_p} A_y$, and n, m, k, p, and y are integers greater than or equal to 0. A set of geometric logic formulas is called a *geometric theory*.

explain why GL is useful to us

2.1.4 Satisfiability, Entailment

satisfiability...

A set of formulas Σ is said to *entail* a formula σ ($\Sigma \models \sigma$)) if the set of all models satisfied by Σ is a subset of the set of all models satisfied by σ. The notation used for satisfiability and entailment is very similar, in that the operator used is the same, but they can be distinguished by the type of left operand.

2.1.5 Variable Binding, Environments

2.2 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 $a_k \in |\mathbb{A}|$ for any $k \in \{0, \ldots, n\}$, $\langle a_0, \ldots, a_n \rangle \in R^{\mathbb{A}}$ 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 $a_k \in |\mathbb{A}|$ for any $k \in \{0, \ldots, n\}$, $\langle a_0, \ldots, a_n \rangle \in R^{\mathbb{A}}$ 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 $h : \mathbb{M} \to \mathbb{N}$. The identity function is a homomorphism from any model $\mathbb{M}$ to itself. Homomorphisms are transitive, so $\mathbb{A} \preceq \mathbb{B} \wedge \mathbb{B} \preceq \mathbb{C}$ implies $\mathbb{A} \preceq \mathbb{C}$. However, $\mathbb{M} \preceq \mathbb{N} \wedge \mathbb{N} \preceq \mathbb{M}$ does not imply that $\mathbb{M} = \mathbb{N}$.

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

2.2.1 Significance

2.2.2 Minimal Models

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

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

2.2.3 Relation to the Chase

The chase is a function that, when given a gemoetric theory, will generate all minimal models for that theory.

A Chase code

```
1  module Chase where
2  import Parser
3  import Helpers
4  import Debug.Trace
5  import Data.List
6
7  chaseVerify :: [Formula] -> [Formula]
8  -- verifies that each formula is in positive existential form and performs some
9  -- normilization on implied/constant implications
10 chaseVerify formulae =
11    let isNotPEF = not.isPEF in
12    map (\f -> case f of
13       Implication a b ->
14          if isNotPEF a || isNotPEF b then error ("implication must be in positive existential
15          else f
16       _ ->
17          if isNotPEF f then error ("formula must be in positive existential form: " ++ showFor
18          else (Implication Tautology f)
19    ) formulae
20
21 chase :: [Formula] -> [Model]
22 -- runs the chase algorithm on a given theory and returns a list of models that
23 -- satisfy it
24 chase formulae = chase' (chaseVerify formulae) ([],[(mkModel [] [])])
25
26 chase' :: [Formula] -> ([Model],[Model]) -> [Model]
27 -- used by the chase function to hide the model identity argument
28 chase' formulae (done,[]) = done
29 chase' formulae (done,pending) =
30    let self = chase' formulae in
31    let (p:ending) = pending in
32    trace ("running chase on " ++ show (done,pending)) $
33    if all (\f -> holds p (UniversalQuantifier (freeVariables f) f)) formulae then
34       trace ("  all formulae in theory hold for model " ++ showModel p) $
35       trace ("  moving model into done list") $
36       self (union done [p],ending)
37    else
38       let possiblySatisfiedModels = attemptToSatisfyFirstFailure p formulae in
39       trace ("  at least one formula does not hold for model " ++ showModel p) $
40       trace ("  unioning " ++ show ending ++ " with [" ++ intercalate "," (map showModel poss
41       self (done, union ending possiblySatisfiedModels)
42
43 attemptToSatisfyFirstFailure :: Model -> [Formula] -> [Model]
44 -- checks if each formula holds, sequentially, until one does not, then tries
45 -- to satisfy that formula
46 attemptToSatisfyFirstFailure model (f:ormulae) =
47    let self = attemptToSatisfyFirstFailure model in
48    if holds model (UniversalQuantifier (freeVariables f) f) then self ormulae
49    else attemptToSatisfy model f
50
51 attemptToSatisfy :: Model -> Formula -> [Model]
52 -- returns a model that is altered so that the given formula will hold
53 attemptToSatisfy model formula =
54    let f' = UniversalQuantifier (freeVariables formula) formula in
55    trace ("  attempting to satisfy (" ++ showFormula formula ++ ")") $
56    attemptToSatisfy' model [] f'
57
58 attemptToSatisfy' :: Model -> Environment -> Formula -> [Model]
59 -- hides the environment identity in the `attemptToSatisfy` function arguments
```

```
60  attemptToSatisfy' model env formula =
61     let (domain,relations) = model in
62     let domainSize = length domain in
63     let self = attemptToSatisfy' model in
64     -- trace ("  attempting to satisfy (" ++ showFormula formula ++ ") with env " ++ show env)
65     case formula of
66        Tautology -> [model]
67        Contradiction -> []
68        Or a b -> union (self env a) (self env b)
69        And a b -> concatMap (\m -> attemptToSatisfy' m env b) (self env a)
70        Implication a b -> if holds' model env a then self env b else []
71        Atomic predicate vars ->
72           let newRelation = mkRelation predicate (length vars) [genNewRelationArgs env vars (fr
73           let newModel = mkModel (mkDomain domainSize) (mergeRelation newRelation relations) in
74           trace ("    adding new relation: " ++ show newRelation) $
75           [newModel]
76        ExistentialQuantifier [] f -> self env f
77        ExistentialQuantifier (v:vs) f ->
78           let f' = ExistentialQuantifier vs f in
79           let nextDomainElement = fromIntegral $ (length domain) + 1 in
80           if any (\v' -> holds' model (hashSet env v v') f') domain then
81              trace ("    " ++ showFormula formula ++ " already holds") $
82              [model]
83           else
84              trace ("    adding new domain element " ++ show nextDomainElement ++ " for variabl
85              attemptToSatisfy' (mkDomain nextDomainElement,relations) (hashSet env v nextDomain
86        UniversalQuantifier [] f -> self env f
87        UniversalQuantifier (v:vs) f ->
88           let f' = UniversalQuantifier vs f in
89           concatMap (\v' -> self (hashSet env v v') f') domain
90        _ -> error ("formula not in positive existential form: " ++ showFormula formula)
91
92  genNewRelationArgs :: Environment -> [Variable] -> DomainElement -> [DomainElement]
93  -- for each Variable in the given list of Variables, retrieves the value
94  -- assigned to it in the given environment, or the next domain element if it
95  -- does not exist
96  genNewRelationArgs env [] domainSize = []
97  genNewRelationArgs env (v:ars) domainSize =
98     let self = genNewRelationArgs env in
99     case lookup v env of
100       Just v' -> v' : (self ars domainSize)
101       _ -> (domainSize+1) : (self ars (domainSize+1))
```

References

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