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

25th 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

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

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.

(Explain why GL is useful)

2.2 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 $\langle a_0, \ldots, a_n \rangle$ where $a_k \in |\mathbb{A}|$ for any k and $0 \leq k \leq n$, $\langle a_0, \ldots, a_n \rangle \in R^{\mathbb{A}}$ 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 and $0 \leq k \leq n$, $\langle a_0, \ldots, a_n \rangle \in R^{\mathbb{A}}$ if and only if $\langle h(a_0), \ldots, h(a_n) \rangle \in R^{\mathbb{B}}$.

2.2.1 Significance

2.2.2 Minimal Models

2.2.3 Relation to the Chase

3 For later reference:

preceq: $\mathbb{M} \preceq \mathbb{N}$

Figure 1: Caption in report

A Chase code

```haskell
module Chase where
import Parser
import Helpers
import Debug.Trace
import Data.List

chaseVerify :: [Formula] -> [Formula]
-- verifies that each formula is in positive existential form and performs some
-- normilization on implied/constant implications
chaseVerify formulae =
   let isNotPEF = not.isPEF in
   map (\f -> case f of
      Implication a b ->
         if isNotPEF a || isNotPEF b then error ("implication␣must␣be␣in␣positive␣existential␣
         else f
      _ ->
         if isNotPEF f then error ("formula␣must␣be␣in␣positive␣existential␣form:␣" ++ showFor
         else (Implication Tautology f)
   ) formulae

chase :: [Formula] -> [Model]
-- runs the chase algorithm on a given theory and returns a list of models that
-- satisfy it
chase formulae = chase' (chaseVerify formulae) ([],[(mkModel [] [])])

chase' :: [Formula] -> ([Model],[Model]) -> [Model]
-- used by the chase function to hide the model identity argument
chase' formulae (done,[]) = done
chase' formulae (done,pending) =
   let self = chase' formulae in
   let (p:ending) = pending in
   trace ("running␣chase␣on␣" ++ show (done,pending)) $
   if all (\f -> holds p (UniversalQuantifier (freeVariables f) f)) formulae then
      trace ("␣␣all␣formulae␣in␣theory␣hold␣for␣model␣" ++ showModel p) $
      trace ("␣␣moving␣model␣into␣done␣list") $
      self (union done [p],ending)
   else
      let possiblySatisfiedModels = attemptToSatisfyFirstFailure p formulae in
      trace ("␣␣at␣least␣one␣formula␣does␣not␣hold␣for␣model␣" ++ showModel p) $
      trace ("␣␣unioning␣" ++ show ending ++ "␣with␣[" ++ intercalate ",␣" (map showModel poss
      self (done, union ending possiblySatisfiedModels)

attemptToSatisfyFirstFailure :: Model -> [Formula] -> [Model]
-- checks if each formula holds, sequentially, until one does not, then tries
-- to satisfy that formula
attemptToSatisfyFirstFailure model (f:ormulae) =
   let self = attemptToSatisfyFirstFailure model in
   if holds model (UniversalQuantifier (freeVariables f) f) then self ormulae
   else attemptToSatisfy model f

attemptToSatisfy :: Model -> Formula -> [Model]
-- returns a model that is altered so that the given formula will hold
attemptToSatisfy model formula =
   let f' = UniversalQuantifier (freeVariables formula) formula in
   trace ("␣␣attempting␣to␣satisfy␣(" ++ showFormula formula ++ ")") $
   attemptToSatisfy' model [] f'

attemptToSatisfy' :: Model -> Environment -> Formula -> [Model]
-- hides the environment identity in the `attemptToSatisfy` function arguments
```

```haskell
attemptToSatisfy' model env formula =
  let (domain,relations) = model in
  let domainSize = length domain in
  let self = attemptToSatisfy' model in
  -- trace ("  attempting to satisfy (" ++ showFormula formula ++ ") with env " ++ show env)
  case formula of
    Tautology -> [model]
    Contradiction -> []
    Or a b -> union (self env a) (self env b)
    And a b -> concatMap (\m -> attemptToSatisfy' m env b) (self env a)
    Implication a b -> if holds' model env a then self env b else []
    Atomic predicate vars ->
      let newRelation = mkRelation predicate (length vars) [genNewRelationArgs env vars (fr
      let newModel = mkModel (mkDomain domainSize) (mergeRelation newRelation relations) in
      trace ("    adding new relation: " ++ show newRelation) $
      [newModel]
    ExistentialQuantifier [] f -> self env f
    ExistentialQuantifier (v:vs) f ->
      let f' = ExistentialQuantifier vs f in
      let nextDomainElement = fromIntegral $ (length domain) + 1 in
      if any (\v' -> holds' model (hashSet env v v') f') domain then
        trace ("    " ++ showFormula formula ++ " already holds") $
        [model]
      else
        trace ("    adding new domain element " ++ show nextDomainElement ++ " for variabl
        attemptToSatisfy' (mkDomain nextDomainElement, relations) (hashSet env v nextDomain
    UniversalQuantifier [] f -> self env f
    UniversalQuantifier (v:vs) f ->
      let f' = UniversalQuantifier vs f in
      concatMap (\v' -> self (hashSet env v v') f') domain
    _ -> error ("formula not in positive existential form: " ++ showFormula formula)

genNewRelationArgs :: Environment -> [Variable] -> DomainElement -> [DomainElement]
-- for each Variable in the given list of Variables, retrieves the value
-- assigned to it in the given environment, or the next domain element if it
-- does not exist
genNewRelationArgs env [] domainSize = []
genNewRelationArgs env (v:ars) domainSize =
  let self = genNewRelationArgs env in
  case lookup v env of
    Just v' -> v' : (self ars domainSize)
    _ -> (domainSize+1) : (self ars (domainSize+1))
```

References

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