Interactive Model Finding with Hominy

by

Taymon A. Beal

A Major Qualifying Project

submitted to the faculty

of

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

in

Computer Science

by

March 2015

APPROVED:

Professor Daniel Dougherty, Project Advisor

Abstract

We present a new extension to the Hominy model finding utility, allowing it to be used interactively. This extension provides a command line interface with a read-eval-print loop (REPL) and a Web-based graphical interface. These allow the user to interactively discover the consequences of applying particular augmentations to models. We define a query language to facilitate this exploration. The semantics of this query language are defined in terms of a directed graph structure on the space of finite models of a geometric theory.

Acknowledgements

We gratefully acknowledge Daniel Dougherty for his work as advisor of this project, and Salman Saghafi for providing invaluable assistance in working with the Razor codebase.

Contents

1 Introduction **1**

1.1 Model Finding 1

1.2 Hominy 2

1.3 This Project 2

2 Models and Theories **4**

2.1 Syntax 4

2.1.1 Geometric Logic 6

2.1.2 Converting Functions to Relations 8

2.2 Models 8

2.3 Homomorphism and Minimality 10

2.4 The Chase 10

3 Augmentations and the Model-Space Graph **13**

3.1 Augmentations (Informal) 13

3.2 Augmentations (Formal) 14

3.3 The Model-Space Graph (Informal) 16

3.4 The Model-Space Graph (Formal) 17

4 User Interfaces **19**

4.1 The Command-Line Interface 19
4.1.1 Syntax 19
4.1.2 Formal Semantics 20
4.1.3 REPL Behavior 21
4.2 The Web Interface 22

5 Implementation **24**
5.1 Organization of the Codebase 24
5.2 Handling Graphlocs 34
5.3 Implementing the Web Interface 34

6 Conclusions **36**
6.1 Future Work 36

A Code Listing **38**

Chapter 1

Introduction

1.1 Model Finding

Model finding [14] [25] [21] [3] is an approach within formal methods. Its purpose is to discover properties of something that can be formally specified, such as security policies, protocols, and software designs. Model finding is an alternative to logical deduction; instead of directly determining the logical consequences of a formal theory, the problem is framed as finding scenarios, or models, which are consistent with that theory.

Historically, most formal methods tools have not been readily usable by non-experts in formal logic, but in recent years there has been an emerging class of formal methods tools designed to be usable by non-experts; this area is known as lightweight formal methods [10] [8]. Model finding is a useful approach in lightweight formal methods because it benefits from concreteness.

Previous work in lightweight model finding has included the Alloy Analyzer [9], the Margrave Policy Analyzer (based on Alloy) [7] [16], and Aluminum [15].

1.2 Hominy

Hominy is a lightweight model finding utility. It offers three advantages over other model-finding tools:

- Hominy can find first-order models of unbounded size. Other model-finding tools such as Alloy require a maximum model size to be specified; these tools work by reducing first-order theories to propositional theories (which can be done only if the model size is bounded), and then finding models using an off-the-shelf SAT solver. Hominy instead uses the Chase algorithm, which does not require the model size to be bounded.

- Hominy can trace the provenance of a fact or domain element in a model; that is, it can identify which components of the theory require that fact or element to be present in the model.

- Hominy provides interactivity. The focus of this project has been to add this functionality.

1.3 This Project

This project adds to Hominy an interactive mode, wherein users can navigate different possible models of a theory after that theory has been specified. Given a particular model, the user can augment that model by adding new observations or model elements. Hominy will then follow the logical consequences of these augmentations and present the user with the resulting augmented model, if one can be generated. Observations can be specified in the language of the theory, with the additional facility of being able to reference specific elements of the model to be augmented.

Hominy is part of a long-term project on model-finding at Worcester Polytechnic Institute. The current generation of this work in represented in the Razor tool [19]. The innovations described in this project have been incorporated into Razor, though sometimes in slightly altered form.

Chapter 2

Models and Theories

Definitions 1–7, 11–16, and 17 can be found in any standard reference on mathematical logic, such as Enderton [6].

2.1 Syntax

Definition 1. In first-order logic, a *signature* $\mathcal{L}$ is specified by a tuple $(\mathcal{R}, \mathcal{F}, \mathcal{A})$, where: [6]

- $\mathcal{R}$ is a finite set of *relation symbols*.
- $\mathcal{F}$ is a finite set of *function symbols*.
- $\mathcal{A}$, the *arity function*, is a function of type $\mathcal{R} \cup \mathcal{F} \to \mathbb{N}$.

For a relation or function symbol s, we refer to $A(s)$ as the *arity* of s. A function symbol whose arity is 0 is also known as a *constant symbol*.

Given a signature $\mathcal{L} = (\mathcal{R}, \mathcal{F}, \mathcal{A})$, we can define a number of syntactic forms over $\mathcal{L}$.

Definition 2. We assume an infinite set $\mathcal{V}$ of *variables*. A *term*, defined inductively, is either:

- a variable; or
- a *function application* of the form $f(t_1, \ldots, t_n)$, where $f \in \mathcal{F}$, $\mathcal{A}(s) = n$, and each t_i is a term.

Definition 3. An *atom* is either:

- a *relation application* of the form $r(t_1, \ldots, t_n)$, where $r \in \mathcal{R}$, $\mathcal{A}(r) = n$, and each t_i is a term; or
- an *equation* of the form $t_1 = t_2$, where t_1 and t_2 are terms.

Definition 4. A *formula*, defined inductively, is either:

- *truth* ($\top$); or
- *falsehood* ($\bot$); or
- an atom; or
- a *negation* $\neg a$, where a is a formula; or
- a *conjunction* $a_1 \wedge a_2$, where a_1 and a_2 are formulas; or
- a *disjunction* $a_1 \vee a_2$, where a_1 and a_2 are formulas; or
- an *implication* $a_1 \implies a_2$, where a_1 and a_2 are formulas; or
- a *universal quantification* $\forall v.a$, where v is a variable and a is a formula; or
- an *existential quantification* $\exists v.a$, where v is a variable and a is a formula.

Definition 5. A term or formula has a set F of *free variables*. The set of free variables of a term is calculated as follows:

- $F(v) = \{v\}$

- $F(f(t_1,\ldots,t_n)) = \bigcup_i F(t_i)$

The set of free variables of a formula is calculated as follows:

- $F(\top) = \varnothing$
- $F(\bot) = \varnothing$
- $F(r(t_1,\ldots,t_n)) = \bigcup_i F(t_i)$
- $F(t_1 = t_2) = F(t_1) \cup F(t_2)$
- $F(\neg a) = F(a)$
- $F(a_1 \wedge a_2) = F(a_1) \cup F(a_2)$
- $F(a_1 \vee a_2) = F(a_1) \cup F(a_2)$
- $F(a_1 \implies a_2) = F(a_1) \cup F(a_2)$
- $F(\forall v.a) = F(a) \setminus \{v\}$
- $F(\exists v.a) = F(a) \setminus \{v\}$

Definition 6. A *sentence* is a formula with no free variables ($F(a) = \varnothing$).

Definition 7. A *theory* is a set of sentences.

2.1.1 Geometric Logic

Hominy does not support unrestricted first-order logic; instead, it uses a restricted subset of it called *geometric logic* [20] [22] [23] [24].

Definition 8. A *positive existential formula*, defined inductively, is either:

- *truth* ($\top$); or

- *falsehood* ($\bot$); or
- an atom; or
- a *conjunction* $a_1 \wedge a_2$, where a_1 and a_2 are positive existential formulas; or
- a *disjunction* $a_1 \vee a_2$, where a_1 and a_2 are positive existential formulas; or
- an *existential quantification* $\exists v.a$, where v is a variable and a is a positive existential formula.

Definition 9. A *geometric sentence* is a sentence of the form $\forall v_1, \ldots, v_n.\alpha \implies \beta$, where α and β are positive existential formulas.

By convention, the leading universal quantifier may be omitted; all variables which are not explicitly existentially quantified are implicitly universally quantified.

Lemma 10. *Every first-order theory is equisatisfiable with some geometric theory.*

Proof: To convert an arbitrary theory to a geometric theory, first convert each sentence to conjunctive normal form [6]. Each such sentence is then expressible as a conjunction of zero or more clauses of the form $a_1 \vee \cdots \vee a_m \vee \neg b_1 \vee \cdots \vee \neg b_n$, where each a_i and each b_i is an atom. (If both m and n is zero, $\bot$ is used for the empty disjunction.) Each such clause is then converted to an equivalent geometric sentence of the form $b_1 \wedge \cdots \wedge b_n \implies a_1 \vee \cdots \vee a_m$. (If n is zero, $\top$ is used for the empty conjunction.)

Note that, traditionally, converting a first-order sentence to conjunctive normal form involves removing all existential quantifiers through skolemization; however, because existential quantifiers are allowed on the right-hand side of the implication in a geometric sentence, those existential quantifiers which will be placed there do not need to be skolemized.

2.1.2 Converting Functions to Relations

Any theory using function symbols can be converted to an equisatisfiable theory without function symbols. This is done as follows:

1. For each sentence in the theory, if that sentence contains a function application that is an argument to a function or relation or is on the right-hand side of an equation, rewrite that sentence by replacing the atom containing that application with $\forall v.f(\dots) = v \implies a$, where v is an otherwise unused variable, $f(\dots)$ is the function application, and a is the atom except that the function application is replaced with v. Repeat this process until the only function applications are directly on the left-hand side of equations.

2. Change each function symbol in the signature into a relation symbol and increase its arity by 1. Replace each equation of the form $f(v_1,\dots,v_n) = v_r$ with a relation application of the form $f(v_1,\dots,v_n,v_r)$.

3. For each n-ary function symbol that was changed into an $n+1$-ary relation symbol, add to the theory a sentence of each of the following forms:

   - $\forall v_1,\dots,v_n.\exists x.f(v_1,\dots,v_n,x)$
   - $\forall x,y,v_1,\dots,v_n.f(v_1,\dots,v_n,x) \wedge f(v_1,\dots,v_n,y) \implies x = y$

2.2 Models

Definition 11. Given a signature $\mathcal{L} = (\mathcal{R},\mathcal{F},\mathcal{A})$, a *model* $\mathcal{M}$ for $\mathcal{L}$ is specified by a tuple $(\mathcal{U}, I)$, where: [6]

- $\mathcal{U}$, the *universe*, is a set.

- I, the *interpretation function*, maps each relation symbol $r \in \mathcal{R}$ to a relation that is a subset of $\mathcal{U}^{\mathcal{A}(r)}$.

Although mathematical logic traditionally requires universes to be nonempty, it will be useful for purposes described later to consider the *empty model* $\varnothing_{\mathcal{M}}$. The universe of the empty model is the empty set. The interpretation function of the empty model maps every relation symbol to the empty set.

Definition 12. Given a model $\mathcal{M} = (\mathcal{U}, I)$, an *environment* for $\mathcal{M}$ is a function of type $\mathcal{V} \to \mathcal{U}$.

Definition 13. A *substitution* $\eta[x \mapsto e]$ of an element $e \in \mathcal{U}$ for a variable x onto an environment η for an environment η for $\mathcal{M}$ is defined such that

$$\eta[x \mapsto e](v) = \begin{cases} e & : v = x \\ \eta(v) & : v \neq x \end{cases}$$

Definition 14. *Satisfaction* ($\models$) is a relation between models, environments, and formulas. Given a signature $\mathcal{L} = (\mathcal{R}, \mathcal{F}, \mathcal{A})$, for a model $\mathcal{M} = (\mathcal{U}, I)$ for $\mathcal{L}$ and an environment η for $\mathcal{M}$, satisfaction is defined as follows (with all formulas assumed to be over $\mathcal{L}$):

- For truth: $\mathcal{M} \models_\eta \top$.
- For falsehood: $\mathcal{M} \not\models_\eta \bot$.
- For relation applications: $\mathcal{M} \models_\eta r(t_1, \ldots, t_n)$ iff $(e_1, \ldots, e_n) \in I(r)$, where each $e_i = \eta(t_i)$.
- For equations: $\mathcal{M} \models_\eta t_1 = t_2$ iff $\eta(t_1) = \eta(t_2)$.
- For conjunctions: $\mathcal{M} \models_\eta a_1 \wedge a_2$ iff $\mathcal{M} \models_\eta a_1$ and $\mathcal{M} \models_\eta a_2$.
- For disjunctions: $\mathcal{M} \models_\eta a_1 \vee a_2$ iff $\mathcal{M} \models_\eta a_1$ or $\mathcal{M} \models_\eta a_2$.

- For existential quantifications: $\mathcal{M} \models_\eta \exists v.a$ iff there exists an element $e \in \mathcal{U}$ such that $\mathcal{M} \models_{\eta[v \mapsto e]} a$.
- For implications: $\mathcal{M} \models_\eta \alpha \implies \beta$ iff $\mathcal{M} \models_\eta \alpha$ implies $\mathcal{M} \models_\eta \beta$.

Definition 15. *Universal satisfaction* is a relation between models and formulas. Given a signature $\mathcal{L}$, for a model $\mathcal{M}$ for $\mathcal{L}$ and κ as a formula over $\mathcal{L}$, $\mathcal{M} \models \kappa$ iff $\mathcal{M} \models_\eta \kappa$ for every environment η for $\mathcal{M}$.

2.3 Homomorphism and Minimality

Definition 16. Given a signature $\mathcal{L} = (\mathcal{R}, \mathcal{F}, \mathcal{A})$, for two models $\mathcal{M}_1 = (\mathcal{U}_1, I_1)$ and $\mathcal{M}_2 = (\mathcal{U}_2, I_2)$ for $\mathcal{L}$, a *homomorphism* from $\mathcal{M}_1$ to $\mathcal{M}_2$ is a function $h : \mathcal{U}_1 \to \mathcal{U}_2$ such that, for every $r \in \mathcal{R}$ and every $(e_1, \ldots, e_n) \in \mathcal{U}_1^{\mathcal{A}(r)}$, if $(e_1, \ldots, e_n) \in I_1(r)$, then $(h(e_1), \ldots, h(e_n)) \in I_2(r)$.

Definition 17. Existence of homomorphisms defines a preorder over models for a given signature [19]. $\mathcal{M}_1 \preceq \mathcal{M}_2$ iff a homomorphism exists from $\mathcal{M}_1$ to $\mathcal{M}_2$. A model $\mathcal{M}$ is *minimal* iff, for every model $\mathcal{M}'$ for that signature, $\mathcal{M} \preceq \mathcal{M}'$.

Definition 18. An *isomorphism* between $\mathcal{M}_1$ and $\mathcal{M}_2$ is a homomorphism from $\mathcal{M}_1$ to $\mathcal{M}_2$ that is a bijective function whose inverse is a homomorphism from $\mathcal{M}_2$ to $\mathcal{M}_1$.

Definition 19. A *set of support* for a set S of models is a subset S_0 of S such that for every model $\mathcal{M} \in S$ there exists a model $\mathcal{M}_0 \in S_0$ such that $\mathcal{M}_0 \preceq \mathcal{M}$.

2.4 The Chase

Given a geometric theory $\mathcal{T}$ and a model $\mathcal{M}$ with the same signature as $\mathcal{T}$, the Chase algorithm calculates [19] a set of support for $\{\mathcal{M}_1 \models \mathcal{T} \mid \mathcal{M} \preceq \mathcal{M}_1\}$.

The function chase$(\mathcal{T}, \mathcal{M})$ is calculated as follows [1] [4] [12]:

1. Check if $\mathcal{M} \models \mathcal{T}$. If so, yield $\mathcal{M}$, then halt.
2. Select a geometric sentence $\alpha \implies \beta$ in $\mathcal{T}$ and an environment η such that $\mathcal{M} \not\models \alpha \implies \beta$.
3. If $\beta = \bot$, then halt with failure. Otherwise, select an existentially quantified conjunction
   $\exists v_1 \ldots v_n . a_1 \wedge \cdots \wedge a_n$ from β.
4. For each v_i, generate a new domain element e_i.
5. For each a_i:
   - If a_i is a relation application $(r(t_1, \ldots, t_n))$, mutate $\mathcal{M}$ by adding that $r(t_1, \ldots, t_n)$ to it as a fact.
   - If a_i is an equation $(t_1 = t_2)$, mutate $\mathcal{M}$ by replacing each instance of $\eta'(t_2)$ with $\eta'(t_1)$.
6. Repeat from step 1.

Previous publications on Hominy have presented the Chase as a nondeterministic algorithm [18]. However, the interactive component of Hominy introduced in this project requires that the models yielded by the Chase be ordered consistently. Therefore, the selections in steps 2 and 3 must be made according to some well-defined deterministic procedure. The choice of selection procedure is arbitrary.

A theory $\mathcal{T}$ is satisfiable iff the Chase does not fail when run on that model. Furthermore, for every model $\mathcal{M}_1$ of a theory $\mathcal{T}$, there exists a model $\mathcal{M}_2 \in \text{chase}(\mathcal{M}_1, \varnothing_{\mathcal{M}})$, and there exists a homomorphism from $\mathcal{M}_2$ to $\mathcal{M}_1$. These results are demonstrated in Saghafi [19].

The Chase is not guaranteed to terminate. Since every formula in first-order logic is equisatisfiable with some geometric theory, no algorithm that finds models for geometric theories can be guaranteed to halt, as the satisfiability of an arbitrary formula in first-order logic is undecidable [19].

Chapter 3

Augmentations and the Model-Space Graph

In this chapter, we define a directed graph structure on the space of finite models of a geometric theory. It will provide a formal semantics for the augmentation operations of Hominy.

3.1 Augmentations (Informal)

Prior to this project, Hominy was capable only of finding the set of support for its input theory. However, if the initial input model is nonempty, the Chase can also be used to *augment* existing models of a theory that were generated by a previous run of the Chase on that same theory.

In order to do this, one must first specify exactly how the model is to be augmented. Two kinds of augmentations are supported: specifying a new logical fact which is required to be true in the new model (but is not true in the existing model), and requiring a new element to be added to the model.

When specifying a new logical fact that will be required to be true (called an *observation* [19]), existing elements of the model to be augmented may be referred to. To make this possible, observations are sentences in an expanded language that includes special symbols for existing model elements, in addition to the symbols defined in the original signature.

Once a model has been augmented in this way, it may no longer satisfy the original theory. So the next step is to run the Chase again, using the original theory and the augmented model as inputs. All models which are output by this run of the Chase will satisfy the theory, as the Chase always requires; furthermore, the augmented model will be a submodel of every model output this way.

From a user's perspective, this capability is useful because it means that the initial set of support generated by the Chase need not be the end of the story. A user can ask what happens if one of the models is augmented in a particular way, and the Chase will demonstrate the consequences of this. This allows the user to gain a more concrete understanding of why a given model contains the facts and elements that it does.

3.2 Augmentations (Formal)

Definition 20. We assume a countably infinite set $\mathcal{E}$ of *model element symbols*. For every countable model $\mathcal{M} = (\mathcal{U}, I)$, we define a one-to-one correspondence $\rho : S \to \mathcal{U}$, where $S \subseteq \mathcal{E}$.

Definition 21. A *fact* is of the form $r(e_1, \ldots, e_n)$, where $r \in \mathcal{R}$, $\mathcal{A}(r) = n$, and $e_i \in \mathcal{U}$ for each e_i.

Every model can be written as a universe and a set of facts which are true in that model.

Definition 22. An *observation* for a model $\mathcal{M}$ is a positive existential formula not containing $\bot$ or disjunctions, written in an expanded signature that adds each model element symbol in S as a constant (nullary function) symbol. (Disjunctions are disallowed because it is not useful for the user to specify that one of two facts should be added to a model; they might as well specify which fact should be added [19].) The interpretation $I_{\mathcal{M}}(e)$ of a model element symbol is $\rho(e)$. An observation must be a sentence and does not implicitly universally quantify any variables; they must all be existentially quantified.

Because an observation has no free variables, satisfaction of it is independent of environment.

We define two operations to augment models: add and new_element.

Definition 23. Given a model $\mathcal{M} = (\mathcal{U}, F)$, new_element$(\mathcal{M}) = (\mathcal{U} \cup \{e\}, F)$, where e is a new element such that $e \notin \mathcal{U}$.

Definition 24. Given a model $\mathcal{M} = (\mathcal{U}, F)$ and an observation o, add$(\mathcal{M}, o)$ is defined as follows:

- For truth: add$(\mathcal{M}, \top)$ = $\mathcal{M}$
- For relation applications: add$(\mathcal{M}, r(e_1, \ldots, e_n)) = (\mathcal{U}, F \cup \{r(e_1, \ldots, e_n)\})$
- For equations: add$(\mathcal{M}, e_1 = e_2) = (\mathcal{U} \setminus \{\rho(e_2)\}, F_N)$,
  where F_N is F with each instance of e_2 replaced with e_1.
- For conjunctions: add$(\mathcal{M}, a_1 \wedge a_2) =$ add(add$(\mathcal{M}, a_1), a_2)$
- For existential quantifications:

$$\text{add}(\mathcal{M}, \exists v.a) = \begin{cases} \mathcal{M} & : \mathcal{M} \models \exists v.a \\ \text{add}(\text{new\_element}(\mathcal{M}), a_N) & : \mathcal{M} \not\models \exists v.a \end{cases}$$

where a_N is a with each instance of v replaced with the model element symbol for the new element added with new_element.

(Note that, because all variables in an observation must be existentially quantified, and add replaces existentially quantified variables with model elements, all terms in relation applications and equations that are inputs to add will be model elements.)

3.3 The Model-Space Graph (Informal)

When a new fact or element is added directly to a model of a theory, the new model may no longer satisfy that theory. However, running the Chase on the new model generates a stream of new models, each of which satisfies the theory.

The space of all models satisfying a theory can be thought of as an infinite graph. Each model is a node in the graph, and each possible augmentation of a model provides an edge from that model to a the first of a new set of models, which corresponds to the stream produced by applying that augmentation to a model.

One can think of this model-space graph as being organized into rows. A single model corresponds to a node in the graph, and a stream of models corresponds to a row. The bottom row of the graph is the set of support for the theory. Rows with augmentations are above the models they originated from. Because there are many possible augmentations that can be applied to any model, each model is connected to many rows above it.

For instance, the above diagram shows part of the model-space graph for the following theory:

$$\begin{aligned} \top &\implies \exists f.\text{File}(f) \\ \text{File}(f) &\implies \text{Symlink}(f) \vee \text{Regular}(f) \\ \text{Symlink}(s) &\implies \exists f.\text{LinkTo}(s, f) \end{aligned}$$

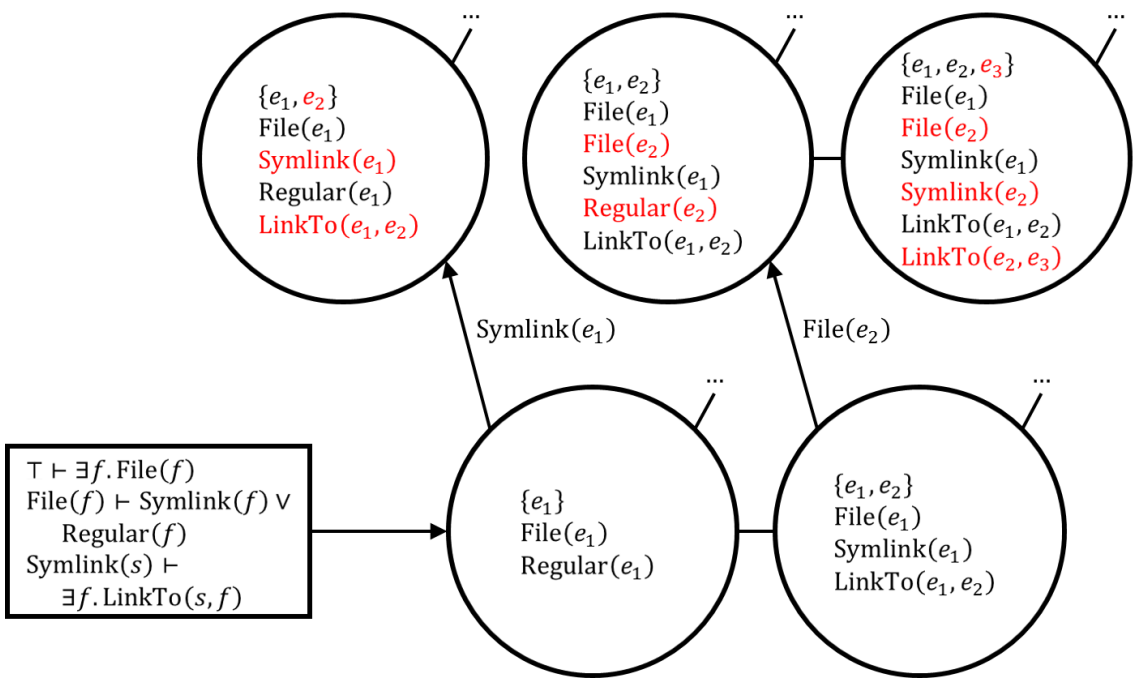

Figure 3.1: Part of the graph representing the space of all models of a theory that might be used for a filesystem.

Hominy's user interfaces are centered around exploration of the model-space graph. Users are able to navigate back and forth between models within a stream, and to move up and down in the graph by applying and removing augmentations.

A model in the graph can be specified by a sequence of augmentations to apply and indices associated with each step. To find the model so specified, Hominy generates the given theory's set of support and identifies the model at the starting index. For each augmentation, it then applies that augmentation, generating a new model stream, and uses the model at the corresponding index in that stream as the basis for the next augmentation.

3.4 The Model-Space Graph (Formal)

The Chase outputs a countable set of models. Consequently, we can assume a function index(S, n), where S is a set of models output by the Chase and n is a natural number less

than the cardinality of S. It returns an element of S, and provides a one-to-one mapping between S and the natural numbers less than the cardinality of S. The exact definition of index is arbitrary and not specified here.

Definition 25. A *graphloc* is specified by a tuple $(\mathcal{T}, i_0, ((a_1, i_1), \ldots, (a_n, i_n)))$, where *thy* is a geometric theory, each i_i is a natural number, and each a_i is either new_element or $add(o)$, where o is an observation.

Definition 26. Given a graphloc G, model_at(G) is recursively defined as follows:

- model_at$((\mathcal{T}, i_0, ())) = \text{index}(\text{chase}(\mathcal{T}, \varnothing_{\mathcal{M}}), i_0)$
- model_at$((\mathcal{T}, i_0, ((a_1, i_1), \ldots, (a_{n-1}, i_{n-1}), (a_n, i_n)))) =$
  $\text{index}(\text{chase}(\mathcal{T}, a_n(\text{model\_at}((\mathcal{T}, i_0, ((a_1, i_1), \ldots, (a_{n-1}, i_{n-1})))))), i_n)$

The function model_at is partial; if any i_i is greater than or equal to the cardinality of the output from the corresponding run of the Chase, the value is undefined.

Chapter 4

User Interfaces

4.1 The Command-Line Interface

4.1.1 Syntax

The Hominy command language has the following syntax:

⟨*command*⟩ ::= ⟨*expression*⟩ | ⟨*assignment*⟩ | ‘`exit`’

⟨*assignment*⟩ ::= ⟨*identifier*⟩ ‘`:=`’ ⟨*expression*⟩ | ‘`save`’ ⟨*identifier*⟩

⟨*expression*⟩ ::= ⟨*theory_literal*⟩ | ⟨*string_literal*⟩ | ‘`@`’ ⟨*identifier*⟩ | ‘`~`’
| [⟨*expression*⟩ ‘`.`’] ⟨*operation*⟩ | ⟨*quantified*⟩

⟨*operation*⟩ ::= ‘`add`’ ⟨*quantified*⟩ | ‘`new_element`’ | ‘`next`’ | ‘`previous`’ | ‘`first`’
| ‘`remove_last`’ | ‘`remove_all`’

⟨*theory_literal*⟩ ::= ‘`[`’ ⟨*sequent*⟩ ‘`;`’ ⟨*sequent*⟩ ‘`]`’

⟨*sequent*⟩ ::= [⟨*conjunction*⟩ ‘`=>`’] ⟨*disjunction*⟩

⟨*disjunction*⟩ ::= ‘`Falsehood`’ | ⟨*quantified*⟩ ‘`|`’ ⟨*quantified*⟩

⟨*quantified*⟩ ::= [‘`exists`’ ⟨*identifier*⟩ ⟨*identifier*⟩ ‘`.`’] ⟨*conjunction*⟩

⟨*conjunction*⟩ ::= ‘`Truth`’ | ⟨*atom*⟩ ‘`&`’ ⟨*atom*⟩

⟨*atom*⟩ ::= ⟨*relational_fact*⟩ | ⟨*equality*⟩

⟨*relational_fact*⟩ ::= ⟨*identifier*⟩ [‘`(`’ [⟨*term*⟩ ‘`,`’ ⟨*term*⟩] ‘`)`’]

⟨*equality*⟩ ::= ⟨*term*⟩ ‘`=`’ ⟨*term*⟩

⟨*term*⟩ ::= ⟨*identifier*⟩ | ⟨*function_value*⟩ | ‘`#`’ ⟨*natural_number*⟩

⟨*function_value*⟩ ::= ⟨*identifier*⟩ ‘`(`’ [⟨*term*⟩ ‘`,`’ ⟨*term*⟩] ‘`)`’

The lexical syntax of the Hominy command language, including the definitions of ⟨*identifier*⟩ and ⟨*string_literal*⟩, is the same as that of Haskell except that a command cannot span more than one line. ⟨*natural_number*⟩ is a lexeme defined as a sequence of one or more ASCII decimal digits.

4.1.2 Formal Semantics

The semantics of the Hominy query language map expressions, as defined in the above concrete syntax, onto graphlocs as defined in the previous chapter. This is a partial mapping; some expressions have undefined values. This mapping connects the operational semantics of the Hominy query language to the denotational semantics specified in the previous chapter.

Definition 27. This mapping, ϕ, is defined as follows:

- $\phi(\langle \textit{theory\_literal} \rangle) = (\mathcal{T}, 0, ())$, where $\mathcal{T}$ is the theory specified by ⟨*theory_literal*⟩.

For $\phi(\langle \textit{expression} \rangle \texttt{.} \langle \textit{operation} \rangle)$, let $(\mathcal{T}, i_0, ((a_1, i_1), \ldots, (a_n, i_n))) = \phi(\langle \textit{expression} \rangle)$. Then:

- $\phi(\langle \textit{expression} \rangle \texttt{.add} \ \langle \textit{quantified} \rangle) = (\mathcal{T}, i_0, ((a_1, i_1), \ldots, (a_n, i_n), (\text{add}(o), 0)))$, where o is the observation specified by ⟨*quantified*⟩.

- $\phi(\langle \textit{expression} \rangle \texttt{.new\_element}) = (\mathcal{T}, i_0, ((a_1, i_1), \ldots, (a_n, i_n), (\text{new\_element}, 0)))$.

- $\phi(\langle \textit{expression} \rangle \texttt{.next}) = (\mathcal{T}, i_0, ((a_1, i_1), \ldots, (a_n, i_n + 1)))$,
  or, if $n = 0$, $(\mathcal{T}, i_0 + 1, ())$.

- $\phi(\langle \textit{expression} \rangle \texttt{.previous}) = (\mathcal{T}, i_0, ((a_1, i_1), \ldots, (a_n, i_n - 1)))$,
  or, if $n = 0$, $(\mathcal{T}, i_0 - 1, ())$. If $i_n = 0$, it is undefined.

- $\phi(\langle \textit{expression} \rangle \texttt{.first}) = (\mathcal{T}, i_0, ((a_1, i_1), \ldots, (a_n, 0)))$,
  or, if $n = 0$, $(\mathcal{T}, 0, ())$.

- $\phi(\langle \textit{expression} \rangle \texttt{.remove\_last}) = (\mathcal{T}, i_0, ((a_1, i_1), \ldots, (a_{n-1}, i_{n-1})))$.
  If $n = 0$, it is undefined.

- $\phi(\langle \textit{expression} \rangle \texttt{.remove\_all}) = (\mathcal{T}, 0, ())$.

4.1.3 REPL Behavior

The Hominy command-line interface takes the form of a read-eval-print loop, or REPL. Users are prompted to enter a command in the terminal, the command is executed, and the results are displayed in the terminal if applicable. This process repeats until the user exits with the `exit` command.

An expression entered by itself is a display command. The expression is interpreted as a graphloc as specified above, the model_at function is used to calculate a model from that graphloc, and that model is output to the console. If either the graphloc or the model is undefined, an error message is displayed instead.

An assignment command interprets the given expression as a graphloc and then saves that graphloc under the specified name. If the graphloc is undefined, an error message is displayed instead.

`~`, `@`⟨*identifier*⟩, and ⟨*string_literal*⟩, are all expressions interpreted as graphlocs.

`~` refers to the graphloc from the most recently entered display command that was interpreted as a defined graphloc. Its value is undefined if no such display command has been entered.

`@`⟨*identifier*⟩ refers to the graphloc most recently saved under the given name by an assignment command. Its value is undefined if no graphloc has been assigned to that name.

⟨*string_literal*⟩, when evaluated, opens the file whose path is the specified string and attempts to read a geometric theory from it, using the same syntax as ⟨*theory_literal*⟩ except that sequents may be separated by either newlines or semicolons. If a theory is successfully read this way, ⟨*string_literal*⟩ is equivalent to a ⟨*theory_literal*⟩ of that theory. Otherwise, its value is undefined.

Some features of the Hominy command language syntax are syntactic sugar:

- `save` ⟨*identifier*⟩ is equivalent to ⟨*identifier*⟩ `:= ~`.
- ⟨*operation*⟩, as a command by itself, is equivalent to `~.`⟨*operation*⟩.
- ⟨*quantified*⟩, as a command by itself, is equivalent to `~.add` ⟨*quantified*⟩.

4.2 The Web Interface

Hominy offers a Web-based graphical user interface. A user can access it by running the software as a local web server on the user's own computer, or the Web service can be hosted on an external server. The server-side code is written in pure Haskell using the Happstack framework [5].

The Hominy Web interface is completely stateless; it does not use cookies or any other mechanism for including state over HTTP, and no data is stored on the server except for caching, which is not relied on to ensure correctness. Each node in the model-space

graph has its own URL; a text representation of that model's 3-tuple is passed as a query parameter in the URL. Visiting the same URL (relative to the host) always returns the same page, regardless of how much time has passed or whether the same instance of the server process is running. As a result, a user who uses the Back button in their browser will find that it works as expected, and users can bookmark model pages and (if Hominy is hosted on an external server) send model URLs to one another, and these URLs will return the same results in the future and for other users as they did the first time.

At the home page, the user can enter a theory or upload it as a file. When the theory is submitted, if it is a syntactically valid theory, the user is redirected to the page for the first model in the initial stream for that theory. If not, they are redirected to an error page.

At the page for a node in the model-space graph, if there is a model at that node, that model's domain and a list of all facts true in that model are displayed. If there is no model at that node, a message is displayed to this effect. In addition, buttons are displayed corresponding to the operations in section 2; the Add button is accompanied by a text box to allow the user to enter an augmentation, and the Previous and Remove Last buttons are disabled if these respective operations cannot be applied to the current node. Each button redirects the user to the page for the model that would be returned by that operation.

To prevent unnecessary recalculation of previously calculated models, the Hominy web server caches the model corresponding to each model URL requested. If the same URL is requested again, the server generates the page based on the cached model instead of running the Chase again. If the server has been restarted or the cache is otherwise unavailable, the server will run the Chase again and the same results will be returned; caching serves only to speed up responses to multiple requests for the same model.

Chapter 5

Implementation

Our work was implemented in Haskell [13], on top of the existing Hominy codebase developed by Salman Saghafi.

A benefit of using a functional language with strong data types was that the actual code to perform the various operations specified in our formalism was often closely analogous to the formal definitions of those operations as given in this report.

5.1 Organization of the Codebase

As is typically the case for software projects of significant size, the code is organized into a hierarchical set of modules. The modules that already existed before this project were organized as follows:

- `Chase`, which defines the high-level implementation of the Chase.
- `CC`, a directory containing alternative implementations of the Chase, based on congruence closure.
  - `CC.CC`, an interface module which specifies which implementation should be used.

  - `CC.Incremental`, an implementation which uses incremental string rewriting.
  - `CC.Naive`, an implementation which also uses string rewriting transformations.
  - `CC.RelAlg`, an implementation which uses the relational algebraic libraries used in the `RelAlg` directory.
  - `CC.Shostak`, an implementation which uses Shostak's algorithm.

- `Formula`, a directory containing datatype definitions for formulas and parsers for those datatypes.
  - `Formula.SyntaxFol`, which defines unrestricted first-order formulas.
  - `Formula.SyntaxGeo`, which defines geometric formulas.

- `Problem`, a directory containing datatype definitions related to models and operations on those datatypes.
  - `Problem.BaseTypes`, which defines datatypes for unique identifiers used in computing the results of the Chase.
  - `Problem.Model`, which defines the datatype for a model.
  - `Problem.Observation`, which defines the datatype for an observation.
  - `Problem.Operations`, which defines operations on a model performed as part of the Chase.
  - `Problem.Provenance`, which defines datatypes used for tracing what caused a fact to be added to the model.
  - `Problem.Structures`, which defines datatypes used internally by the Chase.

  - `Problem.RelAlg`, a directory containing operations for the Chase implementation based on relational algebra.
    * `Problem.RelAlg.Operations`, which provides relational-algebraic implementations of the operations underlying Problem.Operations.
    * `Problem.RelAlg.RelAlg`, which defines the basic relational-algebraic datatypes used by this implementation and basic operations on them.

- `Test`, a directory containing test code modules corresponding to the other modules listed here.

- `Tools`, a directory containing miscellaneous functionality.
  - `Tools.Config`, which defines configuration options specifiable by the user.
  - `Tools.FolToGeo`, which converts arbitrary first-order formulas to geometric ones.
  - `Tools.GeoUnification`, which implements a unification algorithm used in the handling of equations.
  - `Tools.Herbrand`, which implements Herbrandization in order to remove universal quantifiers from formulas.
  - `Tools.Logger`, which implements logging.
  - `Tools.Narrowing`, which provides an alternate implementation of unification.
  - `Tools.Skolem`, which implements Skolemization in order to remove existential quantifiers from formulas.

- `Utils`, a directory containing utility functions used throughout the system.

  - `Utils.FolUtilities`, which provides utility functions for arbitrary first-order formulas.
  - `Utils.GeoUtilities`, which provides utility functions for geometric formulas.
  - `Utils.Trace`, a wrapper around Utils.Utils that exports only tracing-related functionality.
  - `Utils.Utils`, which provides utility functions for tracing and pretty-printing.

- `WeaklyAcyclic`, a directory containing an alternate implementation of the Chase designed to deal with the special case of weakly-acyclic theories.
  - `WeaklyAcyclic.WeaklyAcyclic`, which defines this implementation.

In many cases, modules were separated into an implementation module, which contains the actual code, and an interface module, which selectively exports only a subset of the names defined in the implementation module. Only the interface module is imported by other code, and the subset of names that it exports constitutes the API that other code is written against; this aids in the separation of interface and implementations. In cases where modules are divided this way, only the interface module is named in the above list. For example, the actual file `Chase.hs` is an interface module; it contains no code except for imports from the corresponding implementation module, `IChase.hs`, which is located in the same directory and is not listed separately above.

Part of the work on this project was to separation of concerns in the codebase. This separation was made more important by the presence of multiple different interactive interfaces, which caused tight coupling to be less viable than it had been when only batch processing of theories was supported. In particular, adding a particular observation to a model previously required working directly with the Chase's internal data structures that

contain internal state for that algorithm; this was not a problem when only the Chase itself would perform such additions, but would have made it awkward to do so with user-specified observations from the REPL or Web interface.

To this end, new APIs were created for specifying models, theories, and augmentations, and for calling the Chase. Because it was not feasible to refactor all the existing code at once, the existing code was moved into its own module tree, in a top-level directory named `Chase`. An API layer was written to translate models and theories from the new data structures into the preexisting ones, and vice versa. The design was such that new code would call the preexisting Chase code only through this layer, allowing it to be written using the new APIs while the old ones still existed for the benefit of preexisting code.

The following new modules were written:

- `Datatypes`, which defines data structures to represent models, geometric theories, and their logical components. These were designed to reflect the logical structure of geometric theories and models, as they might be exposed to the user, rather than reflecting implementation details of the Chase. The datatypes for theories are specific to geometric theories and are not designed to support unrestricted first-order logic. Graphlocs and the basic operations on them are also specified here. This module exports the following definitions:
  - `Sequent`, a datatype for geometric sentences.
  - `TAtom`, a datatype for atomic expressions in theories.
  - `TTerm`, a datatype for term expressions in theories.
  - `MAtom`, a datatype for atomic expressions in user-specified observations.
  - `MTerm`, a datatype for term expressions in user-specified observations.
  - `Model`, a datatype for models.

  - `Fact`, a datatype for facts within models.
  - `GraphLoc`, a datatype for graphlocs.
  - `VariableSymbol`, a datatype for variables in theories and user-specified observations.
  - `PredicateSymbol`, a datatype for relation symbols in facts, theories, user-specified observations.
  - `FunctionSymbol`, a datatype for function symbols in theories and user-specified observations.
  - `ModelElementSymbol`, a datatype for model element symbols in user-specified observations.
  - `add`, which implements the operation `add` on graphlocs.
  - `undoConstraint`, which implements the operation `remove_last` on graphlocs.
  - `previousLoc`, which implements the operation `previous` on graphlocs.
  - `nextLoc`, which implements the operation `next` on graphlocs.
  - `origin`, which implements the operation `remove_all` on graphlocs.

- `Chase`, which provides an interface to the existing implementation of the Chase using the new APIs. This module exports the following definitions:
  - `chase`, which implements model_at, resolving a graphloc into a model.
  - `chasifySequent`, which translates a geometric sentence as defined in `Datatypes` into the data structures used by `Chase.Formula.SyntaxGeo`.
  - `chasifyDisj`, which is analogous to `chasifySequent` but operates on disjunctions.

- `chasifyConj`, which is analogous to `chasifySequent` but operates on conjunctions.
- `chasifyConnective`, a helper function that abstracts out operations common to `chasifyDisj` and `chasifyConj`.
- `chasifyExQuant`, which is analogous to `chasifySequent` but operates on existential quantifications.
- `chasifyAtomicFormula`, which is analogous to `chasifySequent` but operates on atoms.
- `chasifyTAtom`, which is similar to `chasifyAtomicFormula`, but returns the datatype used specifically for atoms in `Chase.Formula.SyntaxGeo`, which is distinct from the one used for formulas in general.
- `chasifyTTerm`, which is analogous to `chasifySequent` but operates on terms in theories.
- `chasifyMTerm`, which is analogous to `chasifySequent` but operates on terms in user-specified constraints.
- `chasifyConstraint`, which translates an observation as defined in `Datatypes` into the data structures used by `Chase.Problem.Observation`.
- `dechasifyModel`, which takes a model, as expressed in the data structures used by `Chase.Problem.Structures`, and translates it back into a `Model` as defined in `Datatypes`.
- `dechasifyModelElement`, which takes a model element symbol, as expressed in the data structures used by `Chase.Formula.SyntaxGeo`, and translates it back into a `ModelElementSymbol` as defined in `Datatypes`.

- `Utility`, which provides helper functions used in other modules. This module exports the following definition:

  - `(!!!)`, an infix operator similar to the list indexing operator `(!!)`, but which handles out-of-bounds lookups using Maybe.

- `WebParse`, which implements a protocol for serializing graphlocs as concise strings of characters which are permitted in URL query string parameters. Parsec, a monadic parser combinator library for Haskell [11], was used to implement this. This module exports the following definitions:

  - `pGraphLoc`, a deserializer for graphlocs.
  - `pSequent`, a deserializer for geometric sentences.
  - `pTAtom`, a deserializer for atoms as used in theories.
  - `pTPredicate`, a deserializer for relation applications as used in theories.
  - `pTEquality`, a deserializer for equations as used in theories.
  - `pTTerm`, a deserializer for terms as used in theories.
  - `pTVariable`, a deserializer for references to variables in theories.
  - `pTFunction`, a deserializer for function applications as used in theories.
  - `pMAtom`, a deserializer for atoms as used in user-specified observations.
  - `pMPredicate`, a deserializer for relation applications as used in user-specified observations.
  - `pMEquality`, a deserializer for equations as used in user-specified observations.
  - `pMTerm`, a deserializer for terms as used in user-specified observations.
  - `pMVariable`, a deserializer for references to variables in user-specified observations.

- `pMFunction`, a deserializer for function applications as used in user-specified observations.
- `pModelElement`, a deserializer for references to model element symbols in user-specified observations.
- `pVariableSymbol`, a deserializer for variables.
- `pPredicateSymbol`, a deserializer for predicate symbols.
- `pFunctionSymbol`, a deserializer for function symbols.
- `pModelElementSymbol`, a deserializer for model element symbols.
- `pInt`, a deserializer for natural numbers used in model element symbols.
- `encodeGraphLoc`, a function that serializes graphlocs.
- `encodeSequent`, a function that serializes geometric sentences.
- `encodeTAtom`, a function that serializes atoms as used in theories.
- `encodeTTerm`, a function that serializes terms as used in theories.
- `encodeMAtom`, a function that serializes atoms as used in user-specified observations.
- `encodeMTerm`, a function that serializes terms as used in user-specified observations.
- `encodeVariableSymbol`, a function that serializes variables.
- `encodePredicateSymbol`, a function that serializes relation symbols.
- `encodeFunctionSymbol`, a function that serializes function symbols.
- `encodeModelElementSymbol`, a function that serializes model element symbols.

- WebServ, which implements the Web interface. This is a Main module and is executable as a program, which binds to a local port and runs a Web server which serves the Hominy web interface. This module exports the following definitions:
  - `main`, which starts the Web server.
  - `awi`, the router which handles the different possible URLs which can be served.
  - `find`, the handler for the `/find` URL.
  - `add`, the handler for the `/add` URL.
  - `error_`, the handler for the `/error` URL.
  - `home`, the handler for the `/` URL.
  - `template`, a helper function which provides boilerplate HTML that all responses are wrapped in.
  - `locURL`, a helper function that generates an absolute URL for a given graphloc.
- hominy.hs, which implements the command-line interface. This is a Main module and is executable as a program, which starts the read-eval-print loop. This module exports the following definitions:
  - `main`, which starts the REPL.
  - `modelLoop`, the main loop function which processes a single iteration of the REPL, then recursively calls itself until the program exits.
  - `resolveModelExpr`, which implements ϕ as defined in the previous chapter, resolving an expression in the Hominy query language into a graphloc. This function also takes an environment of bound names as input, in order to resolve @-expressions.
  - `resolveGraphLoc`, which implements model_at, resolving a graphloc into a full model.

– `geoFormulas`, which reads and parses a geometric theory from a file.

5.2 Handling Graphlocs

When a user is visiting a particular model, in order for them to navigate back and forth in the stream and downwards (i.e., removing previously applied augmentations), it is necessary to keep track not only of which model the user is visiting, but also of how they got there. This is why a graphloc stores the entire sequence of augmentations that have been applied, and the index corresponding to each one.

Parsec, a monadic parser combinator library for Haskell [11], was used to implement a parser for the Hominy query language, as used in the command line interface. Parsec was also used to implement a serializer that encodes a graphloc as a series of characters legal in URL query parameters. This allows the current graphloc to be stored in the query string of the URL for the web interface, obviating the need for cookies or server-side storage.

5.3 Implementing the Web Interface

The Web interface was implemented using Happstack [5]; specifically, happstack-lite was used to handle basic request routing, and the Blaze HTML5 combinator library (included as part of Happstack) was used to render the HTML content of each page. Bootstrap [17] was used for the front-end. Models and theories were embedded in the HTML in LaTeX format; the MathJax library [2] was used to render these on the client side, in a way which offered cross-browser compatibility.

The Hominy Web service recognizes the following URLs:

- `/`: The homepage, which offers a text box in which the user enters an initial theory.

- `/find`: The page corresponding to a particular graphloc. Accepts a single query parameter, `q`, which is expected to contain a graphloc serialized by the URL-safe algorithm implemented in the WebParse module. Redirects to `/error` if this parameter does not contain a syntactically valid serialized graphloc. Otherwise, displays the model found at that graphloc through model_at, and provides buttons for operations on that graphloc, including a text box for adding observations. These are simply hyperlinks to other `/find` pages, except for the `add` operation.

- `/add`: The target of the form where additional observations can be entered. Accepts two query parameters, `loc`, which is expected to contain a serialized graphloc, and `constraint`, which is expected to contain a quantified expression as specified by the concrete syntax for the Hominy command language. This expression is parsed and the resulting observation is added to the graphloc through the `add` operation; the page then redirects to the `/find` page for the new graphloc. The primary purpose of this URL is to handle the parsing for the concrete syntax of observations.

- `/error`: Displays an error message. The user is redirected here if a syntax error occurs.

Chapter 6

Conclusions

The work presented here transforms Hominy from a batch-mode-only model finding utility into an interactive one. This interactivity works by imposing a graph structure onto the space of all finite models of a given geometric theory, then providing a command language which the user can use to navigate this graph. In this way, the user can explore the logical consequences of applying a particular augmentation to an existing model. We have developed a command-line interface for Hominy with a read-eval-print loop, and a Web-based graphical interface.

6.1 Future Work

Potential future improvements to the interactive scenario generation facilities of Hominy could include the following:

- The Web interface is currently fairly primitive and could be substantially improved with respect to user experience. In particular, it would be useful to provide graphical representations of models instead of only offering relatively user-unfriendly lists of facts. Graphical representations of logical models are an area of ongoing research.

- Visit Pataranutaporn has done work in tracing the provenance of a fact or element in a model, so as to answer the question of what part of the theory logically requires that fact or element to be present in the model. If this were integrated into the interactive facilities of Hominy, users could query the provenance of facts or elements of models.

- Users might wish to attach names to model elements and refer to them in augmentations by these names instead of by not-particularly-meaningful numbers.

- Users might wish to see the consequences of removing a fact or element from a model, similarly to how they can currently see the consequences of adding a fact or element. The major work to be done here is in determining what the semantics of such an operation would be, as it is not immediately clear.

Appendix A

Code Listing

```
-- hominy.hs

{-| This module is the primary interface to Hominy!
-}


module Main where
import System.Environment
import System.Console.GetOpt
import System.Console.Readline
import System.Exit (exitWith, ExitCode (..))
import System.IO (hPutStrLn, stderr, hFlush, stdout)
import Text.Read (readMaybe)

import Control.Monad
import Control.Applicative

import Data.Maybe
import Data.List
import qualified Data.Map as Map
```

```
import Chase.Formula.SyntaxGeo (Theory, Sequent, Term(..), Elem(..),
    parseSequent,
                                Formula(..), parseCommand, Command(..),
                                    ModelExpr(..),
                                ModelOperation(..), Atom(..))
import Chase.Utils.Utils (isRealLine, isNonEmptyLine)
import Chase.Tools.Config
import Chase.Tools.FolToGeo
import qualified Chase.Problem.Model as Model
import Chase.Chase (chase, chase', chaseWithModel, runChase,
    runChaseWithProblem, deduceForFrame)
import Chase.Problem.Observation
import Chase.Problem.Operations
import Chase.Problem.Provenance
import Chase.Problem.Structures

import qualified Codec.TPTP as TP
import Chase.TPTP.TPTPToGeo as T2G

-- Preexisting option parsing code omitted

main :: IO ()
main = do
  -- get the arguments
  args <- getArgs

  -- Parse options, getting a list of option actions
  let (actions, nonOptions, errors) = getOpt RequireOrder options args

  -- Here we thread startOptions through all supplied option actions
```

```
  config <- foldl (>>=) (return defaultConfig) actions
  modelLoop config Map.empty Nothing

data GraphLoc = GraphLoc Theory Int [(Maybe Formula, Int)]

modelLoop :: Config -> Map.Map String GraphLoc -> Maybe GraphLoc -> IO
    ()
modelLoop config bindings lastLoc = do
  let loop = modelLoop config
      continue = loop bindings lastLoc
  userLine <- readline "> "
  case userLine of
    Nothing -> return ()
    Just userInput -> if Chase.Utils.Utils.isNonEmptyLine userInput
       then
       addHistory userInput >> case (parseCommand userInput) of
        Nothing -> putStrLn "Syntax error." >> continue
        Just cmd -> case cmd of
          Display expr -> do
            maybeLoc <- resolveModelExpr expr bindings lastLoc
            case maybeLoc of
              Nothing -> putStrLn "Invalid expression." >> continue
              Just loc -> case resolveGraphLoc config loc of
                Nothing -> putStrLn "Model not found." >> continue
                Just (prob,_) -> putStrLn (show $ problemModel prob) >>
                     loop bindings maybeLoc
          Assign var expr -> do
            maybeLoc <- resolveModelExpr expr bindings lastLoc
            case maybeLoc of
              Nothing -> putStrLn "Invalid expression." >> continue
              Just loc -> loop (Map.insert var loc bindings) lastLoc
```

```
          Exit -> return ()
        else continue

resolveModelExpr :: ModelExpr -> Map.Map String GraphLoc -> Maybe
    GraphLoc -> IO (Maybe GraphLoc)
resolveModelExpr expr bindings lastLoc = case expr of
  ThyLiteral thy -> return $ Just $ GraphLoc thy 0 []
  LoadFromFile filename -> do
    maybeThy <- geoFormulas filename
    return $ case maybeThy of
      Nothing -> Nothing
      Just thy -> Just $ GraphLoc thy 0 []
  ApplyOp preExpr op -> do
    maybeLoc <- resolveModelExpr preExpr bindings lastLoc
    return $ case maybeLoc of
      Nothing -> Nothing
      Just (GraphLoc thy initialIndex steps) -> case op of
        AddConstraint aug -> Just $ GraphLoc thy initialIndex (steps ++
            [(Just aug,0)])
        NewElement -> Just $ GraphLoc thy initialIndex (steps ++ [(
           Nothing,0)])
        RemoveConstraint -> case steps of
          [] -> Nothing
          _ -> Just $ GraphLoc thy initialIndex $ init steps
        NextModel -> Just $ case steps of
          [] -> GraphLoc thy (succ initialIndex) []
          _ -> let (aug,lastIndex) = last steps in
            GraphLoc thy initialIndex (init steps ++ [(aug,succ
               lastIndex)])
        PreviousModel -> case steps of
          [] -> case initialIndex of
```

```
            0 -> Nothing
            _ -> Just $ GraphLoc thy (pred initialIndex) []
          _ -> let (aug,lastIndex) = last steps in
            case lastIndex of
              0 -> Nothing
              _ -> Just $ GraphLoc thy initialIndex (init steps ++ [(
                  aug,pred lastIndex)])
        FirstModel -> case steps of
          [] -> Just $ GraphLoc thy 0 []
          _ -> let (aug,_) = last steps in
            Just $ GraphLoc thy initialIndex (init steps ++ [(aug,0)])
        Origin -> Just $ GraphLoc thy 0 []
  LastResult -> return lastLoc
  ModelVar var -> return $ Map.lookup var bindings

resolveGraphLoc :: Config -> GraphLoc -> Maybe (Problem, FrameMap)
resolveGraphLoc config (GraphLoc thy initialIndex steps) =
  case steps of
    [] -> let (frms,initialProblem) = buildProblem thy
              stream = runChase config Nothing frms initialProblem in
          if length stream > initialIndex then Just ((stream !!
              initialIndex),frms) else Nothing
    _ -> case resolveGraphLoc config (GraphLoc thy initialIndex (init
        steps)) of
      Nothing -> Nothing
      Just (prob@Problem {problemModel = oldModel, problemLastConstant
          = oldConst},frms) ->
        let (aug,lastIndex) = last steps
            (preDeducedObs,_) = processHead $ case aug of
              Just fmla -> fmla
              Nothing -> Exists "x" $ Atm $ R "=" [Var "x", Var "x"]
```

```
            ([postDeducedObs], intermediateConst) = case aug of
              Just _ -> deduceForFrame oldConst oldModel preDeducedObs
              Nothing -> deduceForFrame (succ oldConst) oldModel
                  preDeducedObs
            (newModel, _, newConst) = Model.add oldModel oldConst
                postDeducedObs UserProv
            stream = runChaseWithProblem config frms prob {problemModel
                = newModel, problemLastConstant = newConst} in
        if length stream > lastIndex then Just ((stream !! lastIndex),
            frms) else Nothing

geoFormulas :: String -> IO (Maybe Theory)
geoFormulas fName = do
  src <- readFile fName
  let inputLines = lines src
      realLines  = filter isRealLine inputLines
      inputFmlas = mapM (parseFolToSequent False) realLines
  return $ concat <$> inputFmlas

-- Chase.hs

module Chase where

import Control.Applicative ((<$>))
import qualified Data.Map as Map
import qualified Text.Parsec as Parsec

import qualified Datatypes
import qualified Chase.Chase as Chase
import qualified Chase.Formula.SyntaxGeo as SyntaxGeo
import qualified Chase.Problem.Model as Model
```

```
import qualified Chase.Problem.Observation as Observation
import qualified Chase.Problem.Operations as Operations
import qualified Chase.Problem.Provenance as Provenance
import qualified Chase.Problem.Structures as Structures
import qualified Chase.Problem.RelAlg.RelAlg as RelAlg
import qualified Chase.RelAlg.DB as DB
import qualified Chase.Tools.Config as Config
import Utility ((!!!))
import qualified WebParse

import Datatypes

chase :: Datatypes.GraphLoc -> Maybe Datatypes.Model
chase (Datatypes.GraphLoc theory index steps) =
  let (frms,initialProblem) =
        Operations.buildProblem $ map chasifySequent theory
      nextStep row stepIndex stepsLeft = do
        currentNode <- row !!! stepIndex
        case stepsLeft of
          [] -> return currentNode
          (constraint,nextIndex):restSteps ->
            let (newModel,_,newConst) =
                  (Model.add (Structures.problemModel currentNode)
                   (Structures.problemLastConstant currentNode)
                   [chasifyConstraint constraint] Provenance.UserProv)
                      in
              (nextStep
               (Chase.runChaseWithProblem Config.defaultConfig frms
                currentNode {Structures.problemModel = newModel,
                             Structures.problemLastConstant = newConst})
               nextIndex restSteps) in
```

```
  dechasifyModel <$>
  nextStep (Chase.runChase Config.defaultConfig Nothing frms
     initialProblem)
  index steps

chasifySequent :: Datatypes.Sequent -> SyntaxGeo.Sequent
chasifySequent (Datatypes.Sequent premises consequents) = SyntaxGeo.
    Sequent {
  SyntaxGeo.sequentBody = chasifyConj chasifyAtomicFormula premises,
  SyntaxGeo.sequentHead = chasifyDisj chasifyExQuant consequents
  }

chasifyDisj :: (a -> SyntaxGeo.Formula) -> [a] -> SyntaxGeo.Formula
chasifyDisj f = chasifyConnective f SyntaxGeo.Or SyntaxGeo.Fls

chasifyConj :: (a -> SyntaxGeo.Formula) -> [a] -> SyntaxGeo.Formula
chasifyConj f = chasifyConnective f SyntaxGeo.And SyntaxGeo.Tru

chasifyConnective :: (a -> SyntaxGeo.Formula)
                  -> (SyntaxGeo.Formula -> SyntaxGeo.Formula
                      -> SyntaxGeo.Formula)
                  -> SyntaxGeo.Formula -> [a] -> SyntaxGeo.Formula
chasifyConnective _ _ base [] = base
chasifyConnective func connective _ xs = foldr1 connective $ map func
    xs

chasifyExQuant :: ([Datatypes.VariableSymbol], [Datatypes.TAtom])
                  -> SyntaxGeo.Formula
chasifyExQuant (vars,atom) =
  foldr SyntaxGeo.Exists (chasifyConj chasifyAtomicFormula atom) $ do
    Datatypes.VariableSymbol symname <- vars
```

```
    return symname

chasifyAtomicFormula :: Datatypes.TAtom -> SyntaxGeo.Formula
chasifyAtomicFormula = SyntaxGeo.Atm . chasifyTAtom

chasifyTAtom :: Datatypes.TAtom -> SyntaxGeo.Atom
chasifyTAtom atom = case atom of
  Datatypes.TPredicate (Datatypes.PredicateSymbol symname) args ->
    SyntaxGeo.R symname $ map chasifyTTerm args
  Datatypes.TEquality term1 term2 ->
    SyntaxGeo.R "=" [chasifyTTerm term1,chasifyTTerm term2]

chasifyTTerm :: Datatypes.TTerm -> SyntaxGeo.Term
chasifyTTerm term = case term of
  Datatypes.TVariable (Datatypes.VariableSymbol symname) ->
    SyntaxGeo.Var symname
  Datatypes.TFunction (Datatypes.FunctionSymbol symname) args ->
    SyntaxGeo.Fn symname $ map chasifyTTerm args

chasifyConstraint :: Datatypes.MAtom -> Observation.Obs
chasifyConstraint constraint = case constraint of
  Datatypes.MPredicate (Datatypes.PredicateSymbol symname) args ->
    Observation.Fct $ SyntaxGeo.R symname $ map chasifyMTerm args
  Datatypes.MEquality term1 term2 ->
    Observation.Eql (chasifyMTerm term1) (chasifyMTerm term2)

chasifyMTerm :: Datatypes.MTerm -> SyntaxGeo.Term
chasifyMTerm term = case term of
  Datatypes.MVariable (Datatypes.VariableSymbol symname) ->
    SyntaxGeo.Var symname
  Datatypes.MFunction (Datatypes.FunctionSymbol symname) args ->
```

```
    SyntaxGeo.Fn symname $ map chasifyMTerm args
  Datatypes.ModelElement (Datatypes.ModelElementSymbol symnum) ->
    SyntaxGeo.Elm $ SyntaxGeo.Elem $ "e#" ++ show symnum

dechasifyModel :: Structures.Problem -> Datatypes.Model
dechasifyModel Structures.Problem {Structures.problemModel = mdl} =
  Datatypes.Model (map dechasifyModelElement $ Model.modelDomain mdl) $
      do
    (ref,tbl) <- Map.assocs $ Model.modelTables mdl
    let rels = map (map dechasifyModelElement) $ DB.toList tbl
    case ref of
      RelAlg.ConTable symname -> let [[constValue]] = rels in
        return $
        Datatypes.FunctionFact (Datatypes.FunctionSymbol symname) []
          constValue
      RelAlg.RelTable ('@':_) -> []
      RelAlg.RelTable symname -> Datatypes.PredicateFact (Datatypes.
        PredicateSymbol symname) <$> rels
      RelAlg.FunTable symname -> do
        row <- rels
        return $ Datatypes.FunctionFact (Datatypes.FunctionSymbol
          symname) (init row) (last row)
      RelAlg.DomTable -> []

dechasifyModelElement :: SyntaxGeo.Elem -> Datatypes.ModelElementSymbol
dechasifyModelElement (SyntaxGeo.Elem ('e':'#':symnum)) =
  Datatypes.ModelElementSymbol $ read symnum

-- Datatypes.hs

module Datatypes where
```

```
import Data.List (intercalate)

data Sequent = Sequent [TAtom] [([VariableSymbol], [TAtom])]
instance Show Sequent where
  show (Sequent premises consequents) =
    (if null premises then "" else
       intercalate " ∧ " (map show premises) ++ " ⊢ ") ++
    if null consequents then "⊥" else intercalate " ∨ " $ do
      (vars,atoms) <- consequents
      return $
        (if null vars then "" else
           "∃ " ++ intercalate ", " (map show vars) ++ ". ") ++
        (if null atoms then "⊤" else intercalate " ∧ " $ map show atoms
            )

data TAtom = TPredicate PredicateSymbol [TTerm] | TEquality TTerm TTerm
instance Show TAtom where
  show (TPredicate symbol args) =
    show symbol ++ "(" ++ intercalate ", " (map show args) ++ ")"
  show (TEquality term1 term2) = show term1 ++ " = " ++ show term2

data TTerm = TVariable VariableSymbol | TFunction FunctionSymbol [TTerm
    ]
instance Show TTerm where
  show (TVariable symbol) = show symbol
  show (TFunction symbol args) =
    (show symbol ++
     (if null args then "" else "(" ++ intercalate ", " (map show args)
         ++ ")"))
```

```
data MAtom = MPredicate PredicateSymbol [MTerm] | MEquality MTerm MTerm
instance Show MAtom where
  show (MPredicate symbol args) =
    show symbol ++ "(" ++ intercalate ", " (map show args) ++ ")"
  show (MEquality term1 term2) = show term1 ++ " = " ++ show term2

data MTerm = MVariable VariableSymbol | MFunction FunctionSymbol [MTerm
    ] | ModelElement ModelElementSymbol
instance Show MTerm where
  show (MVariable symbol) = show symbol
  show (MFunction symbol args) =
    (show symbol ++
     (if null args then "" else "(" ++ intercalate ", " (map show args)
         ++ ")"))
  show (ModelElement symbol) = show symbol

data Model = Model [ModelElementSymbol] [Fact] deriving Show

data Fact = PredicateFact PredicateSymbol [ModelElementSymbol] |
    FunctionFact FunctionSymbol [ModelElementSymbol] ModelElementSymbol
instance Show Fact where
  show (PredicateFact symbol args) =
    show symbol ++ "(" ++ intercalate ", " (map show args) ++ ")"
  show (FunctionFact symbol args value) =
    (show symbol ++
     (if null args then "" else
        "(" ++ intercalate ", " (map show args) ++ ")") ++
     " = " ++ show value)

data GraphLoc = GraphLoc [Sequent] Int [(MAtom, Int)] deriving Show
```

```
newtype VariableSymbol = VariableSymbol String
instance Show VariableSymbol where show (VariableSymbol symname) =
    symname

newtype PredicateSymbol = PredicateSymbol String
instance Show PredicateSymbol where show (PredicateSymbol symname) =
    symname

newtype FunctionSymbol = FunctionSymbol String
instance Show FunctionSymbol where show (FunctionSymbol symname) =
    symname

newtype ModelElementSymbol = ModelElementSymbol Int
instance Show ModelElementSymbol where
  show (ModelElementSymbol symnum) = "#" ++ show symnum

add :: GraphLoc -> MAtom -> GraphLoc
add (GraphLoc theory startingIndex steps) constraint =
  GraphLoc theory startingIndex (steps ++ [(constraint,0)])

undoConstraint :: GraphLoc -> Maybe GraphLoc
undoConstraint (GraphLoc _ _ []) = Nothing
undoConstraint (GraphLoc theory startingIndex steps) =
  Just $ GraphLoc theory startingIndex $ init steps

previousLoc :: GraphLoc -> Maybe GraphLoc
previousLoc (GraphLoc _ 0 []) = Nothing
previousLoc (GraphLoc theory startingIndex []) =
  Just $ GraphLoc theory (pred startingIndex) []
previousLoc (GraphLoc theory startingIndex steps) =
  let (constraint,index) = last steps in
```

```
  if index == 0 then Nothing else
    (Just $
     GraphLoc theory startingIndex (init steps ++ [(constraint,pred
        index)]))

nextLoc :: GraphLoc -> GraphLoc
nextLoc (GraphLoc theory startingIndex []) =
  GraphLoc theory (succ startingIndex) []
nextLoc (GraphLoc theory startingIndex steps) =
  let (constraint,index) = last steps in
  GraphLoc theory startingIndex (init steps ++ [(constraint,succ index)
     ])

origin :: GraphLoc -> GraphLoc
origin (GraphLoc theory _ _) = GraphLoc theory 0 []

-- Utility.hs

module Utility ((!!!)) where

(!!!) :: [a] -> Int -> Maybe a
xs !!! index = if index < 0 then Nothing else getIndex xs index

getIndex :: [a] -> Int -> Maybe a
getIndex xs index = case xs of
  [] -> Nothing
  first:rest
    | index == 0 -> Just first
    | otherwise -> getIndex rest $ pred index

-- WebParse.hs
```

```
{-# LANGUAGE OverloadedStrings #-}

module WebParse where

import Control.Applicative ((<$>))
import Data.List (intercalate)
import qualified Text.Parsec as P
import Text.Parsec ((<|>))
import Text.Parsec.Text.Lazy (Parser)

import qualified Datatypes

pGraphLoc :: Parser Datatypes.GraphLoc
pGraphLoc = do
  theory <- P.sepBy1 pSequent $ P.char ';'
  P.char '!'
  initialIndex <- pInt
  steps <- P.many $ do
    P.char ';'
    constraint <- pMAtom
    P.char '!'
    index <- pInt
    return (constraint,index)
  return $ Datatypes.GraphLoc theory initialIndex steps

pSequent :: Parser Datatypes.Sequent
pSequent = do
  premises <- P.sepBy pTAtom $ P.char '*'
  P.char ':'
  consequents <- flip P.sepBy (P.char '/') $ do
```

```
    existentials <- P.option [] $ P.between (P.char '?') (P.char '.') $
        P.sepBy1 pVariableSymbol $ P.char ','
    atoms <- P.sepBy pTAtom $ P.char '*'
    return (existentials,atoms)
  return $ Datatypes.Sequent premises consequents

pTAtom :: Parser Datatypes.TAtom
pTAtom = pTPredicate <|> pTEquality

pTPredicate :: Parser Datatypes.TAtom
pTPredicate = do
  symbol <- pPredicateSymbol
  args <- P.between (P.char '(') (P.char ')') $ P.sepBy1 pTTerm $ P.
     char ','
  return $ Datatypes.TPredicate symbol args

pTEquality :: Parser Datatypes.TAtom
pTEquality = do
  term1 <- pTTerm
  P.char '='
  term2 <- pTTerm
  return $ Datatypes.TEquality term1 term2

pTTerm :: Parser Datatypes.TTerm
pTTerm = pTVariable <|> pTFunction

pTVariable :: Parser Datatypes.TTerm
pTVariable = do
  P.char '.'
  symbol <- pVariableSymbol
  return $ Datatypes.TVariable symbol
```

```
pTFunction :: Parser Datatypes.TTerm
pTFunction = do
  P.char '@'
  symbol <- pFunctionSymbol
  args <- P.between (P.char '(') (P.char ')') $ P.sepBy1 pTTerm $ P.
      char ','
  return $ Datatypes.TFunction symbol args

pMAtom :: Parser Datatypes.MAtom
pMAtom = pMPredicate <|> pMEquality

pMPredicate :: Parser Datatypes.MAtom
pMPredicate = do
  symbol <- pPredicateSymbol
  args <- P.between (P.char '(') (P.char ')') $ P.sepBy pMTerm $ P.char
      ','
  return $ Datatypes.MPredicate symbol args

pMEquality :: Parser Datatypes.MAtom
pMEquality = do
  term1 <- pMTerm
  P.char '='
  term2 <- pMTerm
  return $ Datatypes.MEquality term1 term2

pMTerm :: Parser Datatypes.MTerm
pMTerm = pMVariable <|> pMFunction <|> pModelElement

pMVariable :: Parser Datatypes.MTerm
pMVariable = do
```

```
  P.char '.'
  symbol <- pVariableSymbol
  return $ Datatypes.MVariable symbol

pMFunction :: Parser Datatypes.MTerm
pMFunction = do
  P.char '@'
  symbol <- pFunctionSymbol
  args <- P.between (P.char '(') (P.char ')') $ P.sepBy pMTerm $ P.char
      ','
  return $ Datatypes.MFunction symbol args

pModelElement :: Parser Datatypes.MTerm
pModelElement = Datatypes.ModelElement <$> pModelElementSymbol

pVariableSymbol :: Parser Datatypes.VariableSymbol
pVariableSymbol = Datatypes.VariableSymbol <$> P.many1 P.alphaNum

pPredicateSymbol :: Parser Datatypes.PredicateSymbol
pPredicateSymbol = Datatypes.PredicateSymbol <$> P.many1 P.alphaNum

pFunctionSymbol :: Parser Datatypes.FunctionSymbol
pFunctionSymbol = Datatypes.FunctionSymbol <$> P.many1 P.alphaNum

pModelElementSymbol :: Parser Datatypes.ModelElementSymbol
pModelElementSymbol = do
  P.char '$'
  symnum <- pInt
  return $ Datatypes.ModelElementSymbol symnum

pInt :: Parser Int
```

```
pInt = read <$> P.many1 P.digit

encodeGraphLoc :: Datatypes.GraphLoc -> String
encodeGraphLoc (Datatypes.GraphLoc theory initialIndex steps) =
  intercalate ";" (map encodeSequent theory) ++ "!" ++ show
     initialIndex ++
  concat [";" ++ encodeMAtom constraint ++ "!" ++ show index |
          (constraint,index) <- steps]

encodeSequent :: Datatypes.Sequent -> String
encodeSequent (Datatypes.Sequent premises consequents) =
  intercalate "*" (map encodeTAtom premises) ++ ":" ++
  intercalate "/"
  [(if null existentials then "" else
      "?" ++ intercalate "," (map encodeVariableSymbol existentials) ++
          ".") ++
   intercalate "*" (map encodeTAtom atoms) |
   (existentials,atoms) <- consequents]

encodeTAtom :: Datatypes.TAtom -> String
encodeTAtom (Datatypes.TPredicate symbol args) =
  encodePredicateSymbol symbol ++ "(" ++
  intercalate "," (map encodeTTerm args) ++ ")"
encodeTAtom (Datatypes.TEquality term1 term2) =
  encodeTTerm term1 ++ "=" ++ encodeTTerm term2

encodeTTerm :: Datatypes.TTerm -> String
encodeTTerm (Datatypes.TVariable symbol) = "." ++ encodeVariableSymbol
   symbol
encodeTTerm (Datatypes.TFunction symbol args) =
  "@" ++ encodeFunctionSymbol symbol ++ "(" ++
```

```
  intercalate "," (map encodeTTerm args) ++ ")"

encodeMAtom :: Datatypes.MAtom -> String
encodeMAtom (Datatypes.MPredicate symbol args) =
  encodePredicateSymbol symbol ++ "(" ++
  intercalate "," (map encodeMTerm args) ++ ")"
encodeMAtom (Datatypes.MEquality term1 term2) =
  encodeMTerm term1 ++ "=" ++ encodeMTerm term2

encodeMTerm :: Datatypes.MTerm -> String
encodeMTerm (Datatypes.MVariable symbol) = "." ++ encodeVariableSymbol
    symbol
encodeMTerm (Datatypes.MFunction symbol args) =
  "@" ++ encodeFunctionSymbol symbol ++ "(" ++
  intercalate "," (map encodeMTerm args) ++ ")"
encodeMTerm (Datatypes.ModelElement symbol) = encodeModelElementSymbol
    symbol

encodeVariableSymbol :: Datatypes.VariableSymbol -> String
encodeVariableSymbol (Datatypes.VariableSymbol symname) = symname

encodePredicateSymbol :: Datatypes.PredicateSymbol -> String
encodePredicateSymbol (Datatypes.PredicateSymbol symname) = symname

encodeFunctionSymbol :: Datatypes.FunctionSymbol -> String
encodeFunctionSymbol (Datatypes.FunctionSymbol symname) = symname

encodeModelElementSymbol :: Datatypes.ModelElementSymbol -> String
encodeModelElementSymbol (Datatypes.ModelElementSymbol symnum) =
  "$" ++ show symnum
```

```
-- WebServ.hs

{-# LANGUAGE OverloadedStrings #-}

module Main where

import Control.Applicative ((<$>))
import Data.List (intercalate)

import Happstack.Lite
  (ServerPart, Response, serve, msum, dir, ok, toResponse, lookText, seeOther,
   setResponseCode)
import qualified Text.Blaze.Html5 as H
import Text.Blaze.Html5 ((!))
import qualified Text.Blaze.Html5.Attributes as A
import Text.Parsec (parse)

import Chase (chase)
import qualified Datatypes
import WebParse (pGraphLoc, pMAtom, encodeGraphLoc)

main :: IO ()
main = serve Nothing awi

awi :: ServerPart Response
awi = msum [dir "find" $ find, dir "add" $ add, dir "error" $ error_, home
    ]

find :: ServerPart Response
find = do
  locEncoded <- lookText "q"
```

```
case parse pGraphLoc "" locEncoded of
  Left _ -> seeOther ("/error" :: String) $ toResponse ("" :: String)
  Right loc -> ok $ template $ do
    case chase loc of
      Just (Datatypes.Model domain facts) -> do
        H.h2 "Model found."
        H.h3 "Domain:"
        H.p $ H.toHtml $ "{" ++ intercalate ", " (map show domain) ++
            "}"
        H.h3 "Facts:"
        H.ul $ foldl1 (>>) $ map (H.li . H.toHtml . show) facts
      Nothing -> H.h3 "No model found."
    H.h2 "Navigation"
    H.ul $ do
      case Datatypes.previousLoc loc of
        Just prevLoc ->
          H.li $ H.a ! A.href (H.toValue $ locURL prevLoc) $ "
              Previous"
        Nothing -> return undefined
      H.li $ H.a ! A.href (H.toValue $ locURL $ Datatypes.nextLoc loc
          ) $ "Next"
      case Datatypes.undoConstraint loc of
        Just upLoc ->
          H.li $ H.a ! A.href (H.toValue $ locURL upLoc) $ "Undo
              Constraint"
        Nothing -> return undefined
      H.li $ H.a ! A.href (H.toValue $ locURL $ Datatypes.origin loc)
          $ "Origin"
    H.form ! A.action "/add" $ do
      H.input ! A.type_ "hidden" ! A.name "loc" ! A.value (H.toValue
          $ encodeGraphLoc loc)
```

```
        H.label $ do
          "Add constraint:"
          H.input ! A.type_ "text" ! A.name "constraint"
        H.input ! A.type_ "submit" ! A.value "Add"

add :: ServerPart Response
add = do
  locEncoded <- lookText "loc"
  constraintEncoded <- lookText "constraint"
  let result = do
        loc <- parse pGraphLoc "" locEncoded
        constraint <- parse pMAtom "" constraintEncoded
        return $ Datatypes.add loc constraint
  case result of
    Left _ -> seeOther ("/error" :: String) $ toResponse ("" :: String)
    Right newLoc -> seeOther (locURL newLoc) $ toResponse ("" :: String
        )

error_ :: ServerPart Response
error_ = do
  setResponseCode 400
  return $ template $ do
    H.h2 "Syntax Error"
    H.p "An error occurred trying to parse the request."

home :: ServerPart Response
home = ok $ template $ do
  H.form ! A.action "/find" $ do
    H.label $ do
      "Enter theory:"
      H.input ! A.type_ "text" ! A.name "q"
```

```
    H.input ! A.type_ "submit" ! A.value "Find"

template :: H.Html -> Response
template body = toResponse $ H.docTypeHtml $ do
  H.head $ do
    --H.meta ! A.charset "UTF-8"
    H.title "Hominy Scenario Generator"
    H.link ! A.href "http://netdna.bootstrapcdn.com/bootstrap/3.1.1/css
        /bootstrap.min.css" ! A.rel "stylesheet"
    H.link ! A.href "http://netdna.bootstrapcdn.com/bootstrap/3.1.1/css
        /bootstrap-theme.min.css" ! A.rel "stylesheet"
    H.script ! A.src "http://ajax.googleapis.com/ajax/libs/jquery
        /1.11.0/jquery.min.js"
    H.script ! A.src "http://netdna.bootstrapcdn.com/bootstrap/3.1.1/js
        /bootstrap.min.js"
  H.body $ do
    H.h1 "Hominy Scenario Generator"
    body
    H.hr
    H.footer $ H.a ! A.href "/" $ "Home"

locURL :: Datatypes.GraphLoc -> String
locURL loc = "/find?q=" ++ encodeGraphLoc loc

-- Chase/Formula/SyntaxGeo.hs

module Chase.Formula.SyntaxGeo where

import qualified Data.Map as Map
import Data.Map(Map)
import Data.Maybe
```

```
import Control.Exception -- for assert
import Debug.Trace
import Data.List(intercalate)

import Control.Applicative hiding (many)
import Text.ParserCombinators.Parsec hiding ( (<|>) )
import Text.Parsec.Token ( TokenParser )
import qualified Text.Parsec.Token as Token
import Text.ParserCombinators.Parsec.Language ( haskellStyle )
import qualified Text.ParserCombinators.Parsec.Expr as Expr
--

import Data.Set ((\\))

-- Preexisting parsing code omitted

-- Query language stuff

data Command = Display ModelExpr | Assign String ModelExpr | Exit

data ModelExpr = ThyLiteral Theory | LoadFromFile String
               | ApplyOp ModelExpr ModelOperation | LastResult |
                  ModelVar String

data ModelOperation = AddConstraint Formula | NewElement |
    RemoveConstraint | NextModel
                     | PreviousModel | FirstModel | Origin

pCommand :: Parser Command
pCommand = pExit +++ pImplicitAssign +++ pAssign +++ pDisplay
```

```
pDisplay :: Parser Command
pDisplay = Display <$> pModelExpr

pImplicitAssign :: Parser Command
pImplicitAssign = do
  symbol "save"
  var <- identifier
  return $ Assign var LastResult

pAssign :: Parser Command
pAssign = do
  var <- identifier
  symbol ":="
  expr <- pModelExpr
  return $ Assign var expr

pExit :: Parser Command
pExit = symbol "exit" >> return Exit

pModelExpr :: Parser ModelExpr
pModelExpr = (pExplicitModelExpr <|> pImpliedOp) +++ pImpliedAdd

pExplicitModelExpr :: Parser ModelExpr
pExplicitModelExpr = do
  base <- pThyLiteral <|> pLoadFromFile <|> pLastResult <|> pModelVar
  ops <- many $ dot >> pModelOperation
  return $ foldl ApplyOp base ops

pImpliedOp :: Parser ModelExpr
pImpliedOp = foldl ApplyOp LastResult <$> sepBy1 pModelOperation dot
```

```
pImpliedAdd :: Parser ModelExpr
pImpliedAdd = ApplyOp LastResult <$> AddConstraint <$> pFmla

pThyLiteral :: Parser ModelExpr
pThyLiteral = ThyLiteral <$> brackets (semiSep1 pSequent)

pLoadFromFile :: Parser ModelExpr
pLoadFromFile = LoadFromFile <$> stringLiteral

pLastResult :: Parser ModelExpr
pLastResult = symbol "~" >> return LastResult

pModelVar :: Parser ModelExpr
pModelVar = do
  symbol "@" <|> symbol "load"
  ModelVar <$> identifier

pModelOperation :: Parser ModelOperation
pModelOperation = (pNewElement +++ pRemoveConstraint +++ pOrigin) <|>
    pNextModel <|> pPreviousModel <|> pFirst <|> pAddConstraint

pAddConstraint :: Parser ModelOperation
pAddConstraint = do
  symbol "add"
  AddConstraint <$> (parens pFmla <|> pFmla)

pNewElement :: Parser ModelOperation
pNewElement = symbol "new_element" >> return NewElement

pRemoveConstraint :: Parser ModelOperation
pRemoveConstraint = symbol "remove_last" >> return RemoveConstraint
```

```
pNextModel :: Parser ModelOperation
pNextModel = symbol "next" >> return NextModel

pPreviousModel :: Parser ModelOperation
pPreviousModel = symbol "previous" >> return PreviousModel

pFirst :: Parser ModelOperation
pFirst = symbol "first" >> return FirstModel

pOrigin :: Parser ModelOperation
pOrigin = symbol "remove_all" >> return Origin

parseCommand :: String -> Maybe Command
parseCommand input = case (parse pCommand "parsing Command" input) of
  Left _ -> Nothing
  Right val -> Just val
```

-- Chase/Formula/UserSyntax.hs

```
module Chase.Formula.UserSyntax where

import Control.Applicative
import Text.ParserCombinators.Parsec hiding ( (<|>) )
import Text.Parsec.Token ( TokenParser )
import qualified Text.Parsec.Token as Token
import Text.ParserCombinators.Parsec.Language ( haskellStyle )
import qualified Text.ParserCombinators.Parsec.Expr as Expr

import Chase.Formula.SyntaxGeo
import Chase.Problem.Observation
```

```
-- These are for interactive mode

natural = Token.natural lexer

parseUserFact :: String -> Obs
parseUserFact input = case parse pUserFact "parsing user fact" input of
  Left err -> error (show err)
  Right val -> val

pUserFact :: Parser Obs
pUserFact = pUserEquality <|> pUserAtom

pUserEquality :: Parser Obs
pUserEquality = do
  t1 <- pUserTerm
  symbol "="
  t2 <- pUserTerm
  return $ Eql t1 t2

pUserAtom :: Parser Obs
pUserAtom = do
  name <- identifier
  termList <- pUserTermList <|> return []
  return $ Fct $ R name termList

pUserTermList :: Parser [Term]
pUserTermList = parens $ commaSep pUserTerm

pUserTerm :: Parser Term
pUserTerm = pUserElement <|> do
```

```
  name <- identifier
  pUserFunction name <|> pUserConstant name <?> "user term"

pUserElement :: Parser Term
pUserElement = do
  symbol "e#"
  n <- natural
  return $ Elm $ Elem $ "e#" ++ show n

pUserFunction :: String -> Parser Term
pUserFunction name = pTermList >>= return . Fn name

pUserConstant :: String -> Parser Term
pUserConstant name = return $ Var name

-- Chase/Problem/IModel.hs

-- Only two functions in this module are original to this project.

prettyModel2 :: Model -> String
prettyModel2 mdl@(Model tbls _) =
  "Domain: {" ++ intercalate ", " (map prettyElement2 (modelDomain mdl)
     ) ++ "}\nFacts: " ++
  (intercalate "; " $ do
    (ref,tbl) <- Map.assocs $ modelTables mdl
    let rels = DB.toList tbl
    case ref of
      ConTable symname -> let [[constValue]] = rels in
        return $ symname ++ " = " ++ prettyElement2 constValue
      RelTable ('@':_) -> []
      RelTable symname -> do
```

```
        row <- rels
        return $ symname ++ "(" ++ intercalate ", " (map prettyElement2
            row) ++ ")"
      FunTable symname -> do
        row <- rels
        return $ symname ++ "(" ++ intercalate ", " (map prettyElement2
            (init row)) ++ ") = " ++ show (last row)
      DomTable -> [])

prettyElement2 = init . tail . tail . show
```

Bibliography

[1] Catriel Beeri and Moshe Y Vardi. A Proof Procedure for Data Dependencies. *Journal of the ACM*, 1984.

[2] Davide Cervone. Mathjax: a platform for mathematics on the web. *Notices of the AMS*, 59(2):312–316, 2012.

[3] K. Claessen and N. Sörensson. New Techniques that Improve MACE-Style Finite Model Finding. In *CADE Workshop on Model Computation-Principles, Algorithms, Applications*, 2003.

[4] Alin Deutsch and Val Tannen. XML Queries and Constraints, Containment and Reformulation. *ACM Symposium on Theory Computer Science*, 2005.

[5] Matthew Elder and Jeremy Shaw. Happstack. `http://happstack.com/page/view-page-slug/1/happstack`.

[6] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[7] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. Verification and Change-Impact Analysis of Access-Control Policies. In *International Conference on Software Engineering*, May 2005.

[8] Daniel Jackson. Lightweight formal methods. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, FME '01. Springer-Verlag, 2001.

[9] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, April 2006.

[10] Daniel Jackson and Jeanette Wing. Lightweight formal methods. *IEEE Computer*, April 1996.

[11] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. 2002.

[12] David Maier, Alberto O Mendelzon, and Yehoshua Sagiv. Testing Implications of Data Dependencies. *ACM Transactions on Database Systems*, 1979.

[13] Simon Marlow et al. Haskell 2010 language report. *Available at* `https://www.haskell.org/onlinereport/haskell2010/`, 2010.

[14] William McCune. MACE 2.0 Reference Manual and Guide. *CoRR*, 2001.

[15] Tim Nelson, Salman Saghafi, Daniel J Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Aluminum: principled scenario exploration through minimality. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013.

[16] Timothy Nelson, Christopher Barratt, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. The Margrave Tool for Firewall Analysis. In *USENIX Large Installation System Administration Conference*, 2010.

[17] Mark Otto and Jacob Thornton. Bootstrap: The world's most popular mobile-first and responsive front-end framework. `http://getbootstrap.com/`.

[18] Justin Pombrio. Protocol analysis via the chase. Major Qualifying Project report, Worcester Polyechnic Institute, 2011.

[19] Salman Saghafi and Daniel J. Dougherty. Razor: Provenance and exploration in model-finding. In *4th Workshop on Practical Aspects of Automated Reasoning (PAAR)*, 2014.

[20] Salman Saghafi, Tim Nelson, and Daniel J. Dougherty. Geometric Logic for Policy Analysis. In *Workshop on Automated Reasoning in Security and Software Verification*, pages 12–20, 2013.

[21] Emina Torlak and Daniel Jackson. Kodkod: A Relational Model Finder. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2007.

[22] Steven Vickers. Geometric Theories and Databases. In *Applications of Categories in Computer Science*, pages 288–314, 1992.

[23] Steven Vickers. Geometric Logic in Computer Science. In *Theory and Formal Methods*, pages 37–54, 1993.

[24] Steven Vickers. Geometric Logic as a Specification Language. In *Theory and Formal Methods*, pages 321–340, 1994.

[25] J. Zhang and H. Zhang. SEM: a system for enumerating models. In *International Joint Conference On Artificial Intelligence*, 1995.