2016

A Functional Approach to Library Construction for Conceptual Reasoning

David William Patrick MacMillan
University of Windsor

A Functional Approach to Library Construction for Conceptual Reasoning

By:

David MacMillan

A Thesis
Submitted to the Faculty of Graduate Studies
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada

2016

A Functional Approach to Library Construction for Conceptual Reasoning

By:
David MacMillan

APPROVED BY:

Dr. R Caron
Department of Mathematics and Statistics

Dr. D Wu
School of Computer Science

Dr. R Kent, Advisor
School of Computer Science

August 16, 2016

Author's Declaration of Originality

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

Abstract

Conceptual Spaces is an emerging theory of knowledge representation that describes relationships between concepts, contexts and observations. Due to its mathematical and functional nature, this theory is quite useful for automated reasoning. However, to the best of our knowledge, there are no software frameworks or packages that allow us to explore the computational applications of this theory. In this thesis, we designed, developed and tested a library containing conceptual data structures and operators, primarily for use in automated reasoning and decision support systems.

Dedication

To Bob and Sue, for their undying support.

Acknowledgements

First and foremost, I would like to thank God for giving me the strength and endurance to complete this work.

Thanks to my supervisor Dr. Robert Kent for his support throughout my time at the university. The opportunities you gave me helped me develop skills that will last a lifetime. Thanks to my committee members for their comments and work in making this thesis better. Thanks also to the support staff at the School of Computer Science for their assistance with the minute details of submitting a thesis.

To my family, my parents Bob and Sue, my brother John and sister-in-law Sam, and my nephews Micah and Josiah: You made me who I am today and I love you all dearly.

Thanks also to the students I worked with during my time here. Bryan St. Amour, Jordan Willis, Paul Preney, Numanul Subhani, I wouldn't have made it this far without your ideas, help and encouragement.

Lastly, to my friends who have supported me outside of the academic setting. Jordan Legg, Jordan Ditty, Peter Brinn, Brittni Carey, Trevor de Boer, Nathan Buck. You helped me realise the potential I have not only as a student but as a man.

Contents

Author's Declaration of Originality **iv**

Abstract **v**

Dedication **vi**

Acknowledgements **vii**

List of Tables **xii**

List of Figures **xiii**

1 Introduction **1**

1.1 Motivation 1

1.2 Problem Statement 4

1.3 Proposed Solution 4

1.4 Contribution 5

1.5 Organisation of Document 6

2 Background **7**

2.1 Automated Reasoning 7
2.1.1 Fuzzy Logic 9
2.2 Conceptual Spaces 10
2.2.1 Applications 16
2.2.2 Automated Reasoning in Conceptual Spaces 17
2.2.3 Summary 18
2.3 Summary 19

3 Functional Programming with Haskell 20

3.1 Functional Programming and Lambda Calculus 20
3.1.1 System F_ω 21
3.2 The Haskell Programming Language 22
3.2.1 Types 23
3.2.2 Maybe 25
3.3 Summary 26

4 Thesis Statement 27

4.1 Thesis Problem 27
4.2 Hypothesis 28
4.3 Objectives 29
4.4 Research Methodology 30

5 Conceptual Reasoning Library in Haskell 31

5.1 Data Structures 31

5.1.1 Domain 32
5.1.2 Property 33
5.1.3 Observation 34
5.1.4 Context 36
5.1.5 Concept 36
5.1.6 Summary 39
5.2 Operators 39
5.2.1 Property-Observation measure 39
5.2.2 Observation-Concept measure 41
5.2.3 Concept-Concept measure 45
5.2.4 Summary 48
5.3 Design Considerations 49
5.4 Limitations 49
5.5 Summary 50

6 Analysis of Results **52**

6.1 Demonstration of Type-Safety 52
6.1.1 Property 53
6.1.2 Domain 53
6.1.3 Observation 54
6.1.4 Context 55
6.1.5 Concept 56
6.2 Proofs of Termination 58

6.2.1 Property-Observation 58
6.2.2 Concept-Observation 59
6.2.3 Concept-Concept 61
6.3 Demonstration of Verifiability 63
6.4 Summary 67

7 Conclusion **68**
7.1 Summary of Thesis Findings 68
7.2 Future Work 69

Bibliography **70**

A CRLH code **74**
A.1 Types.hs 74
A.2 Constructors.hs 75
A.3 Operators.hs 78
A.4 Test.hs 85

Vita Auctoris **87**

List of Tables

6.1 Comparison of results 67

List of Figures

6.1 Routes between ports indicated for (a) fair conditions (b) and stormy conditions [22]. 64

6.2 Matrix of associations involving concepts C^1 and C^2 [22]. 65

6.3 Matrix representation of observation **o** in both subconcepts of C^1 [22]. . . . 65

6.4 Matrix representation of observation **o** in concept C^2 [22]. 66

Chapter 1

Introduction

1.1 Motivation

Imagine a professor who is in a thesis proposal seminar and is trying to determine the validity and quality of the research being proposed. He or she can rely on their past experience, but not as a direct correlation - theses must be unique, after all. However, some professors will have a list of questions for the student: does this thesis contribute to the field, and how? Does the student have enough of a grasp on the background reading and have they explored any related work? Does the student's presentation leave you with the impression that they know what they are trying to accomplish and the confidence that it can be done? Most importantly, is this proposal still a proposal or have they done the majority of the work already?

Now imagine a scenario in which a lab technician is deciding how to allocate some newly purchased computers. Four computers were ordered, with differing specifications. The technician knows that these computers have different tasks; he needs two workstations,

a server, and a computer for organizing backups. The technician has a vague idea of what these tasks require. How does he go about categorizing the machines?

Consider a hunter who has set up his camouflaged shelter during deer hunting season. For various reasons, he is only allowed to shoot female deer. As the hunter cannot spend his entire life living in his shelter waiting for deer to show up, he installs a camera to take pictures while he is away. The camera itself is connected to a sensor that takes a picture when movement is detected. In this scenario, the hunter has a number of (time-stamped) pictures of a certain part of the forest. Using these pictures, he can determine what is the best time to hide in his shelter to attempt to shoot the deer. He may also consider moving his shelter if his photographs show no deer.

Another real-world example to consider is the management of a baseball team. The manager has a set number of players on her team, and a eight fielding positions to fill. For the sake of this example we will ignore the existence of pitchers. No two positions are exactly alike, so making direct comparisons is not always a possibility. The manager knows that all of her players have a certain level of skill in a number of areas (running, catching, throwing) but some positions require other talents - the second baseman and shortstop must be skilled in communicating with each other, for example. Now let us say that the team's left fielder comes down with an injury during the season. The Manager examines her roster and finds she does not have another left fielder. She cannot send out a team with only seven position players, so what does she do?

In all of these situations, reasoning involves comparing what is observed to preexisting knowledge of the categories at hand. The professor, for example, cannot accept a thesis proposal which is identical to one previously presented, though some areas of research

have room for several similar theses. The technician, on the other hand, may have a precise, ideal specification for a backup machine. If one of the computers matches that, he is in luck. However, if his specification is more vague, or presented in terms that are not compatible with the system specifications (Ex. A backup computer should have "a lot" of storage space), how does he make a decision?

Furthermore, the hunter must be able to identify the animals in the picture. The pictures themselves may not be of very high quality; the cameras used have generally been adapted for low-light situations. The hunter must be able to identify animals based on their physical traits, as there are significant penalties for hunting animals out of season. In the example of managing a baseball team, we can consider the overlap that can occur between different specializations within a more generalized field. The team manager in question may find that since there is a significant skill set overlap between playing left field and playing centre field, and she has an unused centre fielder, she can have her backup centre fielder play left field until her left fielder is healthy again.

What we are dealing with here is called *conceptual reasoning*, that is, reasoning about concepts. The Oxford English dictionary [27] defines a *concept* as:

> a general idea or notion, a universal; a mental representation of the essential or typical properties of something, considered without regard to the peculiar properties of any specific instance or example.

Considering the abstract nature of this definition, reasoning about such things is relatively foreign to the kind of automated reasoning systems used in artificial intelligence. A framework that could make these conceptual ideas less abstract would be useful in that it

could increase the applicability of such automated reasoning systems.

1.2 Problem Statement

Categorization - also called classification - is an open area of research in artificial intelligence. Conceptual Spaces is an emerging theory of cognition about categorization and reasoning. To the best of our current knowledge, there are no software tools to help researchers test effectiveness and find new applications.

Most of the relevant foundations for pursuing research into computational cognition require mappings of notions from cognitive science, including philosophy of semantics, especially as it pertains to the use of language in describing the world of objects and relationships between them. To this end, we may reference some elements of category theory [16] in the discussion of our system.

The theory of Conceptual Spaces, initially defined by Peter Gärdenfors [7], has been reformulated by Rickard, Aisbett and Gibbon [22] for applications in Computer Science; however, to the best of our knowledge there are no software tools available for researchers to further develop, or apply, this theory to new and existing problems.

1.3 Proposed Solution

To this end, we intend to build a library of data structures and operators to allow users to model conceptual spaces. We will show our library to be type safe, prove the termination of our operators and verify our library by repeating select calculations from Rickard *et al*

in the literature.

1.4 Contribution

Conceptual similarity as a method of categorization applies to a number of open areas of research in Computer Science, including but not limited to data cleaning, pattern recognition, machine learning, decision support, semantic interoperability and computer vision. There may also be some applicability to conceptual mathematics and category theory [16] as well. Moreover, should conceptual spaces lend itself well to modelling some elements of category theory, our work will be useful in other aspects of that field of study.

In this thesis we have done the following:

- We developed a type-safe, composable library of conceptual data structures and operators. (Chapter 5)
- We showed that the library is type-safe and discussed its handling of invalid data. (Section 6.1)
- We showed that the operators terminate given valid input. (Section 6.2)
- We demonstrated the verifiability of the library by repeating select calculations from the literature. (Section 6.3)

1.5 Organisation of Document

In this thesis we will present a functional programming library for conceptual reasoning. In Chapter 2 we will discuss automated reasoning and conceptual spaces as it was originally defined and currently exists in the literature. In Chapter 3 we will discuss functional programming in general and the Haskell programming language in particular. In Chapter 4 we formalize the thesis problem, our hypothesis, objectives and research methodologies. Chapter 5 will describe our solution and implementation, including the reasons behind choices we made in our implementation. Chapter 6 will include our proofs of termination, demonstration of type-safety and show the verifiability of our operators by repeating a selection of examples from the literature. Chapter 7 will conclude the thesis and discuss areas of improvement and future research. A full list of references is available following Chapter 7. Appendix A contains the full source code for our library, including brief instructions detailing its proper use.

Chapter 2

Background

We begin with a discussion of automated reasoning followed by an overview of conceptual spaces.

2.1 Automated Reasoning

Reasoning is the process of drawing conclusions from facts. For the conclusions themselves to be certain, they must follow directly from the facts. There are three main methods of reasoning:

- deduction, in which we have a series of rules, in the form "If A then B", from which we can draw conclusions through the use of *Modus Ponens* and *Modus Tollens* [26]. For example, Modus Ponens tells us that if we know that all rugby players are nice people and we know that Mr. Jones is a rugby player, we know that Mr. Jones is a nice guy. Modus *Tollens*, on the other hand, tells us that if we know that all rugby players are nice people and that Mr. Jones is not a nice person, then Mr. Jones is *not*

a rugby player,

- induction, in which we form hypotheses from existing observations and test to see if they hold given new data. As a result, the information drawn from induction is not certain, but rather probable. For example, if after observing several rugby matches, a sports fan may hypothesize that all matches are played at full-contact. However, this hypothesis will change after watching a child's match, which is played using a 'two-hand touch' rule,

- abduction, in which we determine the best hypothesis to explain an observation. The best example of abduction in everyday life would be a doctor diagnosing an illness or injury after hearing a description of symptoms. Given that a doctor cannot observe the causes of the symptoms, their conclusion is a 'best guess'. In the event that there are a number of competing theories, a doctor must test a given theory to determine if it is true.

According to Wos *et al*, *Automated* reasoning is "concerned with programs that aid in solving problems and in answering questions where reasoning is required" [30], and an automated reasoning *program* is one that "employs an unambiguous and exacting notation for representing information, precise inference rules for drawing conclusions, and carefully delineated strategies to control those inference rules" [30]. Given this definition, deductive reasoning appears to be the most appropriate style of reasoning for such programs. The trouble with such systems is that the world is rarely unambiguous.

As a result, more recent systems are focused on reasoning under uncertainty. Most of these frameworks make use of probability in an attempt to draw a number of possible

responses. These implementations include Bayesian Probability [29], Fuzzy Logic [32], Dempster-Shafer Theory [25], and Subjective Logic [14]. These theories have been used in a variety of reasoning applications and fuzzy logic specifically is used in the theory of Conceptual Spaces.

2.1.1 Fuzzy Logic

Fuzzy Logic is a multi-valued logic designed for approximate reasoning. It differs from the classical forms of logic in that it deals with approximate truth values rather than the precise values of classical logic. The term "fuzzy logic" was first used by Zadeh in describing *fuzzy set theory* [31], a theory that discussed the idea of partial or approximate membership in sets.

A fuzzy set is a set A on a domain X, where A is equipped with a function f_A where the value determined by $f_A(x)$ is on the range [0,1] and describes the likelihood that x is in the set A. As an example, let A be the fuzzy set of real numbers that are much larger than 1. The function f_A maps values from the domain of real numbers to the codomain [0,1]. We can describe some potential representative values of f_A, for example $f_A(0) = 0$, $f_A(1) = 0$, $f_A(10) = 0.05$, $f_A(1000) = 1$. From here we have "a precise, albeit subjective, characterization of A" [31].

Fuzzy logic is an adaptation of fuzzy set theory, adapted to classical forms of logic. Given a predicate G and a value x, let $G(x)$ be a function that maps x onto the interval [0,1]. We can say that the value of $G(x)$ represents the extent to which x satisfies the predicate G. For example, let us consider the predicates Hot and $Cold$. Given the variable $cool$, we

could say that $Hot(cool) = 0.05$ and $Cold(cool) = 0.75$ as *cool* is significantly closer to *Cold* than to *Hot*. That is, *cool* is more *Cold* than it is *Hot*. It is important to note, however, that a different observer may assign different values to these associations.

Fuzzy logic has the AND, OR and NOT operators of Boolean logic, but since fuzzy logic is multi-valued, these operators have been redefined. In fuzzy logic, the AND operator selects the lower of the two, while the OR operator selects the greater. The NOT operator is equal to 1 minus the truth value. For example:

$Cold(cool)$ OR $Hot(cool) = 0.75$ OR $0.05 = 0.75$,

whereas $Cold(cool)$ AND $Hot(cool) = 0.75$ AND $0.05 = 0.05$,

and NOT$(Cold(cool))$ = NOT$(0.75) = 0.25$.

There have been numerous applications of fuzzy logic and fuzzy sets since they were introduced in 1965. In the following section, we will discuss the theory of conceptual spaces and how fuzzy logic has been useful in reformulating conceptual spaces for use in computer science.

2.2 Conceptual Spaces

Conceptual Spaces is a theory of concept representation that was initially proposed by Peter Gärdenfors [7] and later improved by Jane Aisbett and Greg Gibbon [1] and John Rickard [21], among others. Rickard, Aisbett and Gibbon later combined their work [22], which is the basis for this thesis.

The purpose of Conceptual Spaces is to try to quantify and qualify concepts. Recall that a concept is an abstract representation of an idea. The point of such a framework is

to help the user make their ideas less abstract without making them so concrete that they are not useful for categorization. Let us consider example of managing a baseball team from Chapter 1. If we make our idea of a left fielder (or any other position) too concrete, we would run the risk of not being able to find a replacement player in case of injury. If we said, for example, that a left fielder needs to be able to run fast, have good hand-eye coordination and know how to field different kinds of hits, we would have made our concept concrete enough to find a replacement player, for example. If we were to say that our left fielder also needed to be six feet tall, weigh 200 pounds and be named Michael Saunders, we would have made our concept too specific.

Gärdenfors' initial theory used a geometric representation in which observations are points in an n-dimensional space and concepts are represented as convex regions of these spaces. He also noted the existence of *prototypes*. These prototypes are located at the centre of their respective concept. While an observation will not necessarily have values in every dimension, some dimensions require a value in another. For example, we cannot discuss a coloured light's hue without the light also having a brightness and saturation.

Aisbett and Gibbon's extension [1] of conceptual spaces took advantage of the geometric aspects of Gärdenfors work. Under this new framework, similarity between two properties is measured by the distance between them. The similarity of two concepts is measured by the distance between their prototypes.

Rickard [21] extended Gärdenfors work to represent a concept as a point in a unit hypercube, which describes the concept's properties, salient weights and correlations. The advantage this provides is that it allows a concept to be learned from training data. Rickard also introduces a method for representing observations as a point in the same unit hyper-

cube. Because both the observations and concepts are in a unit hypercube, results from the similarity measures are already normalized.

Another way Rickard extended Gärdenfors work is to represent concepts as fuzzy sets rather than convex regions. By using elements of fuzzy logic and fuzzy set theory, Rickard is able to introduce a metric for measuring similarity between two concepts via mutual subset-hood and a metric for measuring similarity between an observation and a concept via fuzzy subset-hood.

In their combined work [22], Rickard *et al* defined domains as a collection of integral dimensions. The authors chose to use membership functions from fuzzy set theory to describe properties. A property, then, is a fuzzy subset of a domain whose membership function defines the extent to which a given observation relates to that property. Because a property is a fuzzy set, it has a corresponding membership function called p_{ij} where p is the jth property on the ith domain. The authors also define the *intersection* of two properties p_{ij} and p_{ik} on the domain i to be $p_{ij} \cap p_{ik}(\mathbf{x}) : x \rightarrow \min(p_{ij}(x), p_{ik}(x))$. Furthermore, the overlap on any two properties on different domains to be 0 [22].

Concepts are functions on the domain $I(C) \times I(C)$ to the codomain [0,1] where $I(C)$ is the set of properties of the concept C, where for all $a, b \in I(C)$ the following two statements are true: $C(a,a) > 0$ and $C(a,b) = 0$ whenever $C(b,a) = 0$ [22]. The term C_{ab} denotes the *association* of property a and property b. This value is not necessarily equal to C_{ba}, as these associations may have been learned from the co-occurrences of those properties in training data.

The authors also describe the idea of a concept being smaller than another. A concept C is smaller than a different concept D if $I(C) \subset I(D)$ and $C(a,b) = D(a,b)\ \forall a, b \in I(C)$.

From here on, we will borrow Rickard *et al*'s use of matrix notation, where $C(a,b) = C_{ab}$. Furthermore, the authors describe the subconcepts of a concept C as the "maximal concepts smaller than C that have no zero associations." [22] That is, if C' is a subconcept of C, then for any concept C'' that is smaller than C, either C' is not smaller than C'' or $C''_{ab} = 0$ for some $a,b \in I(C')$.

Consider again the example of the baseball manager from Chapter 1. If we model *position player* as a concept, we can model different positions as subconcepts of that concept. In the example of the hunter, a concept could model the quality of a particular hunting location, with subconcepts modelling the quality of that location with respect to different game. In our example of the lab technician, we can consider a concept to be the role a particular computer might be used for. In cases where a role is composed of multiple smaller roles, a subconcept could be considered to be a singular role. For example, while acting as a development machine, a computer might also perform the role of a database or web server.

A context is defined as a set of properties. Rickard *et al* defined a similarity measure between two concepts (C^1 and C^2) in a context G has been stated as follows:

$$s(C^1, C^2) = \frac{\sum_{a,b} \min\left(C^1_{ab}, C^2_{ab}\right)}{\sum_{a,b} \max\left(C^1_{ab}, C^2_{ab}\right)}, \tag{2.1}$$

where the sums are over all pairs of elements in $I(C^1, C^2; G) \equiv (I(C^1) \cup I(C^2)) \cap G$ [22]. We note that in this equation, we are adding the results of applying the OR operator from fuzzy logic and dividing that by the summation of applying fuzzy logic's AND operator, applied to all elements of both concepts.

Again using the example of the baseball team from Chapter 1, when moving a player

to a secondary position, the manager must consider which positions are most similar. In Chapter 1 we discussed the idea of a manager needing to replace a left fielder. The concepts *centre fielder* or *right fielder* are closer to *left fielder* than the concept *catcher* would be, for example. In this framework, what this would mean is that $s(\textit{left fielder}, \textit{centre fielder}) > s(\textit{left fielder}, \textit{catcher})$. In the example of the hunter, this measure could be used to determine which game is more prevalent overall. In the example of the lab technician, some roles are similar to others. A database server and a backup server, for example, both require large volumes of storage space. In that sense, they are similar, however a backup server may not necessarily require the constant availability of a database server.

In situations where the concepts contain different properties, similarity values for these properties can be calculated using the property overlap measure

$$C^1_{ab} = B^r_{aa*} B^s_{bb*} C^1_{a*b*} \tag{2.2}$$

where $B^r_{aa*} \gg B^r_{aa'}$ for all $a' \in I_r \cap I(C^1)$ and $B^s_{bb*} \gg B^r_{bb'}$ such that $a' \neq a*$, $b' \neq b*$ and

$$B^i_{jk} = \frac{\int \min\left(p_{ij}(x), p_{ik}(x)\right) \mathrm{d}m_i}{\int p_{ij}(x)\, \mathrm{d}m_i} \tag{2.3}$$

[22]. In most cases, the discrete form of Equation 2.3 is used, seen here:

$$B_{jk} = \frac{\sum_r \min(p_j(r), p_k(r))}{\sum_i p_j(r)} \tag{2.4}$$

[22].

An observation is described as a vector in a feature space. More specifically, o is defined

as the collection of sets of points $\{o_i\}$ from domains $\{\Delta_i\}_{i\in K}$, where K is the set of all domains. This definition allows for observations to have multiple points on one domain. Because of this, we define the membership $p_{ij}(\mathbf{o})$ to be $\max\{p_{ij}(y) : y \in \mathbf{o}\}$.

Considering again our example of a baseball manager from Chapter 1, an observation would be analogous to the state of an individual player's ability, measured at a certain point in time. In our example of the professor considering a thesis proposal, an observation would be a proposal seminar. In our example of the hunter, we could consider an observation to be a singular photograph taken by the hunter's camera. Lastly, in our example of the lab technician, an observation is analogous to an individual computer.

Having defined the their methods of data representation, the authors then present a mechanic for identifying an observation as an instance of a concept. The similarity for any observation $\mathbf{o}$ to a concept C in a context G is

$$s(C,\mathbf{o}) = \max_{C\text{' a subtype of } C} \left(\frac{\sum_{a,b\in I(C')\cap G} \min\left(C'_{ab}, o_{ab}\right)}{\sum_{a,b\in I(C')\cap G} C'_{ab}} \right). \tag{2.5}$$

This equation is very similar to that of the one used to calculate the similarity of two concepts. Should a context G not be provided, $I(C')$ is used as a context. In the example of our lab technician, this measure would be used to assist him in deploying new equipment to various roles in his lab. In the example of a baseball team from Chapter 1, we could consider this similarity function to be a tool for determining which position a particular player is best suited for at a specific point in time. In the example of the professor reviewing a thesis proposal, this similarity measure could be viewed as the determination process through which the acceptability of a proposal is measured. Similarly, in our example of the

hunter, this measure could be used, with a number of observations, to show whether or not the current location is good for a specific type of game.

2.2.1 Applications

There have been a number of proposed applications of conceptual spaces [34], including semantic interoperability [24], computer vision [4], predication [6] and computing with words [2].

Chella *et al* [4] proposed applying conceptual spaces as an intermediate representation of real-world objects in computer vision. Specifically, they proposed using conceptual spaces as a method of deconstructing physical objects into their component parts. For example, the authors deconstructed a hammer into its handle, as a cylinder, and its head, as a box. They found that their conceptual spaces approach provided results similar to other descriptive frameworks but allowed for measuring similarity between different objects and parts.

Dessalles [6] discusses the difficulty in assigning meaning to phrases and presents a method for doing this, using conceptual spaces as a means of categorizing phrases to meanings. Dessalles extends conceptual spaces with a *contrast* operator, which assists in binding the meaning of words to the contexts in which they are found. The author considers the word ‘big’, applied to different objects. A ‘big’ flea, for example, is much smaller than a ‘big’ galaxy. Furthermore, Dessalles argues that the contrast operator, when used on predicates, is more dynamic than the prototype model used by the traditional Gärdenfors approach.

Scheider and Kuhn proposed conceptual spaces as a method for semantic interoperability through conceptual imitation [24]. By using conceptual spaces as a basis, the authors are able to model a multiple-perspective situation and determine the relationship between pairs of concepts when viewed from different perspectives. The authors use a number of examples including modelling the concept of a mountain when agents are using different training data (mountains in England vs mountains in Asia) and comparing the two on multiple pairs of properties (average wind speed vs average temperature, altitude vs relative relief).

Additional work towards disambiguating information has been proposed by Aisbett, Gibbon and Rickard [2] in an effort to integrate conceptual spaces and Computing With Words [33], a set of tools that might be developed to allow computers to do input/output with words rather than numbers. Applying conceptual spaces to this problem may help solve some of the underlying issues in Computing With Words, mainly the lack of a suitable modelling framework. The authors note a number of similarities between the two systems, but noted a number of problems with Gärdenfors' definition of a concept.

2.2.2 Automated Reasoning in Conceptual Spaces

We have covered the basics of automated reasoning and conceptual spaces. In this section, we will discuss how conceptual spaces can be used for automated reasoning.

If we consider the types of inference rules mentioned in Section 2.1 to exist in the form "If a then b ($a \Rightarrow b$)", then conceptual spaces can be used to help classify raw data into the *predicates* that exist in such rule systems. The classification would work as follows:

Consider a conceptual space consisting of the domain Δ_h representing height and the domain Δ_s representing speed. Δ_h has two properties, t and sh, while Δ_s contains properties f and sl. Next we consider a set of concepts C_{pf}, C_c, representing different positions on a basketball team. The mappings in each of these concepts could be considered an inference rule themselves. Given an observation o, if $(t(o) > 0.85 \wedge sl(o) > 0.7)$, then $s(C_c, o) = 1$, if $(t(o) > 0.75 \wedge t(o) < 0.9 \wedge sl(o) > 0.6 \wedge sl(o) < 0.75)$, then $s(C_{pf}, o) = 1$, etc. This result itself could be expanded even further, if given additional inference rules.

2.2.3 Summary

So, to summarize: a *domain* is a collection of dimensions (that is, axes) equipped with a set of properties. Furthermore, a *property p* is a fuzzy subset of a domain *i*. The property is represented by a membership function $p_j(x)$ from the domain *i* to the codomain [0,1]. The value on the codomain represents the extent to which a given data point 'has' that property.

An *observation* is a collection of points on various domains. As there can be multiple points on any one domain, the membership value of that observation to any given property on that domain is equal to the highest membership value of any point in the observation.

A *context* is a set of properties. A *concept* contains a set of properties and a mapping from pairs of those properties to the interval [0,1]. These properties define the concept, and the mapping defines the extent to which each property is a representation of the concept. Functions that measure the similarity of two concepts, and the similarity of a concept and an observation have been defined. A concept also contains a set of *subconcepts*. These subconcepts are contexts whose properties form a subset of the parent concept's property

set.

2.3 Summary

In this chapter we provided an overview of automated reasoning and fuzzy logic. We described and discussed the theory of conceptual spaces and related it to a number of examples introduced in the motivation for our thesis. In the following chapter, we will describe functional programming, lambda calculus and the Haskell programming language.

Chapter 3

Functional Programming with Haskell

3.1 Functional Programming and Lambda Calculus

Functional programming was initially described by John Backus in his 1977 Turing award lecture "Can Programming Be Liberated from the von Neumann Style? A functional Style and Its Algebra of Programs" [3]. As the name suggests, the entities in functional programming are described by what they *do*, rather than what they *are*. A *purely functional* language is one in which the definition of function is equivalent to the mathematical definition in that the outputs depend solely on the inputs. Much of the basis of functional programming comes from lambda calculus [11].

Lambda calculus is a system of expressing the computational aspect of functions [11]. Developed initially by Church in an attempt to define a mathematical logic in which "every combination of symbols ... if it represents a proposition at all, shall represent particular proposition, unambigouously[sic], and without the addition of verbal explanations" [5]. In one of the earliest forms of lambda calculus, what Hudak calls pure untyped lambda

calculus [11], there are two types of lambda expressions, consisting of identifiers (x) and expressions (e). An expression e can have one of three forms:

- $e ::= x$
- $e ::= e_1 e_2$
- $e ::= \lambda x.e$

Expressions in the form $\lambda x.e$ are called abstractions and they are the notation used for functions, whereas the $e_1 e_2$ form are the application of those functions [11]. Lambda calculus makes heavy use of substitution in expressions. The syntax $[e_1/x]e_2$ indicates that we replace every occurrence of x in e_2 with the expression e_1.

What interests us most is *typed* lambda calculus. The key difference between typed and untyped lambda calculus is that the type system allows, as one might imagine, for *typed* identifiers and expressions. Hudak outlines a number of issues with typed lambda calculus [11], but Hindley [9] and Milner [18] developed a polymorphic type system which, while somewhat restricted, still maintains decidable type inference.

3.1.1 System F_ω

As we are interested in the Haskell programming language and its type system, we will give an overview of System F_ω, an extension to typed lambda calculus. Developed independently by Girard [8] and Reynolds [20], System F introduced the Λ operator. This operator is used to describe the type of a variable, thus defining polymorphic functions in lambda calculus. For example, suppose we have some type σ and a variable x, then x^σ

means that x is of type σ. The identity function for such a type would be $\lambda x^{\sigma}.x^{\sigma}$. A *polymorphic* identity function, then, would be $\Lambda\sigma.\lambda x^{\sigma}.x^{\sigma}$. System F_{ω} thus introduced *kinds*. A *kind* can be considered types of types, or type classes. This system is the basis for Haskell's type system.

3.2 The Haskell Programming Language

Haskell is a non-strict, strongly-typed purely functional programming language [13] that was developed in part for teaching and research [12], including research into programming languages and their features. By *non-strict*, we mean that expressions are only evaluated when needed. This is known as *lazy evaluation* and has the added benefit of having each expression be evaluated at most once [11].

The Glasgow Haskell Compiler includes a number of basic data types, including *Char, Integer, Float, Double, Bool*, etc. Also included are a set of *type classes*, which can be thought of as groups of types that have something in common. The *Eq* type class, for example, is a collection of types that, amongst themselves, can be equated. The *Num* type class is the collection of types that are numbers. Type classes can be members of other type classes if they share the appropriate properties. The *Ord* type class, which consists of all types that can be ordered, also contains the Eq type class.

Since Haskell is a purely functional language, its functions are written as equations, with white space separating the parameters of each function. For example,

```
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = (fibonacci n-1) + (fibonacci n-2)
```

Not all functions are named, however. Haskell has *anonymous* or *lambda* functions, which look very similar to the abstraction notation from lambda calculus.

3.2.1 Types

Each expression in Haskell has a type. As a result, even the expression '*c*' would have the type *Char*. In Haskell, this is expressed as '*c*' :: *Char*. Our function above, *fibonacci*, has the type $Int \rightarrow Int$. When defining a function, it is required to define its *signature*, which is the name of the function, followed by its type. Thus, the signature of our function *fibonacci* would be

```
fibonacci :: Int → Int
```

A *list* is enclosed by square brackets, and all items must be of the same type. For example,

```
a = ["This", "is", "a", "list"]
b = ["This", 3, "isn't"]
```

a is a valid list, while *b* is invalid because of its mixed types. Haskell also contains some very powerful list comprehension tools, which function similarly to set comprehension tools found in set theory. This allows us to construct lists like the following:

```
a = [3,6,7]
let b = [x | y ← a, x ← [y*2]]
```

In the second line, *b* is a list of *x*, where *y* comes from the list *a* and *x* comes from the list of $y * 2$. This results in $b = [6, 12, 14]$. This feature is simplifies programming as it allows the programmer to let the compiler deal with the list management so the programmer can focus on the actual transformation function.

Further to this, Haskell allows for the user to create new types and type classes. There

are two main ways of creating a new type. The *type* keyword is used for renaming an already existing type while copying its constructor. For example, the *String* type is defined as follows:

```
type String = [Char]
```

This line of code defines a *String* as a list of *Char*. The second way to create a new data type is the *newtype* keyword, which allows us to rename a type and re-implement its constructor, but there are two caveats with newtype. Firstly, it can only support one constructor. Secondly, there can only be one field in a data type created with newtype. The *data* keyword is used to combine multiple (existing) types. Let us consider a point on a two-dimensional map. If we wanted to create this data type using the *type* or *newtype* keywords, we could use the following:

```
type Point = (Int,Int)
newtype Point = Point (Int,Int)
```

In this case, the data structure can only be described by its coordinates. If we wanted to attach a label to this point, we must use the *data* keyword as follows.

```
data Point = Point (Int,Int) String
```

This line creates a data type and corresponding constructor called *Point* which takes a tuple of *Int*s and a *String*. Using this syntax, however, accessing the individual items in the data structure requires us to write a function like this:

```
label :: Point → String
label (Point _ x) = x
```

While this is easy for a simple data type like Point, it can get tedious when we start to get longer and more complex data structures. Fortunately, Haskell includes *record* syntax, which allows us to implicitly create the functions needed to access these members. To

leverage the record syntax, we can recreate the Point type as follows:

```
data Point = {
    coordinates :: (Int,Int),
    label :: String
    }
```

There are two functions created using Haskell's record syntax. The function *coordinates* takes a Point and returns (Int,Int), while the function *label* takes a Point and returns a String. The downside to using record syntax is that it only allows one constructor per data structure. Creating a new data type with the *data* keyword does use more memory than either of the other two keywords, but it is the only way to create a multi-field data type.

3.2.2 Maybe

In our research, one of the most useful aspects of Haskell is the *Maybe* type. Maybe has two constructors: *Just x*, and *Nothing*. Consider, for example, a function that takes a number and returns its square root. There are a cases where a square root does not exist (complex or negative numbers, for example). If we were to write such a function, we could use the Maybe type when returning. This way, even if we are given a negative number, our function would terminate and return a value, even if the value is Nothing. The Maybe module includes a number of functions for handling values wrapped in Maybe, including *fromMaybe*, which takes two values, $a :: x$ and $b :: Maybe\ x$. If b is wrapped in the *Just* constructor, it is returned unwrapped. If b is *Nothing*, a is returned.

This allows our library to be built entirely with composable functions without requiring our data types to be well-formed. While our library requires well-formed data types to return useful results, the library itself will always return an output when given an input. This is discussed further in Chapter 6. Another useful aspect of Haskell is its use of list com-

prehension tools, discussed above. This has simplified the implementation of some of our work. We will show in Chapter 5 how we used these tools to simplify the implementation.

For those who are interested in learning more of Haskell, we suggest the book Real World Haskell [19]. We find that for total beginners, the book "Learn You a Haskell For Great Good!" [17] is also a good resource.

3.3 Summary

In this chapter we have introduced functional programming, giving an overview of its history. We discussed lambda calculus, and some extensions that allow for types and type classes. We then discussed the Haskell programming language, including an overview of its type system and its *Maybe* type. In the next chapter, we will present our thesis problem and proposed solution.

Chapter 4

Thesis Statement

In this chapter we will discuss the thesis problem, our hypothesis, objectives and research methodology.

4.1 Thesis Problem

There have been a number of attempts to apply the theory of conceptual spaces to open areas of research in artificial intelligence. Scheider and Kuhn [24] describe an approach to semantic interoperability centred around conceptual imitation, using conceptual spaces as a basis for synthetic learning. Dessalles [6] discussed the possibility of using conceptual spaces for predication. Chella *et al* [4] use conceptual spaces as a generalized framework for modelling complex objects in the field of computer vision, though they make mention of a number of problems, including the definition of a suitable similarity measure. Aisbett, Gibbon and Rickard [2] propose the integration of conceptual spaces and computing with words [33] to further develop those frameworks, as described in Section 2.2.1.

As we stated earlier, there are no software tools currently available for researchers to make use of the theory of conceptual spaces for research, development and testing. While there have been a number of discussions on possible applications in computer science, to the best of our knowledge a complete implementation does not exist. This is a problem because the lack of these tools means that the specific details and problems related to the implementation of this theory would never get addressed. We do not expect ours to be the best possible solution to this problem, merely the first. Nevertheless, we outline some expectations for our library below.

We would expect any such library to be type-safe and compositional. That is to say, we should expect our library to catch nonsensical data types as early as possible. A strong type system would help catch incompatible data types at time of compilation, which would assist us in this area. Additionally, we would like our library to produce output even if it is given nonsensical, though type-safe, input. Lastly, making the library compositional, in addition to the other qualities of the library, would increase the usability of the library by ensuring that computational expressions of arbitrary length can be computed from a small set of functions.

4.2 Hypothesis

We are not trying to implement a particular conceptual space. Rather, we are trying to give an end user the ability to not only construct their own conceptual spaces, but leverage these in their own programs. To do so, we must avoid general mistakes that result in a poor software library.

Specifically, should the library be given invalid data (in whatever form), the library should not cause the program to halt unexpectedly. That is to say our library should be *type-safe*. Furthermore, when the user calls a function from the library, they should eventually receive a result. That is to say, with the exception of being given infinite data, the library should *terminate*.

Finally, as we are basing our system on the work described in [22] and that paper involved some sample calculations, we must be able to reproduce these calculations and their results. That is to say the library should be *verifiable*.

Our hypothesis is this: using Haskell [13], a strongly-typed purely functional programming language, it is possible to construct a general purpose Conceptual Spaces library that has provable behaviours including type-safety, verifiability and termination.

4.3 Objectives

The main objective of our research is to build a library of data structures and operators. Within this objective are a number of smaller goals. Firstly, we want our data structures to match the restrictions on them as outlined in the literature. Secondly, we want our operators to be type-safe and verifiable. Lastly, we want our operators to terminate.

With these specifications in mind, our objectives are as follows:

- Build a library of data structures and operators.
- Demonstrate the type-safety and verifiability of the operators.
- Show that the operators terminate.

4.4 Research Methodology

To complete these objectives,

- we used the Haskell programming language to construct our library (Chapter 5),
- we showed how Haskell's use of strong typing allows for our library to reject nonsensical observations, domains and concepts (Section 6.1),
- we showed termination of the operators via structural induction on the size of the input context (Section 6.2),
- we demonstrated the verifiability of the operators by repeating a selection of examples shown in the literature (Section 6.3).

The library was developed using the Glasgow Haskell Compilation System [28], version 8.0.1.

The next chapter will introduce the Conceptual Reasoning Library in Haskell and discuss some of the software design choices made. The proofs and discussion mentioned earlier will appear in Chapter 6, including a discussion on how the use of Haskell's Maybe type allows us to process incompatible concepts, contexts and observations.

Chapter 5

Conceptual Reasoning Library in Haskell

In this chapter we introduce the Conceptual Reasoning Library in Haskell, CRLH. We begin by describing our data structures and operators. From there, we discuss the various design choices made when constructing CRLH. Finally we discuss the limitations of our library.

5.1 Data Structures

CRLH includes data structures representing domains, properties, observations, concepts and contexts.

In designing our library, we decided that all data points would be encoded as strings of characters. Because conceptual data can include words as well as numbers we found that this choice provided us with a more general method of data representation. We note that

in some ways this limits the extent to which the membership function of a property can be defined. We will discuss this further in Section 5.3.

5.1.1 Domain

Recall from Section 2.2.1 that a domain is composed of a (non-empty) set of dimensions and a (non-empty) set of properties. With this in mind, we designed our domain data structure, which can be found in Appendix A.1. This data structure is represented as a 3-tuple $D = \{N, M, P\}$, where for a given domain D_1, $N(D_1)$ is the domain's name, represented by a String, $M(D_1)$ is the domain's set of dimensions, represented by Strings, and $P(D_1)$ is the domain's set of Properties. We note that while the original description from Section 2.2.1 did not mention a name, we felt that including it would help the end user keep track of their conceptual model.

We chose to use strings to represent dimensions as the dimensions themselves are only used for identifying data points and strings allow us to name the segment a given data point exists on. We wish to use Haskell's record syntax as it simplified the construction of our library. As a result, in this constructor we cannot enforce the requirement that the sets of dimensions and properties are non-empty.

In an effort to handle this shortfall, we designed a secondary constructor. This constructor returns a Domain type (that is, the type returned by the original constructor) but checks to make sure the properties and dimensions exist. As we have the additional field of a domain name, we chose to return an error message to the user should they supply invalid data. As the domain name is not used in any computations and is only included for pro-

grammer convenience, we do not consider an unnamed domain to be malformed. If either the dimensions or properties are missing, we return an empty Domain. We also return an empty Domain in the case that there is a malformed property in the list. We will define a malformed property in the following subsection. If the supplied lists of dimensions and properties pass these two requirements we return a (well-formed) Domain.

5.1.2 Property

In defining a property we again return to Section 2.2.1. A property is a fuzzy subset on a domain, but it is represented as a function from the domain to the codomain [0,1]. Due to the difficult nature of enforcing the restriction on the codomain we have chosen to implement a property as a collection of mappings from values on a selection of the dimensions of the property's domain to the codomain [0,1].

A property is represented as a tuple $P = \{N, A\}$, where for a given property P_1, $N(P_1)$ is the property's name and $A(P_1)$ represents a map from values on various dimensions to a floating point number representing its association value to that data point. We note that while a property exists on a domain, we found that in implementing these data structures we could define this relation as either a member of the domain data structure or the property data structure. For reasons that we will discuss in the next subsection, we chose to this relationship in the domain data structure.

Recall that in our description of the domain data structure we discussed how our use of Haskell's record syntax resulted in our only being able to define one constructor. The same problem arose here and we used the same strategy (the design and construction of

a secondary constructor). Our constructor takes as arguments a string representing the name of the property and a list of tuples, which we will refer to as $[((a,b),c)]$. The first element of these tuples is itself a tuple, where the elements, a and b, are Strings. The value a represents the dimension that the value b exists on. The element c is a Float, which represents the extent to which the value b, on that dimension a, indicates similarity to the property.

Our property constructor checks for two possible failures:

- Firstly, if there are no mappings, the property is undefined and an empty property is returned.
- Secondly, as property similarity values are on the interval [0,1], we should not allow values over 1 or below 0.

In either case, an empty Property (with an appropriate error message) is returned. Otherwise, a well-formed Property is returned.

5.1.3 Observation

Recall from section 2.2.1 that an observation is a collection of points on various domains. From sections 5.1.1 and 5.1.2 we recall that we chose to represent the relationship between properties and domains on the domain data structure. We did this because doing so means that when we want to compare an observation to the potentially relevant properties, we can access a complete set through the domains in which the observation's data points exist.

Our observation data structure can be considered a tuple $o = \{D,T\}$ where, for a given observation o_1, $D(o_1)$ represents the set of domains on which the observation's data exists

and $T(o_1)$ represents the data points on those domains. We represent these data points in a similar fashion to their representation in our Property data structure in that we pair the dimension and value together in a tuple. The set of data points of an observation on a domain is represented as $(Domain, [(Dimension, Value)])$.

In implementing this data structure we used Haskell's record syntax for the sake of uniformity. Furthermore, we developed a secondary constructor to ensure a few conditions for an observation. While only one of these conditions appears in the literature (and only implicitly at that), we feel that the other condition is necessary for a coherent conceptual system.

Our observation constructor checks to make sure of two things:

- Firstly, we check the existence of data points.

  If there are no data points, this observation would not match the definition that exists in the literature. [22]

- Secondly, we make sure that for each data point, each dimension is on the listed domain.

  This condition is *not* found in the literature, but it follows logically that if an observation references a dimension that does not exist, we will have an inconsistent conceptual space.

If either of these two conditions fails, we return an empty observation.

5.1.4 Context

As the definition of a context is so simple, we only mention briefly here that our context data structure is a tuple $G = \{N, P\}$, where, for a given context G_1, $N(G_1)$ is the context's name and $P(G_1)$ is the set of properties that represents the context. The name, as with Properties and Domains, is added for programmer reference. The notion of a set of properties is taken directly from the literature [22].

Arguably, we did not need to design a secondary constructor for our context type, as the only condition that would make a context invalid is when the context's set of properties is empty, which is the style we had been using for our other data structures so far. However, we felt that by exposing a secondary constructor we would make our library more uniform in its presentation to the user.

Our context constructor takes a name, represented again by a String, and a list of properties. The constructor then checks for one of two possible failure states: either the list of properties is empty, or the list contains an invalid property. While the user cannot create an invalid property using our (secondary) constructor, we cannot stop them from using the primary constructor in an undesirable way. To check that the properties are valid, we ensure that each property's map is not empty, using the *Map.null* function from Haskell's Map library.

5.1.5 Concept

When designing our concept data structure we recall again from Section 2.2.1 that a concept is composed of a set of properties, a set of subconcepts and a membership function

from pairs of those properties to the interval [0,1]. We define a concept C to be a 4-tuple $\{N, I, S, M\}$, where, for a given concept C_1, $N(C_1)$ is the name of the concept, $I(C_1)$ is the set of properties, $S(C_1)$ is the set of subconcepts and $M(C_1)$ is a map from $I(C_1) \times I(C_1)$ to $[0,1]$. In order to simplify our notation, we use Rickard *et al*'s notation for describing elements of the map as C^1_{ab} to represent the association between properties a and b. Much like our Property data structure, we chose to emulate this function using Haskell's *Map* data structure, for the same reasons. While Rickard *et al* do not specifically list a set of subconcepts as being part of a concept, for a given concept C we need to keep track of all concepts that are smaller than C. Recall from [22] the definition of a smaller concept:

> A concept C' is *smaller* than C if $I(C') \subset I(C)$ and $C'_{ab} = C_{ab} \forall a, b \in I(C')$

This definition allows us to incorporate a subconcept as a context applied to a concept; that is, a concept that has been filtered through the smaller set of properties in a subconcept. The drawback associated with this implementation is that comparing the individual subconcepts of a concept becomes impractical. Our suggestion to overcome this problem is to define those subconcepts as their own concept.

There are a number of conditions that must be satisfied for a concept to be well-formed.

- From the literature, $\forall_{a,b \in I(C)} C_{ab} = 0 \iff C_{ba} = 0$.

  This is to say that a zero association between two properties must hold for both properties.

- $\forall_{a \in I(C)} C_{aa} \neq 0$.

This is to say that a property's self-association must not be equal to 0. If a property's self-association was equal to 0, that property would not be in that concept.

- We must also ensure that the given subconcepts are valid. A subconcept is valid if and only if its property set is a subset of the larger concept.

To check the first condition, we filter the map for any key-value pairs for which the value is 0, and check the opposite key (ie, if $C_{ab} = 0$, check C_{ba}). We place all such values in a list and add them together. If the sum is not zero, we know that there exists some $C_{a,b}|C_{a,b} = 0 \wedge C_{b,a} \neq 0$, which means that this concept is not well-formed. In this case, an empty concept is returned.

To check the second condition, we place all diagonal values (that is, $C_{a,a} \forall a \in I(C)$) in a list and use Haskell's *minimum* function (which returns the minimum element of a list) to ensure that all values are above 0. If any one value is 0, the concept is not well-formed, and an empty concept is returned.

The third condition requires us to check that all subconcepts of the potential concept are valid. If there are no provided subconcepts, we have nothing to check. However if a concept has subconcepts, we must check that all subconcepts' property sets are subsets of the concept's property set. Should any one of these subconcepts fail this condition, an empty concept is returned.

Our decision to represent subconcepts as contexts makes checking the third condition significantly simpler; since a subconcept uses a reduced map of its parent concept, we do not have to check that $C_{ab} = C'_{ab} \forall a, b \in I(C')$.

5.1.6 Summary

In this section we have detailed the data structures in CRLH and discussed the secondary constructors we designed. We mentioned several times that should the user fail to provide valid data, an empty data structure would be returned. In our next section, we will describe our operators and how they handle these empty data structures.

5.2 Operators

Our library includes operators for measuring the similarity between a concept and an observation, between two concepts and an operator for determining observation property values. There are a number of supporting functions in our library, including a function used by the Concept-to-Concept operator for calculating property overlap in cases where the sets of properties contained by two concepts are not equal.

We will start with the Property-Observation measure, which determines the degree to which an observation is an example of all properties of each domain on which the observation exists. In these descriptions we will refer to the composite parts of the tuples used to describe our data structures in Section 5.1.

5.2.1 Property-Observation measure

This operator takes an observation and returns a map from each property that observation represents to the number on the interval [0,1] representing the degree to which the observation has that property.

This multi-stage process is described as follows: We pass an observation to the *proper-*

Algorithm 1 Property-Observation measure

1: **procedure** PROPERTYRECOGNITION(a) ▷ a is an observation
2: **if** size ($D(a)$) == 0 **then**
3: **return** *an empty map* ▷ Empty observation
4: **else**
5: **return** *Map.fromList* $[x|y \leftarrow T(a),$ $x \leftarrow$ (*propertyRecognition' y*)] ▷ A map from all relevant properties to their similarity value.
6: **end if**
7: **end procedure**

tyRecognition function (outlined in Algorithm 1) and start by checking how many domains the observation was recorded on. If that is 0, then either the observation was not properly formed, or nothing was observed. In either case we return an empty map, as seen on line 3 in Algorithm 1.

Algorithm 2 Property Value Regulator

1: **procedure** PROPERTYRECOGNITION'((a,b)) ▷ a is a domain, b is a list of values on various dimensions in the domain a.
2: **return** $[(x,y)|x \leftarrow P(a),$ ▷ x is the set of properties of the supplied domain
3: $y \leftarrow [\min\ (\sum$ *propertyVal* $(A(x), T(b))), 1]]$ ▷ y represents the value that the provided data points in b have on the property x
4: **end procedure** ▷ Returns a list of properties and the associated similarity values

Otherwise, we pass each domain's list of observation points to Algorithm 2. Algorithm 1 makes reference to two functions, *first* and *second*. Both of these functions take a tuple. The function *first* returns the first element in the tuple, while *second* returns the second. On Algorithm 2 line 2, we get a full list of the domain's properties. Line 3 uses list comprehension tools (on the list of properties) to pass the entire list of data points on that domain and each property (individually) to Algorithm 3. That algorithm, which we will describe shortly, returns a list. We total the list and return the lesser of that total and 1. This ensures that no observation has a similarity higher than 1 to any given property.

Algorithm 3 Property Value Calculator

```
1: procedure PROPERTYVAL(a, b)   ▷ a is a property's map, b is a list of data points on that property's domain.
2:     if b is empty then
3:         return an empty list ▷ This shouldn't ever actually happen; we include this for safety
4:     else
5:         return [x|y ← b,                                  ▷ y contains a data point from b
           x ←[(Map.lookup y a)]]                            ▷ Look up point y in the map a
6:     end if
7: end procedure
```

Algorithm 3 takes a property's map and an observation's set of data points on the property's domain and checks each recorded instance of the observation against the provided property. In the (unlikely) case that the observation has no data points on that domain, we return an empty list. Otherwise, we use list comprehension tools to look up each data point in the provided property map, placing all results in a list. If a data point is not in the map, the value 0 is put into the list. This list is then returned.

This operator is called from the Observation-Concept measure, which we will describe next.

5.2.2 Observation-Concept measure

When comparing an observation to a concept, there may or may not be a context involved. As a result, we have two observation-concept measures. We will begin by describing the context-sensitive measure.

Algorithm 4 takes as arguments a concept, an observation and a context. Its return type is *Maybe Float*, which means it may return a Float, or it may return *Nothing*. In section 5.1, we discussed constructors that checked for invalid data. Algorithm 4 was designed with

Algorithm 4 Context-Sensitive Observation-Concept Measure

1: **procedure** CONCEPTTOOBSWITHCONTEXTCALC(a, b, c) ▷ Compare observation b to concept a with context c.
2: **if** size $(P(c)) == 0$ **then** ▷ Check for an empty context
3: **return** *conceptToObsCalc*(a, b) ▷ Perform a context-free evaluation.
4: **else if** size $(P(a)) == 0$ **then** ▷ Check for an empty concept
5: **return** Nothing
6: **else if** size $(D(b)) == 0$ **then** ▷ Check for an empty observation
7: **return** Nothing
8: **else if** $c \in S(a)$ **then** ▷ Squeeze the concept down to its subconcept
9: **return** *conceptToObsCalc*(x, b), where x is a concept formed from a's property association maps reduced to the context c's property set and contains no subconcepts.
10: **else** ▷ Squeeze the concept down to fit in the context
11: **return** *conceptToObsCalc*(y, b), where y is a concept formed from a's property association maps reduced to the context c's property set and contains filtered subconcepts.
12: **end if**
13: **end procedure**

invalid data handling in mind. Line 2 first checks to see if the supplied context is empty. Recall that equation 2.5 is undefined when $I(C') \cap G = \emptyset$. Because this result cannot be computed, we assume that the context was malformed. In the case of a malformed context, we return the similarity value of a context-free measure on the provided observation and concept. Line 4 checks to make sure that the concept has properties. If it does not, we return Nothing, as the result would be undefined. Line 6 checks to make sure that the observation has any data. If it does not, we return Nothing. We chose to make this choice because while equation 2.5 would arguably return 0, we feel that comparing an observation to a concept it has nothing in common with is fundamentally different than comparing an empty observation to that same concept. If the provided data passes all of these conditions, we move on to handling valid cases.

Line 8 checks whether the provided context c is a subconcept of a. If so, we perform a context-free similarity measure between the provided observation b and a new concept,

created by filtering out from the provided concept's map any keys which contain properties not found in the property set of the provided context c. This new concept will not contain any subconcepts, as we recall from the definition in If c is not a subconcept of a, we again perform a context-free similarity measure, this time between the provided observation b and a new concept, created similar to the method described above, with one main difference: this new concept has a set of subconcepts which consists of any subconcepts from the original concept whose property sets are subsets of the property set of the provided context c.

As we have now described Algorithm 4, we will now proceed to outline and discuss its context-free version below.

Algorithm 5 Context-Free Observation-Concept Measure

1: **procedure** CONCEPTTOOBSCALC(a, b) ▷ Compare observation b to concept a.
2: **if** size $(P(a)) == 0$ **then** ▷ Check for an empty concept
3: **return** Nothing
4: **else if** size $(D(b)) == 0$ **then** ▷ Check for an empty observation
5: **return** Nothing
6: **else if** size $S(a) > 0$ **then**
7: **return** max$(conceptToObsWithContextCalc(a, b, x))$ ▷ $x \in S(a)$.
8: **else**
9: **return** Just $(\frac{\sum calcMin(a,b)}{\sum Map.elems(M(a))})$▷ *Map.elems* returns the values (or elements) of a map, in ascending order of their keys.
10: **end if**
11: **end procedure**

Algorithm 5 is used to perform context-free concept-to-observation similarity measures. Similar to algorithm 4, we make some checks to ensure our supplied concept and observation are valid data structures. Algorithm 5 line 2 checks to see if the provided concept contains any properties. If it does not, we have a malformed concept, in which case we return Nothing. Line 4 checks to see if the supplied observation is well-formed by checking

if it exists on any domains. If it does not, we have an empty observation and so we again return Nothing. At this point, we have checked for all possible cases of invalid data and are finally ready to begin calculating a similarity value.

Line 6 checks for the existence of subconcepts in the provided concept. In the case that the supplied concept contains at least one subconcept, we pass the observation, concept and that subconcept to the context-sensitive similarity function, returning the maximal similarity value of all subconcepts. If there are no subconcepts, line 9 calculates and returns the similarity value of the supplied observation to the supplied concept. This line calls the function *calcMin*, which we will discuss below. Once that function has returned, we total the list and divide it by the sum of the values in the provided concept. We wrap this result in the *Just* constructor and return it to the user.

Algorithm 6 Minimum Concept-Observation calculator

1: **procedure** CALCMIN(a, b) ▷ Given a concept a and an observation b, return the minimum value for each pair of properties in a.
2: **return** *zipWith* min (*Map.elems* $M(a)$) (*calcprop* (*Map.keys* $M(a)$) b)
3: **end procedure**

Algorithm 6 is used to collect the minimum value of each pair of properties in the supplied concept. This algorithm makes use of Haskell's *zipWith* function, which takes a function and two lists. It returns one list, composed of the output of the provided function applied to each pair of elements in the list. In *calcMin*, it takes two lists of numbers and returns a list containing the minimum element at each place in the provided lists, ie. given the lists $[0,1,2]$ and $[1,0,3]$, it returns the list $[0,0,2]$. Should it be given lists of different lengths, the length of the returned list is equal to the shorter of the two provided lists. The two other Haskell functions of note are *Map.keys* and *Map.elems*. *Map.keys* takes a

map and returns its keys in ascending order, while *Map.elems* takes a map and returns its elements (or values) in ascending order of its keys.

Algorithm 7 Minimum Property-Observation calculator

```
1: procedure CALCPROP(a, b)   ▷ Given a list of pairs of properties a and an observation
   b, return a list of the minimum property similarity values
2:     return [x|y ← a,
          x ← [min (Map.lookup (first y) c) (Map.lookup (second y) c)]]
3: where c = propertyRecognition b
4: end procedure
```

The *calcprop* function, shown in Algorithm 7, takes a list of pairs of properties and an observation. Using the property-observation similarity measure described in section 5.2.1, it returns a list of numbers between [0,1], corresponding to the lesser of the two property's similarity values. If the observation has no similarity with a given property, 0 is used.

We also note the use of Haskell's *where* clause on line 3. Because the *propertyRecognition* function can be computationally trivial, we want to avoid calling it needlessly. By using Haskell's *where* clause, we can ensure that the function only gets called once.

Having covered our observation-concept measure, we will now proceed to cover the concept-concept similarity measure.

5.2.3 Concept-Concept measure

When designing our concept-concept similarity measure, we chose to use a design strategy similar to that of the concept-observation similarity measure outlined in Section 5.2.2. Because we chose to use this similar style, we have two concept-concept measures: one that is context-free and one that is context-sensitive. We will begin by describing the context-sensitive measure.

Algorithm 8 Context-Sensitive Concept-Concept Measure

1: **procedure** CONCEPTTOCONCEPTWITHCONTEXTCALC(a, b, c) ▷ Compare concepts a and b in context c
2: **if** c has no properties **then** ▷ Check for an empty context
3: **return** *conceptToConceptCalc*(a, b)
4: **else if** a or b has no properties **then** ▷ Check for an empty concept
5: **return** Nothing
6: **else** ▷ Squeeze the concepts down to fit in the context
7: **return** *conceptToConceptCalc*(x, y) ▷ x and y are concepts derived from a and b, respectively.
8: **end if**
9: **where** $v \leftarrow ((P(a) \cup P(b)) \setminus P(c))$,
$w \leftarrow ((P(a) \cup P(b)) \setminus P(c))$,
$x = Concept\ \{N(a), (P(a) \cap P(c)), S(a), (Map.filter\ (v,w)\ M(a))\}$
$y = Concept\ \{N(b), (P(b) \cap P(c)), S(b), (Map.filter\ (v,w)\ M(b))\}$
10: **end procedure**

The context-sensitive concept-concept measure is outlined in Algorithm 8. Like our observation-concept measure, the concept-to-concept similarity operator checks for invalid data types. Line 2 checks for an empty context. If the context is empty, it returns a context-free evaluation. Our reasoning for not accepting empty concepts in this operator is the same as it was for the context-sensitive observation-concept measure. Line 4 checks that both concepts have properties. If either concept is empty, it returns Nothing. Otherwise, we create new concepts by using the supplied context as a filter for the supplied concepts' maps. We then return a context-free similarity calculation on these new concepts.

Much like the observation-concept measure, all concept-concept measures eventually become context-free, so we will examine that measure next.

We begin our Context-Free Concept-Concept Measure, as outlined in Algorithm 9, by ensuring our concepts are non-empty. This condition is checked on line 2. If either concept is empty, we return Nothing. Otherwise, we create lists from the concepts a and b, which we call z and y, respectively. These lists contain the property relevance values for each pair

Algorithm 9 Context-Free Concept-Concept Measure

procedure CONCEPTTOCONCEPTCALC(a, b) ▷ Compare concept a to concept b
  if a or b has no properties **then** ▷ Check for an empty concept
    return Nothing
  else ▷ Perform the context-free operation
    return Just $(\frac{\sum zipWith \min(z,y)}{\sum zipWith \max(z,y)})$ ▷ z and y are lists formed from a and b, respectively.
  end if
  where $c \leftarrow (P(a) \cup P(b))$,
    $d \leftarrow (P(a) \cup P(b))$,
    $z = [x|x \leftarrow Map.findWithDefault(ifNeeded\ (a,c,d))(c,d)(M(a))]$
    $y = [x|x \leftarrow Map.findWithDefault(ifNeeded\ (b,c,d))(c,d)(M(b))]$
end procedure

of properties in the union of the two concepts' property sets. Should any pair of properties be undefined in either of the concepts, the function *ifNeeded* is called. This function is our implementation of Equation 2.2, using the discrete form of Equation 2.3. We have also included a version of this algorithm that does not call *ifNeeded*. Should a pair of properties not exist in either concept, 0 is returned. Having created these lists, we again use Haskell's *zipWith* function, dividing the minimum values over the maximum values for each list. This quotient is then wrapped in the *Just* constructor and returned. As the *ifNeeded* function is a critical component of the concept-concept similarity measure, we will examine it next.

Algorithm 10 Conceptual Property overlap measure

procedure IFNEEDED(a, b, c) ▷ Calculate the property overlap of properties (b, c) in concept a
  if $(c,d) \in Map.keys\ (M(a))$ **then**
    return $(Map.findWithDefault\ 0\ (b,c)\ M(a))$ ▷ We don't need to calculate the overlap in this case.
  else
    return $z * v * Map.findWithDefault\ 0\ (y,w)\ (M(a))$ ▷ Equation 2.2
  end if
  where $(y,z) = propertyOverlap\ (b, P(a))$
    $(w,v) = propertyOverlap\ (c, P(a))$
end procedure

Algorithm 10 is used to calculate any missing conceptual property values when comparing concepts. Line 2 checks to make sure the property pair is not in the concept to begin with. If it is, we return that pair's value. While this adds a slight computational overhead to the operator, it allows the user to make use of it on its own. If the property pair is not found in the concept, line 5 calculates and return the property overlap value. The *propertyOverlap* function takes a property and a set of properties and returns a tuple in which the first element is a property and the second element is a Float. This result represents the property in the provided set which overlaps most with the supplied property and its overlap value. This is to say, given a property d and a property set $[a,b,c]$, the function would return $(b,0.75)$, for example. Using this, we can say that $second\ z \equiv B_{bb*}$ and $second\ y \equiv B_{cc*}$ from equation 2.2.

5.2.4 Summary

This concludes our discussion on the operators of our library. While we intend for end users to use our type-safe constructors, we cannot force them to, so we developed several guards to ensure invalid data structures are caught by the operators. There are also situations in which using our operators can result in invalid data structures. For example, if a user tried to perform a context-sensitive concept-concept similarity measure in which one of the concepts and the context's property sets were mutually exclusive, our library would pass an empty concept to the context-free operator, which would return Nothing. Having finished discussing the operators, we will now describe briefly some limitations of our library and then conclude the chapter.

5.3 Design Considerations

When designing our library one of the most important aspects to consider was the format we would use in the representation of data. Taking inspiration from Gärdenfors' quality dimensions [7], we wanted to attach some form of meaning to the data, even if only at a representational level. We also wanted to include as many different kinds of data as we could, including textual and numerical data.

As we mentioned earlier, in designing our library we decided to encode all data points in language, represented as strings of characters. This choice was made to allow not only numerical but also textual data to be reasoned about. Rickard's earlier work described data as points in an n-ary space: "Objects in a conceptual space are represented by points, in each domain, that characterize their dimensional values" [21]. We do the same, but like his later combined work, we abandon the strict geometric viewpoint [22] and incorporate the idea of data points being represented by combinations of descriptive phrases.

By representing data as strings we allow for a greater range of concepts and ideas to be reasoned about than if we had limited our library to a numerical form of representation. We accept, however, that this approach is not without its limitations, which we discuss in Section 5.4.

5.4 Limitations

Our library is limited in that the theory of conceptual spaces, properties are membership functions and our implementation of a property is a function in the sense that we have a domain, a codomain and a set of assignments mapping elements from the domain to the

codomain [16].

However, our properties function as somewhat of a black box, in that we cannot define a rule (or set of rules) to manipulate elements of the domain into elements of the codomain. For example, we do not support defining a property function as $f(x) = \frac{1}{3x}$. This design choice was made because of the difficulty in verifying the codomain to be [0,1]. However, we find that should a user have such a function, it would be relatively simple to form a list similar to the type required by our property constructor.

Our choice to represent data as descriptive phrases introduces limitations in our library as well. Even if we could verify the codomain for property functions like those described above, such definitions would not be useful when the domain of such a function is text. Moreover, this library is limited in that should descriptive phrases be found as an inadequate measure for some phenomenon, the library itself would not be useful.

5.5 Summary

In this chapter we introduced CRLH, a library for conceptual reasoning in Haskell. We described our selection of data structures, constructors and operators, while also describing some limitations of our library.

While we have done our best to represent the operators and data structures found in the literature, we accept that there may be errors in our work. Any mistakes found are the sole responsibility of the author. The task of implementing mental objects is not a simple one; we refer to what C.A.R. Hoare said in his 1980 Turing Award lecture [10]:

> I conclude that there are two ways of constructing a software design: One way

> is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

In the next chapter, we will provide an analysis of the type-safety of our library, prove the termination of our operators, and demonstrate the verifiability of CRLH by repeating a selection of example calculations found in the literature.

Chapter 6

Analysis of Results

In this chapter we demonstrate the type-safety of the library, show proofs of termination for our operators and demonstrate the verifiability of the operators by repeating some of the examples shown in the literature.

6.1 Demonstration of Type-Safety

As a function of our exposed constructors and Haskell's strong-typing, we limit the ability for users to create improper instances of our data types. Furthermore, by using Haskell's Maybe type, we are able to ensure that CRLH will report as few errors at run time as possible. If there are to be errors, we would rather they be caught at compile time.

We consider the constructors of our data types to be type-safe as they always return their data type regardless of whether or not they were given valid data. We will show this for all constructors here.

6.1.1 Property

We begin with the *Property* type. To be a valid type, a *Property* needs only a collection of mappings from a point on a dimension to a value between 0 and 1. While a Property's dimensions must all be members of the Property's *Domain*, we cannot enforce this in this constructor. Instead, that is left for the Domain constructor. The property's constructor, seen below, checks:

- The existence of these mappings and
- The values that are mapped to are no larger than 1.

The first condition is checked using Haskell's *length* function, which returns the length of a given list. If the length of the list is 0, then there are no mappings and an empty Property is returned. The second condition is checked through the use of Haskell's *maximum, snd* and *unzip* functions. *Unzip* takes a list of tuples and returns a tuple of lists, *snd* takes a tuple and returns the second element and *maximum* returns the maximum element of a list. The list of mappings is processed through this chain of functions to determine the maximum mapped value. If that value is greater than 1, the constructor returns an empty Property. Otherwise, a (relatively) well-formed property is returned.

```
property :: String → [((String,String),Float)] → Types.Property
property a b
    | length b == 0 = (Types.Property "Error, no mappings!" Map.empty)
    | maximum (snd (unzip b)) > 1 = (Types.Property "Error, invalid mappings!" Map.empty)
    | otherwise = (Types.Property a (Map.fromList b))
```

6.1.2 Domain

Our *Domain* type needs the following things to be valid:

- A collection of dimensions and
- A collection of (well-formed) properties.

For programmer convenience we have also included a name, but that is not strictly required. Our *domain* constructor, seen below, receives these two collections as *lists*, and using Haskell's *length* function, ensures they are non-empty. In the case that the list of dimensions or the list of properties is empty, a *Domain* is returned, with an error message in place of the domain's name. Further to this, we check that all properties are valid by ensuring that their maps are non-empty and all dimensions listed in those maps are also elements of the Domain.

In the case that both of those lists are non-empty and the properties are valid, a well-formed *Domain* type is returned.

```
domain :: String → [String] → [Types.Property] → Types.Domain
domain a b c
    | length b == 0 = (Types.Domain "Error, No Dimensions" Set.empty Set.empty)
    | length c == 0 = (Types.Domain "Error, No Properties" Set.empty Set.empty)
    | checkProperties c = (Types.Domain "Error, Invalid Properties!" Set.empty Set.empty)
    | otherwise = (Types.Domain a (Set.fromList b) (Set.fromList c) )

checkProperties :: [String] → [Types.Property] → Bool
checkProperties a b = or [(or [x | y ← b, x ← [Map.null (propertyMaps y)]]),
(or [x | y ← b, z ← (Map.toList (propertyMaps y)), w ← [fst (z)], x ← [(fst w) `notElem` a]])]
```

6.1.3 Observation

Our Observation type contains a set of *Domains* on which the observation exists and a collection of values on the dimensions of those Domains. Our observation constructor, shown below, takes a list of tuples and returns an *Observation*. These tuples contain a domain and a list of tuples corresponding to the dimensions and values on those dimensions. To be a valid Observation, we must ensure that the list is non-empty, and that the dimensions listed

are on the associated domain.

As before, we check the first condition with Haskell's *length* function. If it returns 0, we have an invalid observation and an empty *Observation* is returned. If the list is non-empty, we pass it to the *checkObservation* function, which ensures that each listed dimension is in its associated domain. We built this on the idea that *if we can't trust all of it, we can't trust any of it*, so should any one data point fail this requirement, the entire observation is assumed to be invalid. In this case, we again return a well-formed, empty Observation. However, should the list pass both of those checks, a well-formed, valid Observation is returned.

```
observation :: [(Types.Domain,[(String,String)])] → Types.Observation
observation a
    | length a == 0 = Types.Observation (Set.empty) []
    | checkObservation a == False = Types.Observation (Set.empty) []
    | otherwise = Types.Observation (Set.fromList (fst (unzip a))) a

checkObservation :: [(Types.Domain,[(String,String)])] → Bool
checkObservation a = and [x |
    y ← a,
    w ← (snd y),
    z ← [(fst w)],
    x ← [Set.member z (dimensions (fst y))]
]
```

6.1.4 Context

Our Context data structure only requires a set of properties and so we only need to ensure two things when checking the validity of a context.

- We must check that the context has properties and,
- We must check that the properties are valid.

The constructor for our Context data structure is provided below. It takes as arguments a String representing its name and a list of properties. Much like the String used in our

Property and Domain data structures, it is included solely for programmer convenience. We again use Haskell's *length* function to make sure the list is non-empty; that is, to make sure its length is greater than 0. To check that the supplied properties are valid, we use the function *checkContextProperties*. This function ensures that all properties in the list are valid, that is, their maps are non-empty. If the data fails either of these two conditions, an empty Context is returned. However, if the supplied properties pass the two checks, we return a valid Context.

```
context :: String → [Types.Property] → Types.Context
context a b
    | length b == 0 = (Types.Context "Error, No Properties" Set.empty)
    | checkContextProperties b = (Types.Context "Error, Invalid Properties" Set.empty)
    | otherwise = (Types.Context a (Set.fromList b))

checkContextProperties :: [Types.Property] → Bool
checkContextProperties a = (or [x | y ← a, x ← [Map.null (propertyMaps y)]])
```

6.1.5 Concept

At its barest level, our *Concept* type needs a set of properties, and maps from pairs of those properties to a Float. A Concept can also have a set of subconcepts, which we represent as contexts. Much like our Context, Property and Domain data structures, we let the user name their Concept, should they so desire.

For a Concept to be valid, the mappings need to do more than simply exist, however. We use the notation $C_{a,b}$ to represent the membership value that the pair of properties a, b have in the concept C. The following conditions must be satisfied:

- If $C_{a,b} = 0$, then $C_{b,a} = 0$ for all $a, b \in C$
- For all $a \in C, C_{a,a} \neq 0$

Our constructor, shown below, also checks to make sure that any Contexts passed as subconcepts are valid for this Concept. That is, they make sure that those concepts' property sets are all subsets of the Concept's property set. The constructor itself takes a String, a list of contexts, and a list of tuples. The first element of these tuples is itself a tuple of properties, and the second element is a Float.

We use Haskell's *length* function to check if the supplied list has any mappings. If not, we return an empty concept. We wrote the function *checkConceptZeros* to check all zero-valued property pairs to ensure that they are consistent. If that function returns False, we return an empty concept, with an appropriate error message stored as the name of the concept. We also wrote a function called *checkConceptSubtypes* which ensures that all listed contexts are subsets of the Concept's property set.

```
concept :: String → [Types.Context] → [((Types.Property,Types.Property),Float)] → Types.Concept
concept a b c
    | length c == 0 = (Types.Concept "Error, no mappings!" Set.empty Set.empty Map.empty)
    | checkConceptZeros c == False = (
            Types.Concept
            "Error, inconsistent maps!"
            Set.empty
            Set.empty
            Map.empty
        )
    | checkConceptSubtypes b
        (
            Set.union
            (Set.fromList (fst (unzip (fst (unzip c)))))
            (Set.fromList (snd (unzip (fst (unzip c)))))
        ) == False = (Types.Concept "Error, inconsistent subconcepts!" Set.empty Set.empty Map.empty)
| otherwise = (
    Types.Concept
    a
        (Set.fromList b)
        (Set.union
            (Set.fromList (fst (unzip (fst (unzip c)))))
            (Set.fromList (snd (unzip (fst (unzip c)))))
        )
        (Map.fromList c)
    )

checkConceptZeros :: [((Types.Property,Types.Property),Float)] → Bool
checkConceptZeros a = and [
    0 == (sum
        [x |
        (y,z) ← Map.keys (Map.filter (== 0) (Map.fromList a)),
        x ← [(Map.findWithDefault 2 (z,y) (Map.fromList a))]]),
    0 /= (minimum
        [x |
```

```
        y ← Map.keys (Map.fromList a),(fst y) == (snd y),
        x ← [(Map.findWithDefault 2 y (Map.fromList a))]]
        )]

checkConceptSubtypes :: [Types.Context] → Set.Set Types.Property → Bool
checkConceptSubtypes a b = and [x |
    w ← a,
    y ← [Types.contextProps w],
    z ← [b],
    x ← [Set.isSubsetOf y z]
    ]
```

6.2 Proofs of Termination

We have shown our data structures to be type-safe and now we will show the operators to terminate, given these type-safe data structures. We will begin by showing that the operator which calculates the similarity of an observation to a property will terminate. Then we will show termination of our concept-observation similarity operator. Finally, we will examine our concept-to-concept similarity measure.

6.2.1 Property-Observation

In this subsection, we will show that the Property-Observation similarity operator terminates. We show this by showing that as long as the total number of properties times the total number of data points is smaller than the maximum size of a map in Haskell.

Postulate: The maximum size of a map in Haskell is given by *maxBound::Int*.

Lemma 1: The number of properties possible for a given observation is never larger than $N * P_{max}$, where N is the number of domains on which the observation exists and P_{max} is the number of properties on the domain with the most properties.

Proof: Trivial, but if it were larger, then the domain with the most properties wouldn't

be the domain with the most number of properties.

Lemma 2: For each data point in the observation, *propertyRecognition'* is called.

Proof: *propertyRecognition'* is called in a list comprehension in *propertyRecognition*. This list comprehension processes the list of data points in the provided observation.

Lemma 3: The number of calls to the function *propertyVal* = $P * D$, where P represents the total number of properties and D represents the total number of data points.

Proof: *propertyVal* is called in a list comprehension that lives in propertyRecognition'. This list comprehension runs once for each of the properties in the dimension of the current data point. Combined with Lemma 2, we know that *propertyRecognition'* is run once per data point.

Lemma 4: For a given property and observation, the function *propertyVal* returns a list no longer than R, where R represents the number of data points on that property's domain. Note: The list returned by *propertyVal* is length R but it is summed.

Theorem: The function *propertyRecognition* terminates (and returns a map) $\iff P * D < maxBound::Int$.

6.2.2 Concept-Observation

To prove termination of our context-free concept-observation similarity measure, we examine its three possible cases. Either an empty concept is provided, an empty observation is provided, or a non-empty concept and observation are provided.

Case 1: An empty concept is provided. In this case, the function returns *Nothing* and terminates.

Case 2: An empty observation is provided. In this case, the function returns *Nothing* and terminates.

Case 3: A non-empty observation and concept are provided. In this case, the function calls *calcMin*, one of our supporting functions. That function calls the function *calcprop*, which terminates if the observation-property similarity measure terminates. We have shown above that the observation-property measure terminates, therefore *calcprop* and *calcMin* terminate. What we are left with is a list of elements of length $I(C) \cap J$, where J is the set of properties that the observation represents. Now, since $|I(C) \cap J| \leq |I(C)|$, we will use mathematical induction on the number of properties in $I(C)$, which we will call $|P|$, to show that the function terminates.

Base case: $|P| = 1$: In this case, we have *conceptToObsCalc* $(C,o) = \frac{\sum[x]}{\sum[y]}$, for some $x, y \in I(C) \cap J$. Since the sum of a list of elements whose length 1 is just the element itself, the summations complete, then the division is performed and the function terminates.

Inductive Hypothesis: Assuming that *conceptToObsCalc* terminates for $|P| = k$, show that it terminates for $|P| = k+1$.

Inductive Proof: Since the length of $|P|$ is k, the list of elements in *conceptToObsCalc* is of length k^2. Therefore, since $|P| = k+1$, the lists are of length $(k+1)^2 = k^2 + 2k + 1$ elements. In this case, we have *conceptToObsCalc* $(C,o) = \frac{\sum A + \sum Y}{\sum B + \sum Z}$, where A and B are lists of length k^2 and Y and Z are lists of length $2k+1$. We know from our inductive hypothesis that summations terminate on lists of length k^2 (and shorter), that $2k \leq k^2$, $\forall k \geq 2$ and from our base case that the summation of a list of elements whose length 1 is just the element itself. From these statements, we know that the function terminates.

To prove termination of our context-sensitive concept-observation similarity measure,

we examine its four possible cases. Either an empty context is provided, an empty concept is provided, an empty observation is provided, or all required data is provided.

Case 1: An empty context is provided. In this case, the function performs a context-free similarity measure, returns the result and terminates.

Case 2: An empty concept is provided. In this case, the function returns *Nothing* and terminates.

Case 3: An empty observation is provided. In this case, the function returns *Nothing* and terminates.

Case 4: A non-empty context, concept and observation are provided. In this case, the function creates a new concept and performs a context-free evaluation. As we have shown above that the context-free similarity measure terminates in all cases, we can conclude that the context-sensitive similarity measure terminates in this case.

We have now shown that, in all cases, the Concept-Observation Similarity Measure terminates.

6.2.3 Concept-Concept

To prove termination of our context-free concept-concept similarity measure, we have two possible cases. Either we are provided with an empty concept, or we are provided with two non-empty concepts.

Case 1: We are provided with an empty concept. In this case, the function returns *Nothing* and terminates.

Case 2: We are provided with two non-empty concepts. In this case we will employ

mathematical induction on the number of properties in the union of each concept's property set. The operator calculates the formula $conceptToConceptCalc(C^1, C^2) = \frac{\sum_{a,b} \min\left(C^1_{ab}, C^2_{ab}\right)}{\sum_{a,b} \max\left(C^1_{ab}, C^2_{ab}\right)} \mid a, b \in (I(C^1) \cup I(C^2)) \cap G$. We will be performing induction on the size of $(I(C^1) \cup I(C^2)) \cap G$, which we will call $|P|$.

Base case: $|P| = 1$: In this case, we have $conceptToConceptCalc\ (C^1, C^2) = \frac{\sum[x]}{\sum[y]}$, for some $x, y \in (I(C^1) \cup I(C^2)) \cap G$. Since the sum of a list of elements whose length is 1 is just the element itself, *conceptToConceptCalc* terminates.

Inductive Hypothesis: Assuming that *conceptToConceptCalc* terminates for $|P| = k$, show that it terminates for $|P| = k + 1$.

Inductive Proof: Since the length of $|P|$ is k elements, the list of elements in the summations in *conceptToConceptCalc* is of length k^2. Therefore, if $|P|$ is now $k+1$, then we have lists of length $\{k+1\}^2 = k^2 + 2k + 1$ elements. As we are adding $2k+1$ elements to lists containing k^2 elements, we have $conceptToConceptCalc\ (C^1, C^2) = \frac{\sum A + \sum Y}{\sum B + \sum Z}$, where Y and Z are lists of length $2k+1$ and $length\ A = k^2$, $length\ B = k^2$. We know from our assumption that the summation terminates on lists of length k^2 (and smaller), that $2k \leq k^2$, $\forall k > 1$ and that lists of length 1 terminate (from our base case). From these statements, we know that the function terminates.

To prove termination of our context-sensitive concept-concept similarity measure, we examine its three cases. The function is provided with an empty context, the function is provided with an empty concept, or the function is provided with two non-empty concepts and a non-empty context.

Case 1: We are provided with an empty context. In this case, a context-free concept-concept evaluation is called. We have shown above that our context-free operator termi-

nates, therefore the context-sensitive operator terminates.

Case 2: We are provided with (at least) one empty concept. In this case, the function returns Nothing and terminates.

Case 3: We are provided with two non-empty concepts and a non-empty context. In this case, the function creates two new concepts and performs a context-free evaluation. As we have shown that the context-free similarity measure terminates, we can conclude that the context-sensitive measure terminates in this case.

We have now shown that, in all cases, the concept-concept similarity measure terminates.

6.3 Demonstration of Verifiability

To verify our library we repeat a selection of examples performed by Rickard *et al* [22]. Their examples discussed the travel routes of maritime vessels in differing weather conditions. The routes themselves can be seen in Figure 6.1. In this example, concept C^1 represents voyages from port A to port D in fair weather. When completing this journey, a ship could take route a, followed by either b or c and then d, in fair weather. In stormy weather, only only route f is usable, but since this concept does not include trips in stormy weather, that route is not included in C^1. Concept C^2 represents voyages from port A to port E in fair weather. The only routes that work for this origin-destination pair is a and e, so those are the only routes included in this concept.

As both concepts are based on journeys completed in fair weather, the *fw* property from the *Weather* domain is also included. An observation describes the specific routes as

followed by an individual ship on a particular journey. The matrices containing our sample data concepts are shown in Figure 6.2. We will go over the individual matrices as the need arises. We will perform two context-free similarity measures between an observation and a concept, as well as a similarity measure between concepts C^1 and C^2.

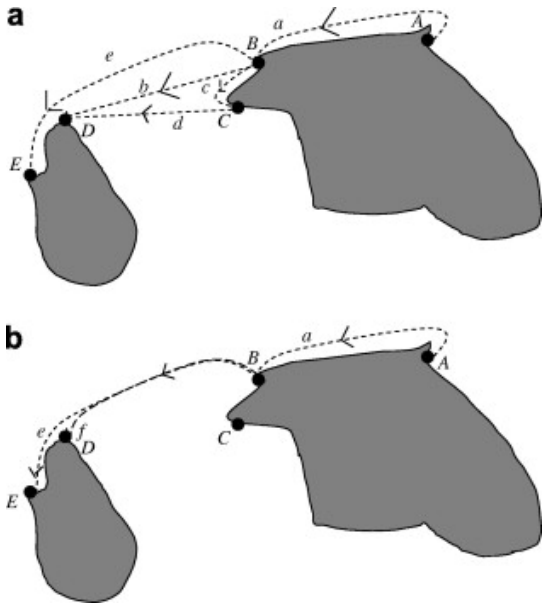

Figure 6.1: Routes between ports indicated for (a) fair conditions (b) and stormy conditions [22].

We begin by performing the context-free concept-concept similarity measure. In Figure 6.2, (a) represents context C^1 and (b) represents concept C^2. Using the list comprehension in *conceptToConceptCalc* and the function *ifNeeded*, we are able to reproduce the matrices (e) and (f), representing the derived associations for concepts C^1 and C^2. There are some minor variations between our output and their matrix; we attribute these differences to rounding errors or typos. We have coded the list comprehension as a separate function *deriveMatrix* in test.hs, see Appendix A.4.

In comparing these two concepts, the result from [22] is $2.87/11.14 = 0.26$[1]. Com-

[1]We are unable to obtain this result using either our library or our own calculations by hand.

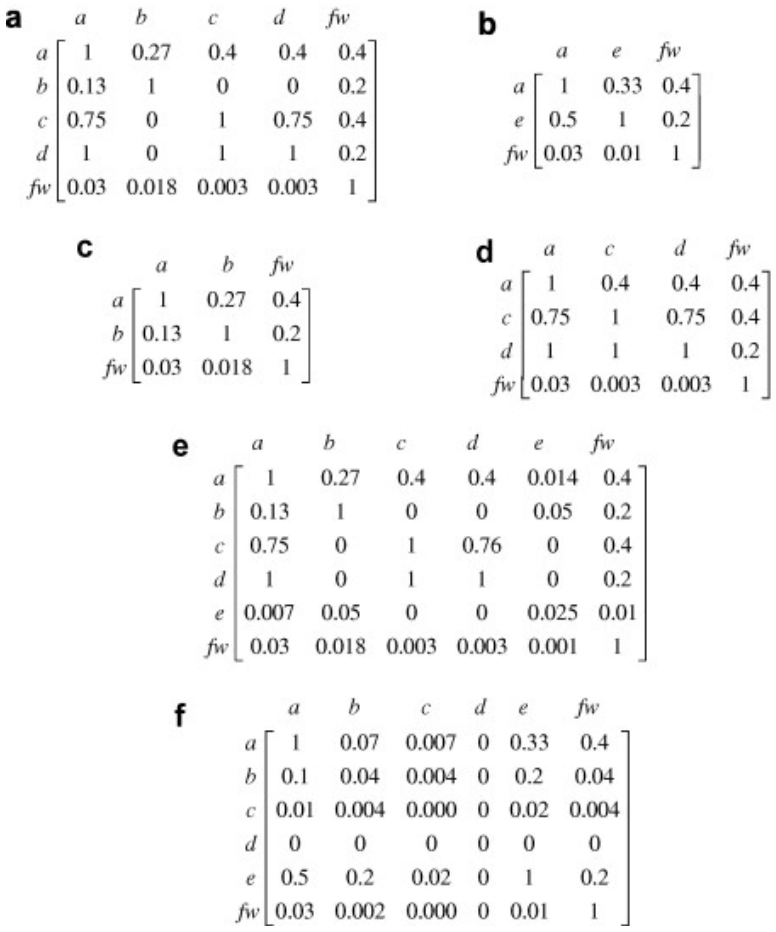

Figure 6.2: Matrix of associations involving concepts C^1 and C^2 [22].

pleting our calculation, CRLH produces the result **Just**[1] **0.21072756**. We recognize the difference, but are unable to explain the discrepancy between our results and theirs. We note what appear to be a number of typographical errors in their matrices, however even correcting for these differences does not assist us in reaching their result.

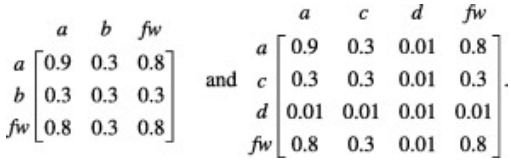

Figure 6.3: Matrix representation of observation **o** in both subconcepts of C^1 [22].

We move on to the concept-observation similarity measure. The observation is defined

[1] "Just" is the result of a successfully returned *Maybe* value in Haskell.

$$\begin{array}{c} \\ a \\ e \\ fw \end{array} \begin{array}{c} \begin{array}{ccc} a & e & fw \end{array} \\ \begin{bmatrix} 0.9 & 0.1 & 0.8 \\ 0.1 & 0.1 & 0.1 \\ 0.8 & 0.1 & 0.8 \end{bmatrix} \end{array}.$$

Figure 6.4: Matrix representation of observation **o** in concept C^2 [22].

as follows: "Suppose an observation **o** has membership 0.8 in fair weather fw, 0.8 in stormy, 0.9 in a, 0.3 in b, 0.3 in c, 0.01 in d, 0.1 in e and 0 otherwise" [22]. The observation can be seen in matrix form in Figure 6.3, for both subconcepts of C^1, and Figure 6.4, for concept C^2. In comparing the observation to the concepts, we consider that concept C^1 has two subconcepts, whose matrix representations are (c) and (d) in Figure 6.2. Concept C^2 has just one subconcept, represented by matrix (b).

In comparing the observation **o** to concept C^1, the result from [22] is $\max(3.05/4.05 = 0.75, 3.40/9.34 = 0.36) = 0.75$. We compare the observation **o** to concept C^1 using the context-free concept-observation similarity measure, *conceptToObsCalc*. We note, however, that because concept C^1 contains two subconcepts, the context-sensitive measure will also be used. From test.hs, the Haskell command used is *conceptToObsCalc conceptTest observationTest*. This returns **Just**[1] **0.75296444** which, rounding to 2 decimal places, is **0.75**, the same value as the one calculated in [22].

Lastly, we compare the observation **o** to concept C^2. In [22], the result was $2.54/4.47 =$ **0.57**. In our calculations we will use the context-free concept-observation measure as before. However, because concept C^2 contains only one subconcept, the context-sensitive measure will not be used. From test.hs, the Haskell command used is *conceptToObsCalc conceptTest2 observationTest*. This command returns **Just**[1] **0.5682326** which rounds to the same value calculated in [22]. Our results are summarized in Table 6.1.

[1] "Just" is the result of a successfully returned *Maybe* value in Haskell.

Table 6.1: Comparison of results

Operator	Result from [22]	CRLH Result	CRLH, Rounded
Concept-Concept	0.26	0.21072756	0.21
Concept-Observation (C^1, o)	0.75	0.75296444	0.75
Concept-Observation (C^2, o)	0.57	0.5682326	0.57

We note that as the property-observation measure was not explicitly defined by Rickard *et al*, we do not have any results with which to verify this operator. However, as the operator is used as a part of the concept-observation similarity measure, we can verify that it performs as expected.

6.4 Summary

We have demonstrated the type-safety of our library by showing that the provided constructors always return a valid data structure when given valid data, and an empty data structure when given invalid data. Having additionally shown termination and verifiability for a selection of operators, we will now conclude the thesis.

Chapter 7

Conclusion

In this chapter we conclude the thesis and describe areas of future research.

7.1 Summary of Thesis Findings

In this thesis, we have presented CRLH, the Conceptual Reasoning Library in Haskell, which implements the theory of Conceptual Spaces. Conceptual Spaces is a recent theory of concept representation [7] with a number of newly-suggested applications [34] in areas from computer vision [4] to linguistics [6].

In Chapter 5 we presented the Conceptual Reasoning Library in Haskell. Section 5.3 discussed the limitations of its current implementation. In Section 6.1 we demonstrated and discussed the type-safe constructors designed for each of our data structures. Section 6.2 presented a proof of termination for the major operators in our library. In Section 6.3 we demonstrated the verifiability of our library by repeating select calculations from the literature [22].

7.2 Future Work

In this section we discuss areas of research not covered by this thesis.

- Develop and implement a method of expressing uncertainty.

  Gärdenfors [7] required the dimensions of a domain to be integral with each other as discussed earlier. Aisbett and Gibbon [1] relaxed this requirement and made mention of using a point at infinity to represent an unknown or incompatible value. However, such a point cannot be easily implemented; another representation is required. We find that the lack of detail in this area allows for further research into applications in uncertain reasoning.

- Enhance our property constructor to allow users to specify a function rather than a series of mappings.

  As we discussed in Chapter 5, our library does not support defining a property's mapping function by taking a standard function, ie. $f(x) = \frac{1}{3x}$. We consider implementing this feature as something beyond the scope of this thesis. Furthermore, as connections between category theory [15], conceptual spaces and prototype theory [23] develop, new comparisons could require capabilities beyond those found in CRLH.

- Develop a reasoning engine to make use of our library.

  As the theory of conceptual spaces does not perform reasoning, but rather gives us something to reason *about*, our library would require a reasoning engine to actually perform automated reasoning.

Bibliography

[1] AISBETT, J., AND GIBBON, G. A general formulation of conceptual spaces as a meso level representation. *Artif. Intell. 133*, 1-2 (Dec. 2001), 189–232.

[2] AISBETT, J., RICKARD, J. T., AND GIBBON, G. *Conceptual Spaces and Computing with Words*. Springer International Publishing, Cham, 2015, pp. 123–139.

[3] BACKUS, J. Can programming be liberated from the von neumann style?: A functional style and its algebra of programs. *Commun. ACM 21*, 8 (Aug. 1978), 613–641.

[4] CHELLA, A., FRIXIONE, M., AND GAGLIO, S. Conceptual spaces for computer vision representations. *Artificial Intelligence Review 16*, 2 (2001), 137–152.

[5] CHURCH, A. A set of postulates for the foundation of logic. *Annals of Mathematics 33*, 2 (1932), 346–366.

[6] DESSALLES, J.-L. *From Conceptual Spaces to Predicates*. Springer International Publishing, Cham, 2015, pp. 17–31.

[7] GARDENFORS, P. *Conceptual spaces the geometry of thought*. MIT Press, Cambridge, Mass, 2000.

[8] GIRARD, J.-Y. The system f of variable types, fifteen years later. *Theoretical Computer Science 45* (1986), 159 – 192.

[9] HINDLEY, R. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society 146* (1969), 29–60.

[10] HOARE, C. A. R. The emperor's old clothes. *Commun. ACM 24*, 2 (Feb. 1981), 75–83.

[11] HUDAK, P. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv. 21*, 3 (Sept. 1989), 359–411.

[12] HUDAK, P., HUGHES, J., PEYTON JONES, S., AND WADLER, P. A history of haskell: Being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages* (New York, NY, USA, 2007), HOPL III, ACM, pp. 12–1–12–55.

[13] HUDAK, P., PEYTON JONES, S., WADLER, P., BOUTEL, B., FAIRBAIRN, J., FASEL, J., GUZMÁN, M. M., HAMMOND, K., HUGHES, J., JOHNSSON, T., KIEBURTZ, D., NIKHIL, R., PARTAIN, W., AND PETERSON, J. Report on the programming language haskell: A non-strict, purely functional language version 1.2. *SIGPLAN Not. 27*, 5 (May 1992), 1–164.

[14] JØSANG, A. A logic for uncertain probabilities. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems 9*, 03 (2001), 279–311.

[15] LAKOFF, G. *Women, Fire and Dangerous Things: What Categories Reveal About the Mind*. University of Chicago Press, Chicago, 1987.

[16] LAWVERE, F. W., AND SCHANUEL, S. *Conceptual mathematics : a first introduction to categories*. Cambridge University Press, Cambridge, UK New York, 2009.

[17] LIPOVACA, M. *Learn You a Haskell for Great Good!: A Beginner's Guide*, 1st ed. No Starch Press, San Francisco, CA, USA, 2011.

[18] MILNER, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17*, 3 (1978), 348 – 375.

[19] O'SULLIVAN, B., GOERZEN, J., AND STEWART, D. *Real World Haskell*, 1st ed. O'Reilly Media, Inc., 2008.

[20] REYNOLDS, J. C. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque Sur La Programmation* (London, UK, UK, 1974), Springer-Verlag, pp. 408–423.

[21] RICKARD, J. T. A concept geometry for conceptual spaces. *Fuzzy Optimization and Decision Making 5*, 4 (2006), 311–329.

[22] RICKARD, J. T., AISBETT, J., AND GIBBON, G. Reformulation of the theory of conceptual spaces. *Information Sciences 177*, 21 (2007), 4539 – 4565.

[23] ROSCH, E. Cognitive representations of semantic categories. *Journal of experimental psychology: General 104*, 3 (1975), 192.

[24] SCHEIDER, S., AND KUHN, W. *How to Talk to Each Other via Computers: Semantic Interoperability as Conceptual Imitation*. Springer International Publishing, Cham, 2015, pp. 97–122.

[25] SHAFER, G. *A Mathematical Theory of Evidence*. Books on Demand, 1976.

[26] STERNBERG, R. *Cognitive Psychology*. Cengage Learning, 2008.

[27] STEVENSON, A. *Oxford Dictionary of English*. Oxford Dictionary of English. OUP Oxford, 2010.

[28] TEAM, T. G. *The Glorious Glasgow Haskell Compilation System User's Guide*, July 2014.

[29] WEBB, G. I., AND PAZZANI, M. J. *Adjusted probability Naive Bayesian induction*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998, pp. 285–295.

[30] WOS, L., OVERBECK, R., LUSK, E., AND BOYLE, J. *Automated Reasoning: Introduction and Applications*. Prentice Hall Professional Technical Reference, 1984.

[31] ZADEH, L. Fuzzy sets. *Information and Control 8*, 3 (1965), 338 – 353.

[32] ZADEH, L. A. Fuzzy logic. *Computer 21*, 4 (Apr. 1988), 83–93.

[33] ZADEH, L. A. Fuzzy logic = computing with words. *IEEE Transactions on Fuzzy Systems 4*, 2 (May 1996), 103–111.

[34] ZENKER, F., AND GARDENFORS, P. *Applications of conceptual spaces : the case for geometric knowledge representation*. Springer, Cham, 2015.

Appendix A

CRLH code

In this appendix we present the source code for our library, CRLH. The library consists of three main files:

- Types.hs, which contains the data structures,
- Constructors.hs, which contains the type-safe constructors for our library,
- Operators.hs, which contains our Concept-to-Concept, Concept-to-Observation and Property-Observation Recognition operators and their supporting functions.

Following these three files is an included Test.hs file, containing sample data used to test the library functions.

A.1 Types.hs

This file contains the data structures of CRLH. The conditions on the data structures outlined by Rickard *et al* [22] are not enforced here, but in our constructors. As a result, this

module is *not* meant to be imported by the end user. Instead, the end user should import Constructors.hs and Operators.hs

```
module Types
where

import qualified Data.Map.Strict as Map
import qualified Data.Maybe as Maybe
import qualified Data.Set as Set

--Data structure for Domain
data Domain = Domain { domString :: String -- Name of Domain
    , dimensions :: Set.Set String -- List of Dimensions.
    , properties :: Set.Set Property
} deriving (Eq, Ord)
instance Show Domain where
show a = "Name: " ++ (show (domString a)) ++ "λnProperties: "
++ [x | y ← (Set.toList (properties a)),x ← (show y) ++ ", "]

-- Data structure for property.

data Property = Property { propertyString :: String,  -- Name of the property.
    propertyMaps :: Map.Map (String,String) Float -- maps.
} deriving (Eq, Ord)
instance Show Property where
show = show.propertyString --show.propertyName

-- Data structure for observation
data Observation = Observation {
    observationDomains :: Set.Set Domain,
    observationMaps :: [(Domain, [(String,String)])]
} deriving (Show)

-- Data structure for Context
data Context = Context {contextName :: String,
    contextProps :: Set.Set Property
} deriving (Eq, Ord)
-- Data structure for Concept
data Concept = Concept {conceptName :: String, -- Name of the concept.
    conceptSubTypes :: Set.Set Context, -- list of subconcepts
    conceptProps :: Set.Set Property, -- all properties in the concept
    conceptMaps :: Map.Map (Property,Property) Float -- Property Association of Concepts.
}
instance Show Concept where
show = show.conceptName


instance Show Context where
show = show.contextName
```

A.2 Constructors.hs

This file contains the constructors for our library. They check that the data supplied matches the requirements for the specific data structure. For more information, see Section 6.1.

```
module Constructors
where
{-this module will provide the type-safe constructors for our data structures.-}
import qualified Data.Map.Strict as Map
import qualified Data.Set as Set
import qualified Types as Types

--------------------------------------------------
--             Exported Functions               --
--   These are constructors for the data types   --
--           designed in this library.          --
--------------------------------------------------

domain :: String → [Types.Property] → Types.Domain
domain a c
    | length c == 0 = (Types.Domain "Error, No Properties" Set.empty Set.empty)
    | otherwise = (Types.Domain a (Set.fromList (getDims c)) (Set.fromList c) )

getDims :: [Types.Property] → [String]
getDims a = [x |
    y ← a,
    x ← fst(
        unzip (
            Map.keys (
                propertyMaps y
            )
        )
    )
]

property :: String → [((String,String),Float)] → Types.Property
property a b
    | length b == 0 = (Types.Property "Error, no mappings!" Map.empty)
    | maximum (snd (unzip b)) > 1 = (Types.Property "Error, invalid mappings!" Map.empty)
    | minimum (snd (unzip b)) < 0 = (Types.Property "Error, invalid mappings!" Map.empty)
    | otherwise = (Types.Property a (Map.fromList b))

--------------------------------------------------
--   Ensure that the properties in each pair are   --
--              in the domain listed.             --
--------------------------------------------------

-- Constructor for observation.  Takes as input a list of tuples.
-- These tuples contain domains paired with a list of tuples.
-- The second list of tuples contains dimension name and value on that dimension.
observation :: [(Types.Domain,[(String,String)])] → Types.Observation
observation a
    | length a == 0 = Types.Observation (Set.empty) []
    | checkObservation a == False = Types.Observation (Set.empty) []
    | otherwise = Types.Observation (Set.fromList (fst (unzip a))) a

checkObservation :: [(Types.Domain,[(String,String)])] → Bool
checkObservation a = or [x |
    y ← a,
    w ← (snd y),
    z ← [(fst w)],
    x ← [Set.member z (dimensions (fst y))]
]
--------------------------------------------------
--             This one might be fine.          --
--------------------------------------------------

concept :: String → [Types.Context] → [((Types.Property,Types.Property),Float)] → Types.Concept
concept a b c
    | length c == 0 = (Types.Concept "Error, no mappings!" Set.empty Set.empty Map.empty)
    | maximum (snd (unzip c)) > 1 = (Types.Concept
       "Error, invalid maps!"
       Set.empty
```

```
        Set.empty
        Map.empty
    )
    | checkConceptZeros c == False = (Types.Concept
        "Error, inconsistent maps!"
        Set.empty
        Set.empty
        Map.empty
    )
    | checkConceptSubtypes b d == False = (Types.Concept
        "Error, inconsistent subtypes!"
        Set.empty
        Set.empty
        Map.empty
    )
    | otherwise = (
        Types.Concept
        a
        (Set.fromList b)
        d
        (Map.fromList c)
    )
    where d =
    Set.union(
        Set.fromList(
            fst(
                unzip(
                    fst(
                        unzip c
                    )
                )
            )
        )
    ) (
        Set.fromList(
            snd(
                unzip(
                    fst(
                        unzip c
                    )
                )
            )
        )
    )


checkConceptZeros :: [((Types.Property,Types.Property),Float)] → Bool
checkConceptZeros a = and [
    0 == (
        sum [x |
            (y,z) ← Map.keys (Map.filter (== 0) (Map.fromList a)),
            x ← [(Map.findWithDefault 2 (z,y) (Map.fromList a))]
        ]
    ),
    0 /= (
        minimum [x |
            y ← Map.keys (Map.fromList a),
            (fst y) == (snd y),
            x ← [(Map.findWithDefault 2 y (Map.fromList a))]
        ]
    )
]

checkConceptSubtypes :: [Types.Context] → Set.Set Types.Property → Bool
checkConceptSubtypes a b = and [x |
    w ← a,
    y ← [Types.contextProps w],
    z ← [b],
```

```
    x ← [Set.isSubsetOf y z]
]

-----------------------------------------------------
--         This one is also probably fine.         --
-----------------------------------------------------

context :: String → [Types.Property] → Types.Context
context a b
    | length b == 0 = (Types.Context "Error, No Properties" Set.empty)
    | checkContextProperties b = (Types.Context "Error, Invalid Properties" Set.empty)
    | otherwise = (Types.Context a (Set.fromList b))

checkContextProperties :: [Types.Property] → Bool
checkContextProperties a = (or [x |
    y ← a,
    x ← [Map.null (propertyMaps y)]
])

-----------------------------------------------------
--                Ported from Types                --
-----------------------------------------------------

properties = Types.properties
dimensions = Types.dimensions
domString = Types.domString
propertyString = Types.propertyString
propertyMaps = Types.propertyMaps
observationDomains = Types.observationDomains
observationMaps = Types.observationMaps
conceptName = Types.conceptName
conceptSubTypes = Types.conceptSubTypes
conceptProps = Types.conceptProps
conceptMaps = Types.conceptMaps
contextName = Types.contextName
contextProps = Types.contextProps
```

A.3 Operators.hs

This section contains our Operators, which are discussed in Section 5.2. Some measures have been put in place to ensure that all operators terminate when given invalid data. For proof of this, see Section 6.2.

```
module Operators
where
import qualified Types as Types
import qualified Data.Map.Strict as Map
import qualified Data.Maybe as Maybe
import qualified Data.Set as Set


propertyRecognition :: Types.Observation → Map.Map Types.Property Float
-- This takes an observation and returns a map from each property
-- to the observation's similarity value for that property.
-- If an empty observation is passed, an empty map is returned.
propertyRecognition a
```

```
    |   Set.size (Types.observationDomains a) == 0 = Map.empty
    |   otherwise = Map.fromListWith max [x |
        y ← Types.observationMaps a,
        x ← propertyRecognition' y
    ]

propertyRecognition' :: (Types.Domain, [(String,String)]) → [(Types.Property, Float)]
-- This takes an observation's values on a domain
-- and returns the similarity values of each property on the given domain.
propertyRecognition' a = [(x,y) |
    x ← (Set.elems (Types.properties(fst a))),
    y ← [min (sum (propertyVal (Types.propertyMaps x) (snd a))) 1],
    y > 0
]

-- This takes a property's map and an observation's values on a domain
-- and returns the similarity values of that observation to that property.
propertyVal :: (Map.Map (String,String) Float) → [(String,String)] → [Float]
propertyVal a [] = []
propertyVal a b = [x |
    y ← b,
    x ← Maybe.catMaybes [(Map.lookup y a)]
]

conceptToObsWithContextCalc :: Types.Concept → Types.Observation → Types.Context → Maybe Float
-- Takes in a concept, an observation and a context.  If all arguments are well-formed,
-- it returns the similarity value of the observation to the concept in that context.
-- Otherwise, it returns nothing.
conceptToObsWithContextCalc a b c
    |   Set.null (Types.contextProps c) = conceptToObsCalc a b
    |   Set.null (Types.conceptProps a) = Nothing
    |   Set.null (Types.observationDomains b) = Nothing
    |   Set.member c (Types.conceptSubTypes a) = conceptToObsCalc
        (Types.Concept
            (Types.conceptName a)
            Set.empty
            (Set.intersection (Types.conceptProps a) (Types.contextProps c))
            (Map.filterWithKey
                (λk _ → k `notElem`
                    ([(x,y) |
                        y ← Set.toList
                        (Set.difference
                            (Types.conceptProps a)
                            (Types.contextProps c)
                        ),
                        x ← Set.toList (Types.conceptProps a)
                    ] ++ [(x,y) |
                        y ← Set.toList
                        (Set.difference
                            (Types.conceptProps a)
                            (Types.contextProps c)
                        ),
                        x ← Set.toList (Types.contextProps c)
                    ] ++ [(x,y) |
                        x ← Set.toList
                        (Set.difference
                            (Types.conceptProps a)
                            (Types.contextProps c)
                        ),
                        y ← Set.toList (Types.conceptProps a)
                    ] ++ [(x,y) |
                        x ← Set.toList
                        (Set.difference
                            (Types.conceptProps a)
                            (Types.contextProps c)
                        ),
                        y ← Set.toList (Types.contextProps c)
                    ])
```

```
                )
                (Types.conceptMaps a)
            )
        )
        b
    |   otherwise = conceptToObsCalc
        (Types.Concept
            (Types.conceptName a)
            (Set.fromList [x | x ← Set.toList (Types.conceptSubTypes a),
                (Types.contextProps x) `Set.isSubsetOf` (Types.contextProps c)]
            )
            (Set.intersection (Types.conceptProps a) (Types.contextProps c))
            (Map.filterWithKey
                (λk _ → k `notElem`
                    ([(x,y) |
                        y ← Set.toList
                            (Set.difference
                                (Types.conceptProps a)
                                (Types.contextProps c)
                            ),
                        x ← Set.toList (Types.conceptProps a)
                    ] ++ [(x,y) |
                        y ← Set.toList
                            (Set.difference
                                (Types.conceptProps a)
                                (Types.contextProps c)
                            ),
                        x ← Set.toList (Types.contextProps c)
                    ] ++ [(x,y) |
                        x ← Set.toList
                        (Set.difference
                            (Types.conceptProps a)
                            (Types.contextProps c)
                        ),
                        y ← Set.toList (Types.conceptProps a)
                    ] ++ [(x,y) |
                        x ← Set.toList
                        (Set.difference
                            (Types.conceptProps a)
                            (Types.contextProps c)
                        ),
                        y ← Set.toList (Types.contextProps c)
                    ])
                )
                (Types.conceptMaps a)
            )
        )
        b

conceptToObsCalc :: Types.Concept → Types.Observation → Maybe Float
-- Takes in a concept and an observation.
-- If all arguments are well-formed, it returns their similarity value.
-- Otherwise, it returns nothing.
conceptToObsCalc a b
    |   Set.null (Types.conceptProps a) = Nothing
    |   Set.null (Types.observationDomains b) = Nothing
    |   Set.size (Types.conceptSubTypes a) /= 0 = Just
        (maximum [x |
            y ← Set.toList (Types.conceptSubTypes a) ,
            x ← Maybe.catMaybes [conceptToObsWithContextCalc a b y]
        ])
    |   otherwise = Just (sum (calcMin a b) / sum (Map.elems (Types.conceptMaps a)))

calcMin :: Types.Concept → Types.Observation → [Float]
calcMin a b = zipWith
    min
    (Map.elems (Types.conceptMaps a))
    (calcprop (Map.keys (Types.conceptMaps a)) b)
```

```
calcprop :: [(Types.Property, Types.Property)] → Types.Observation → [Float]
calcprop a b = [x |
    y ← a,
    x ← [min
        (Map.findWithDefault 0 (fst y) c)
        (Map.findWithDefault 0 (snd y) c)
    ]
]
where c = (propertyRecognition b)

conceptToConceptWithContextCalc :: Types.Concept → Types.Concept → Types.Context → Maybe Float
-- Takes in two concepts and a context.
-- If an empty context is passed, the calculation is performed as if there were no context passed.
-- If either concept is empty, Nothing is returned.
conceptToConceptWithContextCalc a b c
    | Set.null (Types.contextProps c) = conceptToConceptCalc a b
    | Set.null (Types.conceptProps a) = Nothing
    | Set.null (Types.conceptProps b) = Nothing
    | otherwise = conceptToConceptCalc
        (Types.Concept
            (Types.conceptName a)
            (Types.conceptSubTypes a)
            (Set.intersection (Types.conceptProps a) (Types.contextProps c))
            (Map.filterWithKey
                (λk _ → k `notElem`
                    ([(x,y) |
                        x ← Set.toList
                        (Set.difference
                            (Types.conceptProps a)
                            (Types.contextProps c)
                        ),
                        y ← Set.toList (Types.conceptProps a)
                    ] ++ [(x,y) |
                        x ← Set.toList
                        (Set.difference
                            (Types.conceptProps a)
                            (Types.contextProps c)
                        ),
                        y ← Set.toList (Types.contextProps c)
                    ] ++ [(x,y) |
                        y ← Set.toList
                        (Set.difference
                            (Types.conceptProps a)
                            (Types.contextProps c)
                        ),
                        x ← Set.toList (Types.conceptProps a)
                    ] ++ [(x,y) |
                        y ← Set.toList
                        (Set.difference
                            (Types.conceptProps a)
                            (Types.contextProps c)
                        ),
                        x ← Set.toList (Types.contextProps c)
                    ])
                )
                (Types.conceptMaps a)
            )
        )
        (Types.Concept
            (Types.conceptName b)
            (Types.conceptSubTypes b)
            (Set.intersection
                (Types.conceptProps b)
                (Types.contextProps c)
            )
            (Map.filterWithKey
                (λk _ → k `notElem`
```

```
                    ([(x,y) |
                        y ← Set.toList
                        (Set.difference
                            (Types.conceptProps b)
                            (Types.contextProps c)
                        ),
                        x ← Set.toList (Types.conceptProps b)
                    ] ++ [(x,y) |
                        y ← Set.toList
                        (Set.difference
                            (Types.conceptProps b)
                            (Types.contextProps c)
                        ),
                        x ← Set.toList (Types.contextProps c)
                    ] ++ [(x,y) |
                        x ← Set.toList
                        (Set.difference
                            (Types.conceptProps b)
                            (Types.contextProps c)
                        ),
                        y ← Set.toList (Types.conceptProps b)
                    ] ++ [(x,y) |
                        x ← Set.toList
                        (Set.difference
                            (Types.conceptProps b)
                            (Types.contextProps c)
                        ),
                        y ← Set.toList (Types.contextProps c)
                    ])
                )
                (Types.conceptMaps b)
            )
        )

conceptToConceptCalc :: Types.Concept → Types.Concept → Maybe Float
conceptToConceptCalc a b
    | Set.null (Types.conceptProps a) = Nothing
    | Set.null (Types.conceptProps b) = Nothing
    | otherwise = Just (sum
    (zipWith
        min
        z
        y
    )
    /
    sum
    (zipWith
        max
        z
        y
    )
) where z = [x | c ← (Set.toList (Set.union (Types.conceptProps a) (Types.conceptProps b))),
            d ← (Set.toList (Set.union (Types.conceptProps a) (Types.conceptProps b))),
            x ← [
                Map.findWithDefault
                ( ifNeeded a c d )
                (c,d)
                (Types.conceptMaps a)
            ]
        ]
            y = [x | c ← (Set.toList (Set.union (Types.conceptProps a) (Types.conceptProps b))),
            d ← (Set.toList (Set.union (Types.conceptProps a) (Types.conceptProps b))),
            x ← [
                Map.findWithDefault
                ( ifNeeded b c d )
                (c,d)
                (Types.conceptMaps b)
            ]
```

```
        ]

ifNeeded :: Types.Concept → Types.Property → Types.Property → Float
ifNeeded a c d
    | (c,d) `elem` Map.keys (Types.conceptMaps a) = Map.findWithDefault 0 (c,d) (Types.conceptMaps a)
    | otherwise = (snd
        (propertyOverlap
            c
            (Set.toList
                (Types.conceptProps a)
            )
        )
    ) * --B_{aa*}
    (snd
        (propertyOverlap
            d
            (Set.toList
                (Types.conceptProps a)
            )
        )
    ) * --B_{bb*}
    (
        Map.findWithDefault
        0
        (fst
            (propertyOverlap
                c
                (Set.toList
                    (Types.conceptProps a)
                )
            ),
            (fst
                (propertyOverlap
                    d
                    (Set.toList
                        (Types.conceptProps a)
                    )
                )
            )
        )
        (Types.conceptMaps a)
    ) --C_{a*b*}

conceptToConceptCalc' :: Types.Concept → Types.Concept → Maybe Float
-- This function is merely here to show that the property overlap works.
conceptToConceptCalc' a b
    | Set.null (Types.conceptProps a) = Nothing
    | Set.null (Types.conceptProps b) = Nothing
    | otherwise = Just (sum
    (zipWith
        min
        y
        z
    )
    /
    sum
    (zipWith
        max
        y
        z
    )
) where z = [x | c ← (Set.toList (Set.union (Types.conceptProps a) (Types.conceptProps b))),
            d ← (Set.toList (Set.union (Types.conceptProps a) (Types.conceptProps b))),
            x ← [
                Map.findWithDefault
                0
                (c,d)
                (Types.conceptMaps a)
```

```
            ]
        ]
        y = [x | c ← (Set.toList (Set.union (Types.conceptProps a) (Types.conceptProps b))),
            d ← (Set.toList (Set.union (Types.conceptProps a) (Types.conceptProps b))),
            x ← [
                Map.findWithDefault
                0
                (c,d)
                (Types.conceptMaps b)
            ]
        ]


-- This returns (a*,B_{aa*}), where a* is the property that overlaps most with a.
propertyOverlap :: Types.Property → [Types.Property] → (Types.Property,Float)
-- The maximal overlap between two properties.
propertyOverlap a b = maximum' [(x,y) | x ← b, y ← [propertyOverlap' a x]]

--Given a list of tuples, find the element with the highest second member.
maximum' :: Ord a ⇒ [(t, a)] → (t, a)
maximum' (x:xs) = maxTail x xs
    where maxTail currentMax [] = currentMax
    maxTail (m, n) (p:ps)
        | n < (snd p) = maxTail p ps
        | otherwise   = maxTail (m, n) ps

----From Rickard, Aisbett, Gibbon 2007, this calculates the B value of two properties a,b.
propertyOverlap' :: Types.Property → Types.Property → Float
propertyOverlap' a b
    | a == b = 1
    | (
        (&&)
        ((Types.propertyString a) == "PropertyB")
        ((Types.propertyString b) == "PropertyE")
      ) == True = 0.2
    | (
        (&&)
        ((Types.propertyString a) == "PropertyE")
        ((Types.propertyString b) == "PropertyB")
      ) == True = 0.05
    | (
        (&&)
        ((Types.propertyString a) == "PropertyC")
        ((Types.propertyString b) == "PropertyE")
      ) == True = 0.02
    | (
        (&&)
        ((Types.propertyString a) == "PropertyE")
        ((Types.propertyString b) == "PropertyC")
      ) == True = 0.01
    | otherwise = (sum
        (
            [x |
                z ← Map.keys (Types.propertyMaps a),
                x ← [min
                    (Map.findWithDefault
                        0
                        (z)
                        (Types.propertyMaps a)
                    )
                    (Map.findWithDefault
                        0
                        (z)
                        (Types.propertyMaps b)
                    )
                ]
            ]
        )
```

```
    ) / sum (Map.elems (Types.propertyMaps a))
```

A.4 Test.hs

Test.hs includes some sample calculations and all test variables and functions mentioned in Section 6.3.

```
module CRLH where
import qualified Data.Map as Map
import qualified Data.Set as Set
import Constructors
import Operators
import Types

propertyA = property "PropertyA" [(("PortA","A"),0.9)]

propertyB = property "PropertyB" [(("PortB","B"),0.3)]

propertyC = property "PropertyC" [(("PortB","C"),0.3)]

propertyD = property "PropertyD" [(("PortC","D"),0.01)]

propertyE = property "PropertyE" [(("PortB","E"),0.1)]

propertyFW = property "PropertyFW" [(("Weather","FW"),0.8)]

deriveMatrix :: Types.Concept → Types.Concept → ([Float],[Float])
deriveMatrix a b = ([x |
    c ← (Set.toList (Set.union (Types.conceptProps a) (Types.conceptProps b))),
    d ← (Set.toList (Set.union (Types.conceptProps a) (Types.conceptProps b))),
    x ← [
        Map.findWithDefault
        ( ifNeeded a c d )
        (c,d)
        (Types.conceptMaps a)
    ]
],[x |
    c ← (Set.toList (Set.union (Types.conceptProps a) (Types.conceptProps b))),
    d ← (Set.toList (Set.union (Types.conceptProps a) (Types.conceptProps b))),
    x ← [
        Map.findWithDefault
        ( ifNeeded b c d )
        (c,d)
        (Types.conceptMaps b)
    ]]
)

domainTest = domain "A" [propertyA]

domain2 = domain "B" [propertyB,propertyC,propertyE]

domain3 = domain "C" [propertyD]

domainTest2 = domain "Domain2" [propertyFW]

observationTest = observation
[
    (domainTest,
```

```
    [
        ("PortA","A")
    ]),
    (domain2,
    [
        ("PortB","B"),
        ("PortB","C"),
        ("PortB","E")
    ]),
    (domain3,
    [
        ("PortC","D")
    ]),
    (domainTest2,
    [
        ("Weather","FW")
    ])
]

contextTest = context "ContextNameTest"  [propertyA,propertyB,propertyFW]

contextTest2 = context "ContextNameTest2" [propertyA,propertyC,propertyD,propertyFW]

conceptTest = concept "ConceptNameTest" [contextTest, contextTest2]
[
    ((propertyA,propertyA),1.00),
    ((propertyA,propertyB),0.27),
    ((propertyA,propertyC),0.4),
    ((propertyA,propertyD),0.4),
    ((propertyA,propertyFW),0.4),
    ((propertyB,propertyA),0.13),
    ((propertyB,propertyB),1),
    ((propertyB,propertyC),0),
    ((propertyB,propertyD),0),
    ((propertyB,propertyFW),0.2),
    ((propertyC,propertyA),0.75),
    ((propertyC,propertyB),0),
    ((propertyC,propertyC),1),
    ((propertyC,propertyD),0.75),
    ((propertyC,propertyFW),0.4),
    ((propertyD,propertyA),1),
    ((propertyD,propertyB),0),
    ((propertyD,propertyC),1),
    ((propertyD,propertyD),1),
    ((propertyD,propertyFW),0.2),
    ((propertyFW,propertyA),0.03),
    ((propertyFW,propertyB),0.018),
    ((propertyFW,propertyC),0.003),
    ((propertyFW,propertyD),0.003),
    ((propertyFW,propertyFW),1)
]

conceptTest2 = concept "ConceptNameTest2" []
[
    ((propertyA,propertyA),1.00),
    ((propertyA,propertyE),0.33),
    ((propertyA,propertyFW),0.4),
    ((propertyE,propertyA),0.5),
    ((propertyE,propertyE),1),
    ((propertyE,propertyFW),0.2),
    ((propertyFW,propertyA),0.03),
    ((propertyFW,propertyE),0.01),
    ((propertyFW,propertyFW),1)
]
```

Vita Auctoris

David MacMillan was born in 1990 and raised in Oakville, Ontario, Canada. He graduated from Iroquois Ridge High School in 2008 and completed his Bachelor's degree in Computer Science from the University of Windsor in 2012. He completed his Master's degree from the same institution in 2016.