2014

A Subjective Logic Library Constructed Using Monadic Higher Order Functions

Bryan Gary St. Amour
University of Windsor

A Subjective Logic Library Constructed Using Monadic Higher Order Functions

By:

Bryan St. Amour

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

2014

A Subjective Logic Library Constructed Using Monadic Higher Order Functions

By:
Bryan St. Amour

APPROVED BY:

Dr. R Caron
Department of Mathematics and Statistics

Dr. R Frost
School of Computer Science

Dr. R Kent, Advisor
School of Computer Science

September 16, 2014

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

Subjective Logic is a recently emergent probabilistic logic system that allows for reasoning under uncertainty. Though algebraically expressive, there is a lack of software tooling to support computation, such as code libraries, calculators, and software for the development of decision support systems. With this motivation, we present a complete design for a library of opinion data structures and operators constructed from higher order functions that are capable of representing and evaluating well-formed expressions of Subjective Logic. By leveraging monads, mathematical objects from Category Theory, we have enabled our operators to detect and propagate run-time errors without sacrificing compositionality. Furthermore, we have conducted a termination analysis on the expression evaluator and a complexity analysis on a representative subset of the operators. We have also proposed and implemented extensions to the set of Subjective Logic operators. Lastly, we provide examples of how to compute the values of Subjective Logic expressions.

Dedication

This thesis is dedicated to my late grandfather, Arthur Rigo. Though I wonder whether you would understand the content of this thesis, I've no doubt you'd be damn proud of me. This thesis concerns itself with automated reasoning systems, but there is just no reasoning with cancer. Here's to you, kemosabe.

Acknowledgements

No worthwhile academic endevour can be done in isolation; the age of the lone genius working away in his tower are long behind us. Conducting academic research is a team effort, and this thesis would not have come together without the assistance of some key players. I would first like to thank my lab mates Paul Preney, Dave MacMillan, and Jeffery Drake for being there to bounce ideas off of. Especially to Paul: we may have a friendly rivalry when it comes to meta-programming, but you are by far the better programmer.

I would also like to extend my sincerest thanks to my thesis readers: Dr. Rick Caron from Mathematics and Statistics, and Dr. Richard Frost from Computer Science. Your comments and suggestions have helped shape my research in a positive way. Many thanks also go out to my thesis advisor Dr. Robert D. Kent. Bob, thank you for everything you have done over the years, from the excellent conversations on everything from women to quantum tunnelling, to the opportunities that you gave me for developing myself both academically and industrially. Finally, thanks for believing in me even during the times that I had all but given up on myself.

I also wish to acknowledge the love and support of my amazing parents, Gary and Lee Ann St. Amour, and to my awesome sister Kristen. Thank you for always being there for me. I would also like to extend my endless gratitude to my wife, Chelsey St. Amour. I

started the Master's program as your boyfriend, and I'm finishing it as your husband. Words cannot express how thankful I am of your love and support throughout this adventure.

Finally, I would like to thank the Canadian Institutes of Health Research (CIHR) and the Auto21 Network of Centres of Excellence for financial support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Imagine being in a courtroom where a man is being tried for murder. The prosecution has brought forth three witnesses who allegedly observed the event. Witness A is a close friend of the defendant, and has a high opinion of him. Witness B does not like the defendant at all, and has a very negative opinion of him. The third witness, Witness C, has no prior opinion of the defendant, and thus is very uncertain about his character.

The judge has never once interacted with the defendant and therefore must base his entire opinion of him on the evidence brought forth and by the witness testimonies. The judge does, however, have an opinion about each of the three witnesses. The judge golfs regularly with witness A, the judge knows witness B is the pastor at a local church, and witness C is a courtroom regular - always involved in some mischief or other. Therefore, while the judge can construct an opinion of the defendant by analyzing the opinions of the three witnesses and forming a consensus, he also takes into account his knowledge of the

Name of Breed	% of Total
Silky	25%
American	40%
Peruvian	25%
Mixed	10%

Table 1.1: Imaginary distribution of guinea pig breeds

three, and discounts their opinions by his own opinions of them. The judge places great weight on the testimonies of witnesses A and B, and can barely belief a word of witness C's statement.

Now imagine two sensors designed to measure two orthogonal properties of baby guinea pigs. Before they reach a certain age, male guinea pigs must be separated from their mothers (and sisters) because they reach sexual maturity very quickly. Therefore it is important to be able to measure the sex of the guinea pigs quickly and partition them accordingly in order to avoid a combinatorial explosion of new children. Another important measurable trait is the breed of the guinea pig. If the pigs have been brought from many different litters, then it is important to be able to classify them as *Silky*, *American*, or *Peruvian* before sending them to the pet store. This classification cannot be carried out with absolute certainty, as there can be mixed breed guinea pigs as well. Assume for simplicity that guinea pigs have a male/female birth ratio of 50/50, and that the probability of a guinea pig having a certain breed is given in Table 1.1.

Given these two sensors, it is possible to classify the guinea pigs into eight categories. To complicate matters, imagine that your breed-detecting sensor has a tendency to give back inaccurate results, say, 5% of the time. Any reasoning that is to be done with this sensor data must be handled with care, as it has a non-zero rate of error.

	M1	M2	M3
You	0.5	0.3	0.1
Bill	0.2	0.6	0.2
Ted	0.7	0.0	0.0

Table 1.2: Movie Preferences

Lastly, suppose you and two of your friends wish to see a movie. There are three movies currently playing in your local theatre: Star Wars - The Empire Strikes First (M1), Casablanca 2 (M2), and A Slug's Life (M3). Each of you has a preference for each of the three movies, as depicted in Table 1.2. Is it possible for the three of you to come to a reasonable decision for which movie to see?

The above scenarios all share a common theme: they involve reasoning about uncertain or incomplete data. Many real-world reasoning scenarios must deal with this kind of data, and thus any automated system designed to aide decision-makers in these (and many other) kinds of situations must be able to take uncertainty into account.

This thesis is about the engineering of a library for constructing and evaluating expressions in *Subjective Logic*, a recently emergent extension to probabilistic logic [23] with support for reasoning under uncertainty. The library is designed to be a central component of Unified Data Management and Decision Support System (UDMDSS) [37, 41, 39], a decision support system that is under active research and development within our lab. We utilize the *Haskell* programming language [19] as it supports strong typing, has excellent support for programming with *monads* [64], and is overall an elegant *purely functional* programming language for implementing mathematical programs.

1.2 The Problem Addressed

Subjective Logic is a relatively new form of probabilistic logic that is currently under active development [23]. The novelty of Subjective Logic is that it directly handles uncertainty, and each and every operator for manipulating subjective opinions - the primary objects of Subjective Logic - takes this uncertainty into account. The result is a flexible calculus of opinions that can be used to model many kinds of situations that require reasoning under uncertainty [65, 32, 45, 55].

As Subjective Logic is still an area of active research, the operators, opinions, and even nomenclature, are evolving. As a result of this there is, to the best of our knowledge, no implementation of Subjective Logic available for use by application developers and researchers. Audun Josang has provided an implementation of some Subjective Logic operators, however the implementation is incomplete. The implementation was constructed before Subjective Logic had introduced *hyper opinions* and other operators now found in the literature.

1.3 Our Proposed Solution

To combat this scarcity of implementations, we have developed a library of Subjective Logic operators using the *Haskell* programming language. We represent expressions of Subjective Logic as functions from an initial world state to some numeric output, and the operators of Subjective Logic as higher order functions. Therefore simple expressions of Subjective Logic can be combined to form larger more complex equations.

In order to assist us in combining together these equations, we use monads, in particular

a *state monad*. Monads are ubiquitous in Haskell, and are a general design pattern that has been previously used to represent stateful computations [43], input/output [64], and formal [20, 44] and natural language [14] parsers.

In order to demonstrate the effectiveness of our library, we utilize it to implement some example calculations provided by Josang in the literature. Furthermore we prove that our set of operators terminates for all possible valid input equations. Lastly, we perform a complexity analysis on a representative subset of the operators.

We expect that our library will be found useful by the research community, and that it will spur the development of Subjective Logic-based reasoning applications.

1.4 Thesis Contribution

To realize the solution proposed above, in this thesis we have done the following:

- We developed SLHS, a Subjective Logic library that is type-safe, efficient, and compositional, using the Haskell programming language (Chapter 4).
- We contributed two additional operators to Subjective Logic (Section 4.4).
- We proved that the evaluator of SLHS (the function that evaluates the Subjective Logic expressions) terminates for all valid Subjective Logic expressions (Section 5.1).
- We analyzed the time complexity of a representative subset of the Subjective Logic operators (Section 5.2).

- We constructed example applications to demonstrate the effectiveness and ease of use of SLHS (Section 5.5).

1.5 Organization of this Document

The remainder of this document is organized as follows. Chapter 2 introduces the reader to the relevant background information on decision support systems, automated reasoning systems, uncertain reasoning, Subjective Logic, and pure functional programming in Haskell to allow the proceeding chapters to be better understood. Chapter 3 contains the thesis problem, hypothesis, objectives, and methodology. Chapter 4 introduces *SLHS*, a library of Subjective Logic objects and operators, written in the Haskell programming language. Chapter 5 presents a proof of termination, analysis of complexity a sample of operators in SLHS, and a discussion regarding the use of Haskell and monads on the design of the library. It also contains examples of how one can use SLHS to model situations that require uncertain reasoning, and lastly, it discusses the library's role within the larger UDMDSS decision support system. Chapter 6 concludes this thesis and discusses areas for future improvement.

Chapter 2

Background

In this chapter we provide an introduction to the relevant background material pertaining to this thesis. We begin with a discussion of *decision support systems*, followed by an overview of *automated reasoning*. Next we discuss *uncertain reasoning* including *Dempster-Shafer Theory* and *Subjective Logic*. We next discuss various tools for developing uncertain reasoning applications. We conclude with a brief overview of the *Haskell* programming language, as it is the language used for the program examples throughout this thesis.

2.1 Decision Support Systems

Decision support systems are information systems that are designed to aide users with various decision-making tasks [74]. Examples of such tasks are those pertaining to management, planning, or operations. Typically decision support systems work with the kinds of unstructured or underspecified problems faced by managers and decision-makers in many

areas; involve the synthesis of models, analytics, and data; are targeted at non-technical people; and are designed to be flexible and adaptable in the face of new data or changes to the working environment [74].

In his 2002 book, *Decision support systems: concepts and resources for managers* [66], Daniel J Power breaks down Decision support systems into the following taxonomy:

- Communication-driven systems: systems that allow for more than one person to work on a shared task.
- Document-driven systems: systems that allow for the storage, retrieval and manipulation of unstructured data documents.
- Data-driven systems: systems that facilitate the manipulation of internal company data.
- Model-driven systems: systems that allow for access and modification of various models: whether they are financial, simulation, statistical, or other.
- Knowledge-driven systems: systems that contain problem solving expertise for the task at hand, typically encoded as facts and rules.

As a part of the ongoing research in our lab, we have designed the Unified Data Management and Decision Support System (UDMDSS) [37, 41, 39]. UDMDSS was designed to handle the management and analysis of population research surveys. Figure 2.1 shows an overview of the various components of the system. Of particular interest to this thesis is the data analysis component. Of the various tools available for uncertain reasoning such as Fuzzy Set Theory, Bayesian Probability, and Dempster-Shafer Theory, we have chosen

to base UDMDSS's reasoning engine on Subjective Logic [38], a recently emergent extension to probabilistic logic [23]. Each of the mentioned tools have their strengths and weaknesses, and in the next section we discuss the topic of automated reasoning and how they and others can be used for deductive, inductive, and abductive reasoning.

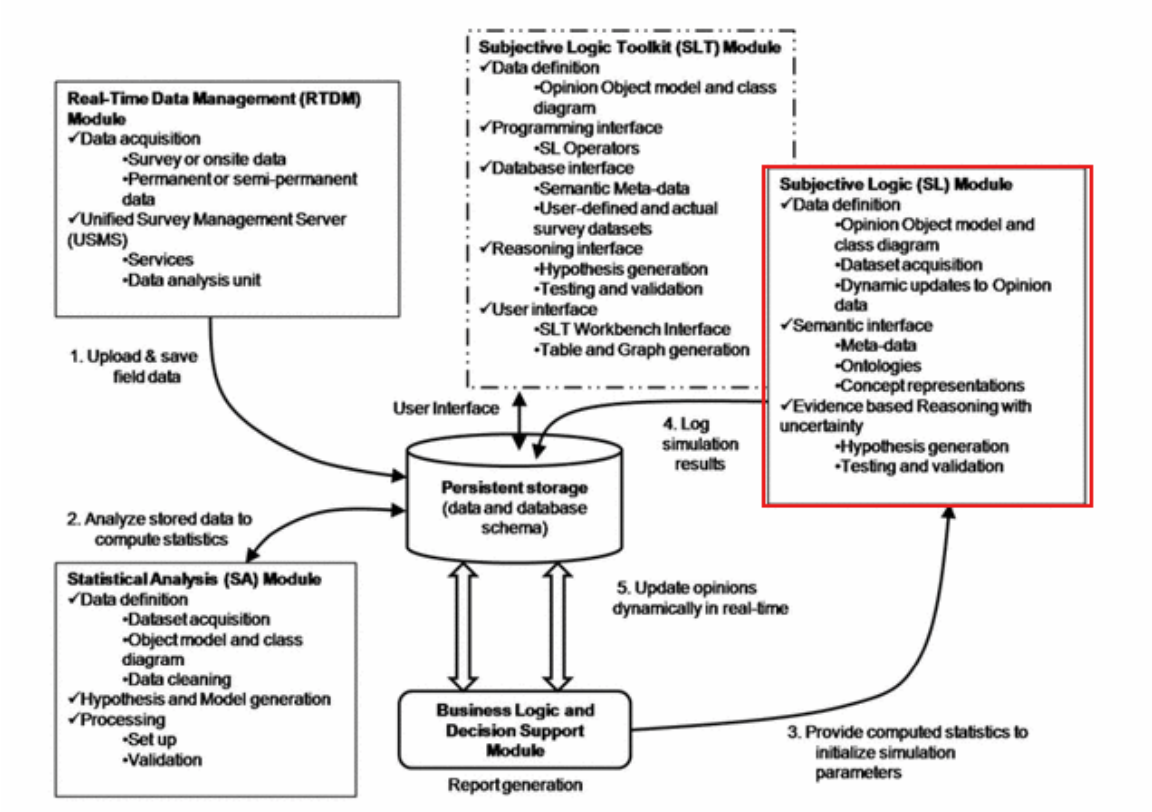

Figure 2.1: Unified Data Management and Decision Support System (UDMDSS) [38]

2.2 Automated Reasoning

Automated reasoning is a topic of Artificial Intelligence that has to do with the construction of systems that can reason with information and draw conclusions. Wos et al define an automated reasoning program to be 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" [81]. Reasoning can either be

- *deductive*: where from a set of initial facts and rules of the form "if X then Y", we can compute the truth or falsity of theorems with absolute certainty through the use of *Modus Ponens* [76] For example: suppose we know for absolute certainty that all professors are cranky, and that Dr. X is a professor. We therefore must conclude that Dr. X is cranky.

- *inductive*: where from some observations we formulate a hypothesis and then verify that hypothesis by testing that it holds for new observations. In contrast with deduction, inductive conclusions should not be certain, but probable, given the supporting evidence [6]. For example, a scientist may, after several observations of birds flying, construct the hypothesis that all birds fly. The scientist must modify her hypothesis upon observing an ostrich.

- *abductive*: where we compute the best possible hypothesis that explains some observation [36]. As an example, physicians must use abductive reasoning every day in their work, as all that they can observe are symptoms, not the causes of those symptoms. Therefore if there exist several competing explanations as to why the patient has a terrible cough, the doctor must abduce the most likely hypothesis, and then test that hypothesis to ensure its validity.

Unlike deduction, neither induction nor abduction can be used to reason with absolute certainty. Since the validity of collected population survey data is not absolutely certain

(data can be missing or unclear, the clerk may have entered the survey data into the system incorrectly, or a whole host of other issues) the focus of this thesis is on the development of a software library that can reason with uncertain information. As will be shown in Section 2.3.4, in the case of Subjective Logic, as the amount of evidence tends toward infinity, the amount of uncertainty tends to zero, leaving a pure probability.

2.3 Reasoning With Uncertain Information

Since the early days of Artificial Intelligence, researchers have been interested in modeling how humans perform various kinds of reasoning [70], and more recently (late 1980's to early 1990's) researchers have developed successful techniques for constructing artificial systems that can reason with uncertain information [70]. Tools that are used by researchers for handling uncertain or incomplete information include, but are not limited to

- Bayesian Probability
- Fuzzy Logic
- Dempster-Shafer Theory
- and more recently, Subjective Logic

In this section we discuss the above mentioned calculi, and in Section 2.4 we discuss various languages, workbenches, and tools that are available for researchers.

2.3.1 Bayesian Probability

Bayesian Probability is an interpretation of the concept of probability that can be seen an extension of propositional logic [5]. It allows for reasoning with propositions whose truth values are uncertain. Being an evidential probability, the prior probability of a proposition (the probability of the proposition being true prior to any evidence being accounted for) is assigned, and as evidence is accounted for, the probability of the proposition is updated through a mechanism called *Bayesian Updating* [57]. Unlike a frequentist view of probability, in which the probability of a proposition represents the frequency of the event occurring, in Bayesian Probability the probability of a proposition represents a state of belief [8].

Reasoning with Bayesian Probability amounts to the following:

1. Represent all sources of uncertainty as statistical random variables [12].

2. Determine and assign a prior probability distribution to the random variables.

3. As more evidence is made available, update the probability distributions by applying Bayes' Formula:

$$P(A|B) = \frac{P(B|A) \times P(A)}{P(B)}$$

$P(A)$ represents the prior probability of the proposition A being true, and $P(A|B)$ is the conditional probability of A being true given B is true. Therefore, as new evidence becomes available, the probability distributions describing the propositions are updated, and these updated probabilities are then used as priors for further calculations with new evidence.

While Bayesian Probability appears to be a fairly simple method of extending propositional logic to handle uncertainty, one issue that arises is when one wants to carry out abductive inference. The *base rate fallacy* occurs when one assumes that $P(A|B) = P(B|A)$ [42], and therefore when one wants to reason backwards from some observable evidence to the likely hypothesis, the conditional probabilities must first be inverted [35]. Subjective Logic, as will be shown, supports both deductive and abductive reasoning as operators, and thus no confusion can occur so long as the correct operator is chosen.

2.3.2 Fuzzy Logic

Fuzzy Logic is a many-valued logic that supports reasoning with approximate truth values, rather than exact truths as in classical logic [63]. The term "fuzzy logic" was first introduced by Zadeh [83] in his description of *Fuzzy Set Theory*, and since then it has been applied to fields such as Control Theory, Automated Reasoning, and Machine Learning [3].

Given a predicate P and a variable x, let $P(x)$ be a function that maps x to a value on the interval $[0,1]$. This function represents the degree of which x satisfies P. For example, consider two predicates *Red* and $Yellow$. Given the variable *orange* representing the colour orange, one observer might say that $Red(orange) = 0.4$, and that $Yellow(orange) = 0.8$. That is, the colour orange is more "yellow" than it is "red". However a different observer might assign a different degree of membership to the colour.

Fuzzy Logic supports the operators AND and OR, just like in classical logic, but since the degrees of truth are continuous values between 0 and 1, a simple truth-table will not suffice for representing the logical operators. Therefore, Fuzzy Logic defines x AND y to

be the minimum value of the two degrees of truth, and x OR y to be the maximum value. The negation of a degree of truth is 1 minus the degree.

Fuzzy Logic has been suggested as a method of handling uncertainty in the design of expert systems by Zadeh [86]. In fact, Zadeh claims that Fuzzy Logic subsumes both Probability Theory and Predicate Logic and allows for uncertainty to be handled in one single conceptual framework. It is claimed, however, by Russell and Norvig in their popular textbook *Artificial Intelligence: A Modern Approach* [70] that Fuzzy Logic is not a method of uncertain reasoning at all, because it simply replaces crisp truth values with approximate ones. Therefore, they claim that Fuzzy Logic is a method of representing vagueness, not uncertainty.

2.3.3 Dempster-Shafer Theory

Dempster-Shafer Theory is a mathematical and philosophical theory of evidence [72]. It is an extension of Bayesian Probability in which probabilities are assigned not to individual random variables, but to sets of them. The belief of an individual random variable is bounded above and below by two values: the *plausibility* of the random variable, and the *belief* of it.

Given a *frame of discernment*, a set containing all mutually exclusive atomic events that are of interest to our reasoning system, one constructs a *basic belief assignment*, or BBA, which assigns a measure of belief between zero and one to subsets of the frame. BBA's are additive: if X is a frame of discernment and m is a BBA over X, then $\sum_{x \subset X} m(x) = 1$. Furthermore, no mass is assigned to the empty set: $m(\emptyset) = 0$.

Given a BBA m over a frame X, one can compute the belief and plausibility of a subset A of X by the following expressions:

- $bel(A) = \sum_{B \subseteq A} m(B)$
- $pl(A) = 1 - bel(\overline{A})$

These two values bound the probability of A from below and above. That is, $bel(A) \leq P(A) \leq pl(A)$. The real novelty of Dempster-Shafer Theory, however, is *Dempster's Rule of Combination*, which states how two BBA's generated by two observations can be combined together [10]. Let m_1 and m_2 be two BBA's over a frame of discernment X. We combine together the two BBA's by computing what is referred to as the *joint mass*, denoted as $m_{1,2}$, by the following equation:

$$m_{1,2}(\emptyset) = 0$$
$$m_{1,2}(A) = (m_1 \otimes m_2) = \frac{1}{1-K} \sum_{B \cap C = A \neq \emptyset} m_1(B) m_2(C)$$

K, which represents the amount of conflicting belief between m_1 and m_2, is

$$\sum_{B \cap C = \emptyset} m_1(B) m_2(C)$$

While fairly straight forward to calculate, it has been shown by Zadeh [84, 85] that Dempster's Rule generates counter-intuitive results when there is a high degree of conflict between the two belief masses, and Josang and Pope claimed that Dempster's Rule actually represents a method of preference combination while serving as an approximation for

other forms of belief combination such as the cumulative or average fusion of two beliefs [34]. Subjective Logic, which we introduce next, contains several operators for combining beliefs together [22, 31, 27, 26], that serve as better tools for combining evidence from different sources in different scenarios. Furthermore, Judea Peal has claimed that it is misleading to interpret belief functions as anything other than the probability that a given proposition is provable from a set of other propositions that have assigned probabilities [59, 58, 60].

Despite these criticisms, Dempster-Shafer Theory has seen much success when applied to problems such as sensor fusion [82, 54, 4] and neural network classification [11, 69].

2.3.4 Subjective Logic

Subjective Logic was introduced by Audun Josang [23] as an extension to probabilistic logic that fixes some of the issues with Dempster-Shafer Theory [34] that have been mentioned in Section 2.3.3. Though it is relatively young and is under constant refinement, Subjective Logic has been shown to be effective across a range of areas that require uncertain reasoning, such as trust network analysis [32, 29], modeling trust on mobile ad-hoc networks [45, 47], and arguing with evidence [55, 30].

Subjective Opinions

The primary building blocks of Subjective Logic expressions are objects called *subjective opinions* [23]. Given a frame of discernment Θ, a subjective opinion over Θ is a 3-tuple consisting of the following elements:

- A *belief vector*, b_Θ, of assigned belief mass that spans the *reduced power set* of Θ. The reduced power set is defined as $R(\Theta) = 2^\Theta \setminus \{\Theta, \emptyset\}$.
- A scalar, u_Θ, that represents the unassigned belief mass $u_\Theta + \sum_{x \in R(\Theta)} b_\Theta(x) = 1$
- A vector of prior belief, a_Θ, that spans the frame Θ

such that the following conditions hold:

1. $\forall x \in R(\Theta), b_\Theta(x) \in [0,1]$
2. $\forall x \in \Theta, a_\Theta(x) \in [0,1]$
3. $u_\Theta \in [0,1]$
4. $u_\Theta + \sum_{x \in R(\Theta)} b_\Theta(x) = 1$
5. $\sum_{x \in \Theta} a_\Theta(x) = 1$

Opinions are written as $\omega_\Theta^A = \langle b_\Theta^A, u_\Theta^A, a_\Theta^A \rangle$, where A is the (optional) agent who owns that particular belief.

Elements of $R(\Theta)$ such that $b_\Theta(x) > 0$ are called *focal elements*. Subjective opinions where the focal elements are all singleton sets - that is, every focal element is simply an element of Θ - are referred to as *multinomial opinions*. Multinomial opinions defined over frames of cardinality 2 are referred to as *binomial opinions*. The most general of opinions, subjective opinions, are also referred to as *hyper opinions*. Lastly, opinions can either be *dogmatic*, when u_Θ is zero, or *uncertain* otherwise. The six classes of subjective opinions are summarized in Table 2.1.

	$\lvert\Theta\rvert = 2$	$\lvert\Theta\rvert > 2$	$\lvert R(\Theta)\rvert = 2^{\lvert\Theta\rvert} - 2$
$u > 0$	Uncertain Binomial	Uncertain Multinomial	Uncertain Hyper
$u = 0$	Dogmatic Binomial	Dogmatic Multinomial	Dogmatic Hyper

Table 2.1: Subjective Logic Opinions

Binomial opinions have a special notation that is used to emphasize the binary nature of the frame of discernment [23]. Given a frame $\Theta = \{x, \neg x\}$, the binomial opinion of x is written as $\omega_x = \langle b_x, d_x, u_x, a_x \rangle$, where

- b_x is the belief of event x being true.
- d_x is the belief of event x being false.
- u_x is the uncertainty of whether x is true or false.
- a_x is the belief of x being true prior to the collection of evidence.

Opinions in Subjective Logic can be mapped to and from probability density functions from Probability Theory [23, 22]. Binomial opinions correspond to *beta probability density functions* (PDFs), multinomial opinions correspond to *dirichlet PDFs*, and hyper opinions correspond to *hyper-dirichlet PDFs*. For evidence-based reasoning this is a boon because the Beta PDF acts as a *conjugate prior* to the binomial distribution, and the Dirichlet PDF is prior to the multinomial [71]. This means that through the mapping, subjective opinions can be used anywhere one could use Bayesian Inference, where the Bayesian Update mechanism updates the opinions to take into account new evidence.

Subjective Logic Operators

Subjective Logic includes a wealth of operators for working with all classes of opinions. It includes the traditional binary logic operators such as *and*, *or*, and *not*, which are upgraded to incorporate uncertainty, as well as the set-theoretic operators *union* and *set-difference*. In the case of absolute belief ($b_x = 1$) or disbelief ($d_x = 1$), these binomial operators behave the same as they would in traditional logic [50, 33].

Subjective logic also includes operators for working with multinomial opinions, such as cumulative and averaging *fusion* and *unfusion* [31, 26, 27, 22]. These operators allow for combining multinomial opinions from different sources. Subjective Logic also includes operators for performing transitive trust analysis [23, 32], where an agent A has an opinion of agent B, and agent B has an opinion of the event X. Agent A, through its opinion of agent B, can derive an opinion of event X by using one of several *discounting* operators. Subjective Logic also includes an operator for *belief constraining* [34], which can be used when multiple agents need to reach a consensus opinion. This operator is in fact equivalent in meaning to Dempster's rule of combination [34].

Lastly, Subjective Logic also includes operators for performing uncertain reasoning [35, 25, 24]. It includes *deduction* and *abduction* operators for subjective opinions, thereby allowing Subjective Logic to be used for intelligence analysis [65], bayesian network analysis [25], and other actions that require reasoning when uncertainty is present.

2.4 Languages and Tools for Automated Reasoning

In Section 2.3 we introduced various systems for automated reasoning. In this section we discuss some languages and tools that have been developed for the previously mentioned systems. Note however that as far as we know, there do not exist any languages or tools for working with Subjective Logic.

2.4.1 Weka

Waikato Environment for Knowledge Analysis (WEKA) is a popular workbench for machine learning [80]. It contains many popular algorithms and visualization techniques for performing data mining, data analysis, and predictive modeling. It is developed in the JAVA programming language, and is distributed as *Free Software* under the GNU General Public License.

Though freely available, Weka requires all data to be described using a fixed number of attributes and all data must be stored in a single file or relational table [68]. There exist tools however for converting data into the format required for Weka [68].

2.4.2 DSI Toolbox

Dempster-Shafer with Intervals (DSI) is a verified MATLAB toolbox for computing with Dempster-Shafer Theory [1]. The authors claim that DSI introduces intervals to a previously developed IPP toolbox [46], and that because of this modification they claim that DSI does not suffer from the same rounding errors that occur in IPP. We follow a similar approach in the design of our library: in order to avoid the possibility of rounding errors

in Subjective Logic, we represent each numeric value as a rational number. As will be explained in Section 4.5, this representation may not always be desirable, as it removes the ability for prior beliefs to be populated with irrational numbers such as $\frac{1}{e}$.

2.4.3 R

R is a programming language and interactive environment for statistical computing [77]. It is popular among statisticians and data miners [13, 79], and is a powerful and free alternative to other non-free statistical tools such as SAS [9] and SPSS [67]. R can be extended through user-defined packages, many of which are available through repositories such as the *Comprehensive R Archive Network (CRAN)* and *Bioconductor*, a project which focuses on the analysis of genomic data in molecular biology.

Though powerful, we believe the language is best suited for designing statistical software, not general purpose programming. For the development of our library for Subjective Logic, we chose to use the Haskell language over R, as we feel that Haskell has better support for everyday programming.

2.4.4 Prolog

Prolog is a Logic Programming Language, which means that every computation must be expressed as a logical statement [75]. Despite this seemingly strange restriction, Prolog is a general-purpose programming language [75].

As mentioned, all computations in Prolog are expressed as logical statements. In particular, expressions in Prolog are *Horn Clauses*: logical expressions of the form

Name	Method of Reasoning	Data Representation	Notes
Prolog	Deduction	Horn Clauses	Unideal for uncertainty. All computations represented as logical deductions.
R	Bayesian Statistics	Data tables	Powerful for statistical computation.
Weka	Machine learning algorithms	Data tables	Vast array of tools. Data must conform to a certain format to be usable.
DSI	Dempster-Shafer Theory	Beliefs	MATLAB workbench. Uses intervals instead of floating point math.

Table 2.2: Summary of Discussed Reasoning Tools

$$head : -X1, X2, ...XN$$

meaning the statement *head* is true only when statements $X1$ through XN are also true [17]. As an example of how one can represent computations in Prolog, the following program computes the factorial of a number:

```
factorial(0, X) :- X = 1.
factorial(N, X) :- NN = N - 1, factorial(NN, X1), X = X1 * N.
```

It was the language of choice for Japan's ambitious fifth generation computing project [73], and Prolog still sees much use in the Natural Language Processing community [7, 61], as it has excellent support for implementing *definite-clause grammars* [62]. Prolog, however, does not have built-in support for uncertainty. Because it is a general purpose programming language, one could theoretically construct an automated reasoning program in Prolog that does handle uncertainty, however it would fight against the spirit of the language.

2.4.5 Summary

There currently exist many tools for developing automated reasoning systems, and we have summarized a few of them in the previous section and in Table 2.2. Due to it being quite young in comparison to other systems, there do not yet exist any comprehensive tools for developing applications with Subjective Logic. In the next section we present an overview of the Haskell programming language, our implementation language for a new Subjective Logic library, and in Chapter 4 we present SLHS: Subjective Logic in Haskell.

2.5 Functional Programming in Haskell

Haskell is a strongly typed, non-strict, pure functional programming language [19] which was initially developed to be a common language for researchers interested in non-strict, pure functional programming languages [18]. By *non-strict*, we mean that Haskell evaluates expressions in a *call-by-need* manner: expressions are only evaluated if and when they are required [15]. Haskell is a *functional programming language*, where the meaning of *functional* is the style of programs as described by John Backus in his Turing award lecture: *Can Programming Be Liberated from the von Neumann Style?*[2]. Lastly, Haskell is *pure* in the sense that all functions are functions in the mathematical sense: they depend only on their inputs to produce their outputs. Haskell does not support the use of global state when writing programs.

In this section we will briefly describe the syntax of Haskell in order to give the reader enough familiarity to understand the code listings of Chapter 4. This section is by no means exhaustive in its treatment of Haskell. For readers who wish to learn Haskell in more depth,

we suggest the book *Real World Haskell* [56].

Functions in Haskell are written as equations, with parameters separated by white space. For example, the function to compute factorials can be written as

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

All expressions in Haskell have *types*. For example, the type of the literal 5 is *Int*. Syntactically this is expressed as *5 :: Int*. The function *factorial* above has the type *Int* $\rightarrow$ *Int*.

Lists in Haskell are enclosed in square braces, and their elements must be of all the same type. As an example, the following is a valid list:

```
names :: [String]
names = [''John'', ''Paul'', ''George'', ''Ringo'']
```

whereas the following is invalid:

```
things = [5, ''seven'', 2/3]
```

Types in Haskell can be organized into *Type Classes*, where each type in a type class must have certain required operations defined over it. For example, consider the following class:

```
class Monoid n where
    id  :: n
    (<>) :: n → n → n
```

which states that a type *n* satisfies the properties of being a *Monoid* if there exists a element *id* of type *n*, and there exists an operator for combining elements of type *n*. Unfortunately the additional requirement of associativity cannot be expressed in Haskell. Instances of the Monoid class can then be defined for individual types:

```
instance Monoid Int where
    id = 0
    x <> y = x + y
```

One type class in particular gets special attention in Haskell. Types that are instances of class *Monad* are very popular in functional programming, and Haskell in particular [64]. Monads are mathematical objects from *category theory* that are prevalent throughout Haskell. They were first introduced by Eugenio Moggi [52] and have subsequently been used for parsing [20, 44, 14], modeling state [43], and much more. Most importantly, Haskell uses monads to handle input/output [64], which allows Haskell to read input from the user, and send output to the computer screen, while remaining a pure functional language. Types that are instances of *Monad* require two operations to be present:

```
class Monad m where
    return :: a → m a
    (>>=)  :: m a → (a → m b) → m b
```

The first function, *return*, injects an object of type a into an object of type ma, where m is some monad. The second operator takes in an object of type ma on the left hand side, and a function f from a to mb on the right hand side, and returns an object of type mb. Informally, the operator unwraps the object of type a from the object of type ma, and then applies the function to it to obtain a result.

2.6 Summary

In this chapter we discussed the key ideas of decision support systems, followed by an overview of automated reasoning, and an introduction to various uncertain reasoning systems. We then introduced Subjective Logic and presented a brief overview of the Haskell programming language. In the next chapter we present our thesis problem, our thesis hypothesis, our research objectives, and our methodology.

Chapter 3

Thesis Statement

In this chapter we describe the problem that this thesis addresses, our thesis hypothesis, and our research objectives. Lastly we outline the methodology that we followed in order to achieve those mentioned objectives.

3.1 Thesis Problem

As mentioned previously, there does not yet exist a comprehensive library of Subjective Logic operators that can be used for research, development, and experimentation. There exists a partial implementation of Subjective Logic operators by Audun Josang[1], but at the time of this writing, to our knowledge no complete implementation exists.

We expect that such a library of operators should be efficient, type-safe, and compositional. The library should be efficient in such a way that values are only computed as needed. The library should be type-safe in order to catch invalid Subjective Logic expres-

[1] `http://folk.uio.no/josang/sl/Op.html`

sions as early as possible. By leveraging a strong type system, the library should be able to catch many errors at the time of compilation. Finally, the library should be compositional in a sense that arbitrarily complicated Subjective Logic expressions should be able to be constructed from a small set of functions and operators.

3.2 Thesis Hypothesis

Motivated by the aforementioned problem, our hypothesis for this thesis is: Using monads and strong typing, it is possible to construct a general purpose Subjective Logic library that is type-safe, efficient, and compositional.

3.3 Objectives

The objectives of our research are the following:

- Develop a library of Subjective Logic operators using monadic higher order functions.
- Demonstrate the type safety of the library.
- Prove that the expression evaluator, the *run* function, terminates for all valid Subjective Logic expressions.
- Analyze the time complexity of a representative subset of the operators.

3.4 Methodology

In order to satisfy the objectives of our research, we have done the following:

- We developed the library using the Haskell programming language due to it's strong type system and excellent support for monadic programming.
- We discuss how Haskell's strong type system allows for our library to reject certain classes of ill-formed Subjective Logic expressions.
- We utilize structural induction on the length of the input Subjective Logic expression to prove that our operators terminate.
- We analyze the time complexity of the operators based on the cardinality of elements in the frame of discernment that have non-zero belief mass assigned to them.

In the following chapter we will discuss the implementation of *SLHS*: Subjective Logic in Haskell. Then, in Chapter 5 we will provide proofs of termination, complexity analysis, and discuss how Haskell's strong type system allows our library to reject a large class of ill-formed Subjective Logic expressions.

Chapter 4

SLHS: Subjective Logic in Haskell

In this chapter we introduce the library *SLHS: Subjective Logic in Haskell*. SLHS is a library for constructing and evaluating expressions of Subjective Logic. It can be embedded into any existing Haskell project, and, through Haskell's *Foreign Function Interface*, can be utilized by other programming languages, most notably *C* and $C++$.

SLHS is designed to be simple to use: all Subjective Logic operators take in Subjective Logic expressions as input, and return Subjective Logic expressions as output, where Subjective Logic expressions are represented as functions that map some data (frames of discernment, belief mass assignments, configuration information) to some value - typically an opinion. Therefore the operators are *higher order functions*. It will be shown that these Subjective Logic expressions, or *SLExpr*s are a kind of *monad*, and therefore when working with *SLExpr*s one may leverage Haskell's excellent support for monadic programming. We use the monad operators provided by Haskell liberally within the implementation of SLHS, and we utilize Haskell's *do-notation* - a syntactic sugar available when writing monadic programs - to keep the code concise and easy to read.

4.1 Core Components

In this section we will introduce components that form the nucleus of the library. These include the implementation details for objects such as the *frame of discernment*, *belief vectors*, as well as the the *SLExpr* type. The Subjective Logic operators are implemented as functions that take and return objects of type SLExpr, and the monadic interface of SLExpr controls how the expressions are combined.

4.1.1 Belief Vectors

We introduce a special type for representing belief vectors - containers whose elements are belief masses. The reason for introducing a new type instead of simply re-using an existing container type is so that in the future if analysis proves that a different container type provides more efficient operations, then the internal represent of our belief vectors can be changed without affecting any other portion of the SLHS code-base. For the time being we have chosen to use Haskell's Map data type, which is a key-value store backed by an efficient red-black tree. It guarantees $O(\log_2 n)$ time for looking up individual elements, and allows us to traverse the entire tree in $O(n)$ time. thus leads to very efficient Subjective Logic operators.

We start with the definition of the Vector type.

```
newtype Vector a = Vector { unVec :: M.Map a Rational }
```

Next we introduce some functions for converting belief vectors to and from standard Haskell lists.

```
fromList :: Ord a ⇒ [(a, Rational)] → Vector a
fromList = Vector ∘ M.fromList
```

```
toList :: Vector a → [(a, Rational)]
toList = M.toList ∘ unVec
```

Finally, we introduce functions for interfacing with vectors.

```
value :: Ord a ⇒ Vector a → a → Rational
value v x = fromMaybe 0 ∘ M.lookup x $ unVec v

map :: (Rational → Rational) → Vector a → Vector a
map f = Vector ∘ M.map f ∘ unVec

mapWithKey :: (a → Rational → Rational) → Vector a → Vector a
mapWithKey f = Vector ∘ M.mapWithKey f ∘ unVec

fold :: (Rational → b → b) → b → Vector a → b
fold f z = M.fold f z ∘ unVec

focals :: Vector a → [a]
focals = M.keys ∘ unVec

elemsWhere :: (a → Bool) → Vector a → [(a, Rational)]
elemsWhere p = filter (λ(k, _) → p k) ∘ toList
```

value retrieves the value associated with a particular key. *map* allows us to apply a function over each value, returning a new transformed vector. The *mapWithKey* function allows us to map a function over the vector that takes the key into account. *fold* allows us to accumulate a vector into a single value by applying an operator between each element. *focals* returns a list of keys that have non-zero mass. Lastly, *elemsWhere* returns a list of key-value pairs, where the key satisfies a certain predicate.

4.1.2 Frames of Discernment

We represent the frame of discernment as a container type that supports set-like operations such as union and intersection. The reason that we provide our own implementation instead of relying solely on the Set data type provided by Haskell is to allow for future modifications to the SLHS library to swap the underlying data structure, either for performance reasons, or for portability.

We first introduce a new type representing a frame of discernment:

```
newtype Frame a = Frame (S.Set a) deriving (Eq, Ord)
```

By declaring this type using Haskell's *newtype* keyword, we are actually creating a kind of strongly discriminating type alias. That is, representationally Frame a is the same as Set a, however one cannot use a frame when expecting a set, and vice versa.

We then expose the set-theoretic operators that are required by the rest of the library implementation.

```
empty :: Frame a
empty = Frame (S.empty)

isEmpty :: Eq a ⇒ Frame a → Bool
isEmpty f = f == empty

union :: Ord a ⇒ Frame a → Frame a → Frame a
union (Frame s1) (Frame s2) = Frame (s1 `S.union` s2)

isSubsetOf :: Ord a ⇒ Frame a → Frame a → Bool
isSubsetOf (Frame s1) (Frame s2) = s1 `S.isSubsetOf` s2

intersection :: Ord a ⇒ Frame a → Frame a → Frame a
intersection (Frame s1) (Frame s2) = Frame (s1 `S.intersection` s2)

difference :: Ord a ⇒ Frame a → Frame a → Frame a
difference (Frame s1) (Frame s2) = Frame (s1 S.\\ s2)


partition :: (a → Bool) → Frame a → (Frame a, Frame a)
partition p (Frame s) = let (s1, s2) = S.partition p s
                        in (Frame s1, Frame s2)

partitionMany :: [a → Bool] → Frame a → [Frame a]
partitionMany [] frm = [frm]
partitionMany (p:ps) frm = let (f1, f2) = partition p frm
                           in f1 : partitionMany ps f2

size :: Frame a → Int
size (Frame s) = S.size s

map :: (Ord a, Ord b) ⇒ (a → b) → Frame a → Frame b
map f (Frame s) = Frame (S.map f s)

fold :: (a → b → b) → b → Frame a → b
fold f z (Frame s) = S.fold f z s

toList :: Frame a → [a]
toList (Frame s) = S.toList s

fromList :: Ord a ⇒ [a] → Frame a
fromList xs = Frame $ S.fromList xs

singleton :: Ord a ⇒ a → Frame a
singleton x = fromList [x]

member :: Ord a ⇒ a → Frame a → Bool
```

```
member x (Frame s) = x `S.member` s

powerSet :: Ord a ⇒ Frame a → Frame (Frame a)
powerSet (Frame s) = fromList frames
  where
    frames = Prelude.map fromList (subsequences (S.toList s))

reducedPowerSet :: Ord a ⇒ Frame a → Frame (Frame a)
reducedPowerSet frm@(Frame s) = Frame $ S.map Frame rpset'
  where
    (Frame pset) = powerSet frm
    pset' = S.map (λ(Frame x) → x) pset
    rpset = pset' S.\\ S.fromList [S.empty]
    rpset' = rpset S.\\ S.fromList [s]

cross :: (Ord a, Ord b) ⇒ Frame a → Frame b → Frame (a, b)
cross (Frame s1) (Frame s2) = fromList [ (x, y) | x ← S.toList s1, y ← S.toList s2 ]
```

The *cross* function computes the cartesian product of two frames, and the functions *powerSet* and *reducedPowerSet* compute the powerSet and reduced powerSet of the input frame.

4.1.3 Belief Holders

Subjective Logic opinions may include an optional belief holder. Belief holders play an important role for operators such as *transitive discounting* [32], where an agent's opinion of an event is computed through its opinion of a secondary agent, who holds an opinion of the event in question. Other operators that utilize this information are the various belief fusion operators that are designed to merge opinions of events collected either from different sensors, or from the same sensor but across different periods of time.

We represent belief holders as a recursive data type in order to be able to capture complex yet "imaginary" belief holders such as "the consensus of agents A, B and C."

```
data Holder a = None
              | Holder a
              | Product (Holder a) (Holder a)
              | Discount (Holder a) (Holder a)
              | Fuse FusionType (Holder a) (Holder a)
              | Constraint (Holder a) (Holder a)
             deriving (Eq, Ord, Show)
```

Since there are different ways in which two belief holders can be fused into an imaginary holder, the *Fuse* data constructor above takes in an argument of type *FusionType*, which is shown below.

```
data FusionType = Cumulative
                | Averaging
                deriving (Eq, Ord, Show)
```

4.1.4 Subjective Logic Values

Values in SLHS are represented by the following type:

```
data SLVal a = SLVal a
             | Err String
             deriving Show
```

Objects of type *SLVal a* either contain a value of type *a*, via the *SLVal* data constructor, or an error message, via the *Err* data constructor. By wrapping values in this intermediate type, we thus allow all operators in SLHS to return either a value on success, or a detailed error message upon failure. This allows us to report issues with Subjective Logic expressions that can only be detected at run-time.

Objects of type *SLVal a* are also monads. The required type class instance is

```
instance Monad SLVal where
  return = SLVal
  SLVal x >>= f = f x
  Err e   >>= _ = Err e
```

4.1.5 Subjective Logic Expressions

Expressions in Subjective Logic are represented as functions from some input state to some output, such as an opinion, or a rational number.

```
newtype SLExpr h a t = SLExpr (SLState h a → SLVal (SLState h a, t))
```

The *SLExpr* type is parametrized over three types:

- The type *h* represents the type that all belief holders within the Subjective Logic expression must have. For example, if *h* is instantiated to *Int*, then all belief holders must be represented by objects that inhabit the *Int* type.

- The type *a* represents the types that make up the frames of discernment within the expression. Any given Subjective Logic expression can contain references to many frames, but for simplicity of implementation, we enforce the rule that all frames must be made up of elements of the same type. For example, all frames could be inhabited by elements of type *UserDefined*, where *UserDefined* is a type that is created by the user of the library.

- The type *t* represents the output type of the function. The output type is, however, wrapped in the *SLVal* type so that we can return meaningful error messages to the users of the library. We also include the updated state in the output.

All functions of type *SLExpr* map objects of type *SLState* to a pair: the new state after evaluation of the expression, and the result of the expression. *SLState* is a simple aggregate type that allows us to thread the frames of discernment and the belief mass assignments over those frames for each belief holder.

```
data SLState h a =
  SLState
  { slsFrames       :: [F.Frame a]
  , slsBeliefVecs   :: M.Map (F.Frame a) (M.Map (Holder h) (BeliefVector (F.Frame a)))
  , slsBaseRateVecs :: M.Map (F.Frame a) (M.Map (Holder h) (BaseRateVector a))
  } deriving (Show)
```

We provide a function *run* that takes as input a Subjective Logic expression and an initial state, and returns the updated state along with the value computed by the expression.

```
run :: SLExpr h a t → SLState h a → SLVal (SLState h a, t)
run (SLExpr f) st = f st
```

If the user does not care about the final state of the computation and only wants to see the final value, we provide the function *run'*:

```
run' :: SLExpr h a t → SLState h a → SLVal t
run' (SLExpr f) st = liftM snd $ f st
```

Lastly, objects of type *SLExpr* form a *monad*, and thus we can take advantage of Haskell's support for programming with monads. We provide the definitions for *bind* and *inject* below. Furthermore, since all monads are applicative functors, and all applicative functors are functors, we provide those definitions also. This allows the user of our library to program in a monadic, applicative, or functorial style.

```
instance Monad (SLExpr h a) where
  return x = SLExpr $ λst → return (st, x)

  ma >>= f = SLExpr $ λst → case (run ma st) of
    Err e           → Err e
    SLVal (st', a) → let mb = f a in case run mb st' of
      Err e   → Err e
      SLVal r → SLVal r

instance Applicative (SLExpr h a) where
  pure = return
  (<*>) = ap

instance Functor (SLExpr h a) where
  fmap = liftA
```

4.2 Opinions

In this section we discuss the implementations of the various kinds of subjective opinions. We start by implementing binomial opinions, and then we present multinomial and hyper opinions.

4.2.1 Binomial Opinions

We represent binomial opinions by four rational numbers corresponding to the belief, disbelief, uncertainty, and base rate of the opinion, along with some additional meta-data: the belief holder and the frame of discernment it is defined over. In code, the binomial opinion looks like the following:

```
data Binomial h a = Binomial { bBelief      :: Rational
                             , bDisbelief   :: Rational
                             , bUncertainty :: Rational
                             , bAtomicity   :: Rational
                             , bHolder      :: Holder h
                             , bX           :: a
                             , bNotX        :: a
                             }
```

Here we use Haskell's *record syntax* to define the data constructor. Haskell automatically creates the top-level functions *bBelief*, *bDisbelief*, *bUncertainty*, *bAtomicity*, *bHolder*, *bX*, and *bNotX* that provide access to the respective items of the record.

Lastly, we also introduce a special *type class* called *ToBinomial* which allows us to define a range of types that can be converted to a binomial opinion. An example of such a type could be a *Beta PDF*. We will re-use this strategy for implementing multinomial and hyper opinions.

```
class ToBinomial op where
  toBinomial :: op h a → Binomial h a

instance ToBinomial Binomial where
  toBinomial = id
```

4.2.2 Multinomial Opinions

Multinomials are represented as records containing a *BeliefVector* to represent the amount of belief assigned to each element of the frame, a scalar rational number to store the uncertainty mass, a *BaseRateVector* which assigns each element in the frame to a base rate, a

belief holder, and a reference to the frame of discernment.

```
data Multinomial h a = Multinomial { mBelief      :: BeliefVector a
                                   , mUncertainty :: Rational
                                   , mBaseRate    :: BaseRateVector a
                                   , mHolder      :: Holder h
                                   , mFrame       :: F.Frame a
                                   }
```

Just as in the case of binomials, we introduce a type class to represent types that can be converted to multinomials. We provide the instance for multinomial opinions (the identity function) as well as an instance for binomial opinions, since binomial opinions are a special case of multinomial opinions.

```
class ToMultinomial op where
  toMultinomial :: Ord a ⇒ op h a → Multinomial h a

instance ToMultinomial Multinomial where
  toMultinomial = id

instance ToMultinomial Binomial where
  toMultinomial (Binomial b d u a h x y) = Multinomial b' u a' h f
    where
      b' = V.fromList [ (x, b), (y, d) ]
      a' = V.fromList [ (x, a), (y, 1 - a) ]
      f  = F.fromList [x, y]
```

4.2.3 Hyper Opinions

Hyper opinions share a similar structural layout to multinomial opinions except the belief vector spans the reduced power set of the frame, and is thus represented as a *BeliefVector* with sub-frames as the keys, instead of elements of the frame.

```
data Hyper h a = Hyper { hBelief      :: BeliefVector (F.Frame a)
                       , hUncertainty :: Rational
                       , hBaseRate    :: BaseRateVector a
                       , hHolder      :: Holder h
                       , hFrame       :: F.Frame a
                       }

class ToHyper op where
  toHyper :: Ord a ⇒ op h a → Hyper h a

instance ToHyper Hyper where
  toHyper = id

instance ToHyper Multinomial where
  toHyper (Multinomial b u a h f) = Hyper b' u a h f
```

```
    where
      b' = V.fromList ∘ map (first F.singleton) ∘ V.toList $ b

instance ToHyper Binomial where
  toHyper = toHyper ∘ toMultinomial
```

4.2.4 The Opinion Type Class

There are certain operations that are common amongst all opinions. One example of such operation is the *probability expectation*: for binomials, the probability expectation is a simple scalar, whereas for multinomial and hyper opinions the probability expectation is a vector over the frame of discernment, and the reduced power set of the frame, respectively.

```
class Opinion op h a where
  type ExpectationType op h a :: ∗

  expectation :: op h a → ExpectationType op h a
  getFrame    :: op h a → F.Frame a
```

In order to accomodate a function such as probability expectation that returns a value of a different type depending on the type of the opinion, we use an *indexed type family* [40]. For each opinion type, we associate an ”expectation type”, which is the type one would obtain when querying the probability expectation of the opinion. The instances for each of the three opinion types follows.

```
instance Ord a ⇒ Opinion Binomial h a where
  type ExpectationType Binomial h a = Rational

  expectation (Binomial b d u a _ _ _)          = b + a ∗ u
  getFrame (Binomial _ _ _ _ _ f1 f2) = F.fromList [f1, f2]

instance Ord a ⇒ Opinion Multinomial h a where
  type ExpectationType Multinomial h a = V.Vector a

  expectation (Multinomial b u a _ f) = V.fromList vals
    where
      vals = map (λk → (k, V.value b k + V.value a k + u)) keys
      keys = F.toList f

  getFrame (Multinomial _ _ _ _ frm) = frm

instance Ord a ⇒ Opinion Hyper h a where
  type ExpectationType Hyper h a = V.Vector (F.Frame a)

  expectation (Hyper b u a _ f) = V.fromList vals
```

```
    where
      vals = map (λk → (k, V.value b k + aval k + u)) keys
      keys = F.toList ∘ F.reducedPowerSet $ f
      aval k = sum ∘ map (V.value a) ∘ F.toList $ k

  getFrame (Hyper _ _ _ _ frm) = frm
```

4.2.5 Belief Coarsening

Coarsening is an operation that takes a hyper opinion and converts it into a binomial opinion. The inputs are an arbitrary hyper opinion and a subset of the frame of discernment for which the hyper opinion is defined over. Coarsening is a two-stage operation: First the frame of discernment is partitioned into two sets: the subset given as input, and everything else. These two subsets, taken together as a set, form a new binary frame with which the new binomial opinion will be defined over. Secondly, the belief masses associated with elements of the power set of the original frame via the hyper opinion input are split up and assigned to the elements of the new frame. The resulting belief mass assignment preserves additivity, and thus the new binomial opinion is valid. The operation for coarsening is given below.

```
coarsen :: (ToHyper op, Ord b)
           ⇒ SLExpr h a (op h b)
           → F.Frame b → SLExpr h a (Binomial h (F.Frame b))
coarsen op theta = liftM2 coarsen' op (return theta)
  where
    coarsen' op theta = Binomial b d u a holder theta (frm `F.difference` theta)
      where
        b = sumSnd ∘ V.elemsWhere subset         $ belief
        d = sumSnd ∘ V.elemsWhere emptyIntersect $ belief
        u = 1 - b - d
        a = sum ∘ F.toList ∘ F.map baseRate      $ theta

        belief   = hBelief ∘ toHyper $ op
        baseRate = V.value (hBaseRate ∘ toHyper $ op)

        holder   = hHolder ∘ toHyper $ op
        frm      = hFrame  ∘ toHyper $ op

        sumSnd         = sum ∘ map snd
        subset         = (`F.isSubsetOf` theta)
        emptyIntersect = F.isEmpty ∘ (`F.intersection` theta)
```

As a convenience, we also offer a function to coarsen a hyper opinion, not by an explicitly given sub-frame, but by those elements of the frame that satisfy a given predicate.

```
coarsenBy :: (ToHyper op, Ord b) ⇒ SLExpr h a (op h b)
             → (b → Bool) → SLExpr h a (Binomial h (F.Frame b))
coarsenBy op pred = op >>= λop' →
  let (theta, _) = F.partition pred ∘ getFrame ∘ toHyper $ op'
  in  coarsen op theta
```

As an example, consider a frame of discernment containing the integer values one through twenty, and a hyper opinion ω^A defined over the frame. We can then construct a binomial opinion $\omega^A_{P(x)} = \langle b_{P(x)}, d_{P(x)}, u_{P(x)}, a_{P(x)} \rangle$, where the predicate $P(x)$ denotes "x is even" by utilizing the *coarsenBy* function:

```
isEven :: Int → Bool
isEven n = n `mod` 2 == 0

evenOpinion = coarsenBy isEven oldOpinion
```

where *oldOpinion* is the initial hyper opinion.

4.2.6 Accessing Opinions

SLHS is built around combining together objects of type *SLExpr*, which are functions from some world state to some value. Since Subjective Logic operators rely on opinions as inputs, we require a method of obtaining the opinions stored in the state that is being threaded through behind the Subjective Logic expressions. The following functions do just that.

We start with fetching hyper opinions, as they are the most general. Given a belief holder h and an index *idx* corresponding to the *idx*'th frame of discernment in the state, *getHyper* returns either a hyper opinion held by h over the *idx*th frame, or a run-time error message.

```
getHyper :: (Ord h, Ord a) ⇒ h → Int → SLExpr h a (Hyper h a)
getHyper holder idx = do
  frames ← liftM slsFrames getState
```

```
  vecs   ← liftM slsBeliefVecs getState
  rates  ← liftM slsBaseRateVecs getState
  if idx > length frames
    then err "getHyper: index out of range"
    else do let frm = frames !! idx
            m ← do case M.lookup frm vecs of
                     Nothing → err "getHyper: no mass assignments for that frame"
                     Just m  → do
                       case M.lookup (Holder holder) m of
                         Nothing → err "getHyper: no mass assignment for that holder"
                         Just m' → return m'

            a ← do case M.lookup frm rates of
                     Nothing → err "getHyper: no base rates for that frame"
                     Just a → do
                       case M.lookup (Holder holder) a of
                         Nothing → err "getHyper: no base rate for that holder"
                         Just a' → return a'

            let u = 1 - V.fold (+) 0 m
            return $ Hyper m u a (Holder holder) frm
```

While the above function looks fairly complicated, it simply unwraps the relevant state data from the *SLExpr* monad, checks to see if the index is within the bounds of the array of frames, and then looks to see if there are any mass assignments for that particular frame. If there are mass assignments for that frame, then we look up the particular mass assignment owned by the belief holder. If one exists, we return it, else we return an error message. We perform a similar unwrapping for checking for base rates, and then compute the uncertainty and return the resulting hyper opinion.

Next we have a way of obtaining multinomial opinions. Since multinomial opinions are a special case of hyper opinions, we first obtain the hyper opinion via a call to *getHyper*, and then check to see if we can safely convert that hyper opinion into a multinomial opinion. If so, we return it, else we return an error message.

```
getMultinomial :: (Ord h, Ord a) ⇒ h → Int → SLExpr h a (Multinomial h a)
getMultinomial holder f = do
  h ← getHyper holder f
  case maybeToMultinomial h of
    Nothing → err "getMultinomial: not a multinomial opinion"
    Just m  → return m
  where
    maybeToMultinomial (Hyper b u a h f) =
      let fs = V.focals b
      in if all (λf → F.size f == 1) fs
         then let bv = V.toList b
```

```
            bv' = map (λ(a, r) → ((F.toList a) !! 0, r)) bv
        in Just $ Multinomial (V.fromList bv') u a h f
    else Nothing
```

The same trick applies to obtaining binomial opinions. We first obtain the relevant multinomial opinion and then see if we can safely convert it into a binomial opinion. If so, great! Otherwise we return an error message to the user.

```
getBinomial :: (Ord h, Ord a) ⇒ h → Int → a → SLExpr h a (Binomial h a)
getBinomial holder f x = do
  m ← getMultinomial holder f
  case maybeToBinomial x m of
    Nothing → err "getBinomial: not a binomial opinion"
    Just b  → return b
  where
    maybeToBinomial x (Multinomial b u a h f) = do
      guard (F.size f == 2)
      guard (x `F.member` f)
      let y = fst ∘ head ∘ V.elemsWhere (/= x) $ b
      let b' = V.value b x
      let d' = V.value b y
      let u' = 1 - b' - d'
      let a' = V.value a x
      return $ Binomial b' d' u' a' h x y
```

In the above code for *maybeToBinomial* we utilize the fact that the *Maybe* type is an instance of the type class *MonadPlus*, which gives us access to the *guard* function. MonadPlus can be thought of the set of types that are monads, but also have the additive properties of monoids: a zero element (in the case of Maybe, the Nothing data constructor), and a method of combining two MonadPlus objects together, which in Haskell is called *mplus* [21]. Unfortunately the rules for identity and associativity cannot be enforced in the language itself.

4.3 Operators

In this section we discuss the implementation details of the Subjective Logic operators that are provided by SLHS. The following notation is used for the operators:

- We denote binary operators with a trailing exclamation mark ! in order to avoid conflicting with Haskell's mathematical operators. For example, binomial addition is denoted as $+!$.

- We use tildes as a prefix to denote $co-$ operations. For example, the binomial comultiplication operator is denoted as $\sim *!$.

- All n-ary operators, where $n > 2$ are denoted as simple functions, instead of symbolic operators.

Every operator is presented in its most general form. For example, instead of presenting two operators for *averaging fusion* (one for multinomial opinions, and another for hyper opinions) we implement only the version for hyper opinions. In order to achieve this level of code reuse, each operator accepts as parameters any object that can be converted into the correct opinion type by virtue of the *ToBinomial*, *ToMultinomial*, and *ToHyper* type classes.

4.3.1 Binomial Operators

We begin our treatment of the Subjective Logic operators by looking at those operators designed to work with binomial opinions. We split this section into two parts: *logical and set-theoretical* operators, and *trust transitivity* operators. The former contains the operators that are generalizations of those found in logic and set theory, such as conjunction, and set union. The latter operators are for modeling trust networks, where agents can formulate opinions based on reputation and trust.

Logical and Set-Theoretical Operators

The logical and set-theoretical binomial operators are those that have equivalent operators in logic and set theory. We will start with binomial addition. Addition of binomial opinions, denoted as $\omega_{x\cup y} = \omega_x + \omega_y$, is defined when x and y are disjoint subsets of the same frame of discernment [50]. Binomial addition is implemented as follows:

```
(+!) :: (ToBinomial op1, ToBinomial op2, Eq h, Eq b, Ord b)
       ⇒ SLExpr h a (op1 h (F.Frame b))
       → SLExpr h a (op2 h (F.Frame b))
       → SLExpr h a (Binomial h (F.Frame b))
opx +! opy = do
  opx' ← liftM toBinomial opx
  opy' ← liftM toBinomial opy
  require (bHolder opx' == bHolder opy') "opinions must have same holder"
  require (getFrame opx' == getFrame opy') "opinions must have the same frame"
  return $ add' opx' opy'


add' :: Ord a
        ⇒ Binomial h (F.Frame a) → Binomial h (F.Frame a) → Binomial h (F.Frame a)
add' opx@(Binomial bx dx ux ax hx xt xf) (Binomial by dy uy ay _ yt yf) =
  Binomial b' d' u' a' hx (xt `F.union` yt) (xf `F.union` yf)
  where
    b' = bx + by
    d' = (ax * (dx - by) + ay * (dy - bx)) / (ax + ay)
    u' = (ax * ux + ay * uy) / (ax + ay)
    a' = ax + ay
```

Here we see a pattern that we will re-use for all operator implementations. We start with a function whose inputs are of type *SLExpr h a t*, where *t* is some type. Within that function, we unwrap the values from the *SLExpr* monad, verify that some requirements are met, and then send those values to a worker function that does the actual computation. We then wrap the result back into the *SLExpr* monad via the *return* function.

Binomial subtraction is the inverse operation of addition. In set theory it is equivalent to the set difference operator [50]. Given two opinions ω_x and ω_y where $x \cap y = y$, the difference, $\omega_{x\setminus y}$ is calculated as follows:

```
(-!) :: (ToBinomial op1, ToBinomial op2, Eq h, Eq b, Ord b)
       ⇒ SLExpr h a (op1 h (F.Frame b))
       → SLExpr h a (op2 h (F.Frame b))
       → SLExpr h a (Binomial h (F.Frame b))
```

```
opx -! opy = do
  opx' ← liftM toBinomial opx
  opy' ← liftM toBinomial opy
  require (bHolder opx' == bHolder opy') "opinions must have same holder"
  require (getFrame opx' == getFrame opy') "opinions must have the same frame"
  return $ subtract' opx' opy'


subtract' :: Ord a
             ⇒ Binomial h (F.Frame a) → Binomial h (F.Frame a) → Binomial h (F.Frame a)
subtract' (Binomial bx dx ux ax hx xt xf) (Binomial by dy uy ay _ yt yf) =
  Binomial b' d' u' a' hx ft ff
  where
    b' = bx - by
    d' = (ax * (dx + by) - ay * (1 + by - bx - uy)) / (ax - ay)
    u' = (ax * ux - ay * uy) / (ax - ay)
    a' = ax - ay
    ft = xt `F.difference` yt
    ff = xt `F.union` xf `F.difference` ft
```

Negation is a unary operator that switches the belief and disbelief and inverts the atomicity of a binomial opinion [23]. Given a binomial opinion ω_x over a frame $X = \{x, \neg x\}$, the negated opinion $\omega_{\bar{x}} = \omega_{\neg x}$.

```
negate :: ToBinomial op ⇒ SLExpr h a (op h b) → SLExpr h a (Binomial h b)
negate op = do
  op' ← liftM toBinomial op
  return $ negate' op'


negate' :: Binomial h a → Binomial h a
negate' (Binomial b d u a h x y) = Binomial d b u (1 - a) h y x
```

Multiplication of two binomial opinions is equivalent to the logical *and* operator [33]. Given two opinions ω_x and ω_y over distinct binary frames x and y, the product of the opinions, $\omega_{x \wedge y}$, represents the conjunction of the two opinions.

```
(*!) :: (ToBinomial op1, ToBinomial op2, Eq h, Ord b, Ord c)
        ⇒ SLExpr h a (op1 h b)
        → SLExpr h a (op2 h c)
        → SLExpr h a (Binomial h (F.Frame (b, c)))
opx *! opy = do
  opx' ← liftM toBinomial opx
  opy' ← liftM toBinomial opy
  require (bHolder opx' == bHolder opy') "opinions must have same holder"
  return $ b_times' opx' opy'

b_times' (Binomial bx dx ux ax hx xt xf) (Binomial by dy uy ay _ yt yf) =
  Binomial b' d' u' a' hx t f
  where
    b' = bx * by + ((1 - ax) * bx * uy + (1 - ay) * ux * by)
         / (1 - ax * ay)
    d' = dx + dy - dx * dy
    u' = ux * uy + ((1 - ay) * bx * uy + (1 - ax) * ux * by)
         / (1 - ax * ay)
```

```
    a' = ax * ay

    t = F.singleton (xt, yt)
    f = F.fromList [(xt, yf), (xf, yt), (xf, yf)]
```

The resulting frame of discernment is a coarsened frame from the cartesian product of $\{x, \neg x\}$ and $\{y, \neg y\}$, where the element whose belief mass is designated the role of "belief" for binomial opinions is $\{(x, y)\}$, and the element whose belief mass is given the role of "disbelief" is $\{(x, \neg y), (\neg x, y), (\neg x, \neg y)\}$.

Binomial co-multiplication is equivalent to the logical *or* operator [33]. Given two opinions, again on distinct binary frames, ω_x and ω_y, the disjunctive binomial opinion $\omega_{x \vee y} = \omega_x \sqcup \omega_y$ is computed by the following function:

```
(~*!) :: (ToBinomial op1, ToBinomial op2, Eq h, Ord b, Ord c)
         ⇒ SLExpr h a (op1 h b)
         → SLExpr h a (op2 h c)
         → SLExpr h a (Binomial h (F.Frame (b, c)))
opx ~*! opy = do
  opx' ← liftM toBinomial opx
  opy' ← liftM toBinomial opy
  require (bHolder opx' == bHolder opy') "opinions must have same holder"
  return $ cotimes' opx' opy'

cotimes' (Binomial bx dx ux ax hx xt xf) (Binomial by dy uy ay _ yt yf) =
  Binomial b' d' u' a' hx t f
  where
    b' = bx + by - bx * by
    d' = dx * dy + (ax * (1 - ay) * dx * uy + (1 - ax) * ay * ux * dy)
         / (ax + ay - ax * ay)
    u' = ux * uy + (ay * dx * uy + ax * ux * dy)
         / (ax + ay - ax * ay)
    a' = ax + ay - ax * ay

    t = F.fromList [(xt, yt), (xf, yt), (xt, yf)]
    f = F.singleton (xf, yf)
```

Binomial multiplication and co-multiplication are duals to one another and satisfy De-Morgan's law: $\omega_{x \wedge y} = \omega_{\overline{\overline{x} \vee \overline{y}}}$ and $\omega_{x \vee y} = \omega_{\overline{\overline{x} \wedge \overline{y}}}$, but they do not distribute over one another [33]. Josang and McAnally claim that binomial multiplication and co-multiplication produce good approximations of the analytically correct products and co-products of Beta probability density functions [33]. Therefore, if one were to construct a *Beta* data type in Haskell representing a beta PDF and create an instance of the *ToBinomial* type class for

it, one could use the above operators to generate good approximations to the products and co-products of beta PDFs with minimal effort.

We next discuss binomial division and co-division, which are the inverses of binomial multiplication and co-multiplication. The binomial division of an opinion ω_x by another opinion ω_y is denoted as $\omega_{x\overline{\wedge}y} = \omega_x / \omega_y$ [33], and is computed as follows:

```
(/!) :: (ToBinomial op1, ToBinomial op2, Eq c)
        ⇒ SLExpr h a (op1 h (F.Frame (b, c)))
        → SLExpr h a (op2 h b)
        → SLExpr h a (Binomial h c)
opx /! opy = do
  opx' ← liftM toBinomial opx
  opy' ← liftM toBinomial opy
  require (lessBaseRate opx' opy') "ax must be less than ay"
  require (greaterDisbelief opx' opy') "dx must be greater than or equal to dy"
  require (bxConstraint opx' opy') "Division requirement not satisfied"
  require (uxConstraint opx' opy') "Division requirement not satisfied"
  return $ divide' opx' opy'
  where
    lessBaseRate x y     = bAtomicity x < bAtomicity y
    greaterDisbelief x y = bDisbelief x ≥ bDisbelief y

    bxConstraint x y = bx ≥ (ax * (1 - ay) * (1 - dx) * by) / ((1 - ax) * ay * (1 - dy))
      where
        (bx, dx, ux, ax) = (bBelief x, bDisbelief x, bUncertainty x, bAtomicity x)
        (by, dy, uy, ay) = (bBelief y, bDisbelief y, bUncertainty y, bAtomicity y)

    uxConstraint x y = ux ≥ ((1 - ay) * (1 - dx) * uy) / ((1 - ax) * (1 - dy))
      where
        (bx, dx, ux, ax) = (bBelief x, bDisbelief x, bUncertainty x, bAtomicity x)
        (by, dy, uy, ay) = (bBelief y, bDisbelief y, bUncertainty y, bAtomicity y)

divide' (Binomial bx dx ux ax hx xt xf) (Binomial by dy uy ay _ yt yf) =
  Binomial b' d' u' a' hx zt zf
  where
    b' = ay * (bx + ax * ux) / ((ay - ax) * (by + ay *uy))
         - ax * (1 - dx) / ((ay - ax) * (1 - dy))
    d' = (dx - dy) / (1 - dy)
    u' = ay * (1 - dx) / ((ay - ax) * (1 - dy))
         - ay * (bx + ax * ux) / ((ay - ax) * (bx + ay * uy))
    a' = ax / ay

    [(_, zt)] = F.toList xt
    zf = head ∘ filter (/= zt) ∘ map snd ∘ F.toList $ xf
```

Lastly co-division, the inverse operation of co-multiplication [33], is denoted as $\omega_{x\overline{\vee}y} = \omega_x \overline{\square} \omega_y$ and is computed as follows:

```
(~/!) :: (ToBinomial op1, ToBinomial op2, Eq c)
         ⇒ SLExpr h a (op1 h (F.Frame (b, c)))
         → SLExpr h a (op2 h b)
         → SLExpr h a (Binomial h c)
opx ~/! opy = do
  opx' ← liftM toBinomial opx
```

```
  opy' ← liftM toBinomial opy
  require (greaterBaseRate opx' opy') "ax must be greater than ay"
  require (greaterBelief opx' opy') "bx must be greater than or equal to by"
  require (dxConstraint opx' opy') "Division requirement not satisfied"
  require (uxConstraint opx' opy') "Division requirement not satisfied"
  return $ codivide' opx' opy'
  where
    greaterBaseRate x y     = bAtomicity x > bAtomicity y
    greaterBelief x y = bBelief x ≥ bBelief y

    dxConstraint x y = dx ≥ (ay * (1 - ax) * (1 - bx) * dy) / ((1 - ay) * ax * (1 - by))
      where
        (bx, dx, ux, ax) = (bBelief x, bDisbelief x, bUncertainty x, bAtomicity x)
        (by, dy, uy, ay) = (bBelief y, bDisbelief y, bUncertainty y, bAtomicity y)

    uxConstraint x y = ux ≥ (ay * (1 - bx) * uy) / (ax * (1 - by))
      where
        (bx, dx, ux, ax) = (bBelief x, bDisbelief x, bUncertainty x, bAtomicity x)
        (by, dy, uy, ay) = (bBelief y, bDisbelief y, bUncertainty y, bAtomicity y)

codivide' (Binomial bx dx ux ax hx xt xf) (Binomial by dy uy ay _ yt yf) =
  Binomial b' d' u' a' hx zt zf
  where
    b' = (bx - by) / (1 - by)
    d' = ((1 - ay) * (dx + (1 - ax) * ux)
          / ((ax - ay) * (dy + (1 - ay) * uy)))
         - (1 - ax) * (1 - bx) / ((ax - ay) * (1 - by))
    u' = ((1 - ay) * (1 - bx) / ((ax - ay) * (1 - by)))
         - ((1 - ay) * (dx + (1 - ax) * ux)
            / ((ax - ay) * (dy + (1 - ay) * uy)))
    a' = (ax - ay) / (1 - ay)

    zt = head ∘ filter (/= zf) ∘ map snd ∘ F.toList $ xt
    [(_, zf)] = F.toList xf
```

In this section we have introduced those binomial operators that have analogs to logic and set theory. In the next section we discuss the binomial operators for modeling *trust transitivity*.

Trust Transitivity Operators

In this section we present the Subjective Logic operators for trust transitivity. If two agents A and B exist such that agent A has an opinion of agent B, and agent B has an opinion about some proposition X, then A can form an opinion of X by *discounting* B's opinion of x based on A's opinion of B.

Subjective Logic offers three methods of discounting: *uncertainty favouring discounting*, *opposite belief favouring discounting*, and *base rate sensitive discounting* [28]. We

begin by constructing a simple data type to represent the three kinds of discounting.

```
data Favouring = Uncertainty | Opposite | BaseRateSensitive
```

By doing so, we are able to expose a single discounting function to the user that selects the kind of discounting based on an input parameter of type *Favouring*:

```
discount :: (ToBinomial op1, ToBinomial op2, Ord h, Ord b)
            ⇒ Favouring
            →SLExpr h a (op1 h h)
            → SLExpr h a (op2 h b)
            → SLExpr h a (Binomial h b)
discount f opx opy = do
  opx' ← liftM toBinomial opx
  opy' ← liftM toBinomial opy
  return $ case f of
    Uncertainty       → discount_u opx' opy'
    Opposite          → discount_o opx' opy'
    BaseRateSensitive → discount_b opx' opy'
```

Depending on the first parameter, the discount function dispatches to one of three implementations: *discount_u*, *discount_o*, or *discount_b*. Their definitions follow below.

```
discount_u :: Binomial h h → Binomial h a → Binomial h a
discount_u (Binomial bb db ub ab hx _ _) (Binomial bx dx ux ax hy fx fy) =
  Binomial b' d' u' a' (Discount hx hy) fx fy
  where
    b' = bb * bx
    d' = bb * dx
    u' = db + ub + bb * ux
    a' = ax


discount_o :: Binomial h h → Binomial h a → Binomial h a
discount_o (Binomial bb db ub ab hx _ _) (Binomial bx dx ux ax hy fx fy) =
  Binomial b' d' u' a' (Discount hx hy) fx fy
  where
    b' = bb * bx + db * dx
    d' = bb * dx + db * bx
    u' = ub + (bb + db) * ux
    a' = ax


discount_b :: (Ord a, Ord h) ⇒ Binomial h h → Binomial h a → Binomial h a
discount_b op1@(Binomial bb db ub ab hx _ _) op2@(Binomial bx dx ux ax hy fx fy) =
  Binomial b' d' u' a' (Discount hx hy) fx fy
  where
    b' = expectation op1 * bx
    d' = expectation op1 * dx
    u' = 1 - expectation op1 * (bx + dx)
    a' = ax
```

In this section we have presented the operators of Subjective Logic for working with binomial opinions. We first introduced the operators that have analogs to the classical

Name	SL Notation	SLHS Notation
Addition	$\omega_{X\cup Y} = \omega_X + \omega_Y$	$opx+!opy$
Subtraction	$\omega_{X\setminus Y} = \omega_X - \omega_Y$	$opx-!opy$
Negation	$\omega_{\bar{x}} = \neg\omega_x$	$negateopx$
Multiplication	$\omega_{X\wedge Y} = \omega_X \cdot \omega_Y$	$opx*!opy$
Co-multiplication	$\omega_{X\vee Y} = \omega_X \sqcup \omega_Y$	$opx\ *!opy$
Division	$\omega_{X\bar{\wedge}Y} = \omega_X / \omega_Y$	$opx/!opy$
Co-division	$\omega_{X\bar{\vee}Y} = \omega_X \bar{\sqcup} \omega_Y$	$opx\ /!opy$
Discounting	$\omega_x^{A:B} = \omega_B^A \otimes \omega_x^B$	$discount\,t\,opa\,opb$

Table 4.1: Summary of binomial operators

operators of logic and set theory, and then introduced operators for modeling transitive trust networks. These operators are summarized in Table 4.1. In the next section we introduce the operators of Subjective Logic for working with multinomial and hyper opinions.

4.3.2 Multinomial and Hyper Operators

In this section we present the multinomial and hyper operators. We start with multinomial multiplication and describe how it differs from binomial multiplication [33], then we introduce the various operators for belief *fusion* and *unfusion* [22, 31, 27, 26]. We then introduce the *deduction* and *abduction* operators for reasoning [35, 25, 24], and lastly we introduce the *belief constraint* operator [34].

Multinomial Multiplication

The multiplication of two multinomial opinions is a separate operator than the product operator defined over binomial opinions. Whereas the binomial product operator is equivalent to the logical *and* operator, multinomial multiplication constructs an opinion over a new frame which is the cartesian product of the frames of the input opinions [33]. In order

to avoid symbolic naming conflicts, we have chosen to name the binomial operator with the symbol $*!$, and we have used the name *times* to denote the multinomial operator.

```
times :: (ToMultinomial op1, ToMultinomial op2, Eq h, Ord b, Ord c)
         ⇒ SLExpr h a (op1 h b) → SLExpr h a (op2 h c)
         → SLExpr h a (Multinomial h (b, c))
times opx opy = do
  opx' ← liftM toMultinomial opx
  opy' ← liftM toMultinomial opy
  return $ m_times' opx' opy'


m_times' :: (Ord a, Ord b) ⇒ Multinomial h a → Multinomial h b → Multinomial h (a, b)
m_times' (Multinomial bx ux ax hx fx) (Multinomial by uy ay hy fy) =
  Multinomial b' u' a' (Product hx hy) (fx `F.cross` fy)
  where
    b' = V.fromList bxy
    u' = uxy
    a' = V.fromList axy

    bxy = [ ((x, y), f x y) | x ← xKeys, y ← yKeys ]
      where
        f x y = expect x y - (V.value ax x * V.value ay y * uxy)

    axy = [ ((x, y), f x y) | x ← xKeys, y ← yKeys ]
      where
        f x y = V.value ax x * V.value ay y

    uxy = minimum [ uxy' x y | x ← xKeys, y ← yKeys ]

    uxy' x y = (uIxy * expect x y) / (bIxy x y + V.value ax x * V.value ay y * uIxy)

    uIxy = uRxy + uCxy + uFxy
      where
        uRxy = sum [ ux * V.value by y | y ← yKeys ]
        uCxy = sum [ uy * V.value bx x | x ← xKeys ]
        uFxy = ux * uy

    bIxy x y = V.value bx x * V.value by y

    expect x y = (V.value bx x + V.value ax x * ux) * (V.value by y + V.value ay y * uy)

    xKeys = F.toList fx
    yKeys = F.toList fy
```

Fusion, Unfusion, and Fission

Hyper opinions can be fused together using two different operators: *cumulative fusion* and *averaging fusion*. Each operator should be used under different circumstances depending on the meaning of the fused opinions [31, 22].

```
cFuse :: (ToHyper op1, ToHyper op2, Ord b)
         ⇒ SLExpr h a (op1 h b) → SLExpr h a (op2 h b) → SLExpr h a (Hyper h b)
cFuse opa opb = do
  opa' ← liftM toHyper opa
```

```
  opb' ← liftM toHyper opb
  return $ cFuse' opa' opb'


cFuse' :: Ord a ⇒ Hyper h a → Hyper h a → Hyper h a
cFuse' (Hyper ba ua aa hx fx) (Hyper bb ub ab hy _)
  | ua /= 0 || ub /= 0 = Hyper b' u' a' (Fuse Cumulative hx hy) fx
  | otherwise          = Hyper b'' u'' a'' (Fuse Cumulative hx hy) fx
  where
    b' = V.fromList ∘ map (λk → (k, bFunc k)) $ keys
    u' = ua * ub / (ua + ub - ua * ub)
    a' = aa

    b'' = V.fromList ∘ map (λk → (k, bB k)) $ keys
    u'' = 0
    a'' = aa

    bFunc x = (bA x * ub + bB x * ua) / (ua + ub - ua * ub)

    keys  = nub (V.focals ba ++ V.focals bb)

    bA = V.value ba
    bB = V.value bb


aFuse :: (ToHyper op1, ToHyper op2, Ord a)
         ⇒ SLExpr h a (op1 h a) → SLExpr h a (op2 h a) → SLExpr h a (Hyper h a)
aFuse opa opb = do
  opa' ← liftM toHyper opa
  opb' ← liftM toHyper opb
  return $ aFuse' opa' opb'


aFuse' :: Ord a ⇒ Hyper h a → Hyper h a → Hyper h a
aFuse' (Hyper ba ua aa hx fx) (Hyper bb ub ab hy _)
  | ua /= 0 || ub /= 0 = Hyper b' u' a' (Fuse Averaging hx hy) fx
  | otherwise          = Hyper b'' u'' a'' (Fuse Averaging hx hy) fx
  where
    b' = V.fromList ∘ map (λk → (k, bFunc k)) $ keys
    u' = 2 * ua * ub / (ua + ub)
    a' = aa

    b'' = V.fromList ∘ map (λk → (k, bB k)) $ keys
    u'' = 0
    a'' = aa

    bFunc x = (bA x * ub + bB x * ua) / (ua + ub)

    keys  = nub (V.focals ba ++ V.focals bb)

    bA = V.value ba
    bB = V.value bb
```

Cumulative *unfusion* is defined for multinomial opinions [26]. It has yet to be generalized to hyper opinions. Given an opinion that represents the result of cumulatively fusing together two opinions, and one of the two original opinions, it is possible to extract the other original opinion.

```
cUnfuse :: (ToMultinomial op1, ToMultinomial op2, Ord a)
```

```
          ⇒ SLExpr h a (op1 h a) → SLExpr h a (op2 h a)
          → SLExpr h a (Multinomial h a)
cUnfuse opc opb = do
  opc' ← liftM toMultinomial opc
  opb' ← liftM toMultinomial opb
  return $ cUnfuse' opc' opb'


cUnfuse' :: Ord a ⇒ Multinomial h a → Multinomial h a → Multinomial h a
cUnfuse' (Multinomial bc uc ac (Fuse Cumulative hx hy) fx) (Multinomial bb ub ab _ _)
  | uc /= 0 || ub /= 0 = Multinomial ba ua aa hx fx
  | otherwise          = Multinomial ba' ua' aa' hx fx
  where
    ba = V.mapWithKey belief bc
    ua = ub * uc / (ub - uc + ub * uc)
    aa = ac

    ba' = bb
    ua' = 0
    aa' = ac

    belief x b = (b * ub - V.value bb x  * uc) / (ub - uc + ub * uc)
```

Likewise, averaging unfusion is the inverse operation to averaging fusion [26].

```
aUnfuse :: (ToMultinomial op1, ToMultinomial op2, Ord a)
          ⇒ SLExpr h a (op1 h a) → SLExpr h a (op2 h a)
          → SLExpr h a (Multinomial h a)
aUnfuse opc opb = do
  opc' ← liftM toMultinomial opc
  opb' ← liftM toMultinomial opb
  return $ aUnfuse' opc' opb'


aUnfuse' :: Ord a ⇒ Multinomial h a → Multinomial h a → Multinomial h a
aUnfuse' (Multinomial bc uc ac (Fuse Averaging hx hy) fx) (Multinomial bb ub ab _ _)
  | uc /= 0 || ub /= 0 = Multinomial ba ua aa hx fx
  | otherwise          = Multinomial ba' ua' aa' hy fx
  where
    ba = V.mapWithKey belief bc
    ua = ub * uc / (2 * ub - uc)
    aa = ac

    ba' = bb
    ua' = 0
    aa' = ac

    belief x b = (2 * b * ub - V.value bb x * uc) / (2 * ub - uc)
```

Fission is the operation of splitting a multinomial opinion into two multinomial opinions based on some ratio ϕ [27] We refer to this as the *split* operator. Like unfusion, fission has not yet been generalized to hyper opinions.

```
cSplit :: (Ord a, ToMultinomial op) ⇒ Rational → SLExpr h a (op h a)
          → SLExpr h a (Multinomial h a , Multinomial h a)
cSplit phi op = do
  op' ← liftM toMultinomial op
  return $ cSplit' phi op'
```

```
cSplit' :: Rational → Multinomial h a → (Multinomial h a, Multinomial h a)
cSplit' phi (Multinomial b u a (Fuse Cumulative h1 h2) fx) = (op1, op2)
  where
    op1 = Multinomial b1 u1 a h1 fx
    op2 = Multinomial b2 u2 a h2 fx

    b1 = V.map (λx → phi * x / norm phi) b
    u1 = u / norm phi

    b2 = V.map (λx → (1 - phi) * x / norm (1 - phi)) b
    u2 = u / norm (1 - phi)

    norm p = u + p * V.fold (+) 0 b
```

Deduction and Abduction

Deduction and abduction of multinomial opinions allows for one to do conditional reasoning with Subjective Logic [35, 25, 24]. We first introduce the operator for performing deduction, which we call *deduce*, and then discuss the operator *abduce* for performing abduction.

Because of the nature of these operators, the frames of discernment which the opinions are defined over must satisfy two properties: they must be *bounded*, and the must be *enumerable*. These constraints on the type of frames allowed is expressed via the type classes *Bounded* and *Enum*. Boundedness simply means that there exists a least and greatest element, and enumerability means that the values of the type must be enumerable.

We begin by introducing deduction.

```
deduce :: (ToMultinomial op, Ord a, Bounded a, Enum a, Ord b, Bounded b, Enum b)
          ⇒ SLExpr h a (op h a)
          → [(a, Multinomial h b)]
          → SLExpr h a (Multinomial h b)
deduce opx ops = do
  opx' ← liftM toMultinomial opx
  return $ deduce' opx' ops


deduce' :: forall a. forall b. forall h.
           (Ord a, Bounded a, Enum a, Ord b, Bounded b, Enum b)
           ⇒ Multinomial h a
           → [(a, Multinomial h b)]
           → Multinomial h b
deduce' opx@(Multinomial bx ux ax hx _) ops = Multinomial b' u' a' hx f
  where
```

```
expt y = sum ∘ map f $ xs
  where
    f x = V.value ax x * V.value (expectation (findOpinion x)) y


expt' y = sum ∘ map f $ xs
  where
    f x = V.value (expectation opx) x * V.value (expectation (findOpinion x)) y


tExpt y = (1 - V.value ay y) * byxs + (V.value ay y) * (byxr + uyxr)
  where
    (xr', xs') = dims y
    byxr       = V.value (mBelief xr') y
    uyxr       = mUncertainty xr'
    byxs       = V.value (mBelief xs') y


xs = [minBound .. maxBound] :: [a]
ys = [minBound .. maxBound] :: [b]


ay = mBaseRate ∘ snd ∘ head $ ops


uYx x = maybe 1 mUncertainty ∘ lookup x $ ops

findOpinion x = case lookup x ops of
  Nothing → Multinomial (V.fromList []) 1 ay hx f
  Just op → op


f = mFrame ∘ snd ∘ head $ ops


dims :: b → (Multinomial h b, Multinomial h b)
dims y = (xr', xs')
  where
    (_, xr', xs') = foldl1' minPair (dims' y)
    minPair a@(u, _, _) b@(u', _, _) | u < u'    = a
                                     | otherwise = b

    dims' y = do xr' ← xs
                 xs' ← xs
                 let xr'' = findOpinion xr'
                     xs'' = findOpinion xs'
                     byxr = V.value (mBelief xr'') y
                     uyxr = mUncertainty xr''
                     byxs = V.value (mBelief xs'') y
                     val  = 1 - byxr - uyxr + byxs
                 return (val, xr'', xs'')


triangleApexU y
  | expt y ≤ tExpt y = (expt y - byxs) / V.value ay y
  | otherwise        = (byxr + uyxr - expt y) / (1 - V.value ay y)
  where
    byxr = V.value (mBelief ∘ fst ∘ dims $ y) y
    uyxr = mUncertainty ∘ fst ∘ dims $ y
    byxs = V.value (mBelief ∘ snd ∘ dims $ y) y


intApexU = maximum ∘ map triangleApexU $ ys


bComp y = expt y - V.value ay y * intApexU
```

```
    adjustedU y | bComp y < 0 = expt y / V.value ay y
                | otherwise   = intApexU


    apexU = minimum ∘ map adjustedU $ ys


    b' = V.fromList [ (y, expt' y - (V.value ay y) * u') | y ← ys ]
    u' = (apexU -) ∘ sum ∘ map (λx → (apexU - uYx x) * V.value bx x) $ xs
    a' = ay
```

Subjective Logic abduction is a two step procedure. Given an opinion over a frame X and a list of conditional opinions over X given Y, we first must invert the conditionals into a list of conditional opinions over Y given X, and then perform Subjective Logic deduction with the new list and the opinion over X.

```
abduce :: (ToMultinomial op, Ord a, Bounded a, Enum a, Ord b, Bounded b, Enum b)
          ⇒ SLExpr h a (op h a)
          → [(b, Multinomial h a)]
          → BaseRateVector b
          → SLExpr h a (Multinomial h b)
abduce opx ops ay = do
  opx' ← liftM toMultinomial opx
  return $ abduce' opx' ops ay


abduce' :: forall a. forall b. forall h.
           (Ord a, Bounded a, Enum a, Ord b, Bounded b, Enum b)
           ⇒ Multinomial h a
           → [(b, Multinomial h a)]
           → BaseRateVector b
           → Multinomial h b
abduce' opx@(Multinomial bx ux ax hx fx) ops ay = deduce' opx ops'
  where
    ops' = map multinomial xs

    multinomial x = (x, Multinomial b' u' a' hx (F.fromList ys))
      where
        b'  = V.fromList bs
        u'  = uT x
        a'  = ay
        bs  = map (λy → (y, f y)) ys
        f y = expt y x - V.value ay y * uT x


    expt y x = numer / denom
      where
        numer = V.value ay y * V.value (expectation (findOpinion y)) x
        denom = sum ∘ map f $ ys
        f y   = V.value ay y  * V.value (expectation (findOpinion y)) x


    uT x = minimum ∘ map f $ ys
      where
        f y = expt y x / V.value ay y


    ax = mBaseRate ∘ snd ∘ head $ ops
```

```
    xs = [minBound .. maxBound] :: [a]
    ys = [minBound .. maxBound] :: [b]


    findOpinion y = case lookup y ops of
      Nothing → Multinomial (V.fromList []) 1 ax hx (F.fromList xs)
      Just op → op
```

Belief Constraining

The final operator we discuss is the *belief constraint* operator [34]. This operator takes as input two objects that are convertible to hyper opinions and returns a hyper opinion as output. This function is equivalent in meaning to Dempster's rule of combination from Dempster-Shafer Theory [34].

```
constraint :: (ToHyper op1, ToHyper op2, Ord b)
              ⇒ SLExpr h a (op1 h b)
              → SLExpr h a (op2 h b)
              → SLExpr h a (Hyper h b)
constraint op1 op2 = do
  op1' ← liftM toHyper op1
  op2' ← liftM toHyper op2
  return $ constraint' op1' op2'


constraint' :: (Ord a) ⇒ Hyper h a → Hyper h a → Hyper h a
constraint' h1@(Hyper bA uA aA hx fx) h2@(Hyper bB uB aB hy _) =
  Hyper bAB uAB aAB (Constraint hx hy) fx
  where
    bAB = V.fromList ∘ map (λk → (k, harmony k / (1 - conflict))) $ keys

    uAB = (uA * uB) / (1 - conflict)

    aAB = V.fromList $ map (λk → (k, f k)) keys'
      where
        f x = (axA * (1 - uA) + axB * (1 - uB)) / (2 - uA - uB)
          where
            axA = V.value aA x
            axB = V.value aB x

    harmony x = bxA * uB + bxB * uA + rest
      where
        bxA     = V.value bA x
        bxB     = V.value bB x
        rest    = sum ∘ map combine $ matches
        matches = [(y, z) | y ← keys, z ← keys, F.intersection y z == x]

    conflict = sum ∘ map combine $ matches
      where
        matches = [(y, z) | y ← keys, z ← keys, F.intersection y z == F.empty]

    combine (y, z) = V.value bA y * V.value bB z

    keys = F.toList $ F.reducedPowerSet fx
    keys' = nub (V.focals aA ++ V.focals aB)
```

Name	SL Notation	SLHS Notation
Multiplication	$\omega_{X\cup Y} = \omega_X + \omega_Y$	opx ‘$times$‘ opy
Deduction	$\omega_{Y\|\|X} = \omega_X \circledcirc \omega_{Y\|X}$	$deduce\ opx\ ops$
Abduction	$\omega_{Y\overline{\|\|}X} = \omega_X \overline{\circledcirc} \omega_{X\|Y}$	$abduce\ opx\ opys\ a$
Cumulative Fusion	$\omega_X^{A\diamondsuit B} = \omega_X^A \oplus \omega_X^B$	opx ‘$cFuse$‘ opy
Cumulative Unfusion	$\omega_X^{A\overline{\diamondsuit}B} = \omega_X^A \ominus \omega_X^B$	opx ‘$cUnfuse$‘ opy
Averaging Fusion	$\omega_X^{A\underline{\diamondsuit}B} = \omega_X^A \underline{\oplus} \omega_X^B$	opx ‘$aFuse$‘ opy
Averaging Unfusion	$\omega_X^{A\underline{\overline{\diamondsuit}}B} = \omega_X^A \underline{\ominus} \omega_X^B$	opx ‘$aUnfuse$‘ opy
Fission	$\omega_{X\cup Y} = \omega_X + \omega_Y$	$split\ phi\ opx$
Belief Constraining	$\omega_X^{A\&B} = \omega_X^A \odot \omega_X^B$	opx ‘$constraint$‘ opy

Table 4.2: Summary of multinomial and hyper operators

The operators for multinomial and hyper opinions are summarized in table 4.2.

4.4 Extensions to Subjective Logic

In this section we describe new Subjective Logic operators that do not yet appear in the published literature. While Subjective Logic contains a wealth of operators for reasoning with uncertainty [25, 24], modeling transitive trust networks [23], and analyzing hypotheses [65], the set of all theoretically possible operators is incomplete. If we assume that binomial opinions alone are represented as four 32-bit numbers, then the set of all possible unique operators for binomials would be of cardinality $2^{32} \times 2^{32} = 2^{64} = 18446744073709551616$. Whether any or all of these additional operators are meaningful is up to interpretation, of course.

4.4.1 Hypernomial to Multinomial Coarsening

The first extension to the set of Subjective Logic operators we present is a generalized form of coarsening discussed in section 4.2.5. Currently coarsening is defined to be an operation from multinomials to binomials where a subset of the frame of discernment is chosen to be a new element in a binary frame, and the remaining elements of the frame are taken to be the second element, or the *not* of the first element. We generalize this operation to allow for arbitrary hyper opinions to be coarsened into multinomial opinions defined over frames of cardinality $N \geq 2$.

```
hyperCoarsen :: (Ord a, ToHyper op)
                ⇒ op h a → [F.Frame a] → Multinomial h (F.Frame a)
hyperCoarsen op thetas = Multinomial b' u' a' h f'
  where
    (Hyper b u a h f) = toHyper op

    b' = V.fromList [ (t, bel t) | t ← thetas ]
    u' = 1 - V.fold (+) 0 b'
    a' = V.fromList [ (t, br t / norm) | t ← thetas ]
    f' = F.fromList thetas

    norm = sum [ br t | t ← thetas ]

    bel = sum ∘ map snd ∘ overlaps b
    br  = F.fold (+) 0 ∘ F.map (V.value a)

    overlaps v t = V.elemsWhere (λu → u `F.isSubsetOf` t) v
```

Given a hyper opinion and a list of frames of discernment, we construct a new multinomial opinion over a new frame made up of frames as elements. Focal elements that are contained entirely within one of the listed frames contribute their belief mass to the new multinomial opinion, and the remaining mass is lumped into the uncertainty. The new base rates are simply the sums of the base rates multiplied by the normalizing constant

$$\frac{1}{\sum_{t \in Thetas} \sum_{x \in t} a(x)}$$

where $a(x)$ is the base rate of x from the input hyper opinion.

We do not claim that this is the only method that one could use to coarsen a hyper opinion to a multinomial opinion. We present this as simply one method that one could employ.

4.4.2 Uncoarsening from Binomial to Multinomial

In the case of when a binomial opinion is defined over a binary partitioning of a frame, we can uncoarsen it into a multinomial opinion with the following procedure:

```
uncoarsen :: Ord a ⇒ Binomial h (F.Frame a) → Multinomial h a
uncoarsen (Binomial b d u a h xs ys) = Multinomial b' u a' h f
  where
    f = xs `F.union` ys
    b' = V.fromList $
           [ (x, r) | x ← F.toList xs, let r = b / toRational (F.size xs) ]
           ++
           [ (y, r) | y ← F.toList ys, let r = d / toRational (F.size ys) ]
    a' = V.fromList $
           [ (x, r) | x ← F.toList xs, let r = a / toRational (F.size xs) ]
           ++
           [ (y, r) | y ← F.toList ys, let r = (1 - a) / toRational (F.size ys) ]
```

4.5 Limitations

While SLHS is a robust implementation of the opinions and operators of Subjective Logic, our decision to represent all numbers as arbitrary-precision rational numbers imposes a fundamental restriction on the kinds of data that the library can handle. Any computation that involves the assignment of irrational numbers as belief masses cannot be represented directly in our system. However, it is possible to modify SLHS to be able to handle such values: one simply needs to either change the belief vectors to use values of type *Double* instead of *Rational*, or better yet, represent the numeric type as an additional type parameter to the belief vector. The latter would allow the user to use any numerical type of his or her

choosing.

4.6 Summary

In this chapter we introduced SLHS: Subjective Logic in Haskell, a library for representing and evaluating Subjective Logic expressions. We discussed the core components of the library including the monads that represent the expressions, the battery of Subjective Logic opinions and operators, and we concluded with a new operator that is unique to the library.

We have done our best to ensure that the operators implemented in SLHS mirror the definitions found in the literature; however any errors that may arise are the sole responsibility of the author. As is true for many complex software components, it is expected that errors and deficiencies will be found by the users of SLHS. As the famous computer scientist C.A.R. Hoare said during his 1980 Turing Award lecture [16]:

> 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 present a termination analysis of the library, analyze the complexity of a representative subset of the operators, discuss how we leveraged the strong type system to catch errors at compile time, discuss the role that monads have played in the design of the library, demonstrate the expressive power of the library through example programs, and discuss how SLHS fits within the larger UDMDSS system.

Chapter 5

Results and Analysis

In this chapter we analyze SLHS by proving that the *run* function terminates for all valid input Subjective Logic expressions, analyze the complexity of a representative subset of the Subjective Logic operators, discuss how Haskell's strong type system and its support for monads has affected the design of SLHS, and finally we demonstrate the power of SLHS by showcasing some example computations and discuss how the library fits into the *Unified Data Management and Decision Support System (UDMDSS)* [38, 37].

5.1 Proof of Termination

In this section we perform a termination analysis of the *run* function. The run function takes in a Subjective Logic expression and an initial state, and returns either the computed value or an error message. We prove that run terminates for valid Subjective Logic expressions of arbitrary length.

Our strategy for proving termination is as follows: we utilize a function $|\cdot|$ that maps

each Subjective Logic expression e to a natural number. We let $|e|$ denote the number of sub-expressions contained in e, including e itself. As *run* recursively computes the values of the sub-expressions, we will show that the value of $|e|$ decreases at each step, concluding when $|e|$ is 1, when *run* simply returns the final value. Therefore we can conclude that *run* terminates because the set of naturals, along with the $<$ relation, is *well-founded*. That is there cannot exist infinitely descending chains [53].

Lemma 5.1.1. *The* return *function introduces a new sub-expression.*

Proof. The function *return* in Haskell has the following signature:

```
return :: Monad m ⇒ a → m a
```

That is, for any monad m, for every object x of type a, *return* x constructs an object of type $m\ a$. Since *SLExpr* is a monad, *return* constructs a new subjective logic expression containing a single value. We will use this result to assist us in showing that the *run* function's measure decreases at every step. □

Theorem 5.1.2. *For every subjective logic expression $e = e_1 \cdot e_2 \cdot \cdots \cdot e_k$, where $\cdot$ can be any binary subjective logic operator, the computation* run e *terminates.*

Proof. By induction on the length of e. If we can show that $|run\, e_1 \cdot \cdots \cdot e_k| < |e_1 \cdot \cdots \cdot e_k \cdot e_k|$ for all $k \geq 0$, then *run* terminates.

Base Case $e = e_1$: In this case, $|e| = 1$, and since *run* simply applies the initial state to the function contained within e without adding any new objects of type *SLExpr*, in other words $|run\, e| = 0$, *run* e terminates.

Inductive Hypothesis: Assume *run* terminates for the expression $e = e_1 \cdot e_2 \cdot \cdots \cdot e_k$. Given the expression $e' = e \cdot e_{k+1}$, we must prove that $|run\, e'| < |e'|$.

Inductive Proof: Since we are adding exactly one new sub-expression to e to form e', $|e'| = |e| + 1$. Now, all binary operators of SLHS essentially have the same form:

```
op e1 e2 = e1 >>= λe → e2 >>= λe' → return (combine e e')
```

That is, we unpack each expression and then combine them together in some meaningful way to produce a new value of type *SLExpr*. Let us analyze the first monadic bind operator.

```
e1 >>= λe → e2 >>= λe' → return (combine e e')
```

first calls *run* on $e1$, then passes the result of that computation to the lambda function

```
λe → e2 >>= λe' → return (combine e e')
```

and calls *run* on the result. Inside the nested lambda expression, the second monadic bind operator calls *run* on $e2$, passing the result into the lambda expression

```
λe' → return (combine e e')
```

and then invoking *run* on *that* result. The innermost invocation of *run* makes a call to *return*, thus inserting a new object of type *SLExpr*. Combined together with the two invocations of *run* on the input expressions, we have

$$\begin{aligned} |run\, e \cdot e_{k+1}| &= |run\, e| + |run\, e_{k+1}| + |return\, x| \\ &= |run\, e| + 0 + 1 \\ &< |e| + 1 \\ &= |e \cdot e_{k+1}| \end{aligned}$$

□

5.2 Analysis of Complexity

In this section we analyze the time complexity of a representative subset of the SLHS operators. We analyze the complexity of belief constraining to demonstrate how computationally expensive it is to work with hyper opinions, which are defined over the reduced power set of the frame of discernment. Next we analyze the complexity of belief fission, an operator defined over multinomial opinions. Lastly, we analyze the complexity of multinomial multiplication.

We do not claim that the implementations of the operators are optimal. In fact, our implementations are very sensitive to our choice of data structure for representing belief assignments: the red-black tree. Iterating through the entire belief mass assignment takes $O(n)$ time, but finding an individual element takes $O(\log n)$ time. Alternative representations may possibly be more efficient, and we leave that for future work.

It is also worthy to note that every operator that is implemented solely for binomial opinions has complexity $O(1)$ with respect to the size of the frame of discernment. Recall that binomial opinions are either defined as opinions over a frame of cardinality 2, or are defined over binary partitions of frames. In either case, each equation involves calculating new values for b, d, u, and a without any regard to the actual size of the frame. Therefore each calculation on binomial opinions will be carried out in a constant amount of time.

Theorem 5.2.1. *Belief constraining has time complexity $O((2^n-2)^3 \log(2^n-2))$, where n is the cardinality of the frame of discernment.*

Proof. Since belief constraining is defined over hyper opinions, let $m = 2^n - 2$ be the cardinality of the reduced power set of the frame. Computing the *conflict* requires finding

all elements of the power set that share overlap and adding together their assigned belief masses. This operation takes $O(m^2)$ time for the iteration, and $O(\log m)$ for looking up the belief masses. Therefore conflict takes $O(m^2 \log m)$ time.

Computing the new belief mass requires computing the *Harmony* for every element of the power set. Computing the harmony takes $O(m^2)$ time per element, resulting in a time complexity of $O(m^3 \log m)$ for computing the new belief mass.

Computing the uncertainty requires computing the conflict, which we have already computed as a part of the new belief mass.

Atomicity requires iterating over the entire reduced power set, and thus requires $O(m \log m)$ time.

Therefore the total time complexity for belief constraining is $O(m^3 \log m + m^2 \log m + m \log m) = O(m^3 \log m) = O((2^n - 2)^3 \log(2^n - 2))$. □

Theorem 5.2.2. *Multinomial fission has time complexity $O(n)$, where n is the cardinality of the frame of discernment.*

Proof. Computing the normalizing constant takes $O(n)$ time. Since computing the new beliefs and uncertainties requires iterating over each element of the frame of discernment, each takes $O(n)$ time. Therefore, the time complexity for fission is $O(n)$. □

Theorem 5.2.3. *Multinomial multiplication has time complexity $O(m \log m \times n \log n)$.*

Proof. The *expect x y* function takes $O(\log m + \log n)$ time, since it needs to perform look-ups on each frame. Computing the uncertainty takes $O(m \log m \times n \log n)$ time, computing the new atomicity takes $O(m \log m \times n \log n)$ time, and computing the new belief also takes $O(m \log m \times n \log n)$ time. Therefore the entire time complexity is $O(m \log m \times n \log n)$. □

5.3 The Use of Haskell's Type System

In this section we discuss how SLHS leverages Haskell's type system to catch many errors at compile time, instead of at run time. With SLHS we have taken the motto of *catch what we can at compile time, report what we must at run time*. There are certain properties of well-formed Subjective Logic expressions that can only be caught at run time, such as

- the inputs to the binomial addition operator are subsets of the same frame of discernment.
- the inputs to the transitive discounting operator have different belief owners.
- the subset relation required for binomial subtraction is satisfied.

For other issues however, such as restricting addition to work on binomial opinions only, we can leverage Haskell's strong typing to stop those invalid expressions from even compiling.

Consider the type signature for the binomial addition operator:

```
(+!) :: (ToBinomial op1, ToBinomial op2, Eq h, Eq b, Ord b)
       ⇒ SLExpr h a (op1 h b)
       → SLExpr h a (op2 h b)
       → SLExpr h a (Binomial h b)
```

What this tells us is that addition takes in two parameters, *op1* and *op2*, each wrapped in the *SLExpr* monad. These two opinion types must be convertible to binomials, as they must belong to the type class *ToBinomial*. This signature also tells us that the elements of the frame must be of the same type. Therefore, if any one of the opinions is constructed over the cartesian product of two frames, then both opinions must be constructed over the

cartesian product of two frames. Checking whether the two frames are in fact the same must be deferred until run time, however.

5.4 The Use of Monads

In this section we describe the role that monads have played in the design of SLHS. As mentioned previously, the *SLExpr* type, which is the type used to represent Subjective Logic expressions within SLHS, is a function from a world state, *SLState*, to some output value. *SLExpr* forms a monad, and thus we are able to use all of Haskell's built-in support for monads when writing computations involving objects of type *SLExpr*. In particular, *SLExpr* is a special kind of *state monad*, where the state carried through the computation is an *SLState* object.

Because they are monads, objects of type *SLExpr* can be glued together using the various operators and functions in the Haskell standard library. One function that we use quite frequently in the implementation of SLHS is the *liftM* function, which takes an ordinary function from some type a to type b, and converts it into a function from type $M\,a$ to $M\,b$, where M is any monad. This allows us to use functions such as *toBinomial* directly on objects of type *SLExpr* without having to unwrap them first.

Another benefit of *SLExpr* being a monad is that we are able to use Haskell's *do-notation* in order to simplify our code. Do-notation allows us to write code of the form

```
z = mx >>= λx → my >>= λy → return (x + y)
```

where each and every invocation of the bind operator must be explicitly written, as

```
z = do x ← mx
       y ← my
       return (x + y)
```

This syntactic sugar not only allows the implementation of SLHS to be written more concisely in many cases, but it also extends to users of SLHS as well. Complicated Subjective Logic expressions can be broken down into smaller pieces, and then glued together in a style that looks very imperative:

```
expr = do e1 ← getMultinomial ‘‘Alice’’ 0
          e2 ← getMultinomial ‘‘Bob’’ 0
          e3 ← e1 ‘times‘ e2
          e4 ← e3 ‘cFuse‘ (getHyper ‘‘Clark’’ 0)
          return e4
```

which may help programmers who are more accustomed to writing programs in more mainstream structural languages such as *Python* [78] or *Ruby* [49]. In the next section we demonstrate how problems involving Subjective Logic can be modeled and executed using SLHS.

5.5 Example Computations

In this section we demonstrate the use of SLHS on a selection of examples provided in the Subjective Logic literature.

5.5.1 Going to the Movies

The first situation is taken from the draft Subjective Logic book[1] and it involves three friends trying to figure out which movie they want to see. We start with defining the belief holders as strings:

```
holders = ["Alice", "Bob", "Clark"]
```

[1] http://folk.uio.no/josang/papers/subjective_logic.pdf

and then define the frame of discernment. Here we use a special type to denote the three possible movie choices, where *BD* stands for *Black Dust*, *GM* stands for *Grey Matter*, and *WP* stands for *White Powder*:

```
data Movie = BD | GM | WP deriving (Eq, Ord, Show, Bounded, Enum)
frame = [BD, GM, WP]
```

Now that we have the belief holders and the frame of discernment, we can define the belief vectors. Since Subjective Logic expressions can involve many frames, we define our data set to be a list of tuples: the first argument is the frame which we will associate the data, and the second argument is another list of tuples. This second list of tuples is comprised of the belief owner, and a list of tuples containing subsets of the frame and associated belief mass. The base rate data is defined similarly: for each frame we associate a list of tuples: the first element being the belief holder, and the second element being a list of elements of the frame paired up with a-priori mass.

```
vectors =
  [ (frame,
     [ ("Alice", [([BD], 99%100), ([GM], 1%100), ([WP], 0), ([GM, WP], 0)])
     , ("Bob",   [([BD], 0), ([GM], 1%100), ([WP], 99%100), ([GM, WP], 0)])
     , ("Clark", [([BD], 0), ([GM], 0), ([WP], 0), ([GM, WP], 1)])
     ])
  ]

baseRates =
  [ (frame,
     [ ("Alice", [(BD, 1%3), (GM, 1%3), (WP, 1%3)])
     , ("Bob",   [(BD, 1%3), (GM, 1%3), (WP, 1%3)])
     , ("Clark", [(BD, 1%3), (GM, 1%3), (WP, 1%3)])
     ])
  ]
```

In the above code, the % operator constructs a rational number from the numerator and denominator. Therefore, 1%3 results in the value $\frac{1}{3}$.

Once our data model has been defined, we can now perform calculations. We start by constructing an initial state of the world, and then an expression. The expression in this case is a simple application of the belief constraint operator. We fetch the hyper opinions

owned by the three belief holders for frame 0 (the first and only frame in our list of frames) and constrain the resulting hyper opinions.

```
initial = makeState holders [frame] vectors baseRates

expr = getHyper "Alice" 0 `constraint`
       getHyper "Bob"   0 `constraint`
       getHyper "Clark" 0
```

Lastly, we can run the expression over the initial state of the world. The resulting value is of type *SLVal (Hyper String Movie)*, meaning it is either a hyper opinion with belief owners modeled as strings and frame elements being movies, or a run-time error diagnostic.

```
result = initial >>= run' expr
```

When we run the command *print result* we obtain the following:

```
Hyper:
  Holder: Constraint (Constraint (Holder "Alice") (Holder "Bob")) (Holder "Clark")
  Frame: {BD,GM,WP}
  Belief: <({BD},0 % 1),({BD,GM},0 % 1),({BD,WP},0 % 1),({GM},1 % 1),
           ({GM,WP},0 % 1),({WP},0 % 1)>
  Uncertainty: 0 % 1
  Base Rate: <(BD,1 % 3),(GM,1 % 3),(WP,1 % 3)>
```

The resulting hyper opinion is held by the imaginary owner made up by applying the *Constraint* holder data constructor twice, defined over the frame BD, GM, WP, and has 100% belief allocated to the movie GM, and each movie has a base rate of $\frac{1}{3}$.

Note that the result of the calculation, that the three friends should see the movie *Grey Matter*, does not seem to be the intuitively correct answer. This can be attributed to Clark's opinion, while it seemingly neglects to take into account that neither Alice nor Bob seem to really want to see that movie. One method of fixing this issue could be to introduce a *weighted constraint* operator that places more emphasis on different opinions. Since Alice and Bob seem much more certain regarding which movie they want to see, perhaps more

weight should be given to their opinions, and less to Clark's.

5.5.2 Observing Genetic Mutations

This example also comes from the draft Subjective Logic book. Assume through a process of genetic engineering that we can produce two kinds of chicken eggs: male, or female. Each egg, regardless of gender, can also have genetic mutation S or T. The first sensor determines whether an egg is male or female, and the second sensor measures whether the egg has genetic mutation S or T. This scenario can be modeled by using two frames of discernment

```
type Gender = Int
type Mutation = Int

m = 0
f = 1
s = 2
t = 3

gender = [m, f]
mutation = [s, t]
```

and two belief holders

```
data Sensor = A | B deriving (Eq, Ord, Show)
sensors = [A, B]
```

Due to a limitation of SLHS, we must use the same underlying type for all frames, hence we use integers to represent both genders and mutations.

Since the two sensors measure orthogonal aspects of the eggs, we can combine their observations through multinomial multiplication to produce an opinion over the cartesian product of the two frames. Assume we have two observations:

```
obs1 = [(gender,   [(A, [([m], 8%10), ([f], 1%10)])])]
obs2 = [(mutation, [(B, [([s], 7%10), ([t], 1%10)])])]
observations = obs1 ++ obs2
```

with the following base rates:

```
baseRates = [ (gender,   [(A, [(m, 1%2), (f, 1%2)])])
            , (mutation, [(B, [(s, 1%5), (t, 4%5)])])
            ]
```

We can then compute the opinion over the cartesian product by evaluating the following expression:

```
expression = getMultinomial A 0 `times` getMultinomial B 1
state      = makeState sensors [gender, mutation] observations baseRates
opinion    = state >>= run' expression
```

We can see the resulting multinomial opinion by running the command *print opinion*, which displays the following:

```
Multinomial:
  Holder: Product (Holder A) (Holder B)
  Frame: {(0,2),(0,3),(1,2),(1,3)}
  Belief: <((0,2),37823 % 61000),((0,3),11297 % 61000),
           ((1,2),249 % 2440), ((1,3),39 % 12200)>
  Uncertainty: 273 % 3050
  Base Rate: <((0,2),1 % 10),((0,3),2 % 5),((1,2),1 % 10),((1,3),2 % 5)>
```

The fractions are a little messy, but with a trusty pocket calculator we can verify that the beliefs plus the uncertainty sums to 1. This result is in fact displayed with slightly more accuracy than the result in Josang's draft book.

5.6 Utilization Within UDMDSS

As mentioned in Section 2.1, we have participated in a team effort to design the Unified Data Management and Decision Support System (UDMDSS) as a part of our ongoing research into the development of decision support systems for the management and analysis of population research surveys [37, 39, 41]. SLHS was designed to aide in the development of automated reasoning systems that utilized Subjective Logic, and though it has not been integrated yet, we expect that SLHS will find a place in the heart of the UDMDSS system.

Further development on UDMDSS will see SLHS put to the test of analyzing real health care data using the tools of Subjective Logic.

5.7 Summary

In this chapter we presented a termination analysis for the *run* function of SLHS, proving that it terminates for all valid expressions. We then provided a complexity analysis for a selection of Subjective Logic operators. We also discussed how Haskell's type system is leveraged in SLHS to catch problems with Subjective Logic expressions at compile time, and how the use of monads facilitated a sound design. We also provided some example calculations with SLHS, and we discussed its position within the larger UDMDSS system. In the next chapter we conclude this thesis and discuss areas in which we feel SLHS can be improved.

Chapter 6

Conclusion

In this chapter we present our concluding remarks, as well as discuss possible avenues for future improvements to the SLHS library.

6.1 Conclusion

For this thesis we constructed a Subjective Logic library, SLHS, that uses monadic higher order functions to represent subjective expressions. Subjective Logic is a relatively new extension to probabilistic logic [23] that directly handles uncertainty in each and every operator. The fundamental unit for computation is the *subjective opinion*, which is a combination of belief mass assigned to a frame of discernment, plus a scalar value representing uncertainty.

Within this thesis, we have shown the construction of SLHS in Chapter 4, discussed its current limitations in Section 4.5, shown its termination in Section 5.1, and analyzed a representative subset of the operators in Section 5.2. Furthermore we have discussed

the role that *Haskell*, our language of implementation, has had on SLHS in Section 5.3, and how the use of *monads* simplified our code (Section 5.4). In totality, we have shown that it is possible to construct a Subjective Logic library that is type-safe, efficient, and compositional.

6.2 Future Work

In this section we discuss areas for future experimentation or improvement to SLHS.

6.2.1 Modifications to the Vector Representation

In our implementation of SLHS we chose to represent belief vectors as red-black trees in order to avoid storing the entire frame of discernment in memory: elements of the frame that have zero belief mass assigned to them are simply not stored in the tree. While this representation has some nice theoretical properties, such as the ability to map functions across the vector in $O(n)$ time, and the ability to determine whether an element is or is not a focal element in $O(\log n)$ time, we believe that improvements in the actual run-time of the library may be achieved by switching to using a contiguous array.

6.2.2 Implementing Memoization

We have shown how some of the operators of Subjective Logic scale with respect to the cardinalities of the frames of discernment involved. As we deal with larger and larger frames, computing the results of the individual operators will become more and more time consuming. If a single sub-expression appears many times throughout a more complex subjective

logic expression, it would be beneficial to re-use a previously computed value. Instead, currently we would waste valuable time recomputing the output for the same expression over and over.

One technique to avoid this costly re-computation is *memoization* [51]. At every operator invocation, we compute the value and store it in a table. If at any time we require the same expression to be computed, we first look answer up in the table. In a sense we would use additional memory in order to save time.

6.2.3 Exploiting Parallelism

Many operators of Subjective Logic appear to be easily made to run in parallel, as the new opinions are computed by combining together the belief masses of individual elements of the reduced power set without depending on any other elements. Therefore, attempting to introduce parallelism to the implementations of the operators should be as easy as modifying the underlying *SLExpr* monad to utilize one of the many Haskell libraries for parallel and concurrent computing [48]. Then the operators can be rewritten to compute their results in parallel without any modification to the external interface to the library. While we did not address the issue of parallelism in this thesis, it appears, at least to the authors, to be a useful area of future research.

Chapter 7

Bibliography

[1] Ekaterina Auer, Wolfram Luther, Gabor Rebner, and Philipp Limbourg. A verified matlab toolbox for the dempster-shafer theory.

[2] John Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[3] NitinA Bansod, Vaishali Kulkarni, and SH Patil. Soft computing-a fuzzy logic approach. *Soft Computing*, page 73, 2005.

[4] Otman Basir and Xiaohong Yuan. Engine fault diagnosis based on multi-sensor information fusion using dempster–shafer evidence theory. *Information Fusion*, 8(4):379–386, 2007.

[5] Rudolf Carnap, Rudolf Carnap, and Rudolf Carnap. Logical foundations of probability. 1962.

[6] Irving M Copi, Carl Cohen, and Daniel E Flage. *Essentials of logic*. Pearson/Prentice Hall, 2007.

[7] Michael A Covington. *Natural language processing for Prolog programmers*. Prentice Hall Englewood Cliffs (NJ), 1994.

[8] Bruno De Finetti and Bruno de Finetti. Theory of probability. *Bull. Amer. Math. Soc. 83 (1977), 94-97 DOI: http://dx. doi. org/10.1090/S0002-9904-1977-14188-8 PII*, pages 0002–9904, 1977.

[9] Lora D Delwiche and Susan J Slaughter. *The Little SAS Book: A Primer: a Programming Approach*. SAS Institute, 2012.

[10] Arthur P Dempster. A generalization of bayesian inference. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 205–247, 1968.

[11] Thierry Denoeux. A neural network classifier based on dempster-shafer theory. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 30(2):131–150, 2000.

[12] Maurice J Dupré, Frank J Tipler, et al. New axioms for rigorous bayesian probability. *Bayesian Analysis*, 4(3):599–606, 2009.

[13] John Fox and Robert Andersen. Using the r statistical computing environment to teach social statistics courses. *Department of Sociology, McMaster University*, 2005.

[14] Rahmatullah Hafiz and Richard A Frost. Lazy combinators for executable specifications of general attribute grammars. In *Practical Aspects of Declarative Languages*, pages 167–182. Springer, 2010.

[15] Peter Henderson and James H Morris Jr. A lazy evaluator. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, pages 95–103. ACM, 1976.

[16] Charles Antony Richard Hoare. The 1980 acm turing award lecture. *Communications*, 1981.

[17] Alfred Horn. On sentences which are true of direct unions of algebras. *The Journal of Symbolic Logic*, 16(01):14–21, 1951.

[18] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1. ACM, 2007.

[19] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, et al. Report on the programming language haskell: a non-strict, purely functional language version 1.2. *ACM SigPlan notices*, 27(5):1–164, 1992.

[20] Graham Hutton and Erik Meijer. Monadic parsing in haskell. *Journal of functional programming*, 8(04):437–444, 1998.

[21] Simon L Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[22] A Josang and Robin Hankin. Interpretation and fusion of hyper opinions in subjective logic. In *Information Fusion (FUSION), 2012 15th International Conference on*, pages 1225–1232. IEEE, 2012.

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

[24] Audun Jøsang. Abductive reasoning with uncertainty. In *The Proceedings of the International Conference on Information Processing and Management of Uncertainty (IPMU2008)*, 2008.

[25] Audun Josang. Conditional reasoning with subjective logic. *Journal of Multiple-Valued Logic and Soft Computing*, 15(1):5–38, 2008.

[26] Audun Jøsang. Cumulative and averaging unfusion of beliefs. In *Proceedings of IPMU*, volume 8, page 331, 2009.

[27] Audun Josang. Fission of opinions in subjective logic. In *Information Fusion, 2009. FUSION'09. 12th International Conference on*, pages 1911–1918. IEEE, 2009.

[28] Audun Jøsang, Tanja Ažderska, and Stephen Marsh. Trust transitivity and conditional belief reasoning. In *Trust Management VI*, pages 68–83. Springer, 2012.

[29] Audun Josang and Touhid Bhuiyan. Optimal trust network analysis with subjective logic. In *Emerging Security Information, Systems and Technologies, 2008. SECURWARE'08. Second International Conference on*, pages 179–184. IEEE, 2008.

[30] Audun Jøsang and Viggo A Bondi. Legal reasoning with subjective logic. *Artificial Intelligence and Law*, 8(4):289–315, 2000.

[31] Audun Jøsang, Javier Diaz, and Maria Rifqi. Cumulative and averaging fusion of beliefs. *Information Fusion*, 11(2):192–200, 2010.

[32] Audun Jøsang, Ross Hayward, and Simon Pope. Trust network analysis with subjective logic. In *Proceedings of the 29th Australasian Computer Science Conference-Volume 48*, pages 85–94. Australian Computer Society, Inc., 2006.

[33] Audun Jøsang and David McAnally. Multiplication and comultiplication of beliefs. *International Journal of Approximate Reasoning*, 38(1):19–51, 2005.

[34] Audun Josang and Simon Pope. Dempster's rule as seen by little colored balls. *Computational Intelligence*, 28(4):453–474, 2012.

[35] Audun Jøsang and Francesco Sambo. Inverting conditional opinions in subjective logic.

[36] John R Josephson and Susan G Josephson. *Abductive inference: Computation, philosophy, technology*. Cambridge University Press, 1996.

[37] Robert D Kent, Ziad Kobti, Anne Snowdon, and Akshai Aggarwal. Towards a unified data management and decision support system for health care. In *Intelligent Interactive Multimedia Systems and Services*, pages 205–220. Springer, 2010.

[38] Robert D Kent, Jason McCarrell, Gilles Paquette, Bryan St Amour, Ziad Kobti, and Anne W Snowdon. Application of subjective logic to health research surveys. In *Advances in Intelligent Decision Technologies*, pages 383–392. Springer, 2010.

[39] Robert D Kent, Paul D Preney, Anne W Snowdon, Farhan Sajjad, Gokul Bhandari, Jason McCarrell, Tom McDonald, and Ziad Kobti. Design and implementation of a primary health care services navigational system architecture. In *Intelligent Decision Technologies*, pages 743–752. Springer, 2011.

[40] Oleg Kiselyov, Simon Peyton Jones, and Chung-chieh Shan. Fun with type functions. In *Reflections on the Work of CAR Hoare*, pages 301–331. Springer, 2010.

[41] Ziad Kobti, Anne W Snowdon, Robert D Kent, Gokul Bhandari, Shamual F Rahaman, Paul D Preney, Carol A Kolga, Barbara Tiessen, and Lichun Zhu. Towards a "just-in-time" distributed decision support system in health care research. In *Supporting Real Time Decision-Making*, pages 253–285. Springer, 2011.

[42] Jonathan J Koehler. The base rate fallacy reconsidered: Descriptive, normative, and methodological challenges. *Behavioral and brain sciences*, 19(01):1–17, 1996.

[43] John Launchbury and Simon L Peyton Jones. Lazy functional state threads. In *ACM SIGPLAN Notices*, volume 29, pages 24–35. ACM, 1994.

[44] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical report, Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[45] Xiaoqi Li, Michael R Lyu, and Jiangchuan Liu. A trust model based routing protocol for secure ad hoc networks. In *Aerospace Conference, 2004. Proceedings. 2004 IEEE*, volume 2, pages 1286–1295. IEEE, 2004.

[46] P Limbourg. *Imprecise probabilities for predicting dependability of mechatronic systems in early design stages*. PhD thesis, University of Dusburg-Essen, 2007.

[47] Yining Liu, Keqiu Li, Yingwei Jin, Yong Zhang, and Wenyu Qu. A novel reputation computation model based on subjective logic for mobile ad hoc networks. *Future Generation Computer Systems*, 27(5):547–554, 2011.

[48] Simon Marlow. Parallel and concurrent programming in haskell. In *Central European Functional Programming School*, pages 339–401. Springer, 2012.

[49] Yukio Matsumoto and K Ishituka. Ruby programming language, 2002.

[50] David McAnally and Audun Jøsang. Addition and subtraction of beliefs. *Proceedings of Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU 2004)*, 2004.

[51] Donald Michie. Memo functions and machine learning. *Nature*, 218(5136):19–22, 1968.

[52] Eugenio Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, 1991.

[53] Yiannis Moschovakis. *Notes on set theory*. Springer, 2006.

[54] Robin R Murphy. Dempster-shafer theory for sensor fusion in autonomous mobile robots. *Robotics and Automation, IEEE Transactions on*, 14(2):197–206, 1998.

[55] Nir Oren, Timothy J Norman, and Alun Preece. Subjective logic and arguing with evidence. *Artificial Intelligence*, 171(10):838–854, 2007.

[56] Bryan O'Sullivan, John Goerzen, and Donald Bruce Stewart. *Real World Haskell: Code You Can Believe In*. " O'Reilly Media, Inc.", 2008.

[57] John Allen Paulos. The mathematics of changing your mind. *New York Times (US)*, 2011.

[58] Judea Pearl. On probability intervals. *International Journal of Approximate Reasoning*, 2(3):211–216, 1988.

[59] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.

[60] Judea Pearl. Reasoning with belief functions: an analysis of compatibility. *International Journal of Approximate Reasoning*, 4(5):363–389, 1990.

[61] Francis Jeffry Pelletier. Representation and inference for natural language: A first course in computational semantics. *Computational Linguistics*, 32(2):283–286, 2006.

[62] Fernando CN Pereira and David HD Warren. Definite clause grammars for language analysis-a survey of the formalism and a comparison with augmented transition networks. *Artificial intelligence*, 13(3):231–278, 1980.

[63] Irina Perfilieva and Jiří Močkoř. *Mathematical principles of fuzzy logic*. Springer, 1999.

[64] Simon L Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 71–84. ACM, 1993.

[65] Simon Pope and Audun Josang. Analysis of competing hypotheses using subjective logic. Technical report, DTIC Document, 2005.

[66] Daniel J Power. *Decision support systems: concepts and resources for managers*. Greenwood Publishing Group, 2002.

[67] Dino Quintero, Thomas Ancel, Graeme Cassie, Rodrigo Ceron, Amr Darwish, Guilherme Felix, Jian Jun He, Bharathraj Keshavamurthy, Sreenivas Makineedi, Girish Nikalje, et al. *Workload Optimized Systems: Tuning Power7 for Analytics*. IBM Redbooks, 2013.

[68] Peter Reutemann, Bernhard Pfahringer, and Eibe Frank. A toolbox for learning from relational data with propositional and multi-instance learners. In *AI 2004: Advances in Artificial Intelligence*, pages 1017–1023. Springer, 2005.

[69] Galina Rogova. Combining the results of several neural network classifiers. *Neural networks*, 7(5):777–781, 1994.

[70] Stuart J. Russell and Peter Norvig. *Artificial intelligence - a modern approach: the intelligent agent book second edition*. Prentice Hall series in artificial intelligence. Prentice Hall, 2003.

[71] Robert Schlaifer and Howard Raiffa. Applied statistical decision theory. 1961.

[72] Glenn Shafer et al. *A mathematical theory of evidence*, volume 1. Princeton university press Princeton, 1976.

[73] Ehud Y Shapiro. The fifth generation project - a trip report. *Communications of the ACM*, 26(9):637–641, 1983.

[74] Ralph H Sprague Jr. A framework for the development of decision support systems. *MIS quarterly*, pages 1–26, 1980.

[75] Leon Sterling. *The art of Prolog: advanced programming techniques*. MIT press, 1994.

[76] Robert Sternberg. *Cognitive psychology*. Cengage Learning, 2008.

[77] R Core Team et al. R: A language and environment for statistical computing. 2012.

[78] Guido Van Rossum and Fred L Drake. *Python language reference manual*. Network Theory, 2003.

[79] Ashlee Vance. Data analysts captivated by r's power. *New York Times*, 6, 2009.

[80] Ian H Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[81] Larry Wos, Ross Overbeck, Ewing Lusk, and Jim Boyle. Automated reasoning: introduction and applications. 1984.

[82] Huadong Wu, Mel Siegel, Rainer Stiefelhagen, and Lei Yang. Sensor fusion using dempster-shafer theory [for context-aware hci]. In *Instrumentation and Measurement Technology Conference, 2002. IMTC/2002. Proceedings of the 19th IEEE*, volume 1, pages 7–12. IEEE, 2002.

[83] Lotfi A Zadeh. Fuzzy sets. *Information and control*, 8(3):338–353, 1965.

[84] Lotfi A Zadeh. *On the validity of Dempster's rule of combination of evidence*. Electronics Research Laboratory, University of California, 1979.

[85] Lotfi A Zadeh. A simple view of the dempster-shafer theory of evidence and its implication for the rule of combination. *AI magazine*, 7(2):85, 1986.

[86] Lotfi Asker Zadeh. The role of fuzzy logic in the management of uncertainty in expert systems. *Fuzzy sets and systems*, 11(1):197–198, 1983.

Vita Auctoris

Bryan St. Amour was born in 1987 and raised in Windsor, Ontario, Canada. He completed his undergraduate degree in Computer Science from the University of Windsor in 2010, and his Master's degree in Computer Science from the same institute in 2014.