Dynamic Application of Problem Solving Strategies: Dependency-Based Flow Control

by

Ian Campbell Jacobi

B.S., Computer Science,
Rensselaer Polytechnic Institute (2008)

S.M., Computer Science and Engineering,
Massachusetts Institute of Technology (2010)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Engineer in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2013

Author .....
Department of Electrical Engineering and Computer Science
August 30, 2013

Certified by .....
Gerald Jay Sussman
Panasonic Professor of Electrical Engineering
Thesis Supervisor

Accepted by .....
Professor Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students

Dynamic Application of Problem Solving Strategies:

Dependency-Based Flow Control

by

Ian Campbell Jacobi

Submitted to the Department of Electrical Engineering and Computer Science
on August 30, 2013, in partial fulfillment of the
requirements for the degree of
Engineer in Computer Science

Abstract

While humans may solve problems by applying any one of a number of different problem solving strategies, computerized problem solving is typically brittle, limited in the number of available strategies and ways of combining them to solve a problem. In this thesis, I present a method to flexibly select and combine problem solving strategies by using a constraint-propagation network, informed by higher-order knowledge about goals and what is known, to selectively control the activity of underlying problem solvers. Knowledge within each problem solver as well as the constraint-propagation network are represented as a network of explicit propositions, each described with respect to five interrelated axes of concrete and abstract knowledge about each proposition. Knowledge within each axis is supported by a set of dependencies that allow for both the adjustment of belief based on modifying supports for solutions and the production of justifications of that belief. I show that this method may be used to solve a variety of real-world problems and provide meaningful justifications for solutions to these problems, including decision-making based on numerical evaluation of risk and the evaluation of whether or not a document may be legally sent to a recipient in accordance with a policy controlling its dissemination.

Thesis Supervisor: Gerald Jay Sussman
Title: Panasonic Professor of Electrical Engineering

Acknowledgments

I would like to acknowledge and thank the following individuals for their assistance throughout the development of the work presented in this thesis:

My advisor and thesis supervisor, Gerry Sussman, for his invaluable assistance and advice offered in the process of developing the ideas presented in this thesis.

Alexey Radul for developing the concept of propagator networks and assisting in troubleshooting and revising the propagator network implementation upon which the work in this thesis was based.

K. Krasnow Waterman for proofreading this paper and offering some interesting legal insights on several of the concepts presented in the thesis.

Lalana Kagal for her assistance and advice in developing reasoning systems, the lessons of which were integrated in part in the system proposed in this paper.

Sharon Paradesi for providing additional proofreading assistance and offering suggestions of several pieces of related work.

My lab-mates and other members of the Decentralized Information Group not named above for providing their assistance, criticisms, feedback, and proofreading skills on my presentations of this work as it has evolved over the past five years.

Finally, I would like to acknowledge that funding for this work was provided in part by Department of Homeland Security Grant award number N66001-12-C-0082 "Accountable Information Usage in Distributed Information Sharing Environments"; National Science Foundation Computer Systems Research/Software and Hardware Foundations Grant "Propagator-based Computing", contract number CNS-1116294; National Science Foundation Cybertrust Grant "Theory and Practice of Accountable Systems", contract number CNS-0831442; and a grant from Google, Inc.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

What differentiates human intelligence from that of other animals? Although scientists have proposed a number of different theories and differentiators in the past, including tool use [20], self-recognition [13], and social intelligence [10], each has been dismissed by later evidence of animals which exhibit behavior like that of humans. One of the leading theories at present is the human ability to solve problems in a abstract, "symbolic" manner [29]. Indeed, humans appear to use symbolic problem solving skills in any number of different ways, including the use of logic in statistical analysis based on the scientific method, as well as the application of probabilistic approaches to learning which have formed the foundation of many algorithms considered to fall under the guise of "artificial intelligence" [25].

While the increased understanding of each of these problem solving mechanisms has led to a greater understanding and appreciation of the role of symbolic manipulation in human intelligence, less has been done to properly understand and harness the control of mechanisms that allow humans to make use of any of these approaches in a flexible manner, outside of work relating to Minsky's theory of the Society of Mind [17, 18]. If we are to actually claim we understand the basis of human intelligence, we must also be able to discover the source of flexibility in human thought which allows us to effectively and efficiently solve problems using any number of problem solving methods without any preconceived algorithms of how such control may be implemented.

1.1 Defining Flexibility

If we are to discuss flexibility of human problem solving skills, we must better define what exactly is meant by this term. Human problem solving appears to feature a number of aspects which, while perhaps not entirely unique to humans, are features which we see as integral to the ability to solve problems. Some of these attributes include reactivity, goal-oriented behavior, synthesis of disparate attributes (such as color and location), apparent optimizations for certain kinds of problems, resilience to contradictory inputs, and the ability to evaluate and select from different strategies using what appears to be a single mechanism of thought.

While reactive behavior is common to practically all living things (e.g. subconscious reflexes), humans appear uniquely able to react not only to physical constraints which manifest while solving a problem, but also to theoretical, symbolic constraints, such as those presented in *hypothetical* situations and those which occur in abstract conceptual reasoning (e.g. making reactive decisions about perceived social and financial situations). Indeed, the ability to combine, apply, and otherwise utilize abstract, symbolic values and operations is viewed as such a fundamental component of human thought that its development in the adolescent human psyche is viewed as a transition between the final two developmental stages in the theory of cognitive development proposed by Jean Piaget (concrete operational and formal operational) [4].

Such reactivity is particularly evident in humanity's ability to engage in long-term planning. Indeed, humans appear to be the only species capable of planning beyond their lifetime *and* those of their heirs. When knowledge about specifics will necessarily change over time, there is a need to adjust any strategies seeking extremely long-term results to account for large-scale changes in the environment. Human attempts to address or account for issues such as global warming, technological development, and large-scale societal change are but some examples of such behavior. Although humans often encounter practical difficulties in envisioning and planning on extremely long time-frames, such limitations appear to be cultural in nature (e.g. a drive for short-term profits), rather than inherent to human intelligence. Indeed, some social groups

have proven quite willing to solve problems on time-scales beyond many hundreds of years [1].

Human problem-solving is not only reactive, it is also goal-oriented. On the most basic level, the desire to solve a problem implies that there is a fundamental goal: the solution of the problem. But goal-direction may also be observed in the ability of humans to plan how such problems will be solved, typically by establishing sub-goals which may be achieved on the way to solving the larger problem. Such management of sub-goals appears to act at even a sub-conscious, fundamental level in human thought; a neuroimaging study by Braver and Bongiolatti [2] has implicated the frontopolar region of the prefrontal cortex of the human brain in managing subgoals associated with primary goals in working memory tasks.

The ability to synthesize disparate data appears to be another unique feature of symbolic thought. Research by Herver-Vazquez, et al has shown that humans are capable of synthesizing information about geometric features and landmarks to identify locations [9], an ability not observed in rats. Thus, if a problem solving mechanism is to approach human capabilities, it must be able to synthesize arbitrary features to derive solutions, lest such a mechanism be unable to solve the deceptively simple task of locating an object using both geometric and landmark-based cues, such as locating the Washington Monument in Washington, D.C. based on the fact that it is "at the opposite end of the Reflecting Pool from the Lincoln Memorial."

It is often argued that humans are "efficient" thinkers. Although this is rarely well-supported in practice, as humans easily fall prey to the inefficiencies of classical NP-hard problems, there are indications that humans are uniquely positioned to solve certain kinds of problems. Indeed, recent studies making use of functional magnetic resonance imaging and transcranial magnetic stimulation have begun to identify different regions of the brain which appear to be relevant for social and emotional reasoning [33, 30] as well as "creative" thought such as analogy and metaphor [26, 32]. It is similarly likely that a general-purpose, flexible problem solver must be able to select from efficient, smaller problem solving components which are specialized at solving parts of a larger problem.

Human intelligence is also notable for its ability to handle contradictory beliefs, contrary to the tenets of classical logic, which necessarily imply that simultaneous belief in two contradictory statements necessarily renders all statements true (i.e. *ex falso quodlibet*). This suggests that human thought may not be founded in the realm of classical logic. Instead, humans may be capable of thinking within a superset of such logic.[1]

The flaws of classical logic with respect to non-contradiction are hardly universal however, and other logics may yet prove to be the basis of thought. Modern logics, such as Belnap's four-valued logic [14], that take into account paraconsistency, in which contradictions should not cause an "explosion" of conclusions, or logics that adhere to dialetheism, in which contradictions may exist as facts, seem to embody much more pragmatic modalities for man's rational thought. The rational thinker attempts to resolve observed contradictions by revising his beliefs, not by breaking down.

Most important, however, is humanity's ability to apply a wide variety of different approaches to solving the same problem. For example, the Pythagorean theorem may be demonstrated by way of any one of dozens of methods, including algebraic solutions, geometric rearrangement, and even proofs based on dynamic systems (i.e. physics) [15]. While the symbolic thought of a single human does appear to be constrained to follow a single approach at any given time (i.e. humans have a difficult time thinking about multiple unrelated ideas at the same time), not only are humans free to apply different approaches based on personal judgments made while solving a problem, but humans may even derive and learn new approaches to solve novel problems which have yet to be encountered. These approaches may then be applied in the future. Any system that attempts to achieve a modicum of intelligence must be prepared to be flexible in its approaches to problem solving in these ways, or it is unlikely that it will truly resemble the abilities that may be achieved by a human.

[1] The mere existence of human logicians suffices as demonstrative proof that classical logic may be evaluated using the human mind.

Criteria	Contribution
Customer eats healthy food[2]	-2
Customer eating habits are unknown	-1
Customer eats unhealthy food	+2
Customer is a skydiver	+3
Customer being a skydiver is unknown	+0.5
Customer is not a skydiver	-0.1
Customer is a rock climber	+2
Customer being a rock climber is unknown	+0.25
Customer is not a rock climber	-0.1
Customer is a scuba diver	+1
Customer being a scuba diver is unknown	+0.1
Customer is not a scuba diver	-0.1
Customer rides a motorcycle	+2
Customer riding motorcycles is unknown	+0.2
Customer does not ride motorcycles	-0.1

Table 1.1: Contributions to risk score based on personal behaviors

1.2 A Problem: Issuing Health Insurance

As an example of the flexibility of humans in solving problems, consider the following scenario:

Sally is an insurance underwriter working for Aintno, a health insurance company. As the entirety of the reforms of the Patient Protection and Affordable Care Act have not yet been put into place as of 2013, it is still possible to reject applications for individual health insurance policies. Sally's job is to review the files of prospective customers to determine whether or not they should be issued insurance.

When issuing insurance, Sally must determine Danny's eligibility in accordance with Aintno's eligibility policies, which determine such based on a system which scores the risk of insuring a prospective customer based on a number of different criteria. For each criterion that an individual meets, their risk score is adjusted appropriately (See Table 1.1). Once a final score has been calculated, it is compared with several thresholds. If the risk score is less than the minimum risk score threshold of 2, Aintno will issue an insurance policy to the customer. However, if the risk score is greater than the maximum risk score threshold of 3, Aintno will refuse to issue insurance to

the customer. If the risk score is between 2 and 3, Aintno will attempt to do more work to determine whether insurance should be issued.

When Sally arrives at her desk one morning, she finds that she has been given the file of Danny, a young adult looking for health insurance. As a prospective customer, Danny's eligibility must be determined before insurance may be issued. If he is not eligible, then Sally must note that insurance was denied, and, so as to be able to defend such denial in legal proceedings, must note the reasons which lead to the denial (i.e. the facts which led to the excessively high risk score).

Similarly, if he is eligible, she must finalize an offer to insure Danny. In this case, she must still note the risk score and the sources of the risk, as the sources and amount of risk are relevant to determine what Danny's insurance premiums should be.

In order to determine Danny's risk score, she starts up Aintno's custom risk analysis program and begins to input the data from Danny's file, including his Facebook and Flickr social network profiles[3], which were gleaned from an optional field which had been filled in in Danny's file. As she does, Aintno's risk analysis program mines the two profiles for useful information which may be used to determine Danny's risk.

Initially, the program attempts to score Danny's risk on the basis of criteria other than his diet, as Aintno's policy is to avoid making such determinations if at all possible due to the perceived subjectivity and fluidity of customers' diets. It identifies Danny as high-risk, however, due to his engaging in motorcycling and skydiving. Together with his inability to prove he is not a rock climber and his lack of SCUBA diving experience, Danny has a total risk score of 5.15.

Sally prepares to issue a denial to Danny, but then remembers that, due to a law recently passed in Danny's home state, Aintno is not permitted to mine Facebook for insurance purposes. She directs the program to remove all facts that depended on Facebook. This drops his risk score to 2.15 (as the fact that he engages in skydiving

[3]Although the use of information from social networks has not been used in underwriting as in this example, some insurers already use information from social networking sites in fraud cases [22], and a recent study by Deloitte Consulting and British insurer Aviva PLC found that a predictive model based on consumer-marketing data such as hobbies and TV-viewing habits was judged as largely successful in comparison with traditional underwriting, finding that "the use of third-party data was persuasive across the board in all cases" [27].

eats($subject$, $food$) $\wedge$ unhealthy($food$)	$\rightarrow$	eats($subject$, unhealthy-food)
likes($subject$, $thing$) $\wedge$ is-a($thing$, food)	$\rightarrow$	eats($subject$, $thing$)
likes($subject$, $place$) $\wedge$ is-a($place$, restaurant)	$\rightarrow$	eats-at($subject$, $restaurant$)
works-at($subject$, $place$) $\wedge$ is-a($place$, restaurant)	$\rightarrow$	eats-at($subject$, $restaurant$)
(eats-at($subject$, $place$) $\wedge$ is-a($place$, restaurant) $\wedge$ primarily-serves($place$, $thing$) $\wedge$ is-a($thing$, food))	$\rightarrow$	eats($subject$, $thing$)

Table 1.2: Rules for determining risk

was only found on his Facebook profile), and causes the system to do more work and activate a set of rules regarding Danny's eating habits.

As Danny works at Hal's Hot Dogs (a hot dog restaurant), a number of pre-programmed rules which assist in assessing and accumulating risk scores based on eating habits (See Table 1.2) determine that Danny is a likely consumer of hot dogs, an unhealthy food. As a result, Danny's risk score is returned to an unacceptably high 4.05. Thus confident in Danny's ineligibility, she finalizes the rejection of Danny's policy.

1.3 A Solution (and Related Work)

Certainly individual components of this problem could be solved using existing rule systems, data mining tools and a simple score aggregation algorithm. But what would it take to remove Sally from the equation altogether? Could we automate the process of determining eligibility and eliminate the role of the human altogether?

A naïve approach to such automation would simply result in a domain-specific solution by determining the requirements and rules surrounding Sally's workflow. From a low-level perspective, we may consider the act of *receiving a request for analysis* to drive the process of reading and interpreting Danny's file, which subsequently causes work to be done to mine and analyze information connected through his Facebook and Flickr profiles, not only to better inform about lifestyle choices Danny may not have been asked about in his application, but also to determine whether Danny's application may contain missing or incorrect information.

Evidently, then, automation is possible, but we are left with two subtle issues in accepting this naïve approach to problem solving:

1. Such a domain-specific solution is likely to be brittle and require careful re-tooling as rules, regulations, and inputs change. For example, if Flickr increases the cost (be it computational or financial) of accessing the data it provides, the heuristics used to guide the analysis are likely to change. Rather than querying Flickr for every application, Aintno might only wish to query Flickr for additional details if other parts of Danny's application suggests that he might be lying on his application, but proof is lacking. How can we make this solution *flexible* depending on changing inputs and rules?

2. While we have a method of producing a domain-specific solution corresponding to a workflow to solve a particular problem, this still doesn't address the fundamental act of problem solving in and of itself. It captures nothing of the flexible planning and thought which we associate with true intelligence, as work is likely forced into a procedural rather than declarative mode. Is it possible to generalize this problem-solving approach so that we rely on domain-specific input knowledge rather than a domain-specific problem solver?

In short, such a domain-specific solution does not capture the true nature of human intelligence. Much of the flexibility and power of the human mind is left behind during the process of building the solution according to the constraints of the problem. The fact that a human may calculate this risk in a number of innovative ways is lost when an automated solution is constructed, as such automation ultimately implements only one such method. As a result, naïve, brittle problem-solving mechanisms cannot be reused and may find difficulty in integrating with other domains.

Instead, I propose a system in which knowledge is expressed in terms of orthogonal axes of beliefs. These beliefs may be connected in such a way that control may be made explicit through the expression of appropriate beliefs (e.g. a "need to support" some belief may drive work to uncover proof which supports that belief). These

connections also allow these beliefs to propagate through a network of operators which effect work so as to meet goals expressed as other beliefs.

In some respects, this interlinkage between goals, knowledge, and actions resembles Marvin Minsky's *K-line* theory [16], in which relevant knowledge, stored and organized hierarchically, may be used to configure perceptive units (*P-agents*) to create partial "hallucinations" of perception to assist in achieving goals and solving problems. By synthesizing "missing" perceptions, Minsky argues that existing problems are more readily aligned with previously encountered ones, allowing for the selection of the mechanisms best suited to solving them.

Relevant to the work presented here is Minsky's extension of the K-line theory to include an additional "G-net" of goals which influence which knowledge is likely to be relevant at a given time. Just as knowledge may influence perception through the activation of K-lines, Minsky proposes that goals may influence the activity and application of knowledge (and thus, indirectly, perception) through the connections that exist within the net of goals as well as perceptions.

The problem solving strategy proposed here resembles K-line theory, and treats goals, knowledge, and perceptions as independent beliefs which may be connected by a network of computational propagators which propagate beliefs between various statements of goals, knowledge, and perceptions. In this way, a goal which is established to calculate Danny's risk may establish a belief in another goal which expresses a "need to know" what Danny's eating habits are. This system also provides a mechanism for connecting these goals, knowledge, and perceptions to systems which do practical work (e.g. reasoning, discovery mechanisms, or aggregation of values), and as such moves beyond K-line theory to demonstrate the practicality of such a theory.

It is worth noting that the system presented in this thesis also owes a significant debt to the work done by de Kleer, et al on the AMORD system [5]. In addition to making use of truth maintenance systems to maintain dependencies and enable backtracking in reasoning, the propositional model also allows for the expression of rules and the assumption and retraction of various premises made possible by TMSs as a whole, a concept inspired by the work done in AMORD to do likewise.

Unlike AMORD, however, the "belief propagation" system presents an alternative, more general view of belief. Where AMORD implicitly assumes that a belief is a positive "assertion" of truth, this thesis takes the position that a particular "fact" or *proposition* may have a number of different beliefs associated with it, including acceptance of the proposition (i.e. the affirmative belief that the proposition is true), rejection of the proposition (i.e. the negative belief that the proposition is false), as well as simple beliefs which express the presence or absence of knowledge regarding the proposition entirely. As a result, the propositional system proposed here permits a greater amount of control than AMORD does, due to the greater expressivity of the propositional system.

1.4 Thesis Overview

This thesis seeks to outline the belief-propagation mechanism proposed above, which achieves such flexibility in problem solving by modeling knowledge with respect to support for a given belief. These models are built on top of the propagator network model, described in Chapter 2, and the *proposition model* of knowledge is then proposed in Chapter 3. Chapter 4 then illustrates how the propositional knowledge model may be used to solve problems in a flexible manner. Chapter 5 explains how the belief propagation mechanism may be used to construct meaningful explanations for the results of such problem solving. Finally, Chapter 6 demonstrates an example of the propositional knowledge model while Chapter 7 offers some directions for future work.

Chapter 2

Propagator Networks

The flexible problem solving mechanism described in this thesis depends on the powerful computational substrate called propagator networks, developed by Alexey Radul and Gerald Jay Sussman [21]. This computational substrate provides a simple mechanism for maintaining and updating partial information structures so that partial information about an attribute or value may be refined over time. In this chapter, a short description of the technology is provided in this chapter so that the reader may better understand the mechanism of the propositional system explained in subsequent chapters.

2.1 Propagators

Propagator networks consist of a network of two kinds of elements. *Propagators* are small computational units which do work on various inputs stored in single-storage memories known as *cells*. Any propagator may do work based on data in zero, one, or more cells, and may store output in one or more output cells. This work may range from simple operations such as addition and subtraction to complex calculations and algorithms. In principle, most complex operations are performed by networks of simple propagators representing basic mathematical operators and "switches" in the propagator network, which alternately connect one of several input cells to an output cell.

Propagators serve much the same purpose as electronic components do in an electrical circuit; the ways in which simple propagators are connected help to define the contents of cells at any given point, much as electronic components will influence the voltage and current at any given point in a circuit.

Propagators also resemble electronic components in another way: just as partial circuit diagrams may be abstracted and reused as "compound circuits" (such as with integrated circuits), partial propagator networks may be abstracted and reused as "compound propagators". As these compound propagators become active, they construct the partial propagator network represented by the compound propagator, thus permitting for recursion and the constrution of loops.

For example, a "factorial" propagator might expand into a network consisting of a subtraction propagator to subtract 1 from the input, another factorial propagator to calculate the factorial of one minus the input, a switch to determine whether that factorial should become active (e.g. if the input is less than 1), and a multiplication propagator which multiplies the output of the "inner" factorial propagator with the input of the "outer" factorial propagator. Then, if the input is much greater than one, the inner factorial propagator will be activated to calculate the factorial recursively.

2.2 Cells

The other component of a propagator network, the *cell*, acts as the glue of the propagator network; it stores data which may be used by propagators to do computation. The information stored in a cell may be *updated* at any time by a propagator which sends an "update message" to that cell. This message contains any new information which may be used to inform the contents of that cell by *merging* the contents of that message with the data currently in the cell using an appropriate merge operation.

The merge operation used to combine the update message and the existing information in a cell is selected based on the type of data stored in the cell and the type of data provided in the update message. For example, if a cell stores a numeric interval and receives another numeric interval in an update message, it may merge the update

by taking the *intersection* of the two intervals. In this way, the information stored in the cell may be gradually refined by obtaining ever narrower estimated ranges of the value of the cell.

When a propagator network is constructed, propagators may be registered as *neighbors* of cells so that they may be alerted when the cell's content changes. Once a cell has finished merging its content with the content in the update message, those neighboring propagators will be alerted and given an opportunity to do additional work, typically by making use of the newly updated value of the cell. These alerted propagators may then send updates to other cells. As a result, updates to the content of a cell will effectively *propagate* across the network of propagators and cells.

As the numeric range example demonstrates, the ability to use merge operations to gradually refine the information stored in cells means that propagator networks readily lend themselves to the representation and manipulation of *partial information*. Rather than representing complete knowledge about a value, the concept of partial information means that, while the attribute or variable which is *represented* by the partial information is fixed, the value of the attribute itself may be incomplete, and extended as more information is known.

For example, in the numeric range example above, a cell might represent the outside air temperature, even though the actual value of the cell is a range with lower and upper bounds. In this sense, the inherent errors in measurement may be made explicit, so that, rather than some complete, presumably unmeasurable, value, we give a range that the actual temperature lies within. That is, by providing lower and upper bounds on the temperature, we are giving *partial information* about the temperature, rather than full information.

Cells which store partial information may, with appropriate merge operations, build up the partial knowledge in a cell when an update is received, so as to obtain a better, more informed, value. In the numerical range example above, range intersection is an appropriate merge operation, as it may be used to refine multiple, potentially equally broad, ranges to obtain a more precise answer than any one of the "input ranges" on their own.

2.3 An Example

Consider a system which seeks to measure the local air temperature in Boston in degrees Celsius using two thermometers. Like any practical measurement device, these thermometers have an error range and are not guaranteed to measure the actual temperature. These thermometers do not behave identically, so they may report different temperature ranges. Given this fact, it is possible to obtain a more accurate measurement of the temperature by considering the intersection of their error ranges.

In addition to their measurement flaw, these thermometers have one other practical flaw, in that they measure the temperature using different scales. One measures the temperature in degrees Celsius, while the other measures in degrees Fahrenheit. As a result, care must be taken to ensure that the measurements of the thermometer which reads in Fahrenheit are converted to degrees Celsius.

Given this fact, we may consider solving for the temperature of the thermometers in degrees Celsius using a simple propagator network, given in Figure 2-1-1. Here, the thermometers act as propagators which update cells with their estimate of the temperature range. When the Fahrenheit thermometer reports its temperature (as in Figure 2-1-2), it sends an update to the Fahrenheit temperature cell, cell A. This cell is then connected to a subtraction propagator, which will subtract 32 from the contents of cell A and update the value in the output cell (B) with the newly calculated intermediate value of the conversion (Figure 2-1-3). A second propagator is connected to that intermediate cell B and will multiply that value by $\frac{5}{9}$, and use that value to update the Celsius temperature in cell C (Figure 2-1-4).

Note that in all steps in Figure 2-1, the cells to which updates are sent simply adopt the contents of the update message to be their new content. This is due to the fact that the cells initially contain a null-like value, "nothing". This null-like value is the initial value of a cell, and represents a lack of any partial information about the cell's value whatsoever. Thus, since there is no partial information with which the numeric range in the update data can be merged, the merge operation simply sets the value of the cell with the initial, "more specific" partial value given in the update.

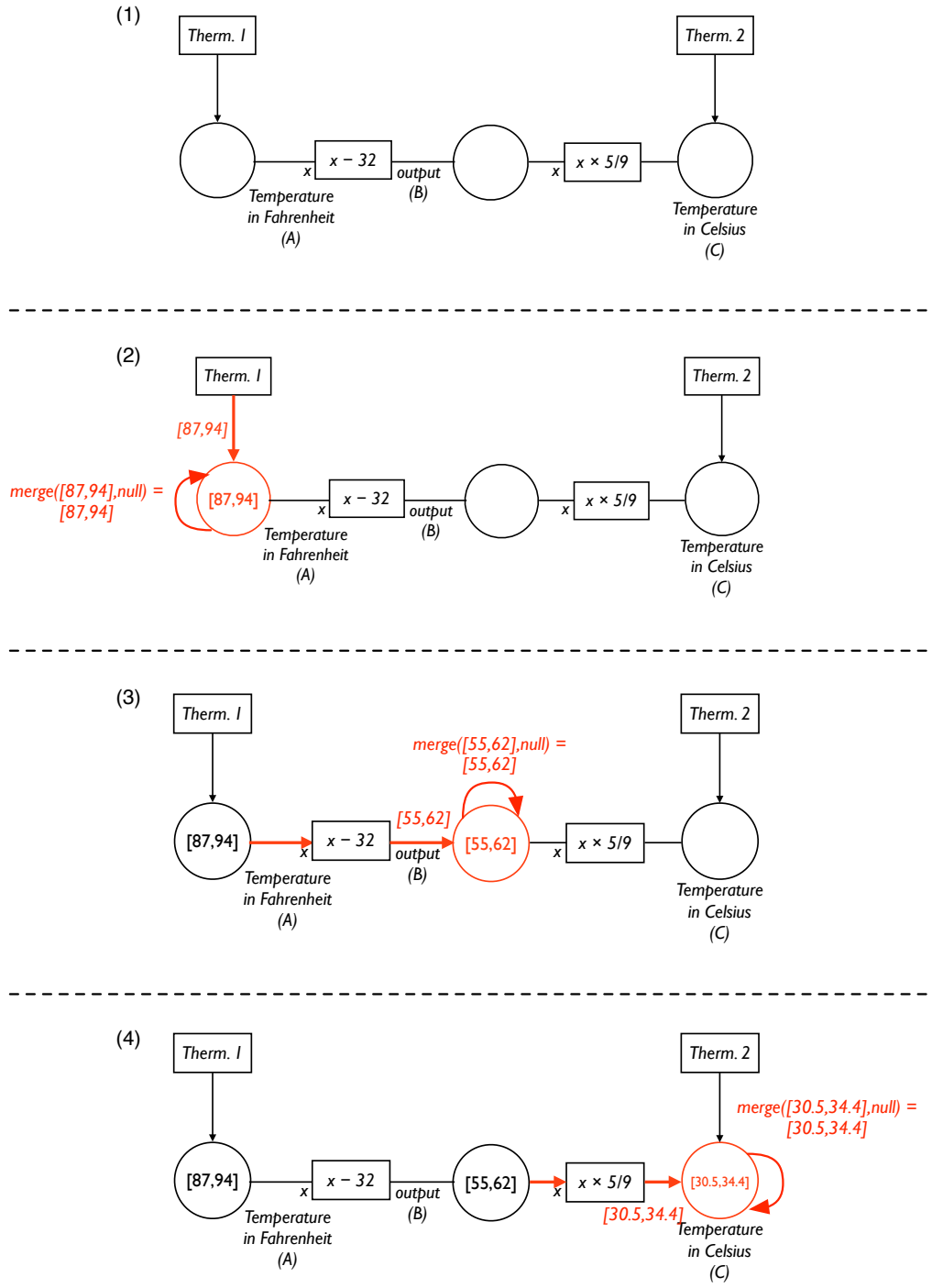


Figure 2-1: A propagator network which combines and converts the outputs of two thermometers, converting between Fahrenheit and Celsius. A temperature range from thermometer 1, in degrees Fahrenheit, (2) is sent to a cell. This alerts a chain of propagators responsible for converting the temperature range to Celsius (3, 4).

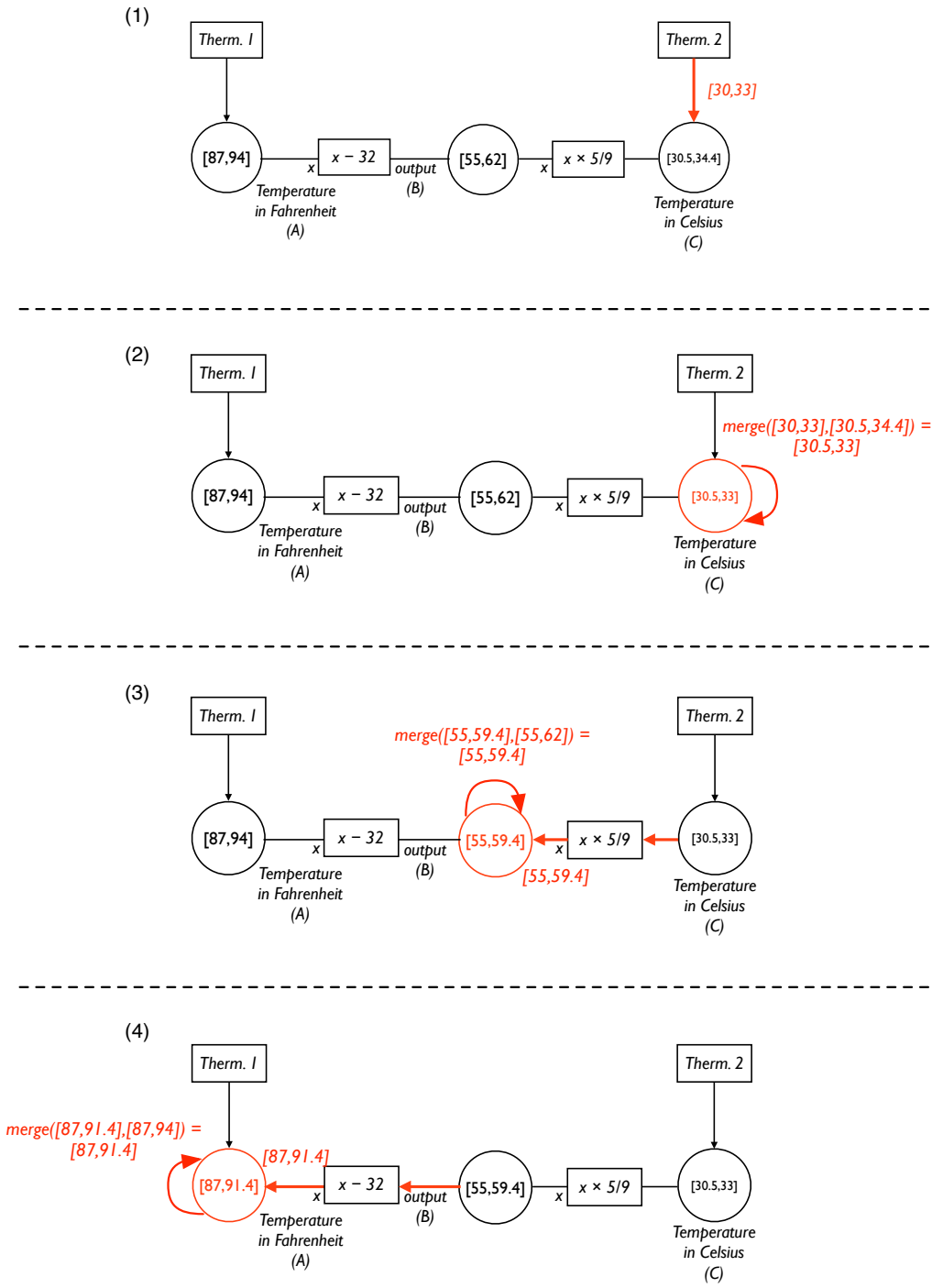


Figure 2-2: Propagator networks merge the existing contents of cells with updates received from propagators. An update in degrees Celsius from thermometer 2 (1) is merged with existing knowledge of the temperature (2) and propagates to the output cell containing the temperature in degrees Fahrenheit (3, 4).

However, when the second thermometer sends its temperature reading in degrees Celsius as an update to cell C (Figure 2-2-1), the cell actually takes the intersection of the current value in cell C and the value in the update and uses that value as the new value of the cell (Figure 2-2-2). Propagators may, of course, be constructed as reversible operations so that an update of an "output" may refine the "input". In Figures 2-2-3 and 2-2-4, the multiplication and subtraction propagators act reversibly and and divide and add to update the temperature range in degrees Fahrenheit in cell A. As a result, the Fahrenheit temperature range in cell A is also an intersection of the two ranges.

2.4 Handling Contradictions

While Figure 2-2 demonstrates the value of refining numerical ranges by taking their intersection, this raises a conundrum: what if the range in the update does not intersect the range currently stored in the cell? Or more generally, what if an update to a cell contains information which contradicts what is already stored in it?

When humans encounter a contradiction in practice, they typically desire to resolve the contradiction to obtain a new, reliable value from which additional work may be derived. Such resolution may be done in many ways, and often results in additional work being done to determine a resolved value. For example, if a temperature range contradicts an existing estimate, it may be appropriate to "kick out" older contributions which may not accurately represent the current value of a changing temperature. It might also be appropriate to determine whether a given thermometer is broken or unusually inaccurate. If so, this may prompt a user to repair or remove the faulty thermometer.

Because the ways in which a contradiction may be resolved may vary drastically depending on the contents of the cell and the nature of a problem to be solved by the network, implementations of propagator networks should be flexible in handling any conflicts. The implementation of propagator networks in the MIT/Scheme programming language, which has been used as the basis for the work in the remainder of

this thesis, normally raises an exception when a contradiction is encountered, causing computation to halt so that humans may examine the nature of the contradiction and resolve it in an appropriate manner.

Requiring human intervention is not a practical solution for most problems, however; it would be untenable to require human input to abort every "dead-end" in a computation by removing inappropriate inputs, especially in the large search spaces that lie at the core of many kinds of problems. While the underlying propagator network mechanism may not directly support contradiction handling, the flexibility of the "cell merge" operation permits basic contradiction handling to be performed as part of the logic of such merge operations. The data structure known as a truth maintenance system, or TMS, has several features which make it particularly appropriate for handling contradiction resolution in their merge operation.

2.5 Truth Maintenance Systems and Backtracking

Truth maintenance systems (TMSs) [6, 7] are data structures which allow for the maintenance of a variable's value based on the set of minimal supports or *premises* for a given value. Although there are several kinds of truth maintenance systems, assumption-based truth maintenance systems (ATMSs) are among the most useful to track all possible premise-set/value pairs known to be valid for a given variable. They may also be "queried" to retrieve the value that is best supported by a given set of premises. Such values may be returned with the minimal set of premises known to support the value, conveniently allowing for the identification of the subset of premises most relevant to the specification of the returned value.

TMSs[1] are appropriate contents for cells, as they may be merged by simply taking the union of the set of value-support pairs in a cell and a set of such pairs in the update message. The union of the sets of value-support pairs may then be treated and stored

[1]It is worth noting that not all of the powers I have mentioned in this section are available in other kinds of truth maintenance systems (such as justification-based truth maintenance systems or logical-based truth maintenance systems). As a result, whenever I mention truth maintenance systems (TMSs) hereafter in this thesis, I refer to assumption-based truth maintenance systems.

as the new contents of the cell, so that smaller support sets for an otherwise identical value may be propagated through neighboring propagators to their output cells.

By storing values as a function of a set of premises, TMSs may mark these sets as "no-good" when contradictions are identified during the merge process (i.e. when two value-support pairs with different values are supported by the set of premises) and resolved through manipulation of the premise set at merge-time. Such premise-set manipulation may change the effective value of a cell without actually introducing a contradiction in the actual value stored in the cell; the contents of the TMS grow and change independently from the set of believed premises, which are not subject to the contradiction behavior of the underlying propagator network itself.

Manipulation of the premises during the cell merge operation introduces the ability to implement algorithms which require a *backtracking* search. When a particular premise propagates to a cell and creates a contradiction in a TMS value when the merge operation is applied, the TMS merge operation may effectively retract that premise so that other premises may be tested for a suitable solution. In short, searching for the solution to a problem using propagators becomes simply a matter of searching for the set of premises which solves a problem without causing any contradictions.

But while backtracking is but a necessary technique for problem solving, it is not the whole story. Propagator networks and truth maintenance systems provide a powerful mechanism which may be used to solve problems, they are still nothing more than a computational platform, and they lack any features which might assist in higher-order problem solving strategies. To actually attack problems and control problem solving based on needs, desires, or beliefs, we need a way to represent these needs, desires, and beliefs beyond the simplistic value maintenance of a TMS. For that, we must turn to the idea of the proposition.

Chapter 3

Five-Valued Propositions

Controlling problem solving systems requires representations of both those beliefs that act as input to the problem solvers and those beliefs which may be used to control the problem solving itself. While the facts about Danny's hobbies may be sufficient to determine that he should not be granted insurance coverage, in practice, it is necessary to be able to express and recognized the *need* for this determination before it may happen. I have represented both the these needs and facts used as input for problem solving in the form of *propositions*.

3.1 Propositions

A *proposition* is any statement which may be believed to be true or false. Propositions differ from traditional logical statements in so far as they have no inherent truth or falsehood in and of themselves. The existence of a proposition does not imply that the proposition is true; propositions represent merely the concept of a statement without any associated belief. To represent the nature of belief in a particular proposition, each proposition is considered in terms of five different axes of belief: *acceptance*, *rejection*, *contradiction*, *knowledge*, and *ignorance*.

The first two axes, acceptance and rejection, correspond to belief in the truth and falsehood of the proposition, respectively. That is, a proposition is *accepted* if it is believed to be true, while a proposition is *rejected* if it is believed false. Note

that truth and falsehood are independent of one another! It is quite possible for a proposition to be both accepted and rejected (albeit temporarily), or to be neither accepted nor rejected. By separating these two concepts as two distinct axes of belief facilitates the expression of complex combinations of the support for and against a particular proposition which may be evaluated differently in different contexts.

The remaining beliefs may be expressed in reference to acceptance and rejection. Acceptance and rejection may be considered together to determine whether there is a *contradiction* in beliefs. Such a contradiction may force problem solvers to backtrack by removing certain assumptions that may have led to such contradictory beliefs.

Where the *contradictory* state of belief is the conjunction of acceptedness and rejectedness, *knowledge* (or knownness) is the disjunction of acceptance and rejection. A proposition is *known* if there is support for the proposition to be either accepted or rejected. Similarly, a proposition may be *unknown* entirely (i.e. the system may be ignorant of any knowledge regarding the proposition) if there is no support for it being either accepted or rejected.

This alternate metric of *ignorance* is what provides for flexibility and control in problem solving; it is possible to use a lack of knownness to determine when work should be done to determine a solution. Often, once a proposition is believed to be true or false, it is unnecessary to expend additional effort to uncover additional support. For example, in the Aintno example, there is no need to calculate a complete risk score for insurance purposes; once sufficient evidence has been gathered to determine that insurance should be denied, the work performed to acquire additional evidence for denial is unnecessary, as the relevant determination (whether the claim should be accepted or rejected) is already made. (But this does not mean that work cannot be done later if the evidence changes!)

What is important for all of these facets of knowledge and belief is that each state of belief is only loosely connected to the others. Excepting the basic logical relationships (e.g. simultaneous acceptance and rejection is contradictory, and a proposition cannot be both known and unknown), support for each belief state is independent and can be used to drive problem solving separately from belief in any

of the other belief states. As a result, each belief state may drive problem solving in a different manner appropriate to the problem. If the problem requires it, *ignorance* may be used to control when work is done to prove acceptance or rejection, but this is not a requirement. Similarly, the belief in the rejection of a given proposition may be used to drive computation to disprove such rejection (e.g. if there is a strong desire to prove acceptance through contradiction).

3.2 Implementation

The evaluation of a proposition with respect to five values is an important foundation of expressing belief, but how are we to actually represent this knowledge and its connections to other beliefs? If we are to properly link propositions so that their beliefs influence each other and so that computation and problem solving is contingent on the nature of belief, we may wish to construct a *network* of propositions using propagator networks, so that different states of belief in a proposition are able to influence and modulate beliefs in other propositions. In this way we may cause computation to occur.

This use of the propagator architecture for "belief propagation" bears many similarities to constraint propagation systems, first described by David Waltz [31]. In particular, the relationships between beliefs which are expressed with respect to statements may be considered edges which constrain the beliefs on either end of the relationship. For example, a rule $A \rightarrow B$ may be represented by a connection between an affirmative belief in A and an affirmative belief in B, as well as a second connection between a negative belief in B and a negative belief in A.

While combining constraint propagation and logical programs is admittedly not a novel idea [12], the propositional architecture exposes program control structures and integrates them into the system of constraints. Rather than simply delegating and expressing logical relationships as constraints themselves, this architecture allows for the amount of work done to "prove" a logical statement to itself be constrained on the basis of belief in other statements.

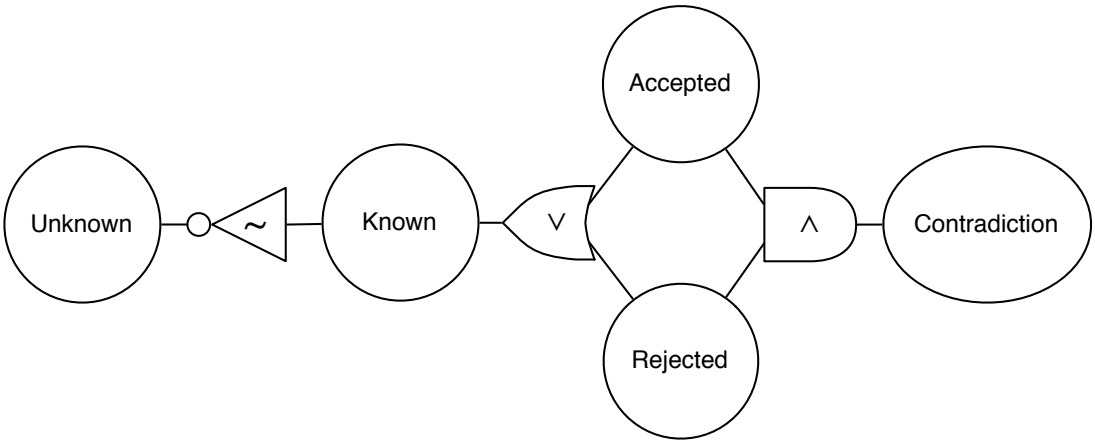


Figure 3-1: A proposition may be described with respect to five values connected in a propagator network of logical operations. Each circle represents a "cell" containing the data in support (or opposition) of that particular belief state, and the currently supported value will propagate its dependencies to related belief states as needed based on the nature of belief.

For example, it should be possible to condition the "acceptance" of a need to do research on our "ignorance" of a proposition for which we would like to have knowledge. That is, if some proposition is unknown, it should translate to an "acceptance" of a second proposition representing the need to know the first, unknown, proposition. Likewise, as the first proposition becomes known, the second proposition, the need to know, should become rejected.

Given all this, when mapping the propositional model to a propagator model, it is sensible to map the individual "beliefs" to cells. The "rules" which relate beliefs may be considered propagators which connect belief cells and combine belief states to generate another.

Given the basic relationships between belief states described previously, we can model a single proposition as a collection of five cells with logical propagators connecting them, as in Figure 3-1. Of special note in this particular model is that the *contradiction* cell is automatically populated with a value of **false**. This captures the implicit assumption that no proposition may be simultaneously accepted and rejected (it may, of course, be simultaneously not accepted and not rejected). Furthermore,

these propagators may be reversible (unlike typical logical circuits which distinguish logical inputs from logical outputs) so the **false** contradictory state will not only support the automated conclusion of *accepted* = ¬*rejected*, but it may also assist in forcing backtracking to occur when there is support for a proposition to be both accepted and rejected (as will be described later in Section 3.3).

3.3 Hypothetical Beliefs and Backtracking

As mentioned in Section 2.5, we gain additional expressive power if we choose to store a truth maintenance system as the contents of each belief cell. Though any given belief cell will evaluate to a given **true** or **false** value depending on whether our belief in the proposition is in accordance with the belief associated with the cell, if a truth maintenance system is utilized, we may modify our beliefs by simply modifying the set of premises which we hold to be true. Such a feature also permits backtracking by removing premises, which I discuss in more detail here by focusing on the concept of the *hypothetical premise*.

If we desire to model certain problem solving approaches, we must be able to adopt premises by supposition, whether because information needed to prove or solve a problem is incomplete, or because the supposition is itself used in the process of proving the opposite belief (i.e. proof by contradiction). These *hypothetical premises* are similar to other "concrete" premises which might exist in the premise set (those which might be founded upon observations, measurements, or other external sources for belief), but hypothetical premises are distinguished by a more transient, context-specific nature.

Hypothetical premises rarely support more than one independent belief directly. Additional beliefs may depend on a hypothetical premise in so far as they depend on the acceptance or rejection of another proposition, but they generally are not supported directly. In contrast, concrete premises may support multiple beliefs directly (e.g. Danny's Facebook interests may serve as a single premise which not only supports the belief that he is a sky-diver, but also the belief that he is a motorcyclist.)

More important, however, is that, unlike concrete premises, hypothetical premises are generally designed to be retracted as additional evidence supports an alternate position. For example, if we assume that Danny is a non-smoker for the sake of determining his insurance rates, and we later encounter evidence that, in fact, Danny actually smokes, it is desirable to reject our previous assumption that he was a non-smoker and re-determine *only* those beliefs which were premised on that hypothetical belief. Thus, we would like to automatically *kick out* the assumption based on the fact that the presence of other, more concrete premises create a contradiction.

Another example of the relative transience of the hypothetical can be found in the argumentation form *reductio ad absurdum*, which assumes that, if the denial of some statement is assumed and a contradiction is encountered, said assumption may be dismissed in favor of a "proof by contradiction" of the contrary. That is, a particular proposition may be believed to be rejected on the basis of a hypothetical premise and, should that premise lead to a contradiction, the premise may be kicked out and acceptance of that same proposition may be supported instead, based on the support which led to the identified contradiction. For example, Aintno might prove that Danny is a non-smoker by contradiction as follows:

1. Assume Danny is a smoker.
   (Add a hypothetical premise supporting smoker(danny))

2. All smokers have chronic obstructive pulmonary disease (COPD).[1]
   $(\forall x.\text{smoker}(x) \rightarrow \text{has-copd}(x))$

3. Therefore, Danny must have COPD.
   (has-copd(danny))

4. Individuals with COPD have a FEV_1/FVC ratio[2] of less than 70%.[3].
   $(\forall x.\text{has-copd}(x) \rightarrow \frac{\text{FEV}_1}{\text{FVC}}(x) < 70\%)$

[1]This is not actually true, but is assumed for simplicity.

[2]The FEV_1/FVC ratio is the ratio of the volume of air expelled in the first second of a forced breath over the maximum volume of air that can be expelled

[3]According to standards set by the National Institute for Clinical Excellence [19]

5. Therefore, Danny must have an FEV_1/FVC ratio of less than 70%.
   $(\frac{FEV_1}{FVC}(\text{danny}) < 70\%)$

6. In fact, Danny has an FEV_1/FVC ratio of 92%.
   $(\frac{FEV_1}{FVC}(\text{danny}) = 92\%)$

7. Such a fact contradicts the prior conclusion about Danny's FEV_1/FVC ratio.
   $(92\% \not< 70\%)$

8. Due to this contradiction, our initial assumption (that Danny is a smoker) must be incorrect, and we may instead support the opposite.
   $(\neg\text{smoker}(\text{danny}))$

In the propagator network model, hypothetical premises may be modeled as premises in the TMS premise set which are specially marked such that they may be automatically kicked out and removed from the system when a contradiction is detected. In effect, the merge operation for truth maintenance systems will preferentially remove hypothetical premises if a contradiction is encountered when merging. New beliefs are then recalculated based on the removed premise.

This principle explains why the contradictory cell is fixed to **false**: it causes a truth value for acceptance to be negated for the rejected belief (so that an accepted proposition is not also rejected, and a rejected proposition is not also accepted). Then, when support arrives to support the contrary belief, a contradiction will be encountered in the truth maintenance system of the accepted or rejected cell, forcing backtracking to occur.

Chapter 4

Building Problem-Solving Strategies

With propagators as the underlying computational model of propositions, it remains to be shown how these propositions may be connected to properly solve problems. As stated in the previous chapter, working with multiple propositions requires the simple extension of the "belief propagation" model to operate between multiple propositions. In this chapter, we demonstrate the use of such a model by focusing on a specific problem, proving the ancestry of a child, using propositions.

4.1 A Simple Rule

Consider the following simple problem: Joe is Mary's father, and Howie is Mary's son. Howie also has a son named Jeff with his wife Jane. Is Joe an ancestor of Jeff?

To a human, this problem is easy to solve given the assumptions that a parent is an ancestor of their child (i.e. $\forall a, b.\text{parent}(a, b) \rightarrow \text{ancestor}(a, b)$) and that ancestry is transitive (i.e. $\forall a, b, c.\text{ancestor}(a, b) \wedge \text{ancestor}(b, c) \rightarrow \text{ancestor}(a, c)$). In effect, we may use a simple pair of rules to draw conclusions about an individual's ancestry from a collection of parent-child relationships.

If the propositional model of reasoning is general enough to represent *all* methods of problem solving so as to integrate them with control, it stands to reason that it should be possible to model any one mechanism for solving problems, such as the application of rules, using propositions. The evaluation of a rule has two components:

1. *A proposition matching the pattern of the antecedent of a rule must be identified.* If we consider all propositions to be abstractly represented as n-ary predicates, then we must be able to discover those specific predicates which match the pattern of the antecedent of the rule such that any variables in the antecedent may be filled by atoms in the specific matching propositional predicates.

   Given a set of variable bindings which fix the values of variables that may be present in the antecedent, the set of all proposition-environment pairs must be found such that the proposition matches the antecedent pattern in arity, and all concrete atoms and "bound" variables are the same. The environment of a given pair then consists of the union of the input variable bindings and the new bindings derived from the alignment of the remaining "unbound" variables in the antecedent pattern with the values in the matching proposition.

   For example, if we take the above rule $\forall a, b.\text{parent}(a, b) \rightarrow \text{ancestor}(a, b)$ to start with, we would need to find some proposition which may be matched with the pattern $\text{parent}(a, b)$, such as $\text{parent}(\text{howie}, \text{jeff})$, which corresponds to the new set of variable bindings $\{a = \text{howie}, b = \text{jeff}\}$.

2. *We must connect the relevant belief in any matching proposition to belief in its consequent (as evaluated in the environment created by the matching the antecedent).* In short, we must create an instance of the rule by binding any variables in the rule to the corresponding terms in the matching proposition.

   Since we have matched $\text{parent}(a, b)$ with $\text{parent}(\text{howie}, \text{jeff})$, we have obtained the relevant bindings $a = \text{howie}$ and $b = \text{jeff}$. Thus, we may consider a specific instance of the parent-ancestor rule $\text{parent}(\text{howie}, \text{jeff}) \rightarrow \text{ancestor}(\text{howie}, \text{jeff})$. Given the nature of this rule, there is necessarily a connection between the *acceptance* of the former proposition of parenthood and *acceptance* of the latter proposition of ancestry. As a result, we must connect the two belief cells of the propositions such that an affirmative (**true**) belief in *accepting* the proposition $\text{parent}(\text{howie}, \text{jeff})$ will propagate, *with its set of premises* to the *acceptance* of the proposition $\text{ancestor}(\text{howie}, \text{jeff})$, as depicted in Figure 4-1.

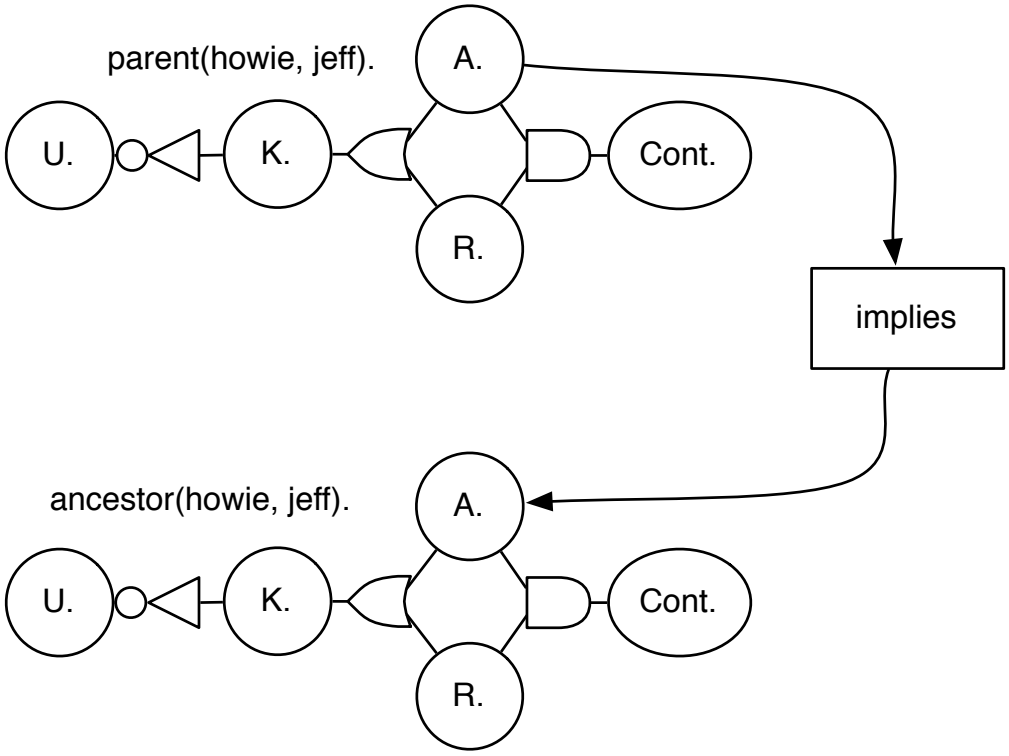


Figure 4-1: A simple network of propositions which propagates the belief in the acceptance of parent(howie, jeff) along with its dependencies to the acceptance of ancestor(howie, jeff). The "implies" propagator ensures that this belief is properly propagated without modification and merged with the contents of the output cell.

```
(define (prove-ancestry-by-parenthood a b)
  (let ((parent-proposition (proposition `(parent ,a ,b)))
        (ancestor-proposition (proposition `(ancestor ,a ,b))))
    (accept ancestor-proposition
            (list 'prove-ancestry-by-parenthood a b)
            (list (accepted parent-proposition)))))
```

Figure 4-2: Proving ancestry through parenthood. `(ancestor a b)` is accepted contingent on `(parent a b)` being accepted, by way of the "prove-ancestry-by-parenthood" rule.

We might represent such a "parenthood" rule using MIT/Scheme code like that of Figure 4-2. In the function `prove-ancestry-by-parenthood`, two propositions are retrieved: `parent-proposition` (i.e. `(parent a b)`), and `ancestor-proposition` (i.e. `(ancestor a b)`). With these two propositions in hand, the `accept` function states that `ancestor-proposition` will be believed to be *accepted*, as informed by (i.e. caused by) `(list 'prove-ancestry-by-parenthood a b)`. That is, we will potentially accept `(ancestor a b)` by way of the fact that we attempted to prove ancestry through parenthood (with the specified arguments a and b).

This acceptance is not a foregone conclusion, however; the `accept` function also states, through its third argument, that this acceptance is based on the acceptance of `parent-proposition`. While we may believe `(ancestor a b)` to be accepted for any number of reasons, we can only believe it is accepted due to parenthood *if* we also believe that we accept `(parent a b)`.

4.2 Knowing the Unknown

While the above example is perfectly acceptable should we know a and b, what if we do not? As stated above, the rule $\forall a, b.\text{parent}(a, b) \rightarrow \text{ancestor}(a, b)$ does not require us to know a and b. Indeed, it is true for all a and b. Would it not be better to be aggressive and proactively find parents, so as to prove ancestry without having to be told which ones we are looking for?

There are several problems which must be surmounted with such an approach. The most obvious difficulty is that of finding all such propositions that match `(parent a b)`. While we can certainly use efficient database storage and indexing algorithms to find all such `(parent a b)` that exist when we are given the rule, we must also be aware that it is very unlikely that we know all parenthood relationships at that particular point in time. Furthermore, if a system is intelligent enough to know the generic existence of parents (in short, $\text{parent}(b, c) \rightarrow \exists a.\text{parent}(a, b)$), then a system may easily get lost simply proving the existence of parents *ad infinitum* rather than discovering a relevant ancestry relationship.

Even lacking such a general rule, from a pragmatic standpoint it is also quite possible that we may not know about certain `(parent a b)` relationships at a given time, due simply to their current "irrelevance." In other words, at any given point there are *unknown unknowns*; not only do we not know whether we accept or reject some arbitrary `(parent a b)`, but there are `(parent a b)` propositions of which we are not even remotely aware!

Thus, when making such a pattern matcher, we must take care that it is *lazy*. Any attempt to prove ancestry through the existence of a parenthood relationship must be ready to be acted on at any time, as new parenthood relationships are introduced and believed to be true. That is, we must be prepared to make the connections between propositions *asynchronously*, by making such connections in *callbacks* which are invoked whenever such a statement is generated. Thus, we elaborate the code as in Figure 4-3, which pushes the connection of the "acceptance" belief cells into the function `accept-ancestor-by-parenthood`. This function may be called when a matching proposition is found.

Note that here we replace the instantiation of a proposition `(parent a b)` with the `find-proposition-matching` function. This function searches for propositions matching the pattern `(parent ?a ?b)`, where the question marks denote named variables a and b. Upon finding any such proposition, `accept-ancestor-by-parenthood` is called, with a first argument containing the matching proposition, and a second argument containing a mapping of the variable names to the values resulting from matching the pattern with the proposition.

For example, given the proposition `(parent howie jeff)`, the *parent-proposition* variable would contain the proposition itself, while the contents of the *environment* variable could be used to determine that the variable `?a` would be bound to `howie` and the variable `?b` would be bound to `jeff`.

These environmental bindings are used in `instantiate-proposition`, in which they are substituted for the `?a` and `?b` variables in the pattern `(ancestor ?a ?b)` to instantiate the proposition `(ancestor howie jeff)`. Then, the acceptance of `(parent howie jeff)` is connected to the acceptance of `(ancestor howie jeff)`

```
;; This function is called with a proposition matching the pattern
;; (parent ?a ?b) and an environment that contains the values that
;; matched the pattern variables ?a and ?b.  It then accepts the
;; corresponding (ancestor ?a ?b) proposition contingent on acceptance
;; of the parent proposition.
(define (accept-ancestor-by-parenthood
         parent-proposition environment)

  ;; In order to accept the corresponding ancestry proposition
  ;; (ancestor ?a ?b) we create the proposition by using the
  ;; environment to fill in ?a and ?b...
  (let ((ancestor-proposition
         (instantiate-proposition '(ancestor ?a ?b) environment)))

    ;; We then accept ancestor-proposition based on ?a being the
    ;; parent of ?b, so long as we accept the proposition
    ;; (parent ?a ?b)
    (accept ancestor-proposition
            (list 'prove-ancestry-by-parenthood
                  (get-binding '?a environment)   ; Get value of ?a
                  (get-binding '?b environment))  ; Get value of ?b
            (list (accepted parent-proposition)))))))

;; Find propositions matching (parent ?a ?b) and call
;; accept-ancestor-by-parenthood for each such proposition.
(define (prove-ancestry-by-parenthood)
  (find-proposition-matching '(parent ?a ?b) '()
    accept-ancestor-by-parenthood))
```

Figure 4-3: Proving ancestry through parenthood generally. For every `(parent ?a ?b)` that is known, `(ancestor ?a ?b)` is accepted contingent on that `(parent ?a ?b)` being accepted, by way of the "prove-ancestry-by-parenthood" rule. Note the introduction of the *environment* variable which contains the variable bindings to a and b.

```
(define (prove-ancestry-by-parenthood)
  (find-proposition-matching '(ancestor ?a ?b) '()
    (lambda (prop-1 environment)
      (find-proposition-matching '(ancestor ?b ?c) environment
        (lambda (prop-2 environment)
          (let ((ancestor-proposition (instantiate-proposition
                                       '(ancestor ?a ?c)
                                       environment)))
            (accept ancestor-proposition
                    (list 'prove-ancestry-transitively
                          (get-binding '?a environment)
                          (get-binding '?b environment)
                          (get-binding '?c environment))
                    (list (accepted prop-1)
                          (accepted prop-2)))))))))
```

Figure 4-4: Proving ancestry through transitivity. For every `(ancestor ?a ?b)` and `(ancestor ?b ?c)` that is known, `(ancestor ?a ?c)` is accepted contingent on those two previous ancestor relationships being accepted, by way of the "prove-ancestry-transitively" rule. Note how the *environment* variable is carried as an argument to the nested `find-proposition-matching` function, and how it implicitly carries the bindings of the named variable *a* through to the inner lambda in which the proposition `(ancestor ?a ?c)` is instantiated.

as in Figure 4-2. Similarly, the `get-binding` function must be used to resolve the bindings to `?a` and `?b`. In this way, the explanation of the method used to conclude acceptance may be constructed.

The observant reader will note that the `find-proposition-matching` function takes an empty list as its second argument. A second argument is helpful when multiple patterns are chained together, as in the ancestor-chaining rule implemented in Figure 4-4. As the variable `?b` must be the same value in both `(ancestor ?a ?b)` and `(ancestor ?b ?c)` in order to prove ancestry through transitivity, the environmental bindings created by matching the former must be passed to the latter so as to partially instantiate the pattern `(ancestor ?b ?c)` with the known value of `?b`. Thus, the second argument acts as an environment in which to evaluate the pattern *before* performing a search, and the empty list merely denotes an empty initial environment containing no variable bindings.

4.3 Making Work Contingent

So far, we have worked under the assumption that the mere existence of a proposition, regardless of our belief in it, is justification enough to connect any possible acceptance of that proposition with the consequent proposition of the rule. This has a distinct downside; we will necessarily make such a connection for *all* propositions that match the specified pattern, even though many of these may never be accepted (e.g. if they may be rejected in the future rather than accepted). If, for example, our simple ancestor problem solver is given every potential parent-child relationship in the United States for a child under the age of 18, this would mean that our problem solver would need to build nearly 75 million minors $\times$ 300 million citizens $= 2.25 \times 10^{16}$ ancestor relationships [11]. This is extremely wasteful, given that only about 150 million of those relationships would ever be accepted, as a child has exactly two biological parents!

It would be far more reasonable to make any problem solver's work contingent, not on the mere existence of some proposition, but on the nature of our belief in it in the first place! Rather than blindly matching every proposition `(parent ?a ?b)`, which would result in doing far more work than our original rule, should we not preferentially connect only those `parent` propositions which we already accept? Should we not control our problem solving based on what we believe?

If so, perhaps an appropriate solution would make the body of the function `accept-ancestor-by-parenthood` contingent on a particular prior belief. For example, only when a matched proposition `(parent ?a ?b)` is believed to be accepted should the connection be made between acceptance of the parent statement and the acceptance of ancestry. In effect, we could make the act of connecting, itself, a propagator which is the neighbor of the accepted state of the parent proposition. Figure 4-5 gives an example of such a compound propagator in action. When coded up properly in MIT/Scheme using the `s:when` propagator, which lazily evaluates its body only when its condition (the first argument) becomes true, such a rule might look like that of Figure 4-6.

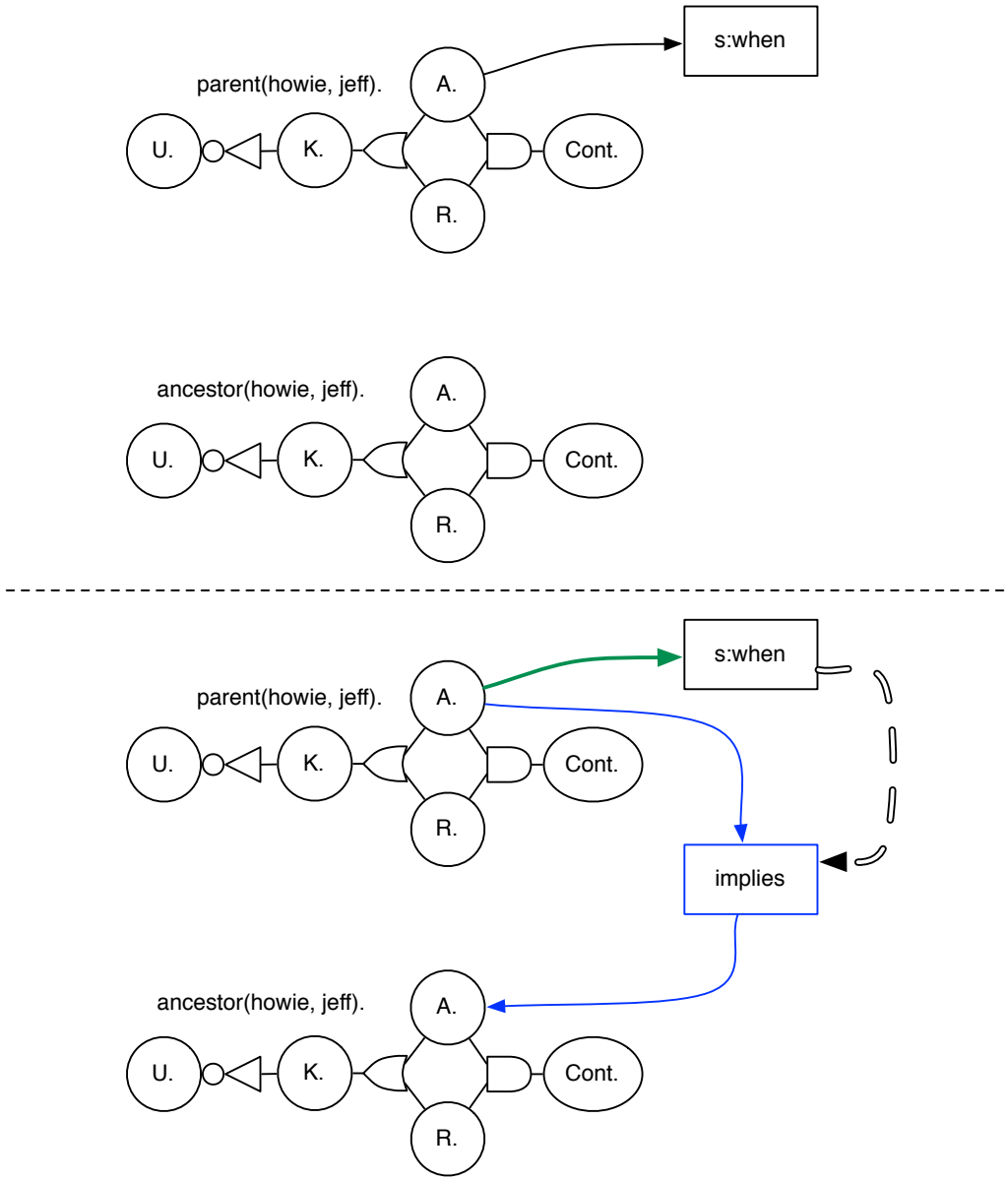


Figure 4-5: Lazily attaching a rule using the `s:when` propagator, which builds the connection between the acceptedness of parent(howie, jeff) and ancestor(howie, jeff) only when the parent relationship is accepted.

```
(define (prove-ancestry-transitively)
  (find-proposition-matching '(ancestor ?a ?b) '()
    (lambda (prop environment)
      (s:when (accepted prop-1)
        (find-proposition-matching '(ancestor ?b ?c) environment
          (lambda (prop-2 environment)
            (s:when (accepted prop-2)
              (let ((ancestor-proposition (instantiate-proposition
                                           '(ancestor ?a ?c)
                                           environment)))
                (accept ancestor-proposition
                        (list 'prove-ancestry-transitively
                              (get-binding '?a environment)
                              (get-binding '?b environment)
                              (get-binding '?c environment))
                        (list (accepted prop-1)
                              (accepted prop-2)))))))))))
```

Figure 4-6: Lazily proving ancestry through transitivity. Even though propositions matching `(ancestor ?a ?b)` and `(ancestor ?b ?c)` might exist, this function will wait until both propositions are actually accepted before accepting the consequent `(ancestor ?a ?c)`, as the `s:when` function will lazily evaluate its body only when the contents of the cell in its first argument (i.e. *accepted* belief state of the given proposition) is true.

```
(define (prove-ancestry-transitively)
  (rule ((a (accepted (a-prop '(ancestor ?a ?b))))
         (b (accepted (a-prop '(ancestor ?b ?c)))))
    (accept (the-prop '(ancestor ?a ?c))
            (list 'prove-ancestry-transitively a b)
            (list a b))))
```

Figure 4-7: A simple syntax for lazily proving ancestry through transitivity. This code effectively expands to the code given in Figure 4-6

4.4 Simplifying the Code

Because of the complexity of rules like that of Figure 4-6, the remainder of this thesis will make use of a simplified syntax given in Figure 4-7. This syntax expands into the syntax of Figure 4-6 through a macro expansion.

The syntax is designed to be easy to understand. In short, the `rule` keyword behaves much as the Scheme `let` keyword, and consists of a list of variable assignments and a body. The list of variable assignments assigns the various cells of a belief state (e.g. `accepted`, `unknown`) to variables, with the `a-prop` keyword effectively acting as a generator which returns propositions matching the specified proposition pattern.

Propositions are matched in order, with the environment for each matching proposition being used to match subsequent proposition (effectively finding each proposition in order, using subsequent nested `find-proposition-matching` functions, each taking, as an argument, the environment returned by the previous proposition). As a result, all sets of propositions matching the list are discovered.

The body of the `rule` will only execute conditionally upon the truth of all belief states in the list of variable assignments, much like the body of the `s:when` propagator. Thus, the connection of the acceptance or rejection of a proposition specified in the body will only occur so long as all of the conditions in the list are true. Furthermore, as each proposition in the variables is found in order, as soon as one of the proposition cells is no longer true, work will cease.

The `accept` inside the body of the rule behaves identically to the function used above, except that `the-prop` acts to expand the proposition using the environment implicitly defined by the enclosing matched variables. That is, `the-prop` returns the proposition that would have been defined by `instantiate-proposition`, given as the two arguments the same pattern and the environment returned by the final proposition in the list of variables of the `rule`. There is also no need to explicitly obtain the accepted state of the matching propositions as those accepted states are automatically stored in the named variables. These variables may then be reused in the body of the `rules`.

Thus, in short, the `rule` keyword effectively defines a rule with the list of belief states of propositions which must all be believed before executing the body in which work is done.

It is worth noting that, like in AMORD [5], it is possible to conceive of *consequent rules* which are only active so long as the parent rule is active. Since the body of a `rule` is simply code which is evaluated upon matching the antecedent of the rule, by placing a `rule` within a `rule` (together with, or in place of, an `accept` function), it is possible to make a secondary rule for which the corresponding propagators are created and connected only when the antecedent beliefs of the *parent* rule are met. This conditional construction of rules gives rule-designers additional control over when rules are instantiated in the first place, providing finer-grain control than a lack of nested rules would provide.

4.5 Controlling for Unknownness

Despite this improvement, there remains yet another inefficiency. While propositions will now only be connected if a parent relationship is accepted, such a problem solver is still useless in certain cases. Proving only 150 million ancestor relations is better than many times that, but it's unlikely that we would care to determine every ancestor relationship. In practice, such an "ancestor finder" would want to focus only on Jeff's ancestry rather than every possible ancestor in the United States.

The previous insight regarding the lazy construction of the propagator network is a crucial one in solving this smaller problem, as we may make use of belief states other than mere acceptance to help *control* the process of problem solving. Rather than creating the relationship between parent and ancestor when we *accept* a parent relationship, why not simply extend the lazy rule mechanism to effectively activate and deactivate its "search for ancestors" based on the goal of proving Jeff's ancestry?

Figure 4-8 (example code in Figure 4-9) depicts how such a network might operate. Both a basic desire to know *and* a lack of knowledge combine to serve as the input to a `s:when` propagator, so that the construction of the conditional network in Figure 4-5

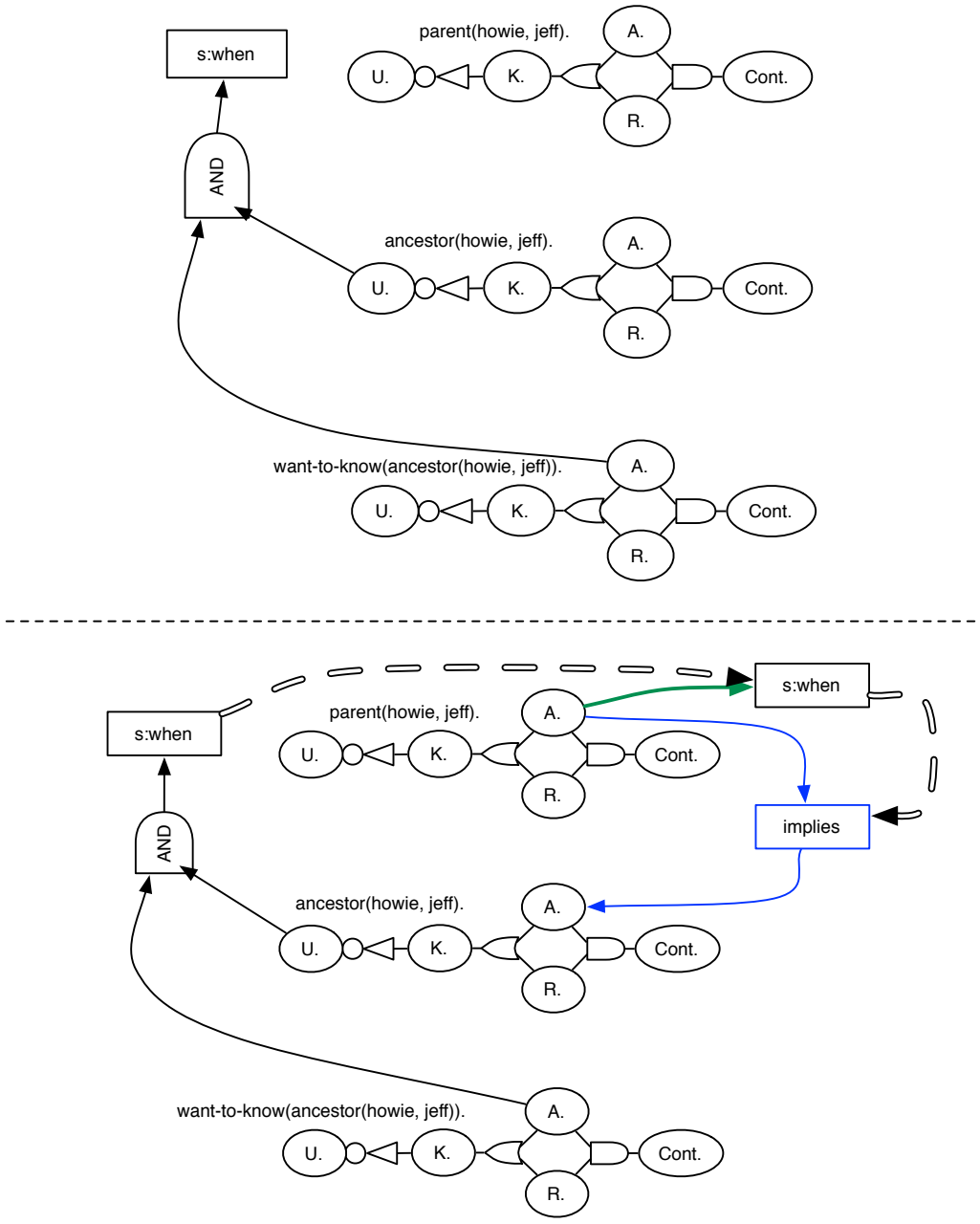


Figure 4-8: Controlling the search for ancestors only when such a search is needed. Only when `(want-to-know (ancestor ?a ?b))` is accepted and the corresponding `(ancestor ?a ?b)` proposition is unknown will a proof be attempted (bottom). If the ancestry proposition is known (i.e. accepted as true or rejected as false) for any reason, work to prove ancestry will cease.

```
(define (prove-ancestry-by-parenthood)
  (rule ((k (accepted (a-prop '(want-to-know (ancestor ?a ?b)))))
         (u (unknown (a-prop '(ancestor ?a ?b)))))
    (rule ((p (accepted (a-prop '(parent ?a ?b)))))
      (accept (the-prop '(ancestor ?a ?b))
              (list 'prove-ancestry-by-parenthood k u p)
              (list p)))))
```

Figure 4-9: Controlling the search for ancestors only when such a search is needed. This code corresponds to the behavior in Figure 4-8.

will not even occur unless there is a need to know the ancestry relationship and the ancestry relationship's unknown belief state is true. Once it is true (bottom), the secondary conditional network is created.

A crucial difference between Figure 4-8 and Figure 4-5, however, is that as the `s:when` in the former is conditioned on the ignorance of the ancestor relationship, once the ancestry is known, the `s:when` is turned off. If the contents of the `s:when` actually listen for a number of possible candidate matches and construct the connection for each one (e.g. in the transitive ancestry case, where there may need to be a number of different ancestry relationships that must be built), this work may be turned off when the ancestry relationship in question has been proved. In short, as soon as we know that there is (or is not) such ancestry, we automatically cease doing additional work to prove it!

Such a system admittedly bears some superficial similarities to the Belief-Desire-Intention model of rational agents [23]; beliefs, combined with the desires and intentions expressed by `want-to-know` propositions, work together to effect work, activate appropriate rules, and do work. But here, we do not propose that all propositional systems *must* adhere to a BDI-like model, but merely demonstrate that the propositional system *allows* for the construction of a BDI-like modality of rational actions and choices. The propositional system is actually far more flexible and may be used to do more than build simple BDI agents. Indeed, the power of the propositional system goes farther and may be used to solve far more complex problems that may even require justifications of the choices made.

Chapter 5

Building Justifications

While a problem solver capable of answering any kind of complex problem is indeed more valuable than one which cannot, the answers it produces are only as reliable as they can be proven to be correct. Mathematical and theoretical approaches may be used to derive the correctness of a given algorithm, but such approaches cannot necessarily account for incorrect inputs or assumptions which may produce erroneous answers. In addition, many practical problems may have a requirement that solutions be capable of being *audited* to confirm correctness of inputs and of the accuracy of outputs for quality control or legal reasons.

Tracing the process by which an answer was constructed has many benefits beyond auditing, accountability, and improving an end-user's confidence in generated answers. Analysis of answer generation may also be used to automate the diagnosis and repair of unexpected behavior, to help optimize future problem solving performance when answering similar questions, or even to help model risks involved in solving problems.

Examining *justifications* for answers may even prove useful in testing hypotheses by identifying which premises contributed to the generation of contradictions. Such a feature may have a practical impact. For example, analysis of the orbit of Mercury using purely Newtonian mechanics predicts the precession of its perihelion, but in 1859, Le Verrier uncovered a conflict between the expected value of this precession given Newtonian mechanics and the actual value resulting from measurements of Mercury's observed position over time. A number of alternate hypotheses to explain

the discrepancy were proposed, but each hypothesis had defects when predicting other astrophysical phenomena which led them to be discarded. Only the replacement of the premises of Newtonian mechanics with those modified by Einstein's theory of general relativity resolved the 43″ per century discrepancy in Mercury's precession while agreeing with other physical phenomena.[1] In practice, then, the "traces of the results" of these alternative theories and hypotheses were analyzed to identify flaws and propose new theories until a resolution was achieved.

Given all these benefits to the production and use of traces of execution, a problem solving system which is capable of providing *justifications* for its answers is more valuable and more useful than one which cannot. Thus, we must ask if it is possible to extend the powerful problem solving capabilities of the propositional system to provide such justifications. If so, the propositional system becomes more valuable for the reasons mentioned above. The remainder of this chapter focuses on how such a mechanism may be built with minimal overhead, using the existing structure of the underlying propagator system to construct such justifications.

5.1 What is a Justification?

Before we can evaluate whether or not the propositional reasoning system can be modified to support the creation of meaningful justifications for its answers, we must first define exactly what is meant by the word *justification*.

A *justification* is, effectively, a story of how a particular state came to exist, that is, the sequence of events that, taken as a whole, caused a particular action, product, or belief to exist. Justifications are typically minimal in extent. For the purposes of this thesis, a justification does not include incorrect decisions and actions which were irrelevant to the production of a given state except in so far as they may have impacted the sequence and timing of the events that are *relevant* to the state being justified.

[1] A comprehensive history of the discovery of, and various solutions to, the problem of Mercury's precessing perihelion, up to and including its resolution due to general relativity, is given in [24].

In short, a *justification* is a description of the causal tree of some state, including both proximal and distal causes, which, if considered as a partially-ordered directed graph from cause to effect, would be sufficient to explain the evolution and production of a given state.

Justifications are related to the concept of document provenance, which consists of "the record of actions taken on [a] particular document over its lifetime," [8] and data provenance, a description of how a particular piece of data came to be and arrived in a given database [3].

Finally, it is worth keeping in mind that we should differentiate justifications from the concept of *dependencies* or premises, discussed previously in Section 3.3, which describe the ultimate *supports* for something. While the former tells us how we got somewhere ("why-provenance" in [3]), the latter tells us only those facts or sources that we depended on to get there ("where-provenance" in the same).

5.2 The Suppes Formalism

So how can we characterize a justification of a particular belief produced by a propositional problem solver? As demonstrated in Chapter 4, the beliefs held by propositional problem solvers evolve over time based on the logical connections (such as implication relationships) between different belief states. What is a natural formalism for justifications which describe this evolution?

Since we speak of logical relationships, a naïve answer would be to say that our justifications should take the form of logical proofs. This is not an unreasonable stance to take, given that all of the examples shown so far involve logical implications, conjunctions, and disjunctions. But what if more complex propagators are used to solve a problem? Not all problems are best formulated in terms of a Boolean algebra. For example, numerical solutions may be more amenable to mathematical algebraic proofs, and it is quite possible that the relationships between belief states as expressed by the propagators that connect them may not be purely Boolean or algebraic in nature. Even so, the generic form of a mathematical proof seems quite reasonable to

{1}	(1) $\mathsf{C} \rightarrow (\mathsf{D} \rightarrow \mathsf{B})$	P
{2}	(2) $-\mathsf{G} \vee \mathsf{C}$	P
{3}	(3) D	P
{4}	(4) G	P
{2, 4}	(5) C	2, 4 T
{1, 2, 4}	(6) $\mathsf{D} \rightarrow \mathsf{B}$	1, 5 T
{1, 2, 3, 4}	(7) B	3, 6 T
{1, 2, 3}	(8) $\mathsf{G} \rightarrow \mathsf{B}$	4, 7 C.P.

Figure 5-1: Example 2 from Chapter 2 of Suppes's *Introduction to Logic*, in which Suppes proves the following (symbolic annotations mine): "If the Cards are third (C), then if Dodgers are second (D) the Braves will be fourth (B). Either the Giants will not be first ($-\mathsf{G}$) or the Cards will be third. In fact, the Dodgers will be second. Therefore, if the Giants are first, then the Braves will be fourth."

consider, as long as we expand the definition of an operation to include any general-purpose propagator.

One wrinkle complicates the adoption of a simple proof mechanism. As mentioned in the previous chapter, navigation of the the answer space (e.g. backtracking) necessarily requires the correct management of the set of *dependencies* which are trusted at any given point in time. Indeed, it should be possible not only to trace the "why-provenance" but also the "where-provenance" for any given belief in our system, so that the proper derivation of a belief may be well understood and demonstrated.

While traditional proofs, such as those typically taught in basic geometry and algebra classes point only to the *processes* by which a derivation or manipulation of data occurs, the proof formulation proposed and utilized by Patrick Suppes [28] captures not only the nature of a derivation, but also the *dependencies* upon which these conclusions are based. An example of such a proof may be seen in Figure 5-1, Example 2 from Chapter 2 of Suppes's *Introduction to Logic*, which demonstrates the application of premises (P), tautological implication (T), and conditional proof (C.P.) to prove the existence of an implication relating to placement in a baseball wild card race.

In the proof, Suppes tracks not only the premises upon which each subsequent statement depends (left column), but also the reasons for derivation (right column).

As such, Suppes's proof formalism demonstrates that it is possible not only to show (8) $\mathsf{G} \rightarrow \mathsf{B}$, but also that it is directly derived from (4) G and (7) B by conditional proof, and depends on the continued acceptance of the premises (1) $\mathsf{C} \rightarrow (\mathsf{D} \rightarrow \mathsf{B})$, (2) $-\mathsf{G} \vee \mathsf{C}$, and (3) D.

Suppes's example also demonstrates the difference between dependencies and justifications. Although the premise (4) G is an integral part of the proof of (8), the conclusion does not actually depend on it being true because implications may be derived using conditional proofs without dependence on the antecedent.

5.3 Building Justifications

Given the power of Suppes's formalism in managing both justifications and dependencies jointly, can we integrate Suppes's formalism into the propositional reasoning system in such a way that we may gain the power of justification generation seamlessly without any additional syntactic changes to our propositional reasoning? I will demonstrate that this is indeed possible, by making use of the innate graph structure of the propagator network.

Since a propagator network may be envisioned as a graph of cells and propagators, it is possible to impose a data structure above and beyond those used in the basic propagator "publish-subscribe" model to represent the abstract structure of propagators as a navigable directed graph. The MIT/Scheme implementation of propagators includes such an abstraction by collecting cells into groups called *diagrams*.

Diagrams may be defined recursively, as diagrams may contain one or more parts which are also diagrams, with cells acting as simple diagrams consisting of no additional *parts*. More complex diagrams may be made from combining cells and diagrams which contain them as component parts of the complex diagram.

As a result, the diagram structure of a propagator network is effectively a pyramidal, directed graph (as in Figure 5-2) which effectively maps cells on the computational level to the "components" to which they belong at decreasing levels of abstraction. A given diagram (or cell) may belong to any number of parent "clubs" (i.e. the parent

diagrams of which the diagram is a part). As a result, the given pyramid of diagrams is not a tree structure but a directed, acyclic graph. When a propagator is created, a diagram is automatically constructed at the first abstraction layer above basic cells. This diagram collects those cells which it reads and writes as "parts" of the initial abstract diagram.

As multiple propagators may claim a cell as a "part" of the parent diagram, certain lower-level diagrams may be easily identified as "boundaries" between different diagrams at a higher abstraction level. For example, cell C in Figure 5-2 belongs not only to `temperature-converter`, but also `weather-model`, and as such may be seen as a cell which is on the *boundary* of the two diagrams.

By annotating the *part* relationships at the time the network is created, it is possible to determine whether cell C is an input to A, an input to B, or both, as the propagators themselves may be analyzed to determine whether or not their implementation will read a cell or write a cell (or both). As such, promises to "only read a cell" may be interpreted as cells which are input to the diagram, while promises to "write only" would similarly be interpreted as output.

Justifications for the content of a cell may thus be constructed by climbing the "diagram pyramid" to an appropriate abstraction level, identifying "input" cells of the given diagram, and then querying those cells for their contents and working to find their inputs in turn. In this manner, a list of antecedent cells and values may be crawled at a given abstraction level using an appropriate graph-traversal algorithm, and the values of those cells may be collected, along with the dependencies and any other information stored in the TMS in each cell.

For example, given the first part of the example in Figure 5-2, suppose that we wish to generate a justification for the value at cell C. If we asked for a justification (Figure 5-3), we would determine which diagrams C was an output for at a given level (say level 2). For these diagrams ($x \times 5/9$), we would then determine the inputs (B) and include those in the justification. Likewise, we would then identify those diagrams for which B is an output *at the same diagram level*, 2 ($x - 32$), and the inputs to those diagrams (A). The full justification would thus include the value and

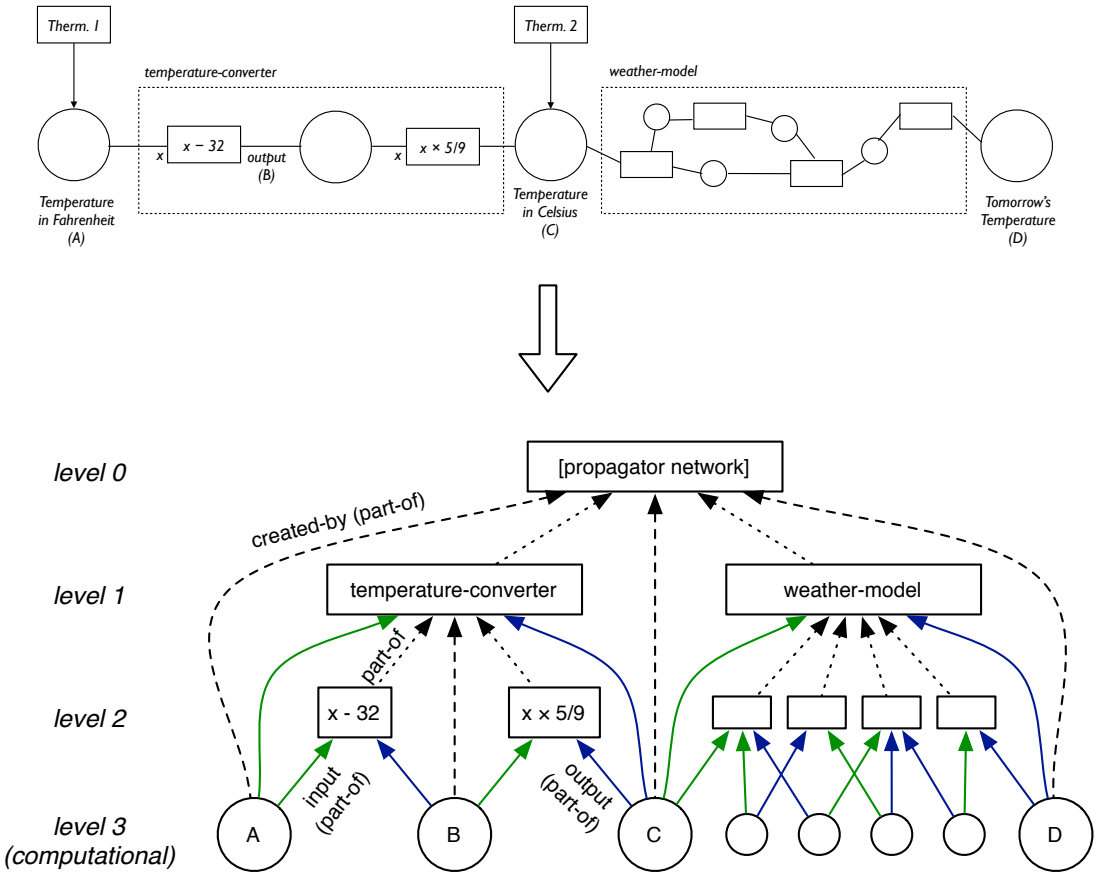


Figure 5-2: A complex propagator network maps to a pyramidal semantic structure. More abstract operations are represented at higher levels. At the top, three cells, A, C, and D, are connected by two compound propagators, `temperature-converter` and `weather-model`. Both compound propagators expand into partial propagator networks. Below, the same network is laid out in terms of its diagrammatic structure. Level 3 represents the concrete cells of the network, while higher levels represent the propagators and compound propagators that connect them. All arrows represent "part-of" relationships between the cells or diagrams at one level and the diagrams that belong to a higher level. Colors and line patterns represent annotations which may be made (e.g. a promise that a given cell is used only as input or as output by a propagator).

supports of each cell crawled in this process, A, B and C, by querying the truth maintenance system stored in each cell. These TMSs are necessarily informed by the propagators that sent updates to them.

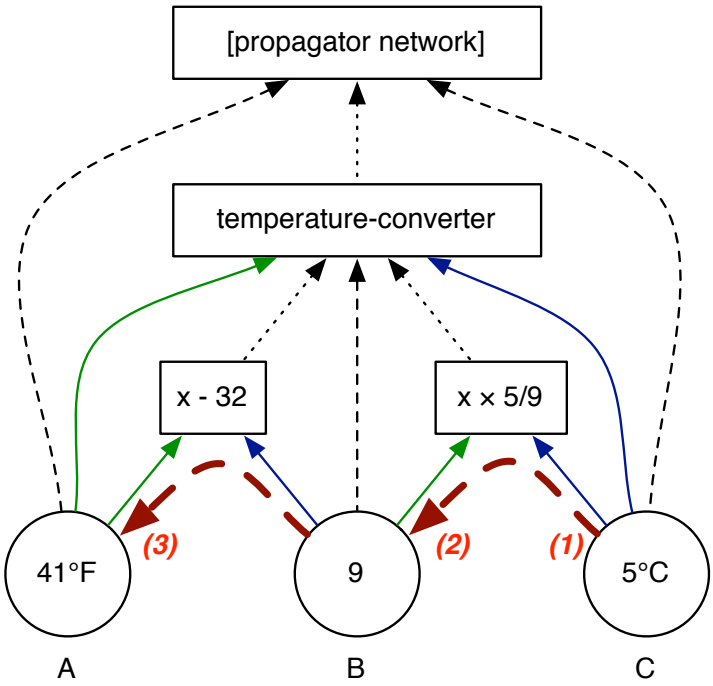
While the example in Figure 5-3 describes a system in which each propagator has only one input, the recursive algorithm for generating justifications is equally applicable to diagrams with an arbitrary number of inputs. Instead of examining each input sequentially, the inputs could be crawled and included in the justification in a breadth-first manner.

5.4 Simplifying Justifications

One significant issue encountered when constructing a justification is the choice of an appropriate abstraction level for the justification. For example, when converting a temperature between degrees Fahrenheit and degrees Celsius, as in Chapter 3, it may be useful to detail each step in the conversion to demonstrate that the conversion is working correctly. On the other hand, if the conversion is part of a much more complex system, such as a numerical weather model, the precise trace of each mathematical step to convert contributes to excessive verbosity of the justification. It may suffice to state that the temperature was converted, leaving the exact mathematical operations implicit.

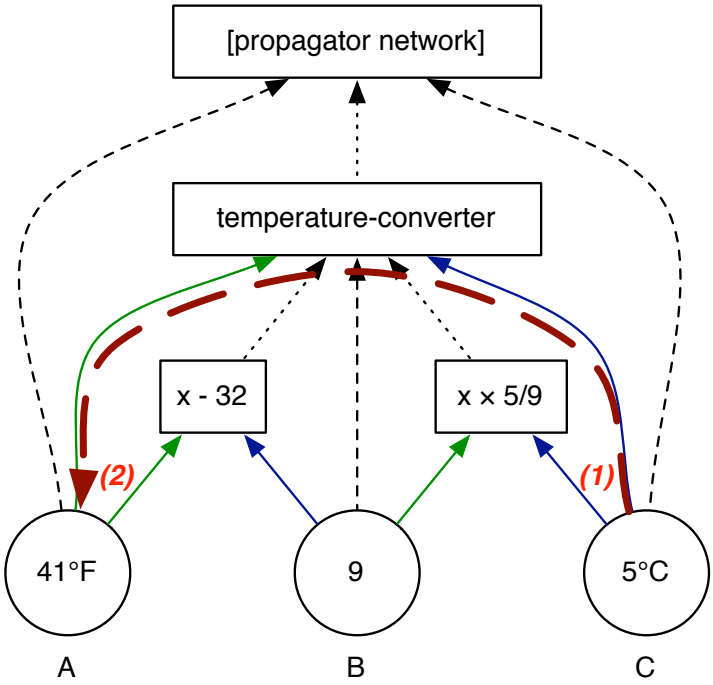
Given this, it would be incredibly useful if we did not necessarily commit to a level of detail until the time the justification was generated. That is, our choice of mechanism for producing justifications should allow us to obtain multiple levels of detail of a justification.

Fortunately, this struggle between the simple and the complex is easily resolved in the diagram model. The pyramidal structure of the diagram-part relationship allows for the creation of any number of abstraction layers between the lowest "computational" units and the highest level which represents the problem solver itself. As a result, it is possible to choose to display a given level of justification (simple to complex) by simply selecting an appropriate abstraction level to generate justifications.




```
((C has-value 5    ; (1)
   by (* B (/ 5 9))
   with-premises thermometer)
 (B has-value 9    ; (2)
   by (- A 32)
   with-premises thermometer)
 (A has-value 41   ; (3)
   by (thermometer-reading)
   with-premises thermometer))
```

Figure 5-3: A justification may be generated by recursively crawling diagrams at a given level from outputs to inputs.




```
((C has-value 5    ; (1)
    by (temperature-converter A)
    with-premises thermometer)
 (A has-value 41   ; (2)
    by (thermometer-reading)
    with-premises thermometer))
```

Figure 5-4: A simpler justification may be generated by crawling diagrams at higher levels.

A simple justification may be generated by simply climbing to one of the highest abstraction levels. For example, instead of generating a detailed justification from level 2 as in Figure 5-3, we may instead generate a simpler justification by crawling the diagrams at level 1, as shown in Figure 5-4.

Chapter 6

Propositions in Practice

So far, I have demonstrated the principles of the propositional reasoning system, but I have yet to demonstrate its viability in solving real world problems such as the insurance company scenario presented at the beginning of this thesis. In this chapter, I intend to walk through the implementation of a propositional problem solver capable of addressing that very problem. In so doing, I hope that the principles which I have laid out in previous chapters will be made obvious.

6.1 The Problem

As stated previously, consider an insurance underwriter, Sally, who works for the insurance company Aintno. As she reviews the file of a prospective customer, Danny, she must make a decision on his eligibility for insurance based on his risk and activity. As part of his application, he has included information about his Facebook and Flickr social networking accounts, among the many other details in his application.

Sally feeds this application to an assisting proposition-based problem solver which must then decide whether or not Danny is eligible for insurance based on a set of scoring criteria (Table 6.1). If Danny's aggregate score is less than 2, Danny is eligible for insurance. If it is greater than 3, Danny is ineligible for insurance. If it is between 2 and 3, more work may be done to prove Danny's eligibility (including a determination of Danny's eating habits).

Criteria	Contribution
Customer eats healthy food	-2
Customer eating habits are unknown	-1
Customer eats unhealthy food	+2
Customer is a skydiver	+3
Customer being a skydiver is unknown	+0.5
Customer is not a skydiver	-0.1
Customer is a rock climber	+2
Customer being a rock climber is unknown	+0.25
Customer is not a rock climber	-0.1
Customer is a scuba diver	+1
Customer being a scuba diver is unknown	+0.1
Customer is not a scuba diver	-0.1
Customer rides a motorcycle	+2
Customer riding motorcycles is unknown	+0.2
Customer does not ride motorcycles	-0.1

Table 6.1: Contributions to risk score based on personal behaviors

This problem solver has the ability to crawl the social network information Danny has provided and use the conclusions drawn from that information (posts, photos, etc.) to justify arguments in favor of a given risk score. Indeed, such justifications are a necessary part of the problem solver; once a score has been generated, Sally must include a justification for the score when she submits her final decision with respect to granting or denying eligibility to Danny.

6.2 Bootstrapping the System: Propositions

For the purposes of this example, I will assume that the problem of extracting beliefs from images and free-form text is solved and that such algorithms are capable of populating belief states of appropriate propositions which are indexed in a database. This is a reasonable assumption to make; algorithms for entity and sentiment extraction from text, and object and gesture recognition in images have already been deployed in active production systems. Additionally, the methods used to solve these problems are not directly relevant to the problem of actually calculating a risk score addressed in this example.

Proposition	Current Belief	Evidence
Danny engages in skydiving	accepted	Facebook post
Danny engages in motorcycling	accepted	Flickr photo
Danny engages in SCUBA diving	rejected	Danny's forms
Danny engages in rock-climbing	unknown	Danny's forms
Hal's Hot Dogs is a restaurant	accepted	Hal's Hot Dogs website
Hal's Hot Dogs primarily sells hot dogs	accepted	Hal's Hot Dogs website
Hot dogs are food	accepted	common knowledge
Hot dogs are unhealthy	accepted	FDA
Danny works at Hal's Hot Dogs	accepted	Danny's forms
Danny likes Hal's Hot Dogs	accepted	Danny's forms

Table 6.2: Propositions which might be believed about Danny

Thus, the propositions that might be discovered by such a system may be simply expressed by populating the belief state of such propositions which have been discovered. Some of these propositions are given in Table 6.2.

As stated in Chapter 3, we may express propositions by virtue of creating appropriate propagator networks for each proposition. The cells created as part of the network may then be populated with an appropriate value associated with each belief state based on the dependencies (i.e. the posts, photos, etc.) from which the belief was established. In my MIT/Scheme problem solver, a proposition may be constructed using the `proposition` function, which is passed the pattern of the proposition to be asserted as its argument (for example, `(proposition '(Danny engages-in skydiving))`).

But how do we populate our initial beliefs? To accomplish this, we "tell" a particular belief state its value. The `tell!` function automatically constructs an update message which is sent to the cell specified as the first argument of the function. The value to be stored is given as the second argument, and all all subsequent arguments are the dependencies for that value. Thus, we might state that the proposition `(Danny engages-in skydiving)` is accepted by way of a Facebook post through the function call `(tell! (accepted (proposition '(Danny engages-in skydiving))) #t 'Facebook)`. In this way, we update the *accepted* belief cell with the true value (`#t`) and a dependency named `Facebook`.

```
(define (true! pattern source)
  (tell! (accepted (proposition pattern)) #t source))

(define (unknown! pattern source)
  (tell! (unknown (proposition pattern)) #t source))

(define (false! pattern source)
  (tell! (rejected (proposition pattern)) #t source))
```

Figure 6-1: The `tell!` function can be simplified to refer to only the `true`, `false` and `unknown` cells.

In practice, we may simplify telling true, false, and unknown values by making appropriate functions which expand to the verbose `tell!` syntax for these common cases, as in Figure 6-1. With these functions in hand, it becomes relatively simple to express basic beliefs, given in Figure 6-2. Thanks to the dependence on the `proposition` function, the `true!`, `unknown!` and `false!` functions automatically create the propagator network for each proposition if it does not already exist. Otherwise, the proposition is retrieved from a database.

6.3 Bootstrapping the System: Rules

Establishing the basic belief states is only half of the solution, however. The core of any problem solver are the rules and algorithms which allow it to actually draw conclusions, and I have not yet demonstrated how such rules might be created. To do so, we turn to Chapter 4, which demonstrated the construction of a number of problem solving rule components.

The scoring rules in Table 6.1 may be readily expressed in terms of simple rules, using the `rule` syntax introduced at the end of the chapter, as each appropriate belief state would be matched to establish the contribution to risk. Some of these rules can be seen in Figure 6-3, where a simple pattern match against an appropriate belief state is sufficient to accept a "contribution to risk" of an appropriate size.

For example, in the case of the risk if a customer is a sky-diver, the proposition `(contribution risk ?subject 3.0 skydiver)` is accepted for a given subject

```
;;; Ground facts about Danny

(true! '(Danny engages-in skydiving) 'Facebook)

(true! '(Danny engages-in motorcycling) 'Flickr)

(false! '(Danny engages-in scubadiving) 'Danny)

(unknown! '(Danny engages-in rockclimbing) 'Danny)


;;; His eating habits

(true! '(hals-hotdogs is-a restaurant) 'Hal)

(true! '(hals-hotdogs primarily-serves hotdogs) 'Hal)

(true! '(hotdogs is-a food) 'common-knowledge)

(true! '(hotdogs is unhealthy) 'FDA)

(true! '(Danny works-at hals-hotdogs) 'Danny)

(true! '(Danny likes hals-hotdogs) 'Danny)


;;; Ground facts about AINTNO

(true! '(risk-accept-threshold AINTNO 2) 'aintno-1)

(true! '(risk-reject-threshold AINTNO 3) 'aintno-2)
```

Figure 6-2: Beliefs captured by the risk-scoring system based on information gleaned from Danny's Facebook and Flickr accounts, as well as his insurance application (the last labeled with a dependency of `'Danny`). In addition, two risk score thresholds are included as belief states.

```
(rule ((s (accepted (a-prop '(?subject engages-in skydiving)))))
  (accept (the-prop '(contribution risk ?subject 3.0 skydiver))
          (list 'risk-estimate 'skydiving)
          (list s)))

(rule ((s (unknown (a-prop '(?subject engages-in skydiving)))))
  (accept (the-prop '(contribution risk ?subject 0.5 skydiver))
          (list 'risk-estimate 'skydiving)
          (list s)))

(rule ((s (rejected (a-prop '(?subject engages-in skydiving)))))
  (accept (the-prop '(contribution risk ?subject -0.1 skydiver))
          (list 'risk-estimate 'skydiving)
          (list s)))
```

Figure 6-3: Rules establishing contributions to risk with respect to whether an individual is a sky-diver

only when the proposition `(?subject engages-in skydiving)` is accepted. Such a contribution is accepted contingent on the acceptance of the `engages-in` statement, but its acceptance also labeled with the *why-provenance* from its second argument. Thus, the acceptance of the risk contribution is due to `(list 'risk-estimate 'skydiving)`, that is, based on a risk-estimate with respect to skydiving.

Rules can, of course, be more complex than these simple sky-diving rules. Aintno's policy may state, for example, that the fact of whether or not Danny eats unhealthy food is only relevant if there is a need to determine it (i.e. if there is sufficient evidence to reject Danny on another basis, there is no reason to investigate Danny's eating habits). In such a case, we can turn to the more complex `want-to-know` formula expressed at the end of Chapter 4. Such an expression might resemble that in Figure 6-4.

But a rule such as that in Figure 6-4 demands a way to establish *the need to determine* whether Danny eats unhealthy food. For that, we might wish to condition the specific "need to know" on a more general belief that "information is lacking". Such a condition would necessarily connect the general directive to find information to score Danny's risk with the specific ways in which the information may be found

```
(rule ((req (accepted (a-prop '(does ?subject eat unhealthy-food)))))

  (rule ((e (accepted (a-prop '(?subject eats ?food))))
         (f (accepted (a-prop '(?food is unhealthy)))))
    (accept (the-prop '(?subject eats unhealthy-food))
            (list 'common-sense 'food)
            (list e f)))

  (rule ((l (accepted (a-prop '(?subject likes ?thing))))
         (t (accepted (a-prop '(?thing is-a food)))))
    (accept (the-prop '(?subject eats ?thing))
            (list 'preference 'food)
            (list t l)))

  (rule ((p (accepted (a-prop '(?subject likes ?place))))
         (r (accepted (a-prop '(?place is-a restaurant)))))
    (accept (the-prop '(?subject eats-at ?place))
            (list 'likes 'restaurant)
            (list p r)))

  (rule ((p (accepted (a-prop '(?subject works-at ?place))))
         (r (accepted (a-prop '(?place is-a restaurant)))))
    (accept (the-prop '(?subject eats-at ?place))
            (list 'works-at 'restaurant)
            (list p r)))

  (rule ((p (accepted (a-prop '(?subject eats-at ?place))))
         (r (accepted (a-prop '(?place is-a restaurant))))
         (s (accepted (a-prop '(?place primarily-serves ?thing))))
         (f (accepted (a-prop '(?thing is-a food)))))
    (accept (the-prop '(?subject eats ?thing))
            (list 'eating-at 'restaurant)
            (list p r s f)))
  )
```

Figure 6-4: Rules that help determine whether or not Danny eats unhealthy food. Such rules might only ever be active (i.e. work might only ever be done to prove that he eats unhealthy food) if there is not sufficient proof to render Danny ineligible for insurance.

```
(rule ((i
        (accepted
         (a-prop '(?company needs-more-information ?subject)))))
  ;; Eating unhealthy food is questionable, but not worth looking at
  ;; unless not enough other information to determine eligibility.
  (accept (the-prop '(does ?subject eat unhealthy-food))
          (list 'digging-deeper)
          (list i))
  (rule ((e (accepted (a-prop '(?subject eats unhealthy-food)))))
    (accept (the-prop
              '(contribution risk ?subject +2 eats-unhealthy-food))
            (list 'risk-estimate 'unhealthy-food)
            (list e)))
  (rule ((e (unknown (a-prop '(?subject eats unhealthy-food)))))
    (accept (the-prop
              '(contribution risk ?subject -1 unknown-food-habits))
            (list 'risk-estimate 'unhealthy-food)
            (list e)))
  (rule ((e (rejected (a-prop '(?subject eats unhealthy-food)))))
    (accept (the-prop
              '(contribution risk ?subject -2 eats healthy-food))
            (list 'risk-estimate 'unhealthy-food)
            (list e))))
```

Figure 6-5: There only exists a need to score the risk from eating unhealthy food if there is not enough evidence to accept or reject an individual's insurance on other grounds. Thus, as long as there is a need for more information, the need to determine whether an individual eats unhealthy food (and appropriate risk scoring) will be established.

(e.g. asking whether Danny eats unhealthy food). Thus, we may include a rule like that in Figure 6-5, which conditions not only the scoring of unhealthy eating habits, but also the establishment of the "need to know," on a general need for more information.

The expression of such a connection may seem at first glance to be needless pedantry which may lead to an infinite regress of determining whether we "need to know that we need to know". Though an implementer will necessarily need to tread with care to determine what is reasonable intent, this first step is actually quite reasonable. As mentioned before, if the cost of discovering whether or not Danny

eats unhealthy food is expensive (it may take considerable processing and time to interpret and extract features and intents from Facebook posts), we likely will only want to do the work to determine whether or not Danny eats unhealthy food if we cannot prove with certainty that Danny is, or is not, eligible for insurance. Thus, the connection between the general "lack of information" and the specific "need to know" Danny's eating habits is actually quite indicative of a need for the level of inherent control which propositional reasoning offers.

With all these rules in place, there remains only one component necessary to actually properly rate risk: the mechanism to accumulate risk scores itself. Though I have thus far demonstrated that problems may be resolved by the rule mechanism described in Chapter 4, such partial propagator networks are not the only networks which may be useful in solving problems. The propagator network model permits a vast number of network constructions which may be used to solve problems in a wide variety of ways. Accumulation may be accomplished by one such "alternative" network structure.

Unlike the basic scoring mechanisms described above, accumulation is a complex operation which cannot be simply implemented with rules alone, due to the need to "undo" an accumulation when premises change. For example, while an assumption of ignorance regarding Danny's sky-diving hobby (or lack thereof) contributes a risk factor of 0.5 to overall accumulated risk, if, at a later time, he is found to indeed engage in sky-diving, the contribution to the overall accumulated risk score must be changed from 0.5 to 3. This change will necessarily alter the overall risk score (such as by increasing it from 3.5 to 5). Furthermore, the overall risk score must then depend on the sources of information which contribute to the *acceptance* of Danny's sky-diving which were not previously present, as the new score of 5 is only true, in part, due to that information.

In order to implement accumulation, an alternative partial network may be constructed like that presented in Figure 6-6. The basic premise of the accumulator structure is that, at any stage in the life of an accumulator, there is an implicit assumption that all contributions to the value of the accumulator are known. Thus, the

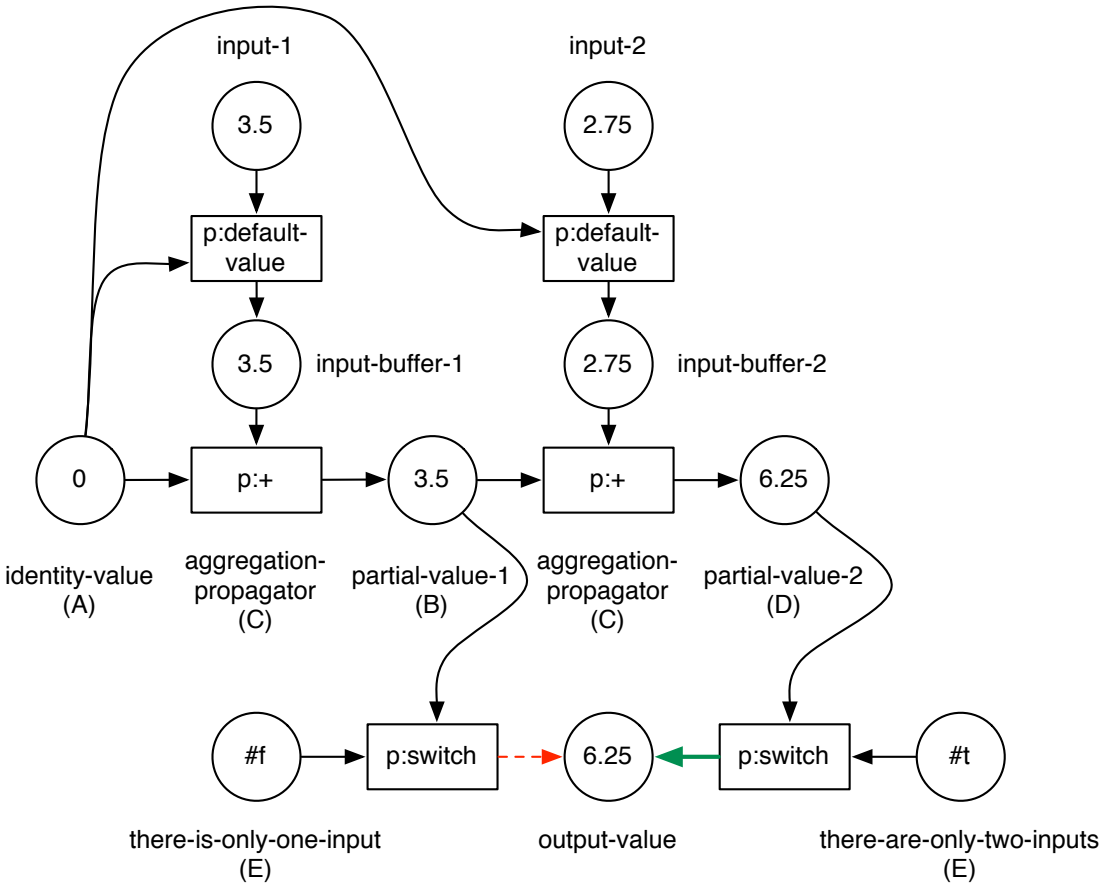


Figure 6-6: A propagator network which accumulates two values by addition (`p:+`). Partial values slowly accumulate until a `p:switch` connects a partial value (partial-value-2) to the final value based on the assumption that there are no more inputs to the accumulator.

basic network of an accumulator constructs a "final" partial accumulated value (D) by chaining partially accumulated values (B) starting with an identity value (A) to additional inputs to the accumulator by way of a known binary operator (such as the `p:+` addition operator) (C). This "final" partial accumulated value is then connected to an output cell through a "switch" propagator which only propagates the "final" value to the output as long as the assumption that there are no more contributions to the accumulator is true (E).

Figure 6-7 illustrates how the number of inputs to an accumulator may increase. When a new contribution is identified, the chain of accumulator inputs may be extended by constructing additional propagators and cells and appending them to the chain of partial accumulations. Following this, the assumption that all contributions were known may be kicked out, disconnecting the old "final" partial accumulation (D) from the output cell. Then, the newly constructed "final" partial accumulation (F) may be attached to the output cell to effectively update the value with the new contribution.

Figure 6-8 illustrates how removing contributions may be accomplished through the use of the `p:default-value` propagator and the buffer cell which sit between the contributing cell and the partially accumulated value. The `p:default-value` propagator provides a default value for an output cell whenever the primary input (i.e. the contributing cell) contains no contribution whatsoever. Thus, when a contribution is "removed", one need only remove the support for the value in the input cell. With no supported value in its primary input, the `p:default-value` propagator will instead connect the alternate input (i.e. the identity cell) to the input buffer cell used as input to the partial accumulation. As a result, only the identity value supports the partial accumulation at that point, so that the actual contribution at the newly unsupported partially accumulated value is effectively eliminated thanks to the use of the identity value for the operation, such as adding 0 or multiplying by 1.

We may, of course, simplify this partial network using a compound propagator, named `p:accumulator`. This propagator constructs an accumulation network and is invoked with five arguments. The first argument is a simple symbolic prefix used

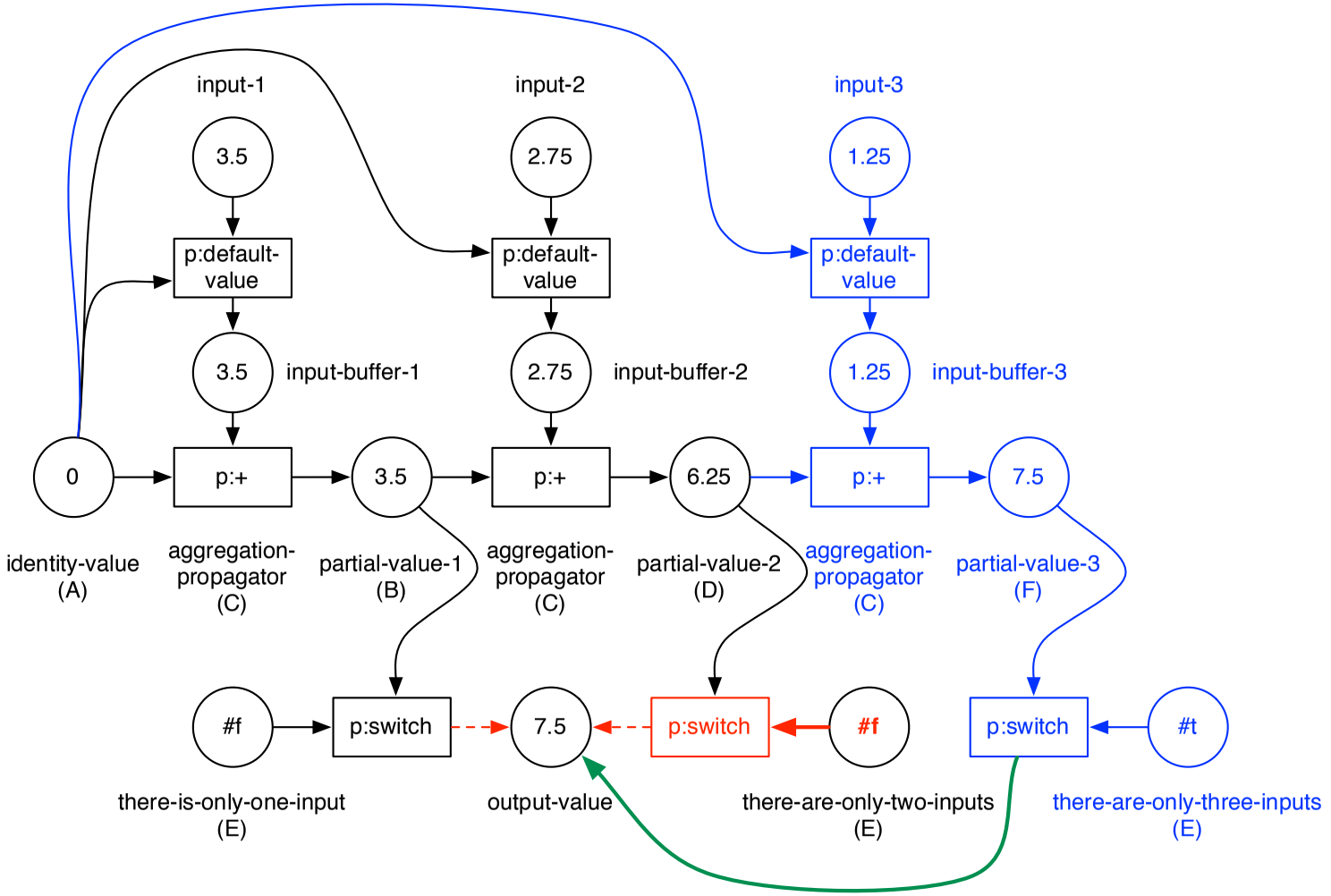


Figure 6-7: Adding a new input to a propagator-based accumulator. Here, input-3 is added as an input, and the assumption that there are only two inputs is kicked out, so that the final value contains only the value of partial-value-3.

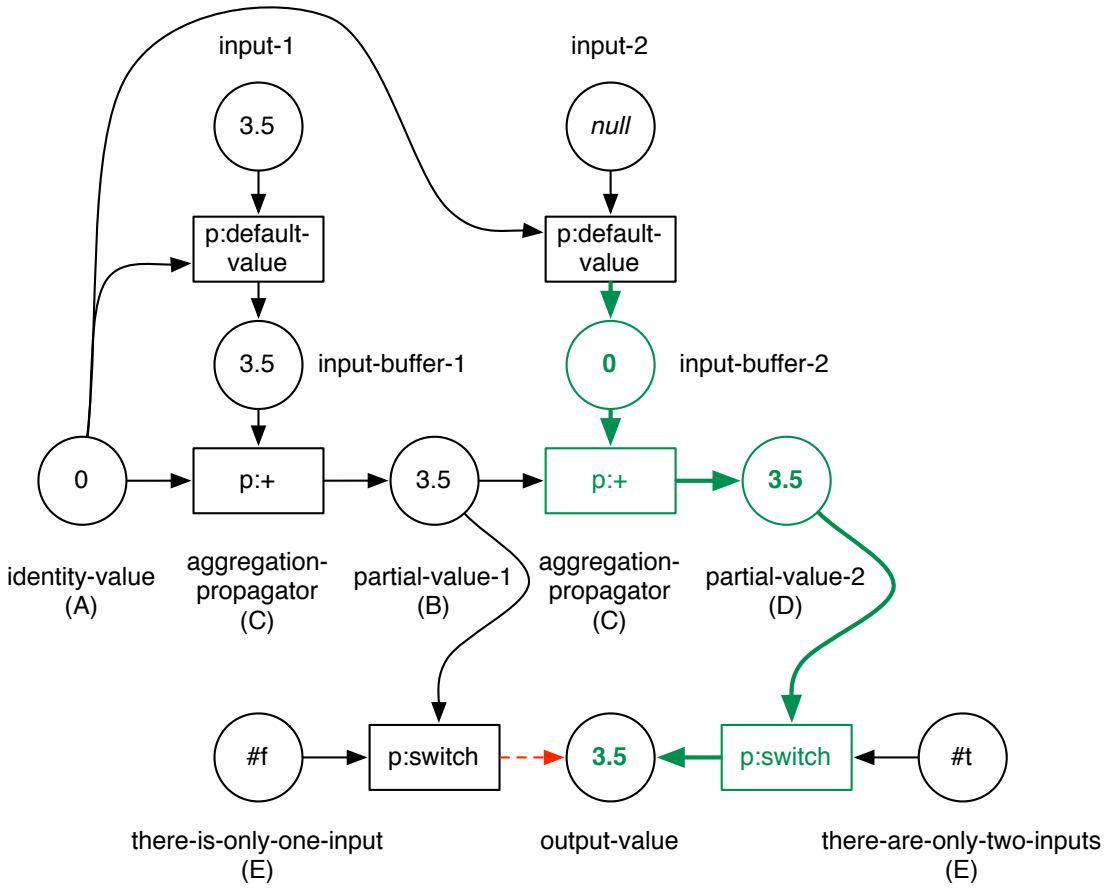


Figure 6-8: Removing a contribution from an input to the propagator-based accumulator in Figure 6-6. Here, input-2 is removed (its value becomes `*the-nothing*`, depicted here as *null*). This causes the `p:default-value` propagator to which it is connected to instead propagate the value from the identity cell (0) to the input of the partial-accumulation.

for labeling and debugging. The second is the name of the binary operator used in accumulation (for example, `p:+` or `p:*`) and the third argument is the identity value for that operation. The fourth argument is the cell to which the output of the accumulator will be connected, and the fifth and final argument is a cell which will contain the propagator which may be invoked to add new contributions to the accumulator.

With this propagator in hand, the accumulation of risk may be handled using the code in Figure 6-9, which constructs an accumulator whenever a request for accumulation `(please-accumulate ?type ?subject ?requester)` is found and accepted. Then, for every proposition representing an accumulator contribution `(contribution ?type ?subject ?num ?reason)` which is accepted, a cell representing the value of that contribution is created, and its value is supported by the acceptance of the contribution proposition. This value is then connected to the accumulator using the `p:add-new!` propagator, at which point its contribution will be reflected in the output cell of the accumulator.

Finally, the output value of the accumulator is mapped to a proposition of the form `(accumulator ?type ?subject ?requester ,accumulation)`, allowing other propositional pattern-matching to make use of the value of the accumulation.

With such a mechanism, Aintno need only express a need to accumulate risk, which may be done using rules and tests for risk thresholds, as in Figure 6-10, in which the need to determine insurability establishes a need for accumulation `(please-accumulate risk ?subject ?company)`, and checks whether a given accumulated risk `(accumulator risk ?subject ?company ?risk-accumulator)` is below the acceptance threshold or above the rejection threshold before accepting or rejecting on that basis.

6.4 The System in Operation

With the rules and propositions about Danny in place, we need only establish a need to determine Danny's insurability as an accepted proposition by way of the assertion

```
(rule ((r
        (accepted
         (a-prop '(please-accumulate ?type ?subject ?requester)))))
  (let* ((name (symbol (pattern-value ?type)
                       ":accumulator-of:"
                       (pattern-value ?subject)
                       ":for:"
                       (pattern-value ?requester)))
         (contribution-counter (make-counter)))
    (let-cells (accumulation p:add-new!)
      (p:accumulator name p:+ 0 accumulation p:add-new!)
      (rule ((c
              (accepted
               (a-prop '(contribution ?type ?subject ?num ?reason)))))
        (let* ((cname
                (symbol name ":contribution:" (contribution-counter)))
               (contrib (make-named-cell cname)))
          (p:switch c (pattern-value ?num) contrib)
          (p:add-new! contrib)))

      (accept (the-prop
               `(accumulator ?type ?subject ?requester ,accumulation))
              (list 'accumulator-result)
              (list r)))))
```

Figure 6-9: Accumulation of values is only done when there is a need to do such accumulation (`(please-accumulate ?type ?subject ?requester)` is accepted). When there is such a need, only accepted numeric contributions of the form `(contribution ?type ?subject ?num ?reason)` matching the request for accumulation are used.

```
(rule ((req
        (accepted
         (a-prop
          '(please-determine-if ?subject is-insurable-by ?company)))))
  (accept (the-prop '(please-accumulate risk ?subject ?company))
          (list 'risk-evaluation)
          (list req))
  (rule ((result
          (accepted
           (a-prop
            '(accumulator risk ?subject ?company ?risk-accumulator))))
         (ath
          (accepted
           (a-prop
            '(risk-accept-threshold ?company ?accept-threshold))))
         (rth
          (accepted
           (a-prop
            '(risk-reject-threshold ?company ?reject-threshold)))))

    (let ((accumulated-risk-cell (pattern-value ?risk-accumulator))
          (accept-threshold (pattern-value ?accept-threshold))
          (reject-threshold (pattern-value ?reject-threshold)))

      (p:> accept-threshold accumulated-risk-cell
           (accepted (the-prop
                      '(?subject insurance-issued-by ?company))))
      (p:> accumulated-risk-cell reject-threshold
           (rejected (the-prop
                      '(?subject insurance-issued-by ?company))))
      (p:and (e:<= accept-threshold accumulated-risk-cell)
             (e:<= accumulated-risk-cell reject-threshold)
             (accepted
              (the-prop
               '(?company needs-more-information ?subject)))))))
```

Figure 6-10: Only when there is a need to determine whether a potential customer is insurable will the accumulation of risk be undertaken. More information is needed to determine Danny's insurability `(accepted (the-prop '(?company needs-more-information ?subject)))` if the risk is between the accept and reject thresholds.

`(true! '(please-determine-if Danny is-insurable-by AINTNO) 'ian)`. The above rules and propagator mechanisms will then work to solve the problem of whether or not Danny is insurable.

Based on this need to determine insurability, the code in Figure 6-10 establishes a need to accumulate risk. At this point, only the simple rule expressions like those in Figure 6-3 will be active to establish contributions to risk based on Danny's engagement in risky hobbies as according to the beliefs established in Figure 6-2.

On their own, these facts are enough to deny Danny insurance as he engages in skydiving and motorcycling, does not engage in SCUBA diving, and nothing is known about whether or not Danny is a rock climber. This belief may be demonstrated readily, as the rejected state of `(Danny insurance-issued-by AINTNO)` is held as true. We may inquire as to why this is the case using the `explain` function `(explain (rejected (proposition '(Danny insurance-issued-by AINTNO))) 1)`, the results of which are given in Figure 6-11. At this description level (0), we find that the rejection of `(danny insurance-issued-by aintno)` is supported (`has-value #t`) based on the fact that the accumulated value in `(out)` (5.15) is greater than the rejection threshold (3). The justification also explains that this accumulated risk value is supported by the premises gathered from `facebook`, `flickr`, `danny`, and `risk:accumulator-of:danny:for:aintno:premise4`. This last premise is the assumption that there are no more contributions to Danny's risk.

We can, of course, ask for more detail by digging into a more detailed justification level (2), which gives the answer in Figure 6-12. This justification demonstrates that contributions to the risk came from the addition of the risks from rock-climbing, scuba-diving, motorcycling, and sky-diving.

But what if Sally remembers that Facebook posts should not be used to support rejections of insurability? In that case, we need only kick out the premises supported by Facebook `(retract! 'Facebook)`, which will drop the accumulated risk within the bounds of issuance ($risk < 2$) and rejection ($risk > 3$) of insurance. In this range, the system now accepts the need to look for more information to determine Danny's insurability `(?company needs-more-information ?subject)`. As a result

```
((((rejected (danny insurance-issued-by aintno)))
  has-value
  #t
  by
  ((>:p) (out) (3))
  with-premises
  facebook
  flickr
  danny
  risk:accumulator-of:danny:for:aintno:premise4)
 ((out) has-value
        5.15
        by
        ((p:accumulator) (0))
        with-premises
        facebook
        flickr
        danny
        risk:accumulator-of:danny:for:aintno:premise4))
```

Figure 6-11: A simple justification, generated by the function call `(explain (rejected (proposition '(Danny insurance-issued-by AINTNO))) 1)`

of accepting this proposition, the system now may accept the need to prove whether or not Danny eats unhealthy food (Figure 6-5) and will attempt to find proof of that proposition (Figure 6-4).

With such rules on unhealthy food now active, the system is again able to reject Danny's insurance application, based on the fact that he eats unhealthy food (specifically, he works at `hals-hotdogs`, a restaurant, implying that he eats their food, `hotdogs`, which, according to the FDA, are unhealthy). This justification can, of course, be obtained using the `explain` function at a suitably deep justification level (2), as in Figure 6-13.

```
((((rejected (danny insurance-issued-by aintno)))
  has-value #t
  by ((>:p) (risk:accumulator-of:danny:for:aintno) (3))
  with-premises facebook flickr danny
  risk:accumulator-of:danny:for:aintno:premise4)
 ((risk:accumulator-of:danny:for:aintno)
  has-value 5.15
  by (((accumulator (p:+)))
      (risk:accumulator-of:danny:for:aintno:contribution:4)
      (risk:accumulator-of:danny:for:aintno:contribution:3)
      (risk:accumulator-of:danny:for:aintno:contribution:2)
      (risk:accumulator-of:danny:for:aintno:contribution:1)
      (risk:accumulator-of:danny:for:aintno:zero))
  with-premises facebook flickr danny
  risk:accumulator-of:danny:for:aintno:premise4)
 ((risk:accumulator-of:danny:for:aintno:contribution:4)
  has-value .25
  by ((switch:p)
      ((accepted (contribution risk danny .25 rockclimber)))
      (.25))
  with-premises danny)
 (((accepted (contribution risk danny .25 rockclimber)))
  has-value #t
  by ((((risk-estimate) (rockclimbing)))
      ((unknown (danny engages-in rockclimbing))))
  with-premises danny)
 (((unknown (danny engages-in rockclimbing)))
  has-value #t
  by (user)
  with-premises danny)
 ((risk:accumulator-of:danny:for:aintno:contribution:3)
  has-value -.1
  by ((switch:p)
      ((accepted (contribution risk danny -.1 scubadiver)))
      (-.1))
  with-premises danny)
 (((accepted (contribution risk danny -.1 scubadiver)))
  has-value #t
  by ((((risk-estimate) (scubadiving)))
      ((rejected (danny engages-in scubadiving))))
  with-premises danny)
 (((rejected (danny engages-in scubadiving)))
  has-value #t
  by (user)
  with-premises danny)
 ((risk:accumulator-of:danny:for:aintno:contribution:2)
  has-value 2.
  by ((switch:p)
      ((accepted (contribution risk danny 2. motorcycler)))
      (2.))
  with-premises flickr)
 (((accepted (contribution risk danny 2. motorcycler)))
  has-value #t
  by ((((risk-estimate) (motorcycling)))
      ((accepted (danny engages-in motorcycling))))
  with-premises flickr)
 (((accepted (danny engages-in motorcycling)))
  has-value #t
  by (user)
  with-premises flickr)
 ((risk:accumulator-of:danny:for:aintno:contribution:1)
  has-value 3.
  by ((switch:p)
      ((accepted (contribution risk danny 3. skydiver)))
      (3.))
  with-premises facebook)
 (((accepted (contribution risk danny 3. skydiver)))
  has-value #t
  by ((((risk-estimate) (skydiving)))
      ((accepted (danny engages-in skydiving))))
  with-premises facebook)
 (((accepted (danny engages-in skydiving)))
  has-value #t
  by (user)
  with-premises facebook)
 ((risk:accumulator-of:danny:for:aintno:zero) has-value 0))
```

Figure 6-12: Danny's hobbies may be used as a basis to reject his insurance, a fact which comes out of the justification generated by the `explain` function, given here and in Appendix A.

```
((((rejected (danny insurance-issued-by aintno)))
  has-value #t
  by ((>:p) (risk:accumulator-of:danny:for:aintno) (3))
  with-premises nothing:1 flickr danny fda common-knowledge hal
                parachutingassociation
                risk:accumulator-of:danny:for:aintno:premise6)
 ((risk:accumulator-of:danny:for:aintno)
  has-value 4.050000000000001
  by (((accumulator (p:+)))
      (risk:accumulator-of:danny:for:aintno:contribution:6)
      (risk:accumulator-of:danny:for:aintno:contribution:5)
      (risk:accumulator-of:danny:for:aintno:contribution:4)
      (risk:accumulator-of:danny:for:aintno:contribution:3)
      (risk:accumulator-of:danny:for:aintno:contribution:2)
      (risk:accumulator-of:danny:for:aintno:contribution:1)
      (risk:accumulator-of:danny:for:aintno:zero))
  with-premises
  (hypothetical 460 nothing:1 in #[entity 461] bool)
  flickr danny fda common-knowledge hal parachutingassociation
  risk:accumulator-of:danny:for:aintno:premise6)
 ((risk:accumulator-of:danny:for:aintno:contribution:6)
  has-value -.1
  by ((switch:p)
      ((accepted (contribution risk danny -.1 skydiver)))
      (-.1))
  with-premises parachutingassociation)
 (((accepted (contribution risk danny -.1 skydiver)))
  has-value #t
  by ((((risk-estimate) (skydiving)))
      ((rejected (danny engages-in skydiving))))
  with-premises parachutingassociation)
 (((rejected (danny engages-in skydiving)))
  has-value #t
  by (user)
  with-premises parachutingassociation)
 ((risk:accumulator-of:danny:for:aintno:contribution:5)
  has-value 2
  by ((switch:p)
      ((accepted (contribution risk danny 2 eats-unhealthy-food)))
      (2))
  with-premises fda common-knowledge hal danny)
 (((accepted (contribution risk danny 2 eats-unhealthy-food)))
  has-value #t
  by ((((risk-estimate) (unhealthy-food)))
      ((accepted (danny eats unhealthy-food))))
  with-premises fda common-knowledge hal danny)
 (((accepted (danny eats unhealthy-food)))
  has-value #t
  by ((((common-sense) (food)))
      ((& ((accepted (danny eats hotdogs)))
          ((accepted (hotdogs is unhealthy))))))
  with-premises fda common-knowledge hal danny)
 (((accepted (hotdogs is unhealthy)))
  has-value #t
  by (user)
  with-premises fda)
 (((accepted (danny eats hotdogs)))
  has-value #t
  by ((((eating-at) (restaurant)))
      ((& ((accepted (danny eats-at hals-hotdogs)))
          ((accepted (hals-hotdogs is-a restaurant)))
          ((accepted (hals-hotdogs primarily-serves hotdogs)))
          ((accepted (hotdogs is-a food))))))
  with-premises common-knowledge hal danny)
 (((accepted (hotdogs is-a food)))
  has-value #t
  by (user)
  with-premises common-knowledge)
 (((accepted (hals-hotdogs primarily-serves hotdogs)))
  has-value #t
  by (user)
  with-premises hal)
 (((accepted (danny eats-at hals-hotdogs)))
  has-value #t
  by ((((works-at) (restaurant)))
      ((& ((accepted (danny works-at hals-hotdogs)))
          ((accepted (hals-hotdogs is-a restaurant))))))
  with-premises hal danny)
 (((accepted (hals-hotdogs is-a restaurant)))
  has-value #t
  by (user)
  with-premises hal)
 (((accepted (danny works-at hals-hotdogs)))
  has-value #t
  by (user)
  with-premises danny)
 ((risk:accumulator-of:danny:for:aintno:contribution:4)
  has-value .25
  by ((switch:p)
      ((accepted (contribution risk danny .25 rockclimber)))
      (.25))
  with-premises danny)
 (((accepted (contribution risk danny .25 rockclimber)))
  has-value #t
  by ((((risk-estimate) (rockclimbing)))
      ((unknown (danny engages-in rockclimbing))))
  with-premises danny)
 (((unknown (danny engages-in rockclimbing)))
  has-value #t
  by (user)
  with-premises danny)
 ((risk:accumulator-of:danny:for:aintno:contribution:3)
  has-value -.1
  by ((switch:p)
      ((accepted (contribution risk danny -.1 scubadiver)))
      (-.1))
  with-premises danny)
 (((accepted (contribution risk danny -.1 scubadiver)))
  has-value #t
  by ((((risk-estimate) (scubadiving)))
      ((rejected (danny engages-in scubadiving))))
  with-premises danny)
 (((rejected (danny engages-in scubadiving)))
  has-value #t
  by (user)
  with-premises danny)
 ((risk:accumulator-of:danny:for:aintno:contribution:2)
  has-value 2.
  by ((switch:p)
      ((accepted (contribution risk danny 2. motorcycler)))
      (2.))
  with-premises flickr)
 (((accepted (contribution risk danny 2. motorcycler)))
  has-value #t
  by ((((risk-estimate) (motorcycling)))
      ((accepted (danny engages-in motorcycling))))
  with-premises flickr)
 (((accepted (danny engages-in motorcycling)))
  has-value #t
  by (user)
  with-premises flickr)
 ((risk:accumulator-of:danny:for:aintno:contribution:1)
  has #(*the-nothing*))
 ((risk:accumulator-of:danny:for:aintno:zero) has-value 0))
```

Figure 6-13: Danny's employment at Hal's Hot Dogs may be used against him if Aintno is unable to prove that he should be insured or not, a fact visible when using the `explain` function. This text is also provided in Appendix A.

Chapter 7

Beyond Propositions

Though I have demonstrated the viability of propositions in constructing and integrating control with knowledge in problem solvers, it is difficult to prove that such propositional reasoning is *viable* for all kinds of problem solving strategies. Certainly the computational power of the underlying propagator network is equivalent to that of a universal Turing machine[1], but the mere ability to emulate any Turing machine does not necessarily mean that propositional approaches are appropriate to all problems.

The true feasibility and flexibility of propositional problem solving can only be uncovered in the course of future work spent exploring and implementing problem solvers, using the propositional model, to solve different kinds of problems. Such work will presumably uncover any weaknesses or flaws in the current system, whether such flaws are due to current implementations of propagator networks or a fundamental incompatibility of propositional problem solving with certain kinds of problems. In the process of developing this propositional problem solving system, however, I have identified a number of directions for future work to better determine the flaws of and improve propositional problem solving. I will discuss these directions in detail in the remainder of this chapter.

[1]Consider a single cell representing the state of the tape of a Turing machine, with propagators attached to it which read the state stored in the cell and update it appropriately.

7.1 Supporting Multiple Worldviews

A crucial assumption made in the implementation of propositions described in this thesis is the assumption that each proposition is associated with *one* set of the five belief states. That is, it is assumed that a belief state for a proposition will only ever be believed or disbelieved; there is to be no superposition of belief.

Within the context of the "agency" of a single problem solving process, this is not necessarily a problem. In the examples provided in previous chapters, the inability to superpose belief and disbelief is a non-issue, because the goal of each (partial) problem solver was to resolve discordant beliefs so as to draw an appropriate conclusion.

A problem arises when there is a need to reconcile or work with multiple beliefs simultaneously, such as when the beliefs of two distinct agents must be coordinated. In such a situation it is unclear how two agents' beliefs should be kept separate. Because each cell contains a simple TMS which contains a single supported truth value, it is impossible to interpret a single *accepted* cell's value in the light of both agents simultaneously. Certainly, one could maintain separate sets of premises and evaluate the single cell in light of each premise set to obtain the different truth values associated with beliefs of the same proposition, but such a system hardly allows us to connect or synthesize the two beliefs in any meaningful fashion. Premise sets are "external" to the propagator network. The contents of cells in a propagator network may only be evaluated with respect to one premise set at a time, so there is no way to properly depend on two (potentially conflicting) premise sets.

There are several possibilities for addressing this need to represent what amounts to two distinct "world views." We could choose to create additional belief states associated with a proposition, each representing an agent and its associated basic belief (e.g. "john accepts", "john rejects", "jane accepts", "jane rejects"). Such a mechanism would allow for a meaningful grouping of all beliefs relating to a single proposition within multiple world views. It may, however, complicate addressing and naming of individual belief states associated with a world view when resolution of each cell name requires knowledge of the agent who holds that state.

Alternatively, we could represent each agent's view of the proposition as a distinct proposition. This semantic reification of the proposition (e.g. `(jane holds (jack parent ben))`) complicates instead the naming of a proposition, leaving the problem of addressing the proposition, itself, unsolved. One benefit of this approach is that rules may be easily constructed to relate two different agents' views (e.g. believes(jane, X) $\rightarrow$ believes(john, X)). That said, it is not immediately obvious which approach is better, given that both approaches construct five additional belief state cells for each agent which holds beliefs relating to a single proposition. As a result, regardless of the approach chosen to representing beliefs in multiple world views with respect to a single proposition, the number of cells for both approaches scales linearly in terms of the number of agents, and both are unwieldy in their naming mechanisms.

An ideal solution would be to surface the underlying premise sets as cells on their own and have propagators which make use of the premise-containing cells in any computation (e.g. by propagating a value from an accepted belief state of one proposition to another). Such a system has the advantage that each proposition retains only five belief state cells, but comes with a distinct disadvantage: as propagators necessarily do computation based on the contents of a cell, care must be taken to *always* compute with respect to at least one set of premises. If this is not done, beliefs may propagate to belief states in other propositions even though the relationships between propositions themselves may differ between the worldviews.

For example, Jane and John may both accept the proposition `(jack parent ben)`, but they may instead disagree on the implication that such an acceptance necessarily means that one must accept `(jack ancestor jim)`. If this is done blindly without consideration of a premise set, one may find that both Jane and John accept the latter proposition, even though they may not both accept the justification generated to get there. Furthermore, as backtracking is currently handled when merging the contents of a cell with the contents of an update message, merge operations would need to be able to directly manipulate the contents of relevant premise-cells, complicating the design of the merge operations used in a propagator network.

7.2 Proof by Contradiction

A practical example of the need for multiple world-views can be observed in the principle of *reductio ad absurdum*, mentioned previously in this thesis. Unlike the *modus ponens* strategy applied to generate the rule framework described in the previous chapters, proofs by contradiction cannot actually be implemented using a simple propagator network in which each cell contains the reasons for supporting a particular belief state. Effective application of proving a contradiction requires the creation of a separate worldview in which the a belief *contrary* to what is to be proven is held true. If we seek to prove that `(jack parent ben)` is to be accepted, we must create a worldview which is identical to the current one except that `(jack parent ben)` is rejected, and we must determine whether such rejection implies a contradiction.

Proving a belief by contradiction not only depends on the ability to represent two worldviews (the original worldview in which we wish to assert the proven belief, as well as the hypothetical worldview in which we attempt to show a contradiction), which is difficult for reasons discussed above, but it also depends on the ability to *connect* the worldviews. That is, the existence of a contradiction in the "contrary" worldview in which `(jack parent ben)` is rejected necessarily creates an effect in the original worldview by supporting the belief that we wished to prove in the first place, `(jack parent ben)`.

While such a system might indeed be made possible through the extension of propagator networks to permit multiple worldviews, *reductio ad absurdum* is not a method of argumentation that is currently feasible with the existing propagator network infrastructure. It thus suggests that there is room for improvement of the underlying propagator network substrate.

7.3 Alternate Belief States

Although I have laid out five belief states for each proposition in this thesis, it is possible that more belief states may be useful or necessary for other computations.

A number of modal qualifications exist for each proposition which may be better served as a belief state, including the *want-to-know* modality expressed in the Aintno example throughout the thesis.

Other modalities may also prove useful, including a modality of indifference (if a problem solver need not factor in or care about the belief in a proposition) and a modality of irrelevance (e.g. if a problem solver wishes to prove that some proposition is true or false regardless of the belief in another proposition). It is unclear if these modalities represent true alternate belief states in addition to the states of acceptance, rejection, contradiction, knowledge, and ignorance. The *want-to-know* modality in particular appears to be a proposition of its own, based on the fact that we may consider a need-to-know itself to be accepted, rejected, or something of which we are ignorant.

On a more fundamental level, however, the boundary between a true belief and a modality deserving of its own proposition is unclear, as any belief may plausibly be reified. For example, we might talk about the rejected belief of a proposition `(accepted (jack parent ben))`, itself describing a belief in the acceptance of `(jack parent ben)`.

Regardless of the viability of various modalities of a proposition, however, the set of five beliefs proposed here should be viewed as a minimal set, rather than a complete set of the beliefs that apply to a given proposition. Additional use cases of propositional problem solving may indeed find that one or more additional belief states are necessary to solve other kinds of problems not detailed in this thesis.

7.4 Probability in Cells

Throughout this thesis, we have discussed beliefs such as acceptance or rejection as if they have firm Boolean values associated with them (i.e. it is either 100% true that we accept a proposition, 100% false, or, of course, 100% null). However, this should not be seen as a judgment made with regard to probabilistic approaches to reasoning and problem solving. Indeed, it is possible to imagine that, rather than

storing a strict Boolean value in a TMS that a probability value is stored instead. These probabilities could then be used in turn to support other propositions on the basis of various probabilistic models and theories such as Bayesian inference.

In order to implement such a system, work must be done to establish a sound data structure to replace the basic TMS and to develop appropriate propagators so as to permit probabilistic calculations. For example, it may be the case that a probabilistic model of belief should connect acceptance and rejection in such a way that a belief that there is to be no contradiction necessarily forces the probability of acceptance to be 1 minus the probability of rejection. Similarly, the data structure stored in cells must be capable of handling probabilities appropriately, although it is possible that a TMS may be sufficient.

7.5 Contributions and Conclusions

Despite these two flaws, the concept of propositional reasoning is compelling. By modeling beliefs separately from the structure of the problem solver itself, it is possible to inject complex control into problem solving and to make use of widely different mechanisms for doing so. For example, simply using knowledge of a solution's existence may make it possible to control the nature of and limit the extent of problem solving. This is demonstrated most effectively in the scenario presented in the previous chapter, in which work was not done to prove whether or not Danny ate unhealthy food unless it was impossible to approve or deny Danny's insurance by any other means.

Such a mechanism permits problem solvers to intelligently select goals. It also provides flexibility in problem solving (there are no constraints in how or in what order Danny's risk must be scored) while still allowing for a level of control to effectively account for hidden costs incurred in the process of problem solving (e.g. by restricting the amount of work spent on Danny's eating habits until the benefits outweighed the costs of the work).

The nature of the underlying propagator network substrate of this system affords other valuable benefits, including complex justification generation based on the struc-

ture through which information and beliefs flow. By grounding such justifications in a semantic-rich programming substrate, justifications are obtained at little cost and the semantics of the propagator network can be used to obtain meaningful justifications at multiple levels of detail.

Although work is clearly still needed to establish the practical viability of propositional techniques in problem solving as a whole, including work on the problems of supporting multiple worldviews and adding support for proof by contradiction to the underlying propagator architecture, propositional reasoning appears to be a functional approach to general-purpose problem solving solutions which may better model the flexibility of human thought for use in a dynamic, ever-changing world.

Appendix A

A Sample Session with Propositional Reasoning

The example scenario given in Chapter 6 provided only a handful of examples of how Danny's insurance might be determined. In practice, users may interact with the propositional system through the standard read-eval-print loop of MIT/Scheme. This appendix contains an extended interactive session and demonstrates explanation generation in the Aintno scenario. Input is given on its own line, while expected output is given within multi-line Scheme comments (beginning with `#|` and ending with `|#`).

```
;;; Ground facts about Danny

(true! '(Danny engages-in skydiving) 'Facebook)

(true! '(Danny engages-in motorcycling) 'Flickr)

(false! '(Danny engages-in scubadiving) 'Danny)

(unknown! '(Danny engages-in rockclimbing) 'Danny)


;;; His eating habits

(true! '(hals-hotdogs is-a restaurant) 'Hal)

(true! '(hals-hotdogs primarily-serves hotdogs) 'Hal)

(true! '(hotdogs is-a food) 'common-knowledge)

(true! '(hotdogs is unhealthy) 'FDA)

(true! '(Danny works-at hals-hotdogs) 'Danny)

(true! '(Danny likes hals-hotdogs) 'Danny)


;;; Ground facts about AINTNO

(true! '(risk-accept-threshold AINTNO 2) 'aintno-1)

(true! '(risk-reject-threshold AINTNO 3) 'aintno-2)

(length (db-alist-alist (content *database*)))
;Value: 19


;;; The problem

(true! '(please-determine-if Danny is-insurable-by AINTNO) 'gjs-1)

(length (db-alist-alist (content *database*)))
;Value: 24
```

```
(cpp (explain
      (rejected (proposition '(Danny insurance-issued-by AINTNO))) 1))
#|
((((rejected (danny insurance-issued-by aintno)))
  has-value
  #t
  by
  ((>:p) (out) (3))
  with-premises
  facebook
  flickr
  danny
  risk:accumulator-of:danny:for:aintno:premise4)
 ((out) has-value
        5.15
        by
        ((p:accumulator) (0))
        with-premises
        facebook
        flickr
        danny
        risk:accumulator-of:danny:for:aintno:premise4))
|#

(cpp (explain
      (rejected (proposition '(Danny insurance-issued-by AINTNO))) 2))


#|
((((rejected (danny insurance-issued-by aintno)))
  has-value #t
  by ((>:p) (risk:accumulator-of:danny:for:aintno) (3))
  with-premises facebook flickr danny
  risk:accumulator-of:danny:for:aintno:premise4)
 ((risk:accumulator-of:danny:for:aintno)
  has-value 5.15
  by (((accumulator (p:+)))
      (risk:accumulator-of:danny:for:aintno:contribution:4)
      (risk:accumulator-of:danny:for:aintno:contribution:3)
      (risk:accumulator-of:danny:for:aintno:contribution:2)
      (risk:accumulator-of:danny:for:aintno:contribution:1)
      (risk:accumulator-of:danny:for:aintno:zero))
  with-premises facebook flickr danny
  risk:accumulator-of:danny:for:aintno:premise4)
 ((risk:accumulator-of:danny:for:aintno:contribution:4)
  has-value .25
  by ((switch:p) ((accepted (contribution risk danny .25 rockclimber))) (.25))
  with-premises danny)
 (((accepted (contribution risk danny .25 rockclimber)))
  has-value #t
  by ((((risk-estimate) (rockclimbing)))
      ((unknown (danny engages-in rockclimbing))))
  with-premises danny)
```

```
 (((unknown (danny engages-in rockclimbing)))
  has-value #t
  by (user)
  with-premises danny)
 ((risk:accumulator-of:danny:for:aintno:contribution:3)
  has-value -.1
  by ((switch:p) ((accepted (contribution risk danny -.1 scubadiver))) (-.1))
  with-premises danny)
 (((accepted (contribution risk danny -.1 scubadiver)))
  has-value #t
  by ((((risk-estimate) (scubadiving)))
      ((rejected (danny engages-in scubadiving))))
  with-premises danny)
 (((rejected (danny engages-in scubadiving)))
  has-value #t
  by (user)
  with-premises danny)
 ((risk:accumulator-of:danny:for:aintno:contribution:2)
  has-value 2.
  by ((switch:p) ((accepted (contribution risk danny 2. motorcycler))) (2.))
  with-premises flickr)
 (((accepted (contribution risk danny 2. motorcycler)))
  has-value #t
  by ((((risk-estimate) (motorcycling)))
      ((accepted (danny engages-in motorcycling))))
  with-premises flickr)
 (((accepted (danny engages-in motorcycling)))
  has-value #t
  by (user)
  with-premises flickr)
 ((risk:accumulator-of:danny:for:aintno:contribution:1)
  has-value 3.
  by ((switch:p) ((accepted (contribution risk danny 3. skydiver))) (3.))
  with-premises facebook)
 (((accepted (contribution risk danny 3. skydiver)))
  has-value #t
  by ((((risk-estimate) (skydiving)))
      ((accepted (danny engages-in skydiving))))
  with-premises facebook)
 (((accepted (danny engages-in skydiving)))
  has-value #t
  by (user)
  with-premises facebook)
 ((risk:accumulator-of:danny:for:aintno:zero) has-value 0))
|#
```

```
(cpp (explain
      (rejected (proposition '(Danny insurance-issued-by AINTNO))) 3))
#|
((((rejected (danny insurance-issued-by aintno)))
  has-value #t
  by ((>:p) (risk:accumulator-of:danny:for:aintno) (3))
  with-premises facebook flickr danny
                risk:accumulator-of:danny:for:aintno:premise4)
 ((risk:accumulator-of:danny:for:aintno)
  has-value 5.15
  by ((switch:p) (risk:accumulator-of:danny:for:aintno:construction:4)
                 (risk:accumulator-of:danny:for:aintno:partial-sum:4))
  with-premises facebook flickr danny
                risk:accumulator-of:danny:for:aintno:premise4)
 ((risk:accumulator-of:danny:for:aintno:construction:4)
  has-value #t
  by (user)
  with-premises risk:accumulator-of:danny:for:aintno:premise4)
 ((risk:accumulator-of:danny:for:aintno:partial-sum:4)
  has-value 5.15
  by ((+:p) (risk:accumulator-of:danny:for:aintno:partial-sum:3)
            (risk:accumulator-of:danny:for:aintno:buffer:4))
  with-premises danny flickr facebook)
 ((risk:accumulator-of:danny:for:aintno:partial-sum:3)
  has-value 4.9
  by ((+:p) (risk:accumulator-of:danny:for:aintno:partial-sum:2)
            (risk:accumulator-of:danny:for:aintno:buffer:3))
  with-premises danny flickr facebook)
 ((risk:accumulator-of:danny:for:aintno:partial-sum:2)
  has-value 5.
  by ((+:p) (risk:accumulator-of:danny:for:aintno:partial-sum:1)
            (risk:accumulator-of:danny:for:aintno:buffer:2))
  with-premises flickr facebook)
 ((risk:accumulator-of:danny:for:aintno:partial-sum:1)
  has-value 3.
  by ((+:p) (risk:accumulator-of:danny:for:aintno:zero)
            (risk:accumulator-of:danny:for:aintno:buffer:1))
  with-premises facebook)
 ((risk:accumulator-of:danny:for:aintno:buffer:1)
  has-value 3.
  by ((p:default-value)
      (risk:accumulator-of:danny:for:aintno:zero)
      (risk:accumulator-of:danny:for:aintno:contribution:1))
  with-premises facebook)
 ((risk:accumulator-of:danny:for:aintno:contribution:1)
  has-value 3.
  by ((switch:p) ((accepted (contribution risk danny 3. skydiver))) (3.))
  with-premises facebook)
```

```
(((accepted (contribution risk danny 3. skydiver)))
 has-value #t
 by ((((risk-estimate) (skydiving)))
     ((accepted (danny engages-in skydiving))))
 with-premises facebook)
(((accepted (danny engages-in skydiving)))
 has-value #t
 by (user)
 with-premises facebook)
((risk:accumulator-of:danny:for:aintno:buffer:2)
 has-value 2.
 by ((p:default-value)
     (risk:accumulator-of:danny:for:aintno:zero)
     (risk:accumulator-of:danny:for:aintno:contribution:2))
 with-premises flickr)
((risk:accumulator-of:danny:for:aintno:contribution:2)
 has-value 2.
 by ((switch:p) ((accepted (contribution risk danny 2. motorcycler))) (2.))
 with-premises flickr)
(((accepted (contribution risk danny 2. motorcycler)))
 has-value #t
 by ((((risk-estimate) (motorcycling)))
     ((accepted (danny engages-in motorcycling))))
 with-premises flickr)
(((accepted (danny engages-in motorcycling)))
 has-value #t
 by (user)
 with-premises flickr)
((risk:accumulator-of:danny:for:aintno:buffer:3)
 has-value -.1
 by ((p:default-value)
     (risk:accumulator-of:danny:for:aintno:zero)
     (risk:accumulator-of:danny:for:aintno:contribution:3))
 with-premises danny)
((risk:accumulator-of:danny:for:aintno:contribution:3)
 has-value -.1
 by ((switch:p) ((accepted (contribution risk danny -.1 scubadiver))) (-.1))
 with-premises danny)
(((accepted (contribution risk danny -.1 scubadiver)))
 has-value #t
 by ((((risk-estimate) (scubadiving)))
     ((rejected (danny engages-in scubadiving))))
 with-premises danny)
(((rejected (danny engages-in scubadiving)))
 has-value #t
 by (user)
 with-premises danny)
((risk:accumulator-of:danny:for:aintno:buffer:4)
 has-value .25
 by ((p:default-value) (risk:accumulator-of:danny:for:aintno:zero)
                       (risk:accumulator-of:danny:for:aintno:contribution:4))
 with-premises danny)
((risk:accumulator-of:danny:for:aintno:zero) has-value 0)
```

```
 ((risk:accumulator-of:danny:for:aintno:contribution:4)
  has-value .25
  by ((switch:p) ((accepted (contribution risk danny .25 rockclimber))) (.25))
  with-premises danny)
 (((accepted (contribution risk danny .25 rockclimber)))
  has-value #t
  by ((((risk-estimate) (rockclimbing)))
      ((unknown (danny engages-in rockclimbing))))
  with-premises danny)
 (((unknown (danny engages-in rockclimbing)))
  has-value #t
  by (user)
  with-premises danny))
|#
```

```
(retract! 'Facebook)

(length (db-alist-alist (content *database*)))
;Value: 32

;;; We have other knowledge!
(false! '(Danny engages-in skydiving) 'ParachutingAssociation)

(length (db-alist-alist (content *database*)))
;Value: 33

;;; But Danny is still rejected, because of his eating habits.

(cpp (explain
      (rejected (proposition '(Danny insurance-issued-by AINTNO))) 1))
#|
((((rejected (danny insurance-issued-by aintno)))
  has-value #t
  by ((>:p) (accumulation) (3))
  with-premises nothing:1
                flickr
                danny
                fda
                common-knowledge
                hal
                parachutingassociation
                risk:accumulator-of:danny:for:aintno:premise6)
 ((accumulation)
  has-value 4.050000000000001
  by ((p:accumulator) (0))
  with-premises nothing:1
                flickr
                danny
                fda
                common-knowledge
                hal
                parachutingassociation
                risk:accumulator-of:danny:for:aintno:premise6))
|#
```

```
(cpp (explain
      (rejected (proposition '(Danny insurance-issued-by AINTNO))) 2))
#|
((((rejected (danny insurance-issued-by aintno)))
  has-value #t
  by ((>:p) (risk:accumulator-of:danny:for:aintno) (3))
  with-premises nothing:1 flickr danny fda common-knowledge hal
                parachutingassociation
                risk:accumulator-of:danny:for:aintno:premise6)
 ((risk:accumulator-of:danny:for:aintno)
  has-value 4.050000000000001
  by (((accumulator (p:+)))
      (risk:accumulator-of:danny:for:aintno:contribution:6)
      (risk:accumulator-of:danny:for:aintno:contribution:5)
      (risk:accumulator-of:danny:for:aintno:contribution:4)
      (risk:accumulator-of:danny:for:aintno:contribution:3)
      (risk:accumulator-of:danny:for:aintno:contribution:2)
      (risk:accumulator-of:danny:for:aintno:contribution:1)
      (risk:accumulator-of:danny:for:aintno:zero))
  with-premises
  (hypothetical 460 nothing:1 in #[entity 461] bool)
  flickr danny fda common-knowledge hal parachutingassociation
  risk:accumulator-of:danny:for:aintno:premise6)
 ((risk:accumulator-of:danny:for:aintno:contribution:6)
  has-value -.1
  by ((switch:p) ((accepted (contribution risk danny -.1 skydiver))) (-.1))
  with-premises parachutingassociation)
 (((accepted (contribution risk danny -.1 skydiver)))
  has-value #t
  by ((((risk-estimate) (skydiving)))
      ((rejected (danny engages-in skydiving))))
  with-premises parachutingassociation)
 (((rejected (danny engages-in skydiving)))
  has-value #t
  by (user)
  with-premises parachutingassociation)
 ((risk:accumulator-of:danny:for:aintno:contribution:5)
  has-value 2
  by ((switch:p) ((accepted (contribution risk danny 2 eats-unhealthy-food)))
                 (2))
  with-premises fda common-knowledge hal danny)
 (((accepted (contribution risk danny 2 eats-unhealthy-food)))
  has-value #t
  by ((((risk-estimate) (unhealthy-food)))
      ((accepted (danny eats unhealthy-food))))
  with-premises fda common-knowledge hal danny)
 (((accepted (danny eats unhealthy-food)))
  has-value #t
  by ((((common-sense) (food)))
      ((& ((accepted (danny eats hotdogs)))
          ((accepted (hotdogs is unhealthy))))))
  with-premises fda common-knowledge hal danny)
```

```
(((accepted (hotdogs is unhealthy)))
 has-value #t
 by (user)
 with-premises fda)
(((accepted (danny eats hotdogs)))
 has-value #t
 by ((((eating-at) (restaurant)))
     ((& ((accepted (danny eats-at hals-hotdogs)))
         ((accepted (hals-hotdogs is-a restaurant)))
         ((accepted (hals-hotdogs primarily-serves hotdogs)))
         ((accepted (hotdogs is-a food))))))
 with-premises common-knowledge hal danny)
(((accepted (hotdogs is-a food)))
 has-value #t
 by (user)
 with-premises common-knowledge)
(((accepted (hals-hotdogs primarily-serves hotdogs)))
 has-value #t
 by (user)
 with-premises hal)
(((accepted (danny eats-at hals-hotdogs)))
 has-value #t
 by ((((works-at) (restaurant)))
     ((& ((accepted (danny works-at hals-hotdogs)))
         ((accepted (hals-hotdogs is-a restaurant))))))
 with-premises hal danny)
(((accepted (hals-hotdogs is-a restaurant)))
 has-value #t
 by (user)
 with-premises hal)
(((accepted (danny works-at hals-hotdogs)))
 has-value #t
 by (user)
 with-premises danny)
((risk:accumulator-of:danny:for:aintno:contribution:4)
 has-value .25
 by ((switch:p) ((accepted (contribution risk danny .25 rockclimber))) (.25))
 with-premises danny)
(((accepted (contribution risk danny .25 rockclimber)))
 has-value #t
 by ((((risk-estimate) (rockclimbing)))
     ((unknown (danny engages-in rockclimbing))))
 with-premises danny)
(((unknown (danny engages-in rockclimbing)))
 has-value #t
 by (user)
 with-premises danny)
```

```
 ((risk:accumulator-of:danny:for:aintno:contribution:3)
  has-value -.1
  by ((switch:p) ((accepted (contribution risk danny -.1 scubadiver))) (-.1))
  with-premises danny)
 (((accepted (contribution risk danny -.1 scubadiver)))
  has-value #t
  by ((((risk-estimate) (scubadiving)))
      ((rejected (danny engages-in scubadiving))))
  with-premises danny)
 (((rejected (danny engages-in scubadiving)))
  has-value #t
  by (user)
  with-premises danny)
 ((risk:accumulator-of:danny:for:aintno:contribution:2)
  has-value 2.
  by ((switch:p) ((accepted (contribution risk danny 2. motorcycler))) (2.))
  with-premises flickr)
 (((accepted (contribution risk danny 2. motorcycler)))
  has-value #t
  by ((((risk-estimate) (motorcycling)))
      ((accepted (danny engages-in motorcycling))))
  with-premises flickr)
 (((accepted (danny engages-in motorcycling)))
  has-value #t
  by (user)
  with-premises flickr)
 ((risk:accumulator-of:danny:for:aintno:contribution:1) has #(*the-nothing*))
 ((risk:accumulator-of:danny:for:aintno:zero) has-value 0))
|#
```

Bibliography

[1] Stewart Brand. *The Clock of the Long Now: Time and Responsibility: The Ideas Behind the World's Slowest Computer*. Basic Books, New York, 2000.

[2] Todd S. Braver and Susan R. Bongiolatti. The role of frontopolar cortex in subgoal processing during working memory. *NeuroImage*, 15(3):523–536, March 2002.

[3] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. Why and where: A characterization of data provenance. In Jan Van den Bussche and Victor Vianu, editors, *Database Theory – ICDT 2001*, pages 316–330. Springer, 2001.

[4] Philip A. Cowan. *Piaget: With Feeling*. Holt, Rinehart and Winston, New York, 1978.

[5] Johan de Kleer, Jon Doyle, Charles Rich, Guy L. Steele Jr., and Gerald Jay Sussman. AMORD: A deductive procedure system. AI Memo 435, MIT Artificial Intelligence Laboratory, Cambridge, MA, January 1978.

[6] Jon Doyle. Truth maintenance systems for problem solving. AI Technical Report 419, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1978.

[7] Kenneth Forbus and Johan de Kleer. *Building Problem Solvers*. MIT Press, Cambridge, MA, 1993.

[8] Ragib Hasan, Radu Sion, and Marianne Winslett. Preventing history forgery with secure provenance. *ACM Transactions on Storage*, 5(4):12:1–12:43, December 2009.

[9] Linda Hermer-Vazquez, Elizabeth S. Spelke, and Alla S. Katsnelson. Sources of flexibility in human cognition: Dual-task studies of space and language. *Cognitive Psychology*, 39(1):3 – 36, August 1999.

[10] Kay E. Holekamp. Questioning the social intelligence hypothesis. *Trends in Cognitive Sciences*, 11(2):65 – 69, February 2007.

[11] Lindsay M. Howden and Julie A. Meyer. Age and sex composition: 2010. 2010 Census Briefs C2010BR-03, United States Census Bureau, Economics and Statistics Administration, U.S. Department of Commerce, May 2011.

[12] Joxan Jaffar and Michael J. Maher. Constraint logic programming: a survey. *The Journal of Logic Programming*, 19–20(Suppl. 1):503–581, May–July 1994.

[13] Gordon G. Gallup Jr. Chimpanzees: Self-recognition. *Science*, 167(3917):86–87, January 23, 1970.

[14] Nuel D. Belnap Jr. A useful four-valued logic. In G. Epstein and J. M. Dunn, editors, *Modern Uses of Multiple-Valued Logic*, volume 2, pages 5–37. Reidel Publishing Company, Boston, 1977.

[15] Elisha S. Loomis. *The Pythagorean Proposition.* The National Council of Teachers of Mathematics, 1968.

[16] Marvin Minsky. K-lines: A theory of memory. AI Memo 516, MIT Artificial Intelligence Laboratory, Cambridge, MA, June 1979.

[17] Marvin Minsky. *The Society of Mind.* Simon and Schuster, New York, 1988.

[18] Marvin Minsky. *The Emotion Machine.* Simon and Schuster, New York, 2006.

[19] Lennart Nathell, Madelene Nathell, Per Malmberg, and Kjell Larsson. COPD diagnosis related to different guidelines and spirometry techniques. *Respiratory Research*, 8, 2007.

[20] Sue Taylor Parker and Kathleen R. Gibson. Object manipulation, tool use and sensorimotor intelligence as feeding adaptations in cebus monkeys and great apes. *Journal of Human Evolution*, 6(7):623–641, November 1977.

[21] Alexey Radul and Gerald Jay Sussman. The art of the propagator. CSAIL Technical Report MIT-CSAIL-TR-2009-002, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, January 2009.

[22] Anita Ramasastry. Will insurers begin to use social media postings to calculate premiums? *Verdict > Consumer Law*, Justia, January 3, 2012. http://verdict.justia.com/2012/01/03/will-insurers-begin-to-use-social-media-postings-to-calculate-premiums, accessed August 26, 2013.

[23] Anand S. Rao and Michael P. Georgeff. BDI agents: From theory to practice. In *Proceedings of the First International Conference on Multiagent Systems*, pages 312–319. AAAI Press, 1995.

[24] N. T. Roseveare. *Mercury's Perihelion: from Le Verier to Einstein.* Clarendon Press, Oxford, 1982.

[25] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 3rd edition, 2009.

[26] Barbara Rutter, Sören Kröger, Rudolf Stark, Jan Schweckendiek, Sabine Windmann, Christiane Hermann, and Anna Abraham. Can clouds dance? Neural correlates of passive conceptual expansion using a metaphor processing task: Implications for creative cognition. *Brain and Cognition*, 78(2):114–122, March 2012.

[27] Leslie Scism and Mark Maremont. Inside Deloitte's life-insurance assessment technology. *Wall Street Journal*, November 19, 2010. http://online.wsj.com/article/SB10001424052748704104104575622531084755588.html, accessed August 26, 2013.

[28] Patrick Suppes. *Introduction to Logic*. Van Nostrand Reinhold Company, New York, 1957.

[29] Ian Tattersall. An evolutionary framework for the acquisition of symbolic cognition by *Homo sapiens*. *Comparative Cognition & Behavior Reviews*, 3:99–114, 2008.

[30] Inge Volman, Karin Roelofs, Saskia Koch, Lennart Verhagen, and Ivan Toni. Anterior prefrontal cortex inhibition impairs control over social emotional actions. *Current Biology*, 21:1766–1770, October 25, 2011.

[31] David L. Waltz. Generating semantic descriptions from drawings of scenes with shadows. Technical Report 271, MIT Artificial Intelligence Laboratory, Cambridge, MA, November 1972.

[32] Carter Wendelken, Denis Nakhabenko, Sarah E. Donohue, Cameron S. Carter, and Silvia A. Bunge. "Brain is to thought as stomach is to ??": Investigating the role of rostrolateral prefrontal cortex in relational reasoning. *Journal of Cognitive Neuroscience*, 20(4):682–693, April 2008.

[33] Liane Young, Joan Alber Camprodon, Marc Hauser, Alvaro Pascual-Leone, and Rebecca Saxe. Disruption of the right temporoparietal junction with transcranial magnetic stimulation reduces the role of beliefs in moral judgments. *PNAS Early Edition*, March 29, 2010. DOI: 10.1073/pnas.0914826107.