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 15, 2013, in partial fulfillment of the
requirements for the degree of
Engineer of Science in Computer Science and Engineering

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: Professor of Electrical Engineering

Chapter 1

Introduction

What makes human intelligence different from that of other animals? Although a number of different theories and differentiators have been presented in the past, including tool use [13], self-recognition [6], and social intelligence [5], one of the leading theories at present is the human ability to solve problems in a abstract, "symbolic" manner [17]. Indeed, humans appear to be able to use their symbolic problem solving skills in any number of different ways, including applying logic to statistical scientific analysis based on the scientific method, as well as the application of the probabilistic approaches to learning that have formed the foundation of many modern algorithms considered to fall under the guise of "artificial intelligence" [15].

While the increased understanding of each of these various 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 mechanisms for control that allow humans to make use of any of these various approaches in a flexible manner, outside of the limited work proposed as part of Minsky's theory of the Society of Mind [11, 12]. If we are to actually claim we understand the basis of human intelligence, we must also be able to discover the source of the flexibility in thought that allows us to effectively and efficiently solve problems using any number of these methods without any preconceived algorithms of how such control may be implemented.

1.1 Background

{More background research prior to...}

{The following might be better placed in the proposition section/related work? Moved around at the least.}

Perhaps the most relevant theoretical work on the mechanisms of learning and application of problem solving skills has been proposed and elaborated by Marvin Minsky's *K-line* theory [10], 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 problems are more readily identified, along with the mechanisms best suited to solving them.

Most 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 perceived as 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 of knowledge (and thus, indirectly, perception) through the connections within the net of goals as well as perceptions.

The mechanism proposed here resembles this mechanism, but effects the declaration of goals through the declaration of the propagator network which connects the beliefs of propositions, rather than through explicit declarations on their own. However, perception and knowledge is encapsulated in the beliefs themselves. {rework}

1.2 Defining Flexibility

If we are to discuss the flexibility of human problem solving skills, we must better define what exactly is meant by this term. Human problem solving appears to have a number of elements which, while perhaps not entirely unique to humans, are features that 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), perceived "efficiency", resilience to contradictory inputs, and the ability to evaluate and make use of one of any number of different strategies using what appears to be a single mechanism of thought.

While reactive behavior is common to practically all living things, humans appear uniquely able to not only react to physical constraints presented upon execution of a particular problem solving mechanism, 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). {a useful citation to some cog sci research here would be appropriate}

Such reactivity is particularly evident in humanity's ability to engage in long-term planning; when knowledge about specifics will necessarily change over time, there is a definite need to be able to adjust any strategies seeking long-term results to account for large-scale changes in the situational environment, such as those created by global warming, technological development, and large-scale societal change. Although humans often encounter practical difficulties in envisioning and planning on extremely long time-frames, such limitations appear to be cultural in nature, rather than necessary limitations of human intelligence, as some social groups have proven quite willing to solve problems with the understanding and need to plan and react to changes on timescales beyond many hundreds of years [1].

In addition to the ease of reactive thinking and planning, human problem-solving appears to be fundamentally goal-oriented. Humans rarely extrapolate and solve problems that are fundamentally irrelevant to a particular goal they have in mind. Even when solving abstract problems, such as those addressed in the field of theoretical physics, there is still necessarily some goal, even though that goal may only exist as a result of a failed attempt to solve another problem. For example, the classic "failed experiment" of Michelson and Morley to prove the existence of the luminiferous aether [9] failed to achieve the goal of proving the existence of the aether, but in so doing, established a new goal to explain the unexpected null result, to be later resolved through Einstein's formulation of the theory of special relativity. Since hu-

mans necessarily solve problems on the basis of goals (even if those goals are derived from unexpected results in solving another problem), any functionality or mechanism which is capable of selecting and controlling problem solving must necessarily be goal-oriented.

Unique to humans, however, is the ability to synthesize disparate data, a function of human thought that appears to be a unique feature of symbolic thought. Research by Herver-Vazquez, et al has shown that humans are uniquely capable of synthesizing geometric features and landmarks to identify locations [4], an ability which is not observed in rats. Thus, it would appear that general problem solving mechanisms seeking to approximate human skills at problem solving must be able to synthesize arbitrary features to derive solutions, lest they be unable to solve the fundamental task of locating an object using both geometric and landmark-based cues.

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 is a perception that humans are uniquely efficient at solving certain kinds of (vastly different) problems. {expand on this... sources? examples? *Gerry: "e.g. nonsense from Penrose"*}

Human intelligence is also notable for its ability to handle contradictory beliefs, which is 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, but that humans are actually capable of thinking within a superset of such logic (as the mere existence of human logicians necessarily proves that classical logic may be evaluated using the human mind).

The flaws of classical logic with respect to non-contradiction are hardly universal however, and other logics may prove beneficial in determining the basis of thought. Modern logics that take into account paraconsistency (such as Belnap's four-valued logic [7]) in which contradictions should not cause an "explosion" of conclusions, or dialetheism, in which contradictions may, in fact, exist and be true, seem to embody a much more pragmatic way to conceive of 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) [8]. While the symbolic thought of a single human does appear to be constrained to follow one specific approach at any given time, man is free not only to apply other approaches based on personal judgments made while solving a problem, but man is even capable of deriving and learning new approaches solely to solve a novel problem that has yet to be encountered, and to apply these approaches in the future. Any such system that attempts to achieve a modicum of intelligence must be prepared to be so flexible in its approaches to problem solving, or it is unlikely that it will truly resemble the abilities that may be achieved by a human.

1.3 A Problem: Issuing Health Insurance

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

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

When issuing insurance, Sally must determination of eligibility in accordance with Aintno's eligibility policies, which determine eligibility 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 a maximum risk score threshold of 3. If the total score is greater than 3, then Aintno will refuse to issue insurance to the customer. Otherwise, Aintno is more than happy to work

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 1.1: Contributions to risk score based on personal behaviors

out an insurance policy based on the value of that score.

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 court, 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 pass the file along to the actuarial department to 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 of risk are relevant to determining Danny's insurance rates.

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, 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.

On Facebook, the program highlights one of his recent Facebook "likes": one

eats(*subject*, *food*) ∧ unhealthy(*food*)	→	eats(*subject*, unhealthy-food)
likes(*subject*, *thing*) ∧ is-a(*thing*, food)	→	eats(*subject*, *thing*)
likes(*subject*, *place*) ∧ is-a(*place*, restaurant)	→	eats-at(*subject*, *restaurant*)
works-at(*subject*, *place*) ∧ is-a(*place*, restaurant)	→	eats-at(*subject*, *restaurant*)
(eats-at(*subject*, *place*) ∧ is-a(*place*, restaurant) ∧ primarily-serves(*place*, *thing*) ∧ is-a(*thing*, food))	→	eats(*subject*, *thing*)

Table 1.2: Rules for determining risk

"Hal's Hot Dog Hut", a hot dog restaurant. Similarly, analysis of several photographs posted on Flickr allows the program to determine that it is probable that Danny is employed part-time at Hal's Hot Dog Hut, even though this is not noted in his file. Knowing little about Hal's Hot Dog Hut, Sally probes further and the program quickly mines the Hal's Hot Dog Hut website to determine that the establishment is a restaurant that sells hot dogs which, according to common knowledge known by the program, are not only food but, according to the FDA, are unhealthy.

With this knowledge in hand, the program is able to apply a number of pre-programmed rules to assist in assessing and accumulating risk scores (See Table 1.2), from which she is able to determine that Danny's risk score is at least an unacceptably high 5.15 thanks in part to the information gleaned from Facebook. Given this score, Sally finds that Danny is ineligible for insurance. She makes note of this and begins to prepare her report.

Before she finishes, however, she realizes that, according to a new law enacted by the state in which Danny lives, insurance agencies are not permitted to use data from Facebook in making insurance issuance decisions. She then returns to the risk assessment program and removes the Facebook profile from the analysis inputs and requests a reanalysis. The program again tabulates Danny's score, and finds it to again tabulate to at least 4.05, thanks to the Flickr account which had not been needed last time (due to the extra cost of determining whether Danny ate unhealthy food).

Thus confident in Danny's ineligibility, she finalizes her report and forwards it along to be conveyed to Danny.

1.4 Solving Aintno

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 attempt to create 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 the Facebook and Flickr profiles mentioned in it, 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, 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 retooling as rules, regulations, and inputs change. For example, if Facebook 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 Facebook for every application, Aintno might only wish to query Facebook 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 that 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, as 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.

1.5 Thesis Overview

This thesis seeks to outline a mechanism that achieves such flexibility in problem solving by modelling 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 (justifications) 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 [14]. This computational substrate provides a simple mechanism for maintaining and updating partial information structures and allows for the refinement of partial knowledge 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 belief propagation 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 do any computation on the inputs stored in those cells, storing the 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, however, most complex operations may be performed by networks of simple propagators representing basic mathematical operators and "switches" in the propagator network alternately connecting one of several input cells to an output cell.

Propagators serve much the same purpose as electronic components do in an elec-

trical circuit; the ways in which simple propagators are connected will necessarily 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 in integrated circuits), partial propagator networks may be abstracted and reused as "compound propagators".

2.2 Cells

The second component of a propagator network, the *cell* acts as the glue of the propagator network, as it stores data that may be used by propagators to do computation. The information stored in a cell may be *updated* at any time by a propagator that treats the cell as output. An update message sent to a cell by a propagator will then be *merged* with the data currently in the cell using an appropriate merge operation based on the type of data stored in the cell and the data provided in the update message. For example, if a cell stores a numeric interval and receives as an update, another numeric interval, it may merge the update by taking the *intersection* of the two intervals, effectively refining the data in the cell.

Once a cell has finished merging its content with that in the update message, it will subsequently alert those propagators which have registered to become *neighbors* of the cell. Usually, these propagators will make use of the data stored in the cell in some way, and thus should be alerted to do additional computation based on the newly merged data in the cell. As these propagators may then send updates to other cells, changes to the content of a cell will effectively *propagate* across the network of propagators and cells.

The principle that a cell must refine its contents in the merge operation means that propagator networks readily lend themselves to representing and manipulating *partial information*. Appropriate merge operations may be constructed to effectively expand and extend the partial knowledge in a cell when an update is received. For example, in the above numerical range example, the intersection operation is appropriate to

refine multiple, possibly equally broad ranges to obtain a more precise answer. {This paragraph needs to be reworded, and I haven't got a good idea how to continue it at the moment.}

2.3 An Example

Consider the example of 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, and it is possible that they will 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 (since any given reading will be no more than two degrees from the actual temperature).

In addition to their measurement flaw, these thermometers have one other practical flaw: only 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 thermometer which reads in degrees Fahrenheit 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 value (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 simply adopt the contents of the

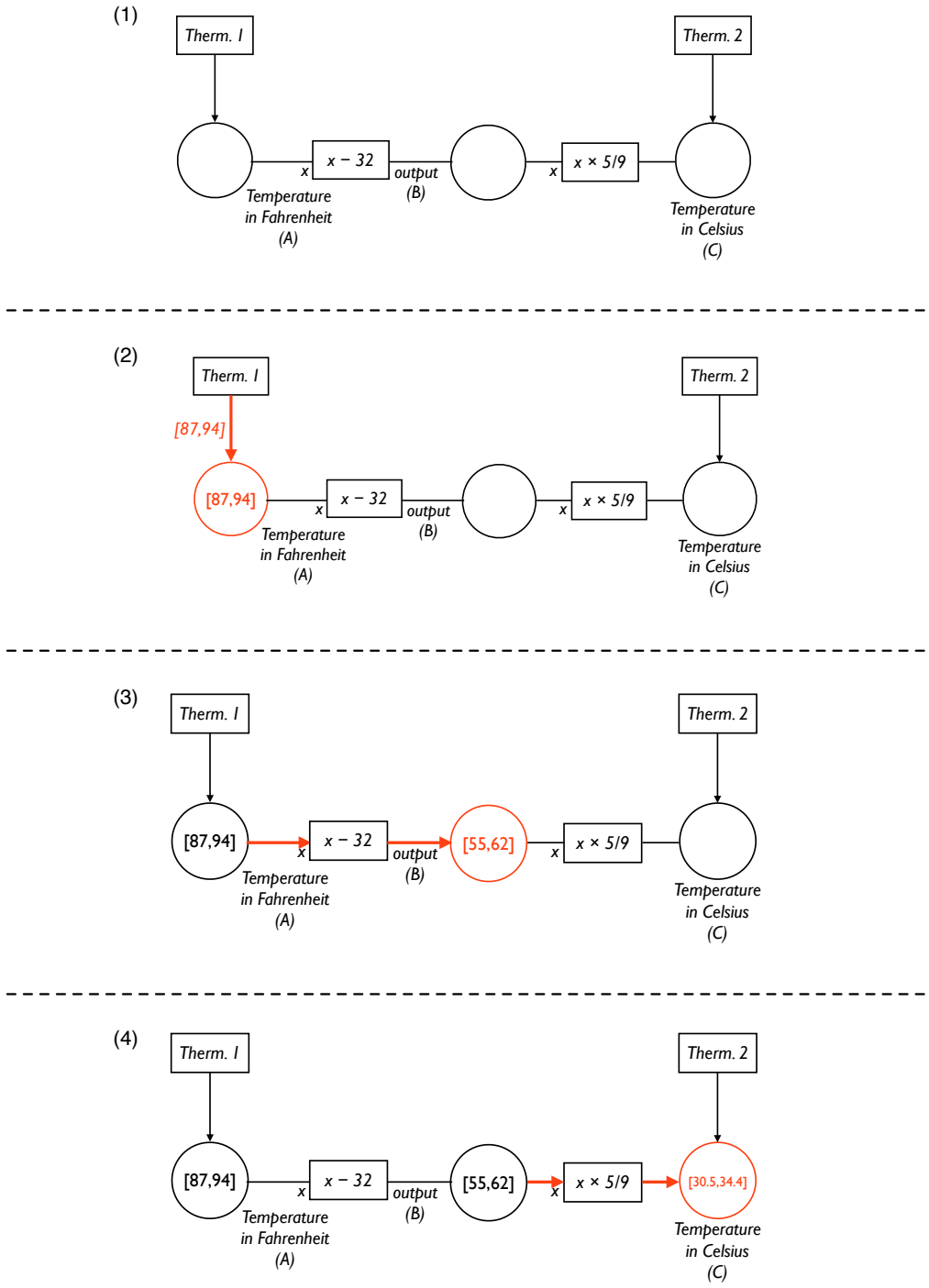

Figure 2-1: A propagator network which combines and converts the outputs of two thermometers, converting between Fahrenheit and Celsius. A temperature range from thermometer 1, in degrees Fahrenheit, (2) is sent to a cell. This alerts a chain of propagators responsible for converting the temperature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

update message to be their new content. This is due to the fact that the cells initially contain nothing. Since there is no information with which the numeric range in the update data can be merged, the merge operation simply sets the value of the cell.

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 are implemented this way and divide and add appropriately 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 contradictory information to 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 it may result in additional work being done. For example, if a temperature range contradicts an existing estimate, it may be appropriate to "kick out" older contributions that 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, and if so, this may prompt the repair or removal of 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 these

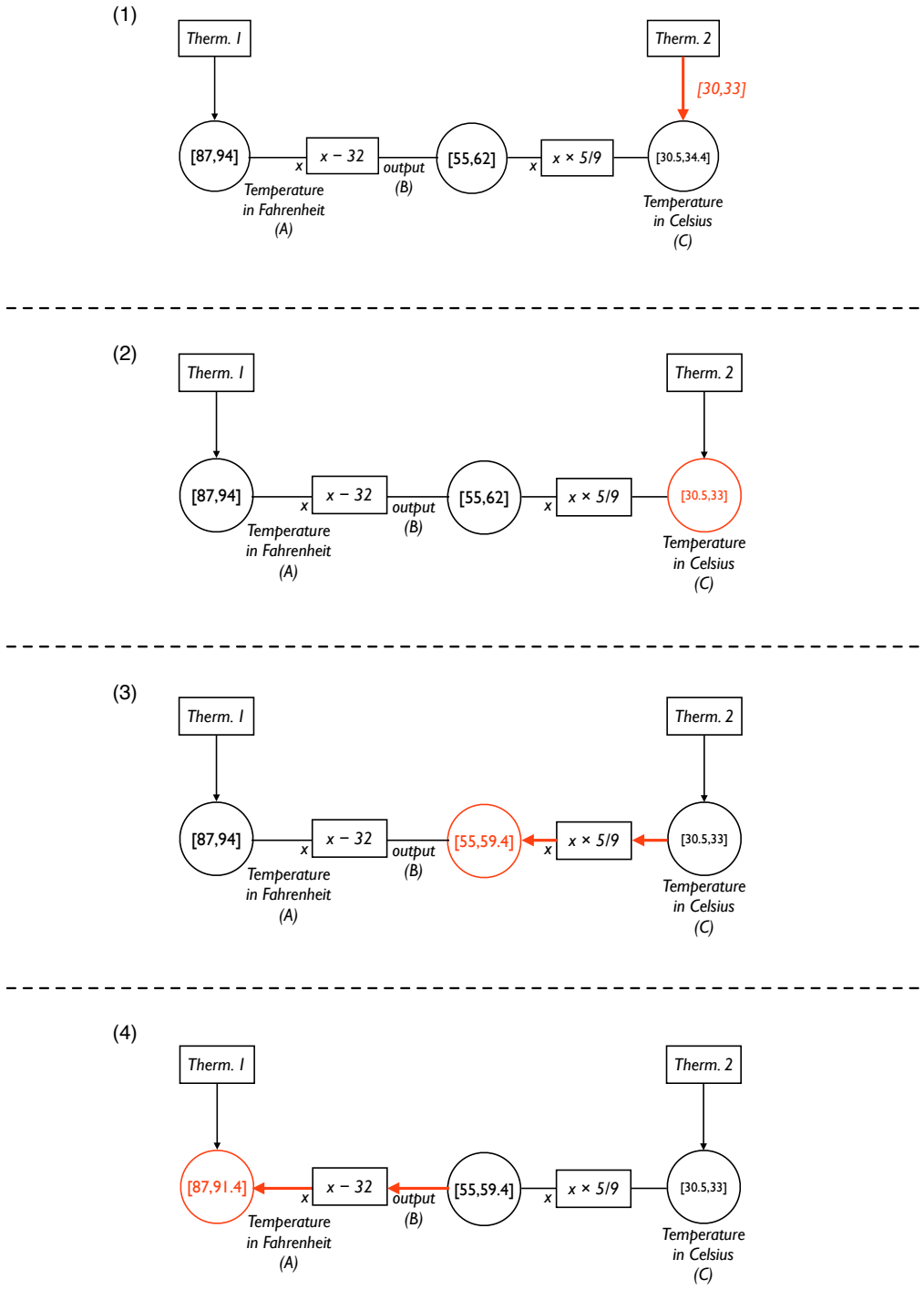

Figure 2-2: Propagator networks merge the existing contents of cells with updates received from propagators. An update in degrees Celsius from thermometer 2 (1) is merged with existing knowledge of the temperature (2) and propagates to the output cell containing the temperature in degrees Fahrenheit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

conflicts. The primary implementation of propagator networks in the 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, especially in the large search spaces that lie at the core of numerous kinds of problems that might wish to be resolved by a general problem solver. While the underlying propagator network mechanism may not directly support contradiction handling, the flexibility of the "cell merge" operation would certainly not prohibit basic contradiction handling to be resolved as part of the logic of such merge operations.

The data structure known as a truth maintenance system, or TMS, has several features that make it particularly appropriate for handling contradiction resolution in their merge operation.

2.5 Truth Maintenance Systems and Backtracking

Truth maintenance systems [2, 3] are a data structure which allows for the maintenance of a variable's value based on the set of minimal supports or *premises* for any given value. A truth maintenance system tracks all possible premise-set/value pairs known to be valid for a given variable, and it may be "queried" to retrieve the value that is 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 relevant to the specification of the returned value.

Truth maintenance systems are appropriate contents for cells, as they may be merged by simply taking the union of the set of values and supports in a cell and those in the update message. The union of the value-support pairs may then be

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

By storing values as a function of a set of premises, contradictions may not only be identified during the merge process (i.e. when two value-support pairs with different values are supported by the set of premises) but also resolved through manipulation of the premise set at merge-time. Such premise-set manipulation may change the effective value stored by a cell without actually introducing a contradiction in the value stored in the *cell*, as 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 that require a *backtracking* search. When the introduction of a particular premise creates a contradiction which is recognized during a cell merge which would otherwise create a contradiction, the TMS merge operation may effectively back out 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 lack any features that 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.

{I should probably say somewhere about how this is similar in some respects to constraint propagation, because it seems relevant, at least in passing}

Chapter 3

Five-Valued Propositions

This section desperately needs some diagrams

{TODO: Read, reference AMORD}

Controlling problem solving systems requires a way to represent not only those beliefs that act as input to the problem solvers, but also, more generally, the nature of 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 propose to unify both the expression of these needs and the expression of input facts by representing both 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 statements of fact in so far as they represent a statement without any assertion of truth or falsehood associated with it; propositions represent merely the concept of a statement itself. To represent the nature of belief in a particular proposition, each proposition is considered in terms of five different axes of belief: acceptance, rejection, *contradictory*, *knownness*, and *unknownness*.

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. 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 that may be evaluated differently in different contexts.

The remaining beliefs may be expressed in reference to acceptance and rejection. The reasons for acceptance and rejection may be considered together to determine whether there is a *contradiction* in beliefs which may force backtracking and the removal of certain assumptions that may have led to such contradictory beliefs.

Where the *contradictory* state of belief is the conjunction of acceptedness and rejectedness, *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 if there is no support for it being either accepted or rejected.

This alternate metric of *unknownness* 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 may be unnecessary to expend additional effort to obtain 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.

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 the opposing relationship between knownness and unknownness), 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,

unknownness may be used to drive work to prove acceptance or rejection, but this is not a requirement by any means. 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 all well and good, 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 need to be able to construct a *network* of propositions, such that different states of belief in a proposition are able to influence and modulate beliefs in other propositions and cause computation to occur.

For example, it should be possible to condition the "acceptance" of a need to do research on the "unknownness" of a proposition for which we would like to know. 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 proposition becomes known, the second proposition, the need to know, should become rejected.

This "propagation" of belief is appropriately handled by the data propagator networks discussed in the previous chapter. In mapping the propositional model to a data propagator model, it seems sensible to map the individual "beliefs" to cells while the "rules" which relate beliefs act as propagators which connect them and forward and combine relevant two or more belief states to generate a third.

Given the basic relationships between belief states described previously, then, 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 the fact that the *contradiction* cell is automatically populated with a value of **false**. This captures the implicit assumption that no proposition may be simultaneously

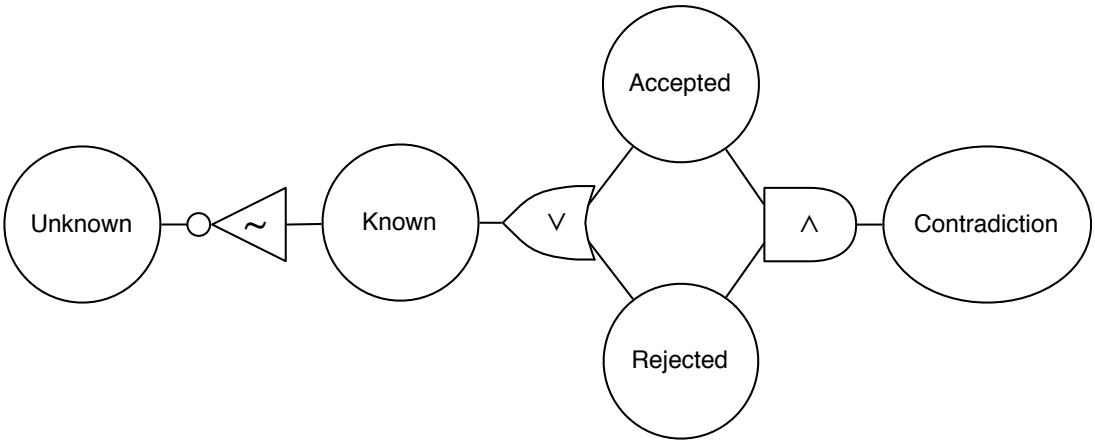

Figure 3-1: A proposition may be described with respect to five values connected in a propagator network of logical operations. Each circle represents a "cell" containing the data in support (or opposition) of that particular belief state, and the currently supported value will propagate its dependencies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

accepted and 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 $accepted = \neg rejected$, but also assists 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 3.4).

3.3 Dependencies and Justifications

The reasons underlying the generation of a solution to a problem are generally valuable in evaluating that solution. Indeed, it usually difficult to "believe" that a particular proposition is true or false without having some measure of confidence in the methodology by which the belief was derived. In other words, any system that attempts to represent the range of human beliefs in a proposition should be prepared to "show its work" if asked, so that other, skeptical agents (be they digital or of flesh-and-blood) may be rightly persuaded to share that belief.

The utility of such *justifications* is many-fold. Not only may such justifications

help to establish shared belief, but they are also useful in diagnosing the derivation of incorrect beliefs and in the development of novel hypotheses by either altering the premises that support the justification or by testing additional hypothetical premises that may alter the justification and final belief.

For example, analysis of the orbit of Mercury using purely Newtonian mechanics predicts the precession of its perihelion, but comparison of this expected value and the observed value by Le Verrier in 1859 led to effort being spent to resolve the conflict. It was only after the promotion of a number of alternate hypotheses that would potentially resolve the conflict that 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.

Such "justification traces" are naturally supported by the existing propagator framework. The semantic-rich "circuit diagrams" of propagator networks easily map to justifications; in order to determine how a cell obtained its value, one need only recursively trace back over the connections of the underlying network, provided that the input propagators that are relevant to a given value are attached to the value itself.

This is accomplished by extending the truth maintenance system structure to additionally store the identity of the propagator which directly informed the cell to provide each truth value and its premises. Given this information, it is possible to follow the *informant* propagators backwards to the cells which they used as input, and, from them, recursively fetch their premises and informants until the complete justification for a belief is constructed.

3.4 Hypothetical Beliefs and Backtracking

Another feature common to a number of problem solving approaches is the concept of the *hypothetical assumption*. Such assumptions play critical roles both in lower-level proofs (e.g. proof by contradiction) as well as more complicated problems for which certain pieces of evidence may be missing or may be assumed to exist without

concrete support.

While these "hypothetical premises" are similar in many respects to the concrete premises described in the previous section, they are distinguished by a more transient, context-specific nature. Hypothetical premises rarely support more than one independent belief directly (additional beliefs may depend on the hypothetical premise in so far as they depend on the acceptance or rejection of another proposition, but they generally are not supported directly).

More importantly, however, is the fact that, unlike more concrete premises, hypothetical premises are generally designed to be retracted as additional evidence supports an alternate position. For example, if we assume that a woman is childless for the sake of determining her insurance rates and later encounter evidence that the woman, in fact, has a son, it is desirable to reject our previous assumption of childlessness and redetermine *only* those beliefs which were premised on that belief. Thus, we would like to automatically *kick out* the assumption based on the fact that 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 supported instead based on the support for the identified contradiction.

In the propagator network model, hypothetical assumptions may be modeled as premises which are marked such that they may be automatically kicked out and removed from the system when a contradiction is detected when a truth maintenance system stored in a cell is updated. In effect, the merge operation of truth maintenance systems will preferentially remove hypothetical assumptions to remove support for either the true or false value. New beliefs are then recalculated based on the removed assumption.

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 rejected, and a rejected proposition is not accepted). Then, when support arrives to support the contrary, a contradiction will be encountered in the truth maintenance system of the accepted or rejected cell, forcing backtracking.

Chapter 4

Building Problem-Solving Strategies

With the computational model chosen, it remains to be shown how these propositions may be connected to properly solve problems. First, however, we must consider how a specific problem might be solved 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 that they may be integrated with control, it stands to reason that it should be possible to model one such mechanism for solving problems, the application of rules, using propositions. The basic action of evaluating a rule has several 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 predicate of the rule such that any variables in the antecedent predicate 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 a concrete instance of the rule by binding any variables in the rule with respect 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

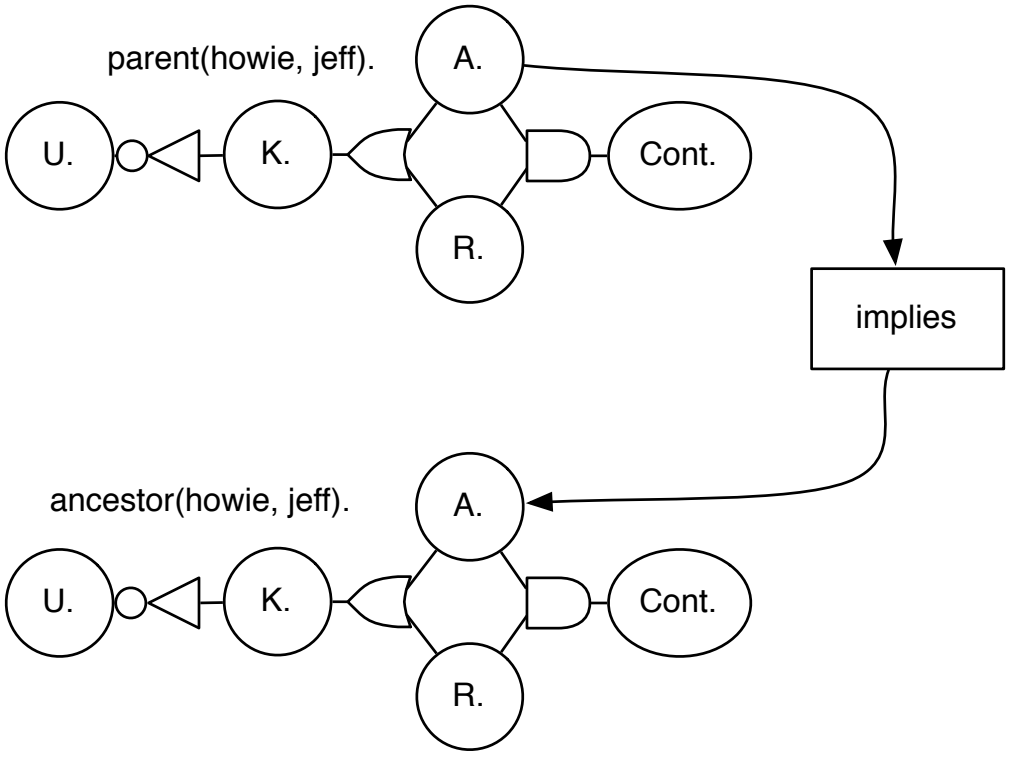

Figure 4-1: A simple network of propositions which propagates the belief in the acceptance of parent(howie, jeff) along with its dependencies to the acceptance of ancestor(howie, jeff). The "implies" propagator ensures that this belief is properly propagated without modification and merged with the contents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

> parent(howie, jeff) will propagate, *with its dependencies* to the *acceptance* of the proposition ancestor(howie, jeff), as depicted in Figure 4-1.

As a result, we might represent such the "parenthood" rule using Scheme code like that of Figure 4-2. In `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

```
(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.

states through its third argument that this acceptance is based on the acceptance of `parent-proposition`. While we may believe `(ancestor a b)` for any number of reasons, we can only believe it is 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 to prove ancestry without having to be told which ones we are looking for?

There are several problems that 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 can use efficient database storage and indexing algorithms to find all such `(parent a b)` that we are aware of at the time we are told 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

```
(define (accept-ancestor-by-parenthood
         parent-proposition environment)
  (let ((ancestor-proposition
         (instantiate-proposition '(ancestor ?a ?b) environment)))
    (accept ancestor-proposition
            (list 'prove-ancestry-by-parenthood
                  (get-binding '?a environment)
                  (get-binding '?b environment))
            (list (accepted parent-proposition)))))))

(define (prove-ancestry-by-parenthood)
  (find-proposition-matching '(parent ?a ?b) '()
    accept-ancestor-proposition-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. {Maybe merge accept-ancestor-by-parenthood back into prove-ancestry-by-parenthood?}

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!

When making such a pattern matcher, then, we must take care that it is *lazy*. Any move to prove ancestry through the existence of a parenthood relationship must be ready to 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` which 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, which instead searches for existing propositions which match the pattern `(parent ?a ?b)`, where the question marks denote named variables a and b. Upon finding any such proposition, the function `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 an appropriate proposition.

For example, given the matching proposition `(parent howie jeff)`, the *prop* 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 then used in turn in `instantiate-proposition` in which they are substituted for the `?a` and `?b` variables in the pattern `(ancestor ?a ?b)` so as to instantiate the proposition `(ancestor howie jeff)` before connecting the acceptance belief of `(parent howie jeff)` to `(ancestor howie jeff)` as in Figure 4-2. Similarly, the `get-binding` function must be used to resolve the bindings to `?a` and `?b` for use with the explanation of the method used to conclude acceptance.

The observant reader will note that the `find-proposition-matching` function takes an empty list as its second argument. Such a second argument is particularly helpful when considering the chaining of multiple patterns, 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)` 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)` by binding 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.

```
(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.

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 tentative acceptance of that proposition with the consequent proposition of the rule. This has a distinct downside in that we will necessarily be doing work for propositions which 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, as a proposition, 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 {cite}, even though only about 150 million of those relationships would ever be accepted!

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 results in doing far more work than our original rule, should we not preferentially connect only those `parent` propositions which we already accept? Do we not want to help control our problem solving based on what we believe?

If so, perhaps an appropriate solution would be to make the body of the function `accept-ancestor-by-parenthood` contingent on a particular 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 a distinct propagator neighbor of the accepted state of the parent proposition, as in Figure 4-5. When coded up properly using the `s:when` propagator, which lazily evaluates its body only when the condition becomes true, such a rule might look like that of Figure 4-6.

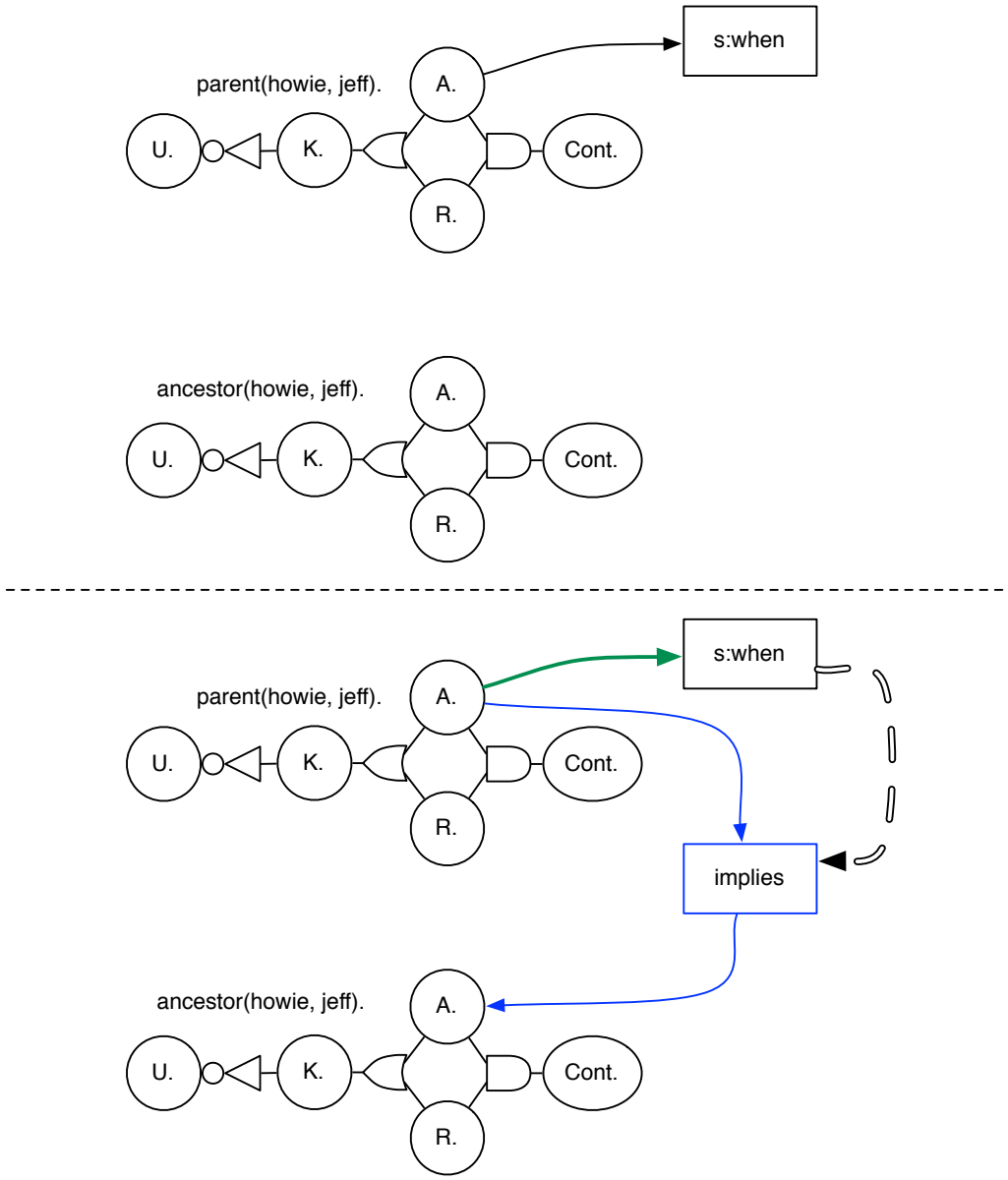

Figure 4-5: Lazily attaching a rule using the `s:when` propagator, which builds the connection between the acceptedness of parent(howie, jeff) and ancestor(howie, jeff) only when the parent relationship is accepted.

```
(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 the 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 relatively straightforward 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 genererator which returns a proposition matching the specified proposition pattern.

Propositions are matched in order, with the environment for each matching proposition being used in turn to match subsequent proposition (effectively finding each proposition in order using subsequent nested `find-proposition-matching` functions, each taking as 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 as 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 with respect to the environment implicitly defined by the enclosing matched variables. That is, `the-prop` returns the proposition defined by `instantiate-proposition` with the same pattern and the environment returned by the final matched 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 states are automatically stored in the named variables which are reused in the body of the `accept`.

Thus, in short, the `rule` keyword effectively defines a rule with a given rule body (the list of belief states of propositions which must all be believed) and rule head (the body in which work is done). {Something about nesting rules?}

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 for specific 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 insight regarding the lazy construction of the propagator network is a crucial one to 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 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 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 unknownness 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 recursive case, where there may need to be a number of different ancestry relationships that must be built transitively), this work may be

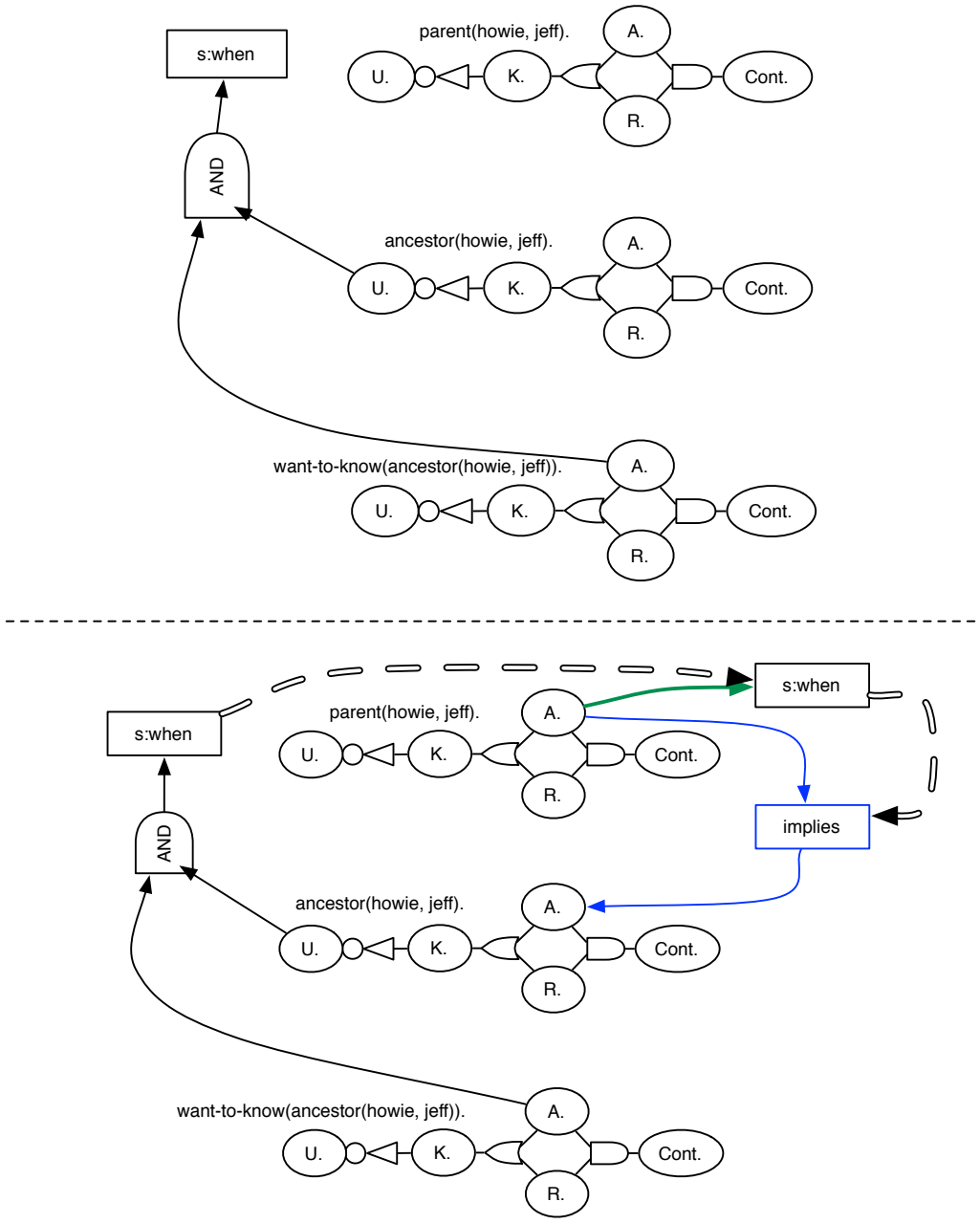

Figure 4-8: Controlling the search for ancestors only when such a search is needed. Only when `(want-to-known (ancestor ?a ?b))` is accepted and the corresponding `(ancestor ?a ?b)` proposition is unknown will a proof be attempted (bottom). If the ancestry proposition is known (i.e. accepted as true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

turned off when the ancestry relationship 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!

Chapter 5

Building Explanations

While a problem solver capable of answering arbitrary complex problems is indeed valuable, the answers it produces are necessarily only as reliable as they can be proven to be correct. Certainly 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. As a result, a problem solving system that is capable of providing *justifications* for its answers is more valuable and useful than one which cannot. Can the problem solver described in the previous chapter be modified or manipulated to produce meaningful justifications? {This seems like it could be strengthened as an argument.}

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 explanations for its answers, we must first define exactly what is meant by the words *explanation* and *justification*.

By explanation, we speak of the common-sense idea that a particular action, product, or belief may be described in terms of the mechanisms by which the action was caused to happen, the product was caused to be made, or the belief was caused to be held. Generally speaking, an explanation consists of 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.

A *justification* is an explanation that is typically minimal in extent. For the purposes of this thesis, a justification does not include incorrect decisions and actions that 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.

In short, an *explanation* is a description of what happened before the state existed, while a *justification* is a description of the causal tree, including both proximal and distal causes, which, if considered as a partially-ordered directed graph of cause to effect, would be sufficient to explain the evolution and production of a given state. While the primary focus of this chapter is on justifications specifically, I will use the words explanation and justification interchangeably to refer to justifications.

{Is this an accepted definition? Should I include relevant citations that discuss this point? Probably so.}

Justifications are similar in many respects to the concept of document provenance, which typically represents "the record of actions taken on [a] particular document over its lifetime," {cite Hasan09} and data provenance, which is a description of how a particular piece of data came to be and arrived in a given database {cite Buneman01, or perhaps wait to expand on it because he introduces where- and why-provenance which we describe in the next paragraph}.

Finally, it is worth keeping in mind that we should differentiate justifications (i.e. the *reasons* for something) from dependencies, which describe the ultimate *supports* for something. While the former tells us how we got somewhere (in the vein of Buneman's "why-provenance"), the latter tells us only those facts or sources that we depended on to get there (à la Buneman's "where-provenance").

5.2 The Suppes Formalism

So how can we characterize a justification of a particular belief produced by a propositional problem solver? As I have demonstrated in Chapter 4, the beliefs held by

propositional problem solvers evolve over time based on the logical connections (often implication relationships) between different belief states. What is a natural formalism for justifications for this evolution?

Since we speak of logical relationships, a naïve conclusion would be to say that our justifications should take the form of logical proofs. This is not an unreasonable stance to take, given the fact that all of the examples that have been shown so far involve logical implications, conjuctions, 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. 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 even be purely Boolean or algebraic in nature!

Even so, a generic sense of a proof seems quite reasonable to consider, as long as we expand the definition of an operation to include any general-purpose propagator.

One wrinkle complicates the adaptation of a simple proof mechanism. As mentioned in the previous chapter, navigation of the the answer space necessarily requires the correct management of the *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 typical proofs only point to the *processes* by which a derivation or manipulation of data occurs, the proof formulation proposed and utilized by Patrick Suppes [16] captures not only the nature of the derivation, but also indicates 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, 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.

{Is this acceptable? I wish to give proper credit, but I do not like lifting wholesale unless it is appropriate. Perhaps another example of my own design would be better?}

{1}	(1) $\mathsf{C} \to (\mathsf{D} \to \mathsf{B})$	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) $\mathsf{D} \to \mathsf{B}$	1, 5 T
{1, 2, 3, 4}	(7) B	3, 6 T
{1, 2, 3}	(8) $\mathsf{G} \to \mathsf{B}$	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 Dodges will be second. Therefore, if the Giants are first, then the Braves will be fourth."

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) $\mathsf{G} \to \mathsf{B}$, but also that it is directly derived from (4) G and (7) B by conditional proof, and depends on acceptance of the premises (1) $\mathsf{C} \to (\mathsf{D} \to \mathsf{B})$, (2) $-\mathsf{G} \vee \mathsf{C}$, and (3) D to be shown.

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 the nature of a conditional proof!

5.3 Building Justifications

Given the power and relevance of Suppes's formalism to propositional reasoning by 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? It turns out 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, it is possible to impose a data structure above and beyond the basic propagator "publish-subscribe" model to abstract the implicity structure of propagators into a navigable directed graph. The MIT-Scheme implementation of propagators includes such an abstraction which implicitly groups all cells into groups called *diagrams*.

Diagrams are defined recursively, as diagrams may contain one or more parts which are also diagrams. Cells are fundamental diagrams consisting of no *parts*, and more complex diagrams may be made from combining cells and diagrams which contain them as descendent parts.

As a result, conceptually, the diagram structure of a propagator network is a pyramidal directed graph (as in Figure 5-2) that 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. parent diagrams of which the diagram is a part), so that the given pyramid of diagrams is not a tree structure but a directed graph. The creation of propagators automatically constructs diagrams at the first abstraction layer above basic cells, connecting 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 when a given diagram has multiple paths to the highest 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 network creation time, it is possible to additionally determine whether cell C is an input to A, an input to B, or both, as the propagators themselves have knowledge as to whether or not their implementation will read a cell or write a cell (or both). As such, promises to "read a cell only" may be interpreted as cells which are input to the diagram, while promises to "write only" would similarly be interpreted as output.

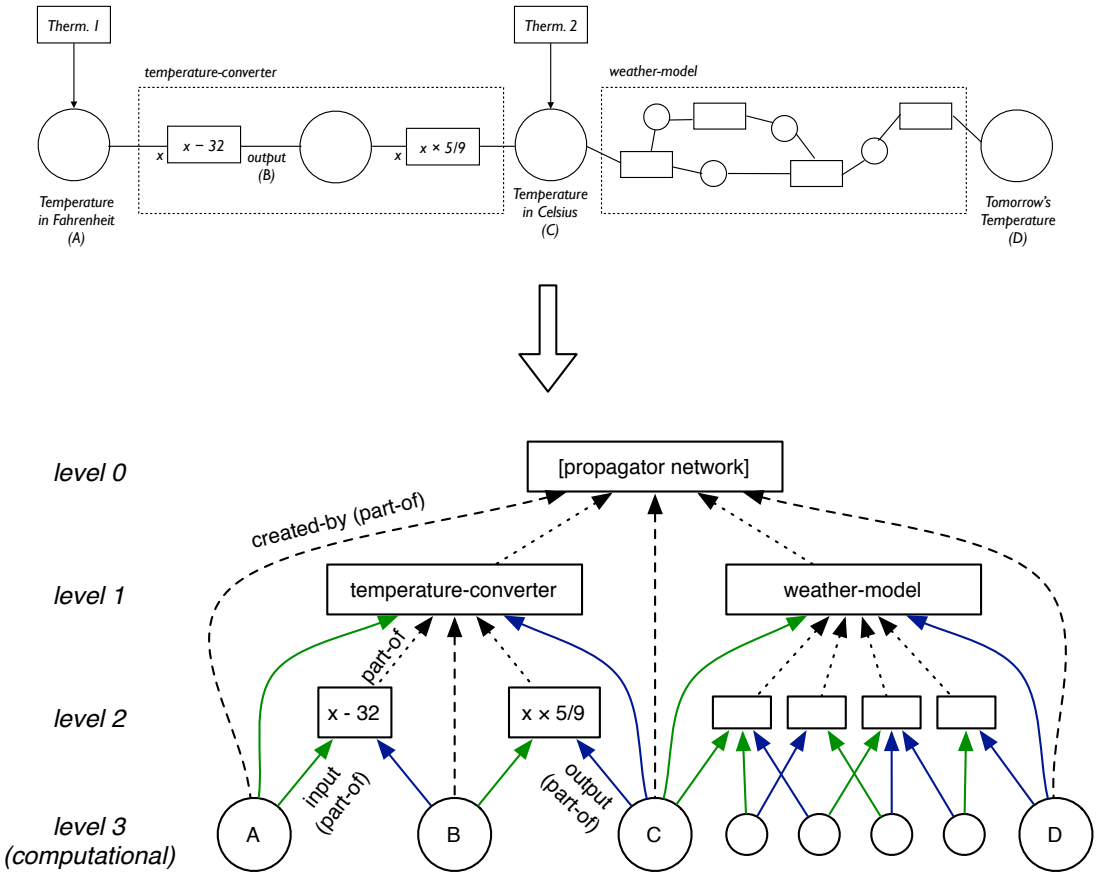

Figure 5-2: A complex propagator network maps to a pyramidal semantic structure with more abstract operations representated at higher levels. At the top, three cells, A, C, and C are connected by two compound propagators, `temperature-converter` and `weather-model`. Both compound propagators expand into additional, more basic, 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 directed arrows represent "part-of" relationships between cells and diagrams at one level and the diagrams that belong to a higher level. Colors and line patterns represent annotations that may be made (e.g. a promise that a given cell is used only as input/output by a propagator).

Justifications for the content of a cell may thus be constructed by climbing the "diagram pyramid" to an appropriate abstraction level, identifying "input" cells, and then querying those cells for their contents and working to their inputs in turn. In this manner, a list of antecedent cells and values may be effectively 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.

{Should this be expanded upon? Probably one other figure in addition to the above, incomplete one. Something about how levels are counted? Practical code?}

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 larger system, such as a numerical weather model, the precise provenance of each mathematical step to convert is less relevant; it may suffice to simply 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 not prevent us from obtaining multiple levels of detail in 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 explanation (simple to complex) by simply selecting an appropriate abstraction level to generate explanations.

A simple explanation may be generated by simply climbing to one of the highest abstraction levels (e.g. level 1 in Figure 5-2); a detailed explanation may be generated at a lower abstraction level (e.g. level 2).

5.5 An Example

{I should probably plot out a detailed example of crawling and generating an explanation here. I don't really want to bring it back to the insurance example, as that probably deserves an entire chapter to itself, so I need another example.}

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 that I have laid out in previous chapters will be made obvious.

6.1 The Problem

As stated previously, consider an insurance agent, 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 greater than 3, Danny is ineligible for insurance.

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,

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

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 largely solved and that such algorithms are capable of populating belief states of appropriate propositions that are indexed in a database. This is a reasonable assumption to make as algorithms for entity and sentiment extraction from text and object and gesture recognition in images is an area of active research that is not particularly relevant to the problem of actually calculating a risk score to be addressed in this example.

As a result, the propositions that might be discovered by such a system are rather simple to express by simply populating the belief state of such propositions that may have been uncovered. Some of these propositions are given in Table 6.2.

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

As stated in Chapter 3, we may express propositions by virtue of creating appropriate partial propagator networks for each proposition and populating the value stored in the TMS in the appropriate belief state with the dependencies (i.e. the posts, photos, etc.) from which the belief was established. In our 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? For this, we may "tell" a particular belief state a value. The `tell!` function automatically constructs an update which is sent to the cell specified as the first argument of the function. The value is then given as the second argument, and the dependencies are all subsequent arguments. Thus, we might state that `(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 of `Facebook`.

In practice, we may simplify stating true, false, and unknown values by making appropriate functions to shorten 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, which are given in Figure 6-2. Each function automatically creates the propagator network for each proposition if it does not already exist. Otherwise, the

```
(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.

proposition is retrieved from a database.

6.3 Bootstrapping the System: Rules

Establishing the basic belief states is only half of the puzzle, however. The core of any problem solver are the rules and algorithms which allow it to actually draw conclusions, and those have yet to be established. To do so, we turn to Chapter 4, which demonstrated the construction of a number of problem solving rule components.

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

```
;;; Ground facts about Danny

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

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

(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 labelled 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

Rules can, of course, be more complex. Aintno's policy may state, for example, that whether or not Danny eats unhealthy food is only relevant if there is a need to determine whether or not he eats unhealthy food (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 is only half of the story. We must still have some 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 a general directive to find enough information to properly score Danny with the specific ways in which the information may be found (e.g. asking whether Danny eats unhealthy food). As such we may include a rule like that in Figure 6-5, which not only conditions the scoring of unhealthy eating habits, but also the establishment of the "need to know" on the general need for more information.

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

```
(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 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 inurance 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.

know". Though any implementer will necessarily need to tread with care to determine how far is far enough in determining 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 images and free-form text), 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" with respect to Danny's eating habits is actually quite indicative of the need for the level of inherent control that 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 some problems may be resolved through reliance on the rule mechanism as described in Chapter 4, such partial propagator networks are not the only networks that may be useful in solving problems. However, the propagator network model permits a vast number of network constructions which may be used to solve problems in ways that rules alone cannot accomplish. Accumulation is one such "alternative" network structure.

Unlike the basic scoring mechanisms described above, accumulation is an example of a complex operation that cannot be simply implemented by using rules alone, due to the need to "undo" an accumulation when premises change. For example, while an assumption of unknownness regarding whether Danny sky-dives contributes a risk factor of 0.5 to overall accumulated risk, if, at a later date, 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, which necessarily will alter the overall risk score (such as by increasing it from 3.5 to 5). Furthermore, the overall risk score must then reflect the sources of information contributing the *acceptance* of Danny's sky-diving which were not previously present.

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

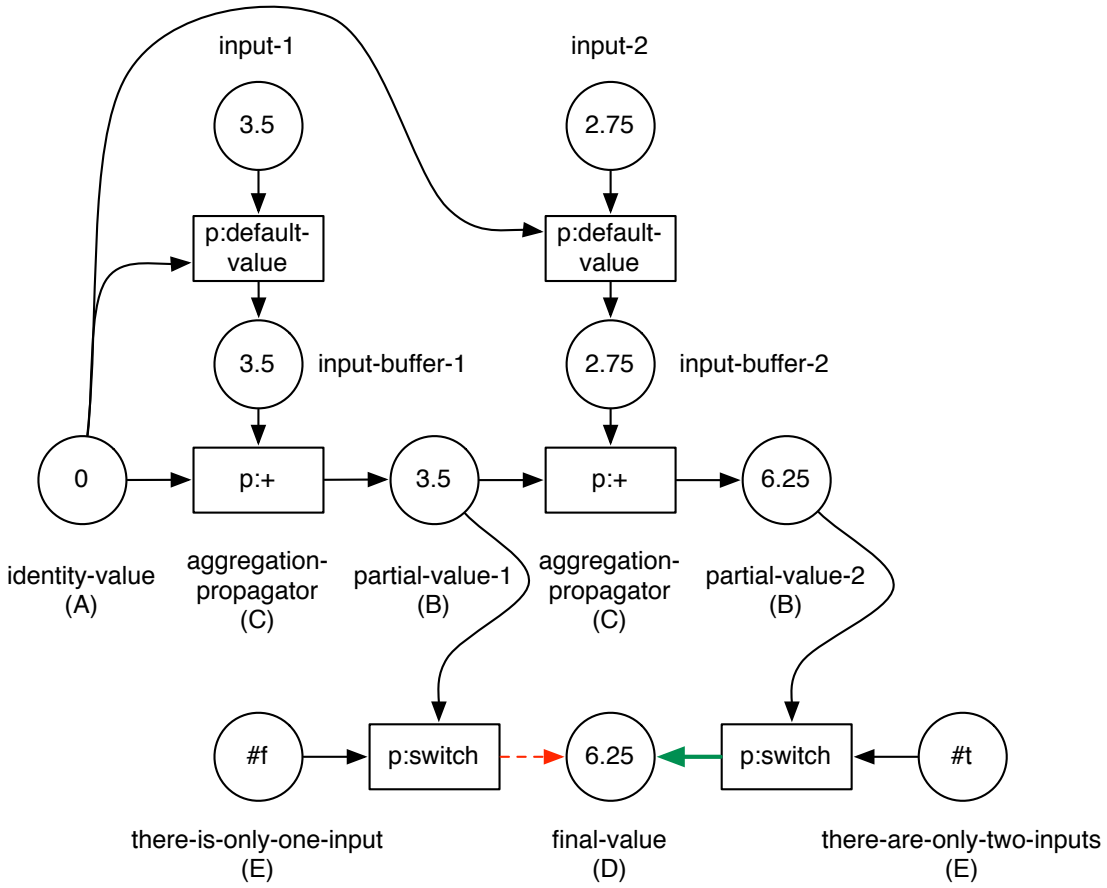

Figure 6-6: A propagator network which accumulates two values by addition (`p:+`). Partial values slowly accumulate until a `p:switch` connects a partial value (partial-value-2) to the final value based on the assumption that there are no more inputs to the accumulator.

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 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 using 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 from the output cell. Then, the newly constructed "final" partial accumulation 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 input buffer cell which sit between the contributing cell and the partially accumulated cell. The `p:default-value` propagator effectively provides a "default value" for an output cell whenever the primary input (e.g. 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 that partial accumulation at that point so that the actual contribution to the accumulation at the newly unsupported partial accumulation is effectively eliminated thanks to the use of the identity value for the operation, such as adding 0 or multiplying by 1.

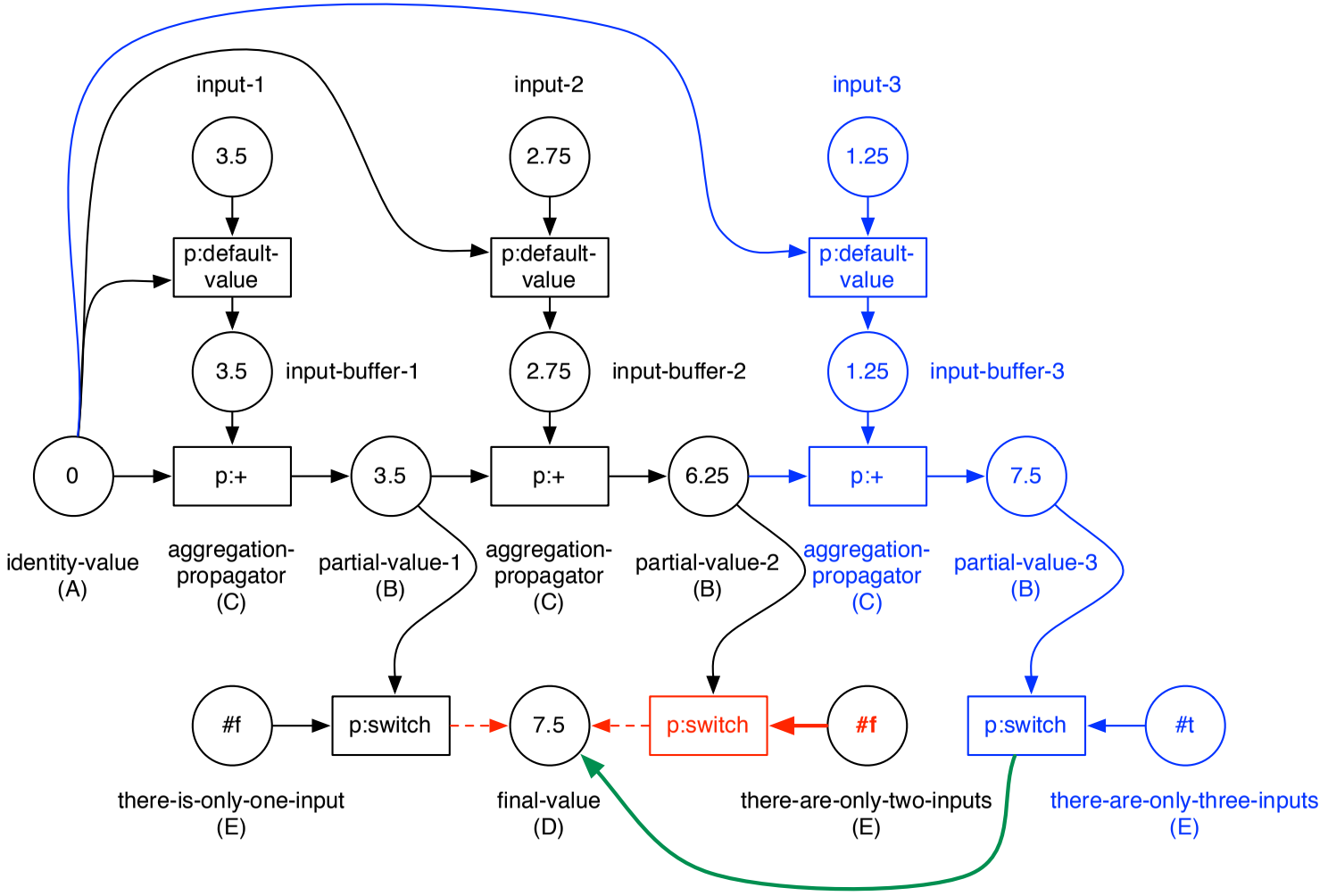

Figure 6-7: Adding a new input to a propagator-based accumulator. Here, input-3 is added as an input, and the assumption that there are only two inputs is kicked out so that the final value contains only the value of partial-value-3.

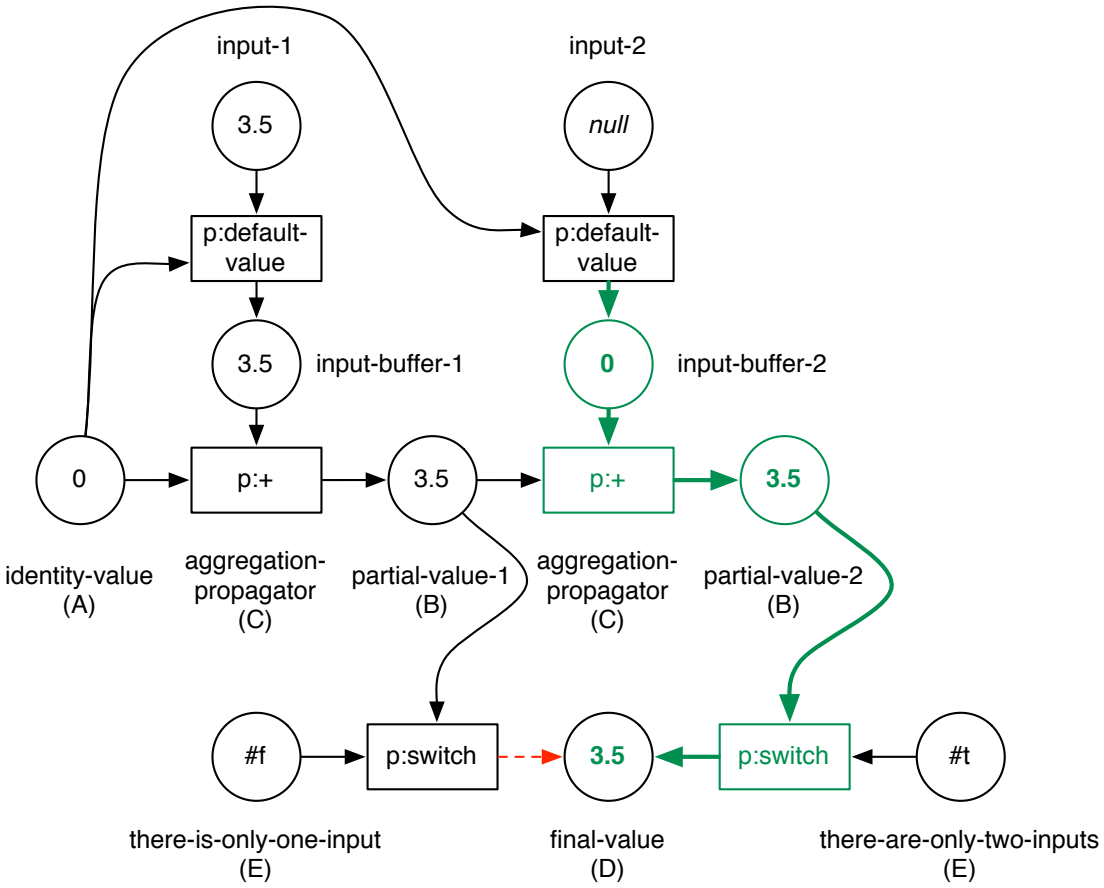

Figure 6-8: Removing a contribution from an input to a propagator-based accumulator. Here, input-2 is removed (its value becomes `*the-nothing*`, depicted here as *null*). This causes the `p:default-value` propagator it is connected to to instead propagate the value from the identity cell (0) to the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

We may, of course, simplify this partial network using a compound propagator, named `p:accumulator`. This propagator effectively handles accumulation and is invoked with five arguments. The first argument is a simple symbolic prefix used for labelling 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 simply using the code in Figure 6-9, which simply constructs an accumulator whenever a request for accumulation is found and accepted (`(please-accumulate ?type ?subject ?requestor)`). Then, for every proposition representing an accumulator contribution (`(contribution ?type ?subject ?num ?reason)`) which is believed to be accepted, a cell representing the value of that contribution is created, the value supported by the acceptance of that proposition. This value is then connected to the accumulator propagator 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 ?requestor ,accumulation)`, allowing other propositional pattern-matching to operate and make use of the value of the accumulation.

With such accumulation in hand, 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.

```
(rule ((r
(accepted
 (a-prop '(please-accumulate ?type ?subject ?requestor)))))
  (let* ((name (symbol (pattern-value ?type)
       ":accumulator-of:"
       (pattern-value ?subject)
       ":for:"
       (pattern-value ?requestor)))
 (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 ?requestor ,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 ?requestor)` 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.

6.4 The System in Operation

With the rule infrastructure and facts about Danny in place, we need only establish a need to determine Danny's insurability as an accepted proposition (`(true! '(please-determine-if Danny is-insurable-by AINTNO) 'ian)`), and the above rules and propagator mechanisms will work to solve the problem as to whether or not Danny is insurable.

Based on the 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 and establish contributions to risk based on Danny's engagement in risky hobbies as established by the beliefs established in Figure 6-2.

On their own, these facts are enough to deny Danny insurance, as noted in the fact that the rejected state of `(Danny insurance-issued-by AINTNO)` is held as true. We may, of course, 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 explanation also explains that this accumulated risk value is supported by the premises gathered from `facebook`, `flicker`, `danny`, and `risk:accumulator-of:danny:for:aintno:premise4` (i.e. the assumption that there are no more contributions to risk).

We can, of course, ask for more detail by digging into a more detailed explanation level (2), which gives the answer in Figure 6-12, demonstrating 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,

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

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

the system now accepts the need to look for more information to determine Danny's insurability (`(?company needs-more-information ?subject)`). As a result of the acceptance of this fact, the system now may accept the need to prove whether or not Danny eats unhealthy food (Figure 6-5), and as a result, will attempt to find proof of whether Danny eats unhealthy food (Figure 6-4).

With such rules on unhealthy food now in hand, 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 explanation can, of course, be obtained using the `explain` function at a suitably deep explanation 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 flicker 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 flicker 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 flicker)
 (((accepted (contribution risk danny 2. motorcycler)))
  has-value #t
  by ((((risk-estimate) (motorcycling)))
      ((accepted (danny engages-in motorcycling))))
  with-premises flicker)
 (((accepted (danny engages-in motorcycling)))
  has-value #t
  by (user)
  with-premises flicker)
 ((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 explanation 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 flicker 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)
  flicker 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 flicker)
 (((accepted (contribution risk danny 2. motorcycler)))
  has-value #t
  by ((((risk-estimate) (motorcycling)))
      ((accepted (danny engages-in motorcycling))))
  with-premises flicker)
 (((accepted (danny engages-in motorcycling)))
  has-value #t
  by (user)
  with-premises flicker)
 ((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 necessarily difficult to prove that such propositional reasoning is *viable* for expressing all kinds of problem solving strategies. Certainly the computational power of the underlying propagator networks is equivalent to that of a universal Turing machine {This would be a bit of an extended proof but it could be done... It's not really appropriate to include here though}, but mere ability to emulate any Turing-based problem solver does not necessarily mean that propositional approaches are natural or appropriate to all problems.

The true flexibility and feasibility of propositional problem solving can only be uncovered in the course of future work spent exploring and implementing problem solvers to solve different kinds of problems. Such work will necessarily uncover any weaknesses or flaws in the current system, whether they be based on flaws in current implementations of the underlying propagator networks or fundamental incompatibilities of the propositional modality with certain kinds of problems. In the process of developing the propositional problem solver system, however, I have identified a number of directions for future work needed to better determine and improve propositional problem solving which I discuss in detail in the remainder of this chapter.

7.1 Supporting Multiple Worldviews

A crucial assumption made in the implementation of propositions presented and described in this paper 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 even 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 in a way to draw an appropriate, concrete conclusion.

A problem, however, 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 different truth values associated with the 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 and as the contents of cells in a propagator network may only be evaluated with respect to one premise set at a time, there is no way to properly depend on two (potentially conflicting) premise sets.

This leaves two viable 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 cell each representing an agent and an associated basic belief (e.g. "john accepts", "john rejects", "jane accepts", "jane rejects"). Such a mechanism would certainly allow for a meaningful grouping of all beliefs relating to a single proposition within multiple world views, but it may complicate addressing and naming of individual belief states associated with a world view when each name

requires the knowledge of the agent who holds that state.

Alternately, we could represent each agent's view of the proposition as a separate 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. It is not immediately obvious which approach is better, given that both approaches effectively construct five additional belief state cells for each agent holding beliefs relating to a single proposition. As a result, regardless of the approach to representing beliefs in multiple world views with respect to a single proposition, both scale linearly and are unwieldy in their naming mechanisms.

Perhaps an ideal solution, however, would be to surface the underlying premise sets as cells on their own and have the propagators which make use of them in any computation (for example 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.* That is, Jane and John may both accept the proposition `(jack parent ben)`, but they may instead disagree on the implication that such an acceptance necessarily implies that one must accept `(jack ancestor jim)`. If this is done blindly, 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.

7.2 Proof by Contradiction

Although the problem of accumulating values may be implemented using the underlying propagator architecture and connected to the propositional reasoning engine as demonstrated in Chapter 6, other kinds of problem solving algorithms may not lend

themselves easily to the implementation of the propagator paradigm as described in this paper.

One such kind of problem solving is of particular interest to the propositional paradigm. While Chapter 4 demonstrated the viability of rules as a mechanism to connect the belief states of propositions to each other through implication relationships, other forms of proof exist in symbolic logic beyond that of *modus ponens* implemented using rules. Unlike *modus ponens*, however, the principle of *reductio ad absurdum* cannot be implemented using a simple propagator network where each cell contains the reasons for supporting a particular belief state, as effective application of proving a contradiction requires creating 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 the proposition is believed to be rejected, and we must determine whether such a rejection implies a contradiction.

Thus, proving a belief by contradiction depends not only 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 that depends on the contrary belief necessarily creates an effect in the original worldview by supporting the belief that we wished to prove in the first place.

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

7.3 Contributions

{Need a better name to tie this all together}

Despite these two flaws, the concept of propositional reasoning is compelling. By modelling beliefs separately from the structure of the problem solver itself, it is possible to inject complex control into problem solving and solve a wide variety of problems using different mechanisms. By simply using knowledge of a solution, for example, it is possible to effectively control and limit the nature and extent of problem solving. This was 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 until it was impossible to deny Danny's insurance by any other mechanism.

Such a mechanism permits intelligent selection of goals and simultaneously provides flexibility in problem solving (there are no constraints in how or in what order Danny's risk must be scored) while still permitting a level of control to effectively include hidden costs in the process of problem solving (for example, 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 the automatic addition of complex explanation generation based on the structure through which information and beliefs flow. By grounding such explanations in an integral part of a semantic-rich underlying programming substrate, explanations are obtained at little cost and can be used to obtain meaningful explanations 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 complex reasoning techniques such as proofs by contradiction to the underlying propagator architecture, propositional reasoning appears to be a functional approach to general-purpose problem solving solutions which may be useful in a decentralized, flexible world. {Better tie this back to the intro}

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, the propositional system is interactive through the standard read-eval-print-loop of MIT/Scheme. This appendix contains an extended 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) 'Flicker)

(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
  flicker
  danny
  risk:accumulator-of:danny:for:aintno:premise4)
 ((out) has-value
        5.15
        by
        ((p:accumulator) (0))
        with-premises
        facebook
        flicker
        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 flicker 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 flicker 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 flicker)
 (((accepted (contribution risk danny 2. motorcycler)))
  has-value #t
  by ((((risk-estimate) (motorcycling)))
      ((accepted (danny engages-in motorcycling))))
  with-premises flicker)
 (((accepted (danny engages-in motorcycling)))
  has-value #t
  by (user)
  with-premises flicker)
 ((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 flicker 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 flicker 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 flicker 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 flicker 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 flicker 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 flicker)
((risk:accumulator-of:danny:for:aintno:contribution:2)
 has-value 2.
 by ((switch:p) ((accepted (contribution risk danny 2. motorcycler))) (2.))
 with-premises flicker)
(((accepted (contribution risk danny 2. motorcycler)))
 has-value #t
 by ((((risk-estimate) (motorcycling)))
     ((accepted (danny engages-in motorcycling))))
 with-premises flicker)
(((accepted (danny engages-in motorcycling)))
 has-value #t
 by (user)
 with-premises flicker)
((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
                flicker
                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
                flicker
                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 flicker 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)
  flicker 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 flicker)
 (((accepted (contribution risk danny 2. motorcycler)))
  has-value #t
  by ((((risk-estimate) (motorcycling)))
      ((accepted (danny engages-in motorcycling))))
  with-premises flicker)
 (((accepted (danny engages-in motorcycling)))
  has-value #t
  by (user)
  with-premises flicker)
 ((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, 2000.

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

[3] K. Forbus and J. deKleer. *Building Problem Solvers*. MIT Press, 1993.

[4] 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*, 39:3 – 36, 1999.

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

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

[7] 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.

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

[9] Albert A. Michelson and Edward W. Morley. On the relative motion of the earth and the luminiferous ether. *Journal of Human Evolution*, 34(203):333 – 345, 1887.

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

[11] Marvin Minsky. *The Society of Mind*. Simon and Schuster, 1988.

[12] Marvin Minsky. *The Emotion Machine*. Simon and Schuster, 2006.

[13] 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, 1977.

[14] 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.

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

[16] Patrick Suppes. *Introduction to Logic*. Van Nostrand Reinhold Company, 1957.

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