Cognitive Equations
Fixed-Point Finding as an Intermediate Layer in Cognitive Architecture

by

Abram Demski

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(Computer Science)

October 2021

Acknowledgements

In 2010, I attended a Soar workshop in Ann Arbor, MI. It was there that I met Dr. Rosenbloom. My gratitude goes to him for believing in my potential as a student. I would also like to thank him for many stimulating discussions over the years, and for reliably providing helpful feedback on my work. I also have him to thank for running the Sigma experiments which appear in this work. I would also like to thank my other committee members for reading and evaluating my work, including Sven Koenig and Morteza Dehghani, and also my qualification committee, which further included Jerry Hobbs and Yan Liu. My thanks also go to Lizsl De Leon for helpful guidance through the process of being a student at USC, as well as all the staff at ICT.

Table of Contents

Acknowledgements **ii**

List Of Figures **v**

Abstract **vii**

Chapter 1: Introduction **1**

Chapter 2: Related Work **11**

2.1 Hybrid Cognitive Architectures 11
2.2 Probabilistic Graphical Models 14
2.3 Fixed-Point Equations and Weighted Logic Programs 18
2.3.1 Halide 20
2.3.2 Hybrid Reasoning in WLP? 22
2.4 Lifted Reasoning 23

Chapter 3: Design **27**

3.1 Overview 27
3.2 The Signia Language 29
3.2.1 First-Order Syntax 29
3.2.2 Second-Order Language 39
3.2.2.1 Second-Order Variables 39
3.2.2.2 Second-Order Arguments 42
3.2.2.3 Second-Order Constraints 43
3.2.2.4 Universal Variables 44
3.3 Spatial Trees 47
3.3.1 BSP Trees: Representing Hyperreal Functions 47
3.3.1.1 Design Considerations 48
3.3.1.2 Representation Details 52
3.3.2 Supporting Variable Dimensionality 57
3.4 Compilation, Initialization, and Execution 62
3.4.1 Compilation 62
3.4.2 Initialization 65

3.4.3 The Execution Cycle 66
3.4.4 Why Merge Results? 70
3.4.5 Why Not Combine Directly? 72
3.5 More Design Details 74
3.5.1 Anti-Signatures 74
3.5.2 Comparison 76
3.5.3 Combination 80
3.5.4 Summarization 83
3.5.4.1 Leaf Summaries 87
3.5.4.2 Problems Integrating TSplits 89
3.5.5 Trimming 91
3.5.6 Merging Ambiguous Cases 94
3.6 Design Alternatives 96
3.6.1 Hyperreals & Constraints 96
3.6.2 Hyperreals vs. Generalized Functions 99
3.6.3 Eliminating Spatial Trees 100

Chapter 4: Analysis **101**
4.1 Stating Algorithms Generically 101
4.1.1 Speed Evaluation 108
4.2 Lifting & Grouping 116
4.2.1 Grouping Like Things Together 117
4.2.2 Grouping Across Predicates 120
4.2.3 Explicit Generalization 122
4.2.4 Grouping Across Explicit Generalizations 125
4.2.5 Automatically Lifting Algorithms 128

Chapter 5: Conclusions & Future Prospects **131**

Reference List **136**

List Of Figures

2.1 $F(a,b,c,d) = f(a,b)g(b,c,d)$ 15

3.1 Constraints Illustrated 34

3.2 Transitivity Output 35

3.3 Merged Result 37

3.4 Example of Region-Array Inefficiency 51

3.5 `BSPTree` and Associated Data-Structures 53

3.6 Sharing Structure 55

3.7 The `RuleSystem` Data-structure 63

3.8 The `Rule` Data-structure 64

3.9 The Execution Cycle 67

3.10 The tuple signature data-types. The type `TSig` is a union of the two types above. 75

3.11 Two `BSPTree`s which are `epsilonequal`, but not `==`. 77

3.12 Left: a `BSPTree`. Right: the result of applying `unfoldboundary` to the `BSPTree`, with `None()` as the second argument. Light blue lines indicate where items were sent. 86

3.13 A sketch of a `BSPTree` containing an inconsistent branch. 92

4.1 BP in WLP, attempt #1. The constraints which would appear in the first rule are not shown. 102

4.2 BP in WLP, attempt #2. Since this is still completely written in first-order WLP, the BP propagation rules must be re-stated in each place they are applied. 104

4.3 BP in WLP, attempt #3. Here, second-order rules are used to make BP generic, so that it can be applied to arbitrary examples without repeating the propagation rules. 105

4.4 Specifying a BP model which applies the generic BP rules from Figure 4.3 to a simple factor graph. 108

4.5 A simple example of a probability function of y given x. 109

4.6 First-order rules implementing a model similar to Figure 4.4. 111

4.7 Performance of Signia vs arrays on very sparse data, in three different conditions. 118

4.8 Comparison of Sigma and Signia on sparse data of higher dimensionality. 119

Abstract

A cognitive architecture is a hypothesis about the fixed structure underpinning intelligence. The field of cognitive architecture emphasises computational implementations of such architectures, encouraging concreteness. However, the implementation of such systems, which aim to demonstrate a wide variety of capabilities rather than achieve high performance on a narrower set of tasks, is quite challenging.

Architectures can be multi-layer, in the sense of specifying lower levels which implement higher levels. An *interface layer* is a layer in such a multi-level design which is especially simple, providing a clean interface between complications above and below. It has been suggested that an interface layer would accelerate development in artificial intelligence generally [7, 6] and cognitive architecture in particular [32].

Weighted logic programming (WLP) has previously been proposed as a general formalism which allows a wide variety of statistical algorithms to be conveniently specified [9]. The present work explores WLP as an interface layer for cognitive architecture.

First, a language extending WLP is developed. The challenge is to provide a version of WLP which is suitable for cases of interest to cognitive architecture. This is accomplished through a *second-order* WLP, allowing for quantification over predicate variables and variable-length argument lists to predicates. This language allows algorithms to be specified in sufficient generality, so that it is reasonable to expect complicated cognitive architectures could be written in WLP. Specifically, first-order WLP is only able to apply to predicates explicitly mentioned; second-order WLP is able to apply to as-yet-unknown predicates. This is critical for specifying general cognitive rules such as probabilistic reasoning.

Second, a data-structure is provided which extends spatial trees to variable-dimensional and uncertain-dimensional data. This allows for the storage of information manipulated by second-order WLP. Moreover, this data-structure facilitates sparse and lifted reasoning.

Third, the utility of this interface layer is tested via the example of belief propagation. Second-order WLP allows a very brief statement of belief propagation, which nonetheless has a high degree of generality.

Chapter 1

Introduction

A cognitive architecture is a computational hypothesis about the fixed structure underlying cognition. A major goal of cognitive architectures is to provide a general framework for artificial intelligence, supporting any number of specific AI tasks. However, in contrast to a toolkit approach, a cognitive architecture tries to tell a coherent story about the nature of intelligence.

The current work inherits much of its conceptual focus from the Sigma cognitive architecture [35]. The Sigma cognitive architecture aims to show how diverse behaviors can arise from a small set of primitives. This involves bridging several conceptual divides, including cognitive vs. sub-cognitive, symbolic vs. non-symbolic, logical vs. probabilistic, and continuous vs. discrete [33].

Cognitive and sub-cognitive: The "central" or "higher" functions (conscious deliberation, planning, meta-cognition, declarative and episodic memory, etc.) vs. the "peripheral" or "lower" functions (perception, motor control, etc.) [23]. This distinction is mentioned here not so much to endorse it, as to state that the present work (together with Sigma) aspires to support both types of processing. IE, term "coginitive architecture" should not be taken to exclude sub-cognitive information processing for our purposes.

Symbolic and nonsymbolic: Knowledge representations with a highly "local" character (in which meaning can typically be ascribed to small pieces), vs. knowledge representations with a "distributed" character (in which the meaning is spread over many features). A central example of the first cluster is symbolic logic, whereas a central example of the second is the artificial neural network. Note that this is does not *necessarily* have a close connection with a cognitive/subcognitive distinction.

Logical and probabilistic: A logical representation is *relational*: there are entities, and different relations between these entities. Logic concerns itself with the structural relationships between beliefs. Probabilistic reasoning, on the other hand, concerns itself with *degrees of belief* and with *evidence.* Note that logical/probabilistic is not a true dichotomy; the two are merely not often seen together.[1] Their combination is the subject of *statistical relational learning* [41].

Continuous and discrete: Support of both discrete and continuous representations aids the crossing of the divides mentioned thus far, as well as helping to facilitate a broader variety of models within a system.

[1]Tools for dealing with the complexities of relational/logical structure have historically been "crisp", dealing only with the Boolean values {0,1}, while tools for dealing with probability/statistics deal with fractional values, in some cases requiring values to be *strictly* between 0 and 1 for reasons such as ergodicity.

Of course, other distinctions between diverse processing types could be pointed to, but these are particularly relevant to the motivation here. The present work will use the term "hybrid cognitive architecture" (nonstandardly[2]) to refer to a cognitive architecture which facilitates all of the above processing types.

In attempting to support multiple types of processing, one might hypothesize separate modules or mechanisms, or, one might instead aim to unify the capabilities – positing underlying mechanisms which give rise to diverse processing types. Sigma is firmly in the latter camp.

Sigma's strategy is to build capabilities on top of a *graphical layer*: a single graphical model which provides the substrate in which the core reasoning takes place.

This has some similarity to Pedro Domingos' call for an *interface layer* for AI, embodied by the Alchemy system [7, 6]. An interface layer is a layer which facilitates diversity in both "lower" architectural layers (the layers which the interface layer is *implemented on top of*), *and* diversity in "higher" layers (the layers *implemented on top of* the interface layer). For example, the LLVM (low level virtual machine) is an interface layer in this sense: it is implemented on many different chip architectures, and many different programming languages compile to LLVM [22].

[2]The Sigma literature uses somewhat more standard terminology, arriving at the term "grand unified mixed hybrid cognitive architecture" to describe systems which bridge three of the four distinctions I mentioned [35]. Using a similar approach, the full four could be described as "grand unified mixed hybrid statistical relational". This seems cumbersome.

Both Sigma and Alchemy hypothesize that graphical models can serve as an interface layer, and both projects utilize a very general type of graphical model known as *factor graphs* in order to represent graphical models. Factor graphs are mainly used to reason about probabilities or probability-like energy functions, although the possibilities are much broader [19].

Alchemy's interface layer, *Markov logic*, supports logical and probabilistic reasoning together in one system. The two are deeply integrated in a unified formalism, which allows for a broad range of models combining the strengths of these two types of reasoning. In other ways, however, Alchemy is restrictive. Bayesian probabilistic reasoning is assumed as an overarching framework. This makes other approaches, such as neural networks, unnatural in Alchemy – or even certain probabilistic models, if they don't fit with the factor graph formalism (such as in [5]).

Cognitive architectures are complicated endeavors, which tend to have long development cycles. Work begins with some some set of assumptions. Implementation proceeds, often with one core developer (with only limited time for development, due to other responsibilities) and others who come and go. The implementation will tend to proceed in a way which allows experiments of interest to happen as soon as possible, but the feedback from these experiments is necessarily limited: a cognitive architecture is a theory of how a whole mind hangs together, so the ultimate test of the early architectural choices cannot be performance on a particular task, as is common in machine learning[3] – instead, one must look at the overall

[3]Unless, perhaps, a task obviously requires a great variety of capabilities for success.

picture after a significant variety of tasks have been successfully tackled.[4] This may realistically be after more than a decade. To name a prominent example, the major assumptions behind the Soar cognitive architecture were re-evaluated after about 20 years of working within the initial core assumptions [20].[5]

By the time evidence accumulates, the questionable assumptions may be entangled with many elements of the codebase, so that re-working those assumptions entails a very significant effort (or, starting anew). This works to delay experimentation with new assumptions even further.

For this reason, it may be helpful to keep the interface layer highly approach-agnostic, so that different architectural assumptions could be tested without re-working the interface layer. An approach-agnostic graphical layer is also more likely to have broad appeal, IE, be useful for multiple different projects.[6] (It should be noted, on the other hand, that approach-agnostic interface layers may be less efficient than those optimized for specific types of reasoning.)

Sigma's interface layer is somewhat more approach-agnostic than Alchemy's; most strikingly, it has proven capable of supporting neural networks [34]. However, its breadth compromises some degree of unity. The the graphical layer takes the

[4]The evaluation of cognitive architectures is a complicated matter. One might say, pessimistically, that the only way to know is via the creation of human-level artificial intelligence. The situation isn't that bad, but, this sentiment points at an important problem: even considering the modern successes of artificial intelligence, how can we guess which ideas will ultimately contribute to our understanding of human intelligence?

[5]The Soar architecture went through many less significant revisions over this time period. I only mean to establish that for core architectural assumptions, things can take this long to play out.

[6]This kind of broad appeal is clearly desirable the vision of an 'interface layer for AI' as discussed in [7].

message-passing concept of the summary-product algorithm (see Section 2.2), and generalizes it by allowing a variety of computations in the message-passing. This does not preserve the overall semantics of factor graphs, which makes it harder to reason about what is being computed by the graph. The overall result is a graphical layer which has a structure *inspired by* factor graphs, but which also includes other behavior, without an underlying theory of how different types of reasoning should interact. See [34] for further details.

Weighted logic programming (WLP) has been proposed as a 'non-probabilistic programming language for probabilistic AI' [8]: a formalism in which many probabilistic models can be expressed conveniently, yet, one which does not force a probabilistic approach. Like Sigma's graphical layer, WLP can be seen as a generalization of the summary-product algorithm. However, WLP comes with a strong story about what is being computed: a WLP program can be understood as a set of equations for which we seek a fixed-point, through iteration methods. This general story can be seen to justify the behavior of Sigma's graphical layer to a large extent, while providing further generalizations.

The present work supports the thesis that ***weighed logic programming can serve as an interface layer for hybrid cognitive architecture, allowing lower-layer implementation to proceed without strong commitments to a particular paradigm.***

The contributions of the present work are as follows:

1. Design of a language for expressing a variant of weighted logic programs, suitable for the support of hybrid architectures, and allowing architectural principles to be stated in a sufficiently abstract manner.
2. Design of a spatial tree data-structure capable of supporting the hybrid reasoning expressed by this language.
3. An implementation of these designs.

These designs and implementation will together be referred to as the Signia system.[7]

The Signia language is discussed in Section 3.2. While similar to Dyna's WLP, discussed in [9], it takes a significantly different approach to reflection (IE rules about rules). Dyna supports the manipulation of Dyna models as data within other Dyna models, a feature which goes unsupported in the present work. Signia instead proposes and implements *second-order WLP*, which allows rules to apply to many predicates. In 'basic' WLP (ie, first-order WLP; see Section 3.2.1), one would need to re-state the same rules of inference over and over for each predicate where they need to be applied. While this is not a significant limitation when programming *specific models* in WLP, it becomes a limitation when writing general principles which should apply to many predicates. Hence, Signia provides features which allow a rule to be written *as a rule of reasoning which applies to arbitrary*

[7]The name Signia refers to the combination of ideas from **Sig**ma and Dy**n**a, and written in the language Jul**ia**. It also connotes insignia, "distinguishing mark or sign" – which pays homage to the graphical-model roots of the line of research.

predicates. This allows algorithms to be written "generically," that is, without prior commitments about which predicates those algorithms will be applied to (see Section 3.2.2).

Signia also generalizes WLP to *continuous domains*: whereas Dyna works with finite grids of numbers, Signia is able to handle piecewise-constant functions made up of convex regions with arbitrary-angle linear boundaries. This serves Signia's goal of supporting hybrid reasoning. See Section 3.3.1 for details of how this is accomplished.

Another way in which Dyna's version of WLP is inadequate for the hybrid reasoning targeted here is its lack of support for *lifted reasoning*. Lifted reasoning refers to the ability to take advantage of logical structure such as is provided by first-order quantifiers to reason about a domain without explicitly reasoning about every item in the domain [4, 38, 16, 39, 25]. This capability may be important in a cognitive architecture context, as the ability to work with these kinds of abstraction is observed in human-level intelligence (although it's possible that humans accomplish this in a very different way).[8]

Lifted belief propagation is an inference algorithm which computes the result of belief propagation over a logically structured factor graph, but without completely instantiating it [38]. Belief propagation is important for Signia because it is a

[8]This work holds no intention to claim that lifting is represented in a similar way in the brain. Rather, the value of this work is in *supporting* ease of development for many types of structures which *themselves* are hypothesized to relate to human cognition.

central example of the summary-product algorithm, which is the source of *Sigma's* generality.[9] Lifted belief propagation is also a cornerstone of Alchemy.

The structure of the rest of this document is as follows: Chapter 2 will put the work in context by providing more background detail. Section 2.1 will discuss hybrid architectures. Section 2.2 will discuss graphical models, focusing on factor graphs and the summary-product algorithm. Section 2.3 discusses weighted logic programming. Section 2.4 discusses lifted reasoning.

Chapter 3 details the Signia system, starting with the Signia language in Section 3.2, and the Signia data-structures in Section 3.3. Both of these sections are split up to first discuss the first-order part of the design (subsections 3.2.1 and 3.3.1), and then the second-order part (subsections 3.2.2 and 3.3.2). Section 3.4 puts all those pieces together by discussing compilation, initialization, and execution. Section 3.5 ties up some loose ends by explaining details which, while important, would have interrupted the flow of earlier sections. Finally, Section 3.6 discusses alternative design choices and some drawbacks to the current design.

Chapter 4 discusses the successes and failures of Signia in terms of its overall design goals. Section 5 demonstrates the ability to write algorithms in a generic way, by doing so for the belief propagation algorithm. Subsection 4.1.1 applies this algorithm to small toy examples to evaluate its efficiency. Section 4.2 examines Signia's sparse and lifted reasoning.

[9] Sigma doesn't use *lifted* belief propagation to enable logical reasoning, however; instead, it uses a custom modification of the summary-product algorithm, as discussed in [34].

Finally, Chapter 5 wraps up, summarizing results and discussing prospects for future research.

Chapter 2

Related Work

Signia builds on a long tradition of cognitive architecture work, but primarily, ideas from the Sigma cognitive architecture. The details of Sigma will not be explained in this document, but in order to convey the motivation for this thesis, much of the motivation for Sigma will need to be explained. To this end, literature on hybrid cognitive architectures will be reviewed in section 2.1, followed by relevant background on probabilistic graphical models in 2.2. Section 2.3 will discuss weighted logic programming, the formalism for fixed-point equations which is used for the fixed-point layer in Signia. Section 2.4 will discuss lifted inference in graphical models.

2.1 Hybrid Cognitive Architectures

The *physical symbol system hypothesis* states that a physical symbol system has the necessary and sufficient means for intelligent action [28]. It has been a major theme, and occasional point of contention, in cognitive architectures. The account in [27] describes it as a pervasive assumption, and one which had been spelled out by logicians long before the advent of artificial intelligence, as early as 1927. Challenges to the hypothesis are discussed in [23], which also includes an early hybrid system which utilized both symbolic and nonsymbolic reasoning. (The paper argues that the terms "symbolic" and "subsymbolic" are misleading, implying a particular relationship between the two within an architecture. Instead, this is split into a symbolic/nonsymbolic distinction, together with a cognitive/subcognitive distinction.) A recent summary of criticism of the physical symbol system hypothesis [29] concludes that none of these attacks are successful, but does indicate a consensus that there must be a "symbol-to-signal conversion" providing an interface between symbolic and nonsymbolic processing.

For the purpose of this document, "symbolic" is understood to mean representations with *local* meaning, such as object-oriented representations (where "objects" have "properties"). In contrast, "nonsymbolic" is taken to mean *distributed* representations, such as semantic vectors extracted from artificial neural networks [24].

Traditional cognitive architectures are primarily symbolic in nature, and must interface with external modules for tasks such as speech processing which (as a matter of consensus) require nonsymbolic reasoning [14]. Other architectures (which might be generically termed connectionist) approach nonsymbolic reasoning exclusively, claiming that symbol-like behavior will emerge from a sufficiently sophisticated nonsymbolic system. This is a fascinating hypothesis, and there has been a recent attempt to articulate and explore it, with some success [2]. However, even if true, such systems make it relatively difficult to examine and manipulate such emergent symbols. Hybrid architectures attempt to capture both symbolic and nonsymbolic parts of intelligence explicitly. A recent discussion of the spectrum of approaches is in [12].

Very recently, the *common model of cognition* (formerly “standard model of the mind”) has emerged as a partial consensus of the cognitive architecture community, especially the ACT-R, Sigma, and Soar groups [21]. The common model of cognition does not represent full agreement on every question, but rather, a significant subset of questions on which there is now agreement – where previously, agreement was lacking. Among the many points of agreement which have emerged is agreement on the need for a hybrid architecture.

Most hybrid architectures have a modular design, implementing symbolic and nonsymbolic processes in independent subsystems, as in [23]. [40] provides a somewhat more integrated approach by separately implementing symbolic and nonsymbolic parts for each cognitive subsystem. Dual, a society-of-mind style architecture, takes this yet further by implementing both symbolic and nonsymbolic functionality for every *agent* in the system [17]. This style of unification still creates a strong distinction between symbolic and nonsymbolic reasoning, but mixes both more thoroughly throughout the system.

Sigma takes a different approach, seeking a unification of symbolic and non-symbolic rather than separate mechanisms supporting each. The graphical layer of Sigma is general enough to implement both symbolic and nonsymbolic computations.[1] Signia takes a very similar approach, introducing representations which unify and generalize diverse types of reasoning.

The terminology in the Sigma group is slightly different from what's been used in this document. Sigma is called a *mixed hybrid* architecture; mixed refers specifically to the ability to use both deterministic and probabilistic models, whereas hybrid refers to the combination of discrete and continuous processing. As mentioned in the introduction, this document will instead use "hybrid" to refer generically to bringing together diverse types of reasoning (including deterministic/probabilistic, statistical/relational, discrete/continuous, and symbolic/nonsymbolic.)

The main tool which Sigma uses in its pursuit of unification is the graphical model, to be discussed next.

2.2 Probabilistic Graphical Models

The core design of Sigma is to re-build the older symbolic style of cognitive architecture *out of* the newer graphical-model style reasoning, to create a system which

[1]An example of significant non-symbolic reasoning in Sigma is the distributed vector representation in [42].

supports both. Sigma uses *factor graphs* as its graphical model of choice, due to their generality.

A factor graph is a bipartite graph, with one set of nodes corresponding to variables (*variable nodes*), and the other set corresponding to functions on those variables (*factor nodes*) [19]. The factor graph represents the multivariate function which is the product of all of the local functions. Factor graphs therefore provide a way to decompose complicated multivariate functions and represent them as a product of local factors. A variable node is linked to a factor node if the factor function takes the variable as an argument, showing the structure of dependencies. For example:

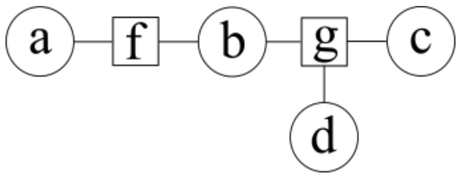


Figure 2.1: $F(a, b, c, d) = f(a, b)g(b, c, d)$

This network could be representing a probabilistic decomposition with $f(a, b) = P(a, b)$ and $g(b, c, d) = P(c, d|b)$. Like other graphical models, the factor graph displays independence information. (We can extract a Markov blanket from the structure.) Unlike others, it emphasizes the factorization over independence structure, illustrating it directly. This makes it appropriate for any factorized function, not only probability distributions. The link structure is useful for a number of message-passing algorithms. The most basic of these is the sum-product algorithm (also known as belief propagation, in the case where the network represents probabilities). We often wish to compute the *marginals* of the network: for each variable x_i, the function summing out all the other variables:

$$\sum_{x_1}\sum_{x_2}\cdots\sum_{x_{i-1}}\sum_{x_{i+1}}\cdots\sum_{x_j}F(x_1, x_2, ..., x_j)$$

This sum is exponential in the number of variables, and therefore intractable. To compute the value efficiently, we apply the distributive law to the factorization of F, pushing the sums in as far as possible so that each can be summed out independently. This gives us a linear-time message passing algorithm if the graph is tree-structured. (For more details, see [19].) It turns out, however, that the same message-passing algorithm makes a surprisingly good approximation for graphs which include loops as well. The messages can be understood as *pretending* that the graph has more independencies than it actually does, and iterating the messages until convergence. This gives us the sum-product algorithm [19]. The same algorithm is also known as loopy belief propagation, especially when the factor graph represents a probability distribution.

The sum-product algorithm can be generalized by replacing the sum and product operations with other operations. For example, an important variation is the max-product algorithm, which can be used to (approximately) find maximum-probability configurations rather than marginal probabilities. As long as the two operations form a commutative semiring, the algorithm is exact on tree-structured factor graphs. I will refer to these variations collectively as “the summary-product algorithm”. The generalized sum operations are sometimes called *summary* operations, since they give a compact result summarizing many items (like summary statistics). The generalized products can be called *combination* operations.

The summary-product algorithm generalizes a number of important algorithms in different domains: loopy belief propagation, turbo coding, constraint propagation, and the Viterbi algorithm – it has even been related to the fast Fourier transform [19].

To take advantage of multiple semirings, Sigma allows different summary and combination operations to be mixed within one graph. This compromises the theoretical justification of the summary-product algorithm (in terms of applying the distributive law to simplify an otherwise intractable computation), meaning that technically, we cannot view Sigma's graphical layer as a factor graph.[2] The present work reinterprets and generalizes Sigma's graphical layer, using the framework of fixed-point computations rather than factor graphs.

[2]The behavior in practice may or may not be well-understood as a factor graph computation, depending on the specific model Sigma is running. See [34] for more details on how Sigma's graphical layer relates to factor graphs.

2.3 Fixed-Point Equations and Weighted Logic Programs

The concept of fixed-point iteration comes from numerical analysis. Given an equation of the form $f(x) = x$, we may seek a solution by taking the limit of a sequence $x_1, x_2, x_3...$ such that $x_{n+1} = f(x_n)$. Many probabilistic inference algorithms such as belief propagation, mean-field approximations, and the EM algorithm are explicitly in this form. A variety of other algorithms can readily be put into this form.

Weighted logic programs (WLP) are a prolog-like syntax for systems of fixed-point equations [11]. WLP is a simple generalization of rule-based systems, but allows numerical algorithms to be easily formulated as fixed-point problems. This makes them a powerful formalism for the unification of diverse AI paradigms.

As already mentioned, Sigma is implemented on top of a factor-graph layer (see Section 2.2). The algebraic generalization of the sum-product algorithm computes approximate solutions to problems which can be stated in terms of two mathematical operations; a *summary* operation (which generalizes the sum), and a *combination* operation (generalizing the product), which together form a commutative semiring.

WLP can be seen as a generalization of this concept, using *aggregation operators* to specify summary-like computations. These are often composed with combination operations which form semirings, but need not be. The concept is similar to map-reduce computations: combination operates independently across many instances, like *map*, while aggregation takes many instances together in order to produce a result, like *reduce*. As in many probabilistic programming or probabilistic logic systems, WLP generalizes the propagation of Boolean values to include real numbers. *Unlike* most such systems, no particular semantics is assigned to the numbers - the user is specifying arbitrary propagation rules without *a priori* constraints on what the numbers should mean or how they should behave.[3] As discussed in Chapter 1, this makes WLP a promising candidate for an approach-agnostic interface layer for cognitive architecture, and artificial intelligence more generally.

Consider a traditional rule in Prolog syntax:

```
squashed(X) :- under(X,Y), heavy(Y).
```

This relates three predicates: `under`, `heavy`, and `squashed`. Any object having an `under` relationship with `heavy` items is `squashed`. Here is one possible weighted-logic version of the rule:

```
squashed(X) max= min(under(X,Y), heavy(Y)).
```

The three predicates are now real-valued, rather than Boolean-valued. The `max=` operator takes the max of all matching instances. This acts like Prolog's '`:-`' operator if `under` and `heavy` contain only Boolean values. In the body, the predicates are combined with `min` to simulate conjunction. (Multiplication would

[3]Note that this also differs from *fuzzy* logic, which (like probability theory) still has a specific theory of how values should combine.

have worked as well – it's all up to how the user wants to generalize the behavior of conjunction.)

Here is a more typical numeric operation with the same structure:

```
vector1(X) += matrix1(X,Y) * vector2(Y).
```

The `+=` aggregation operation sums across all matching instances of the body. As a result of this rule, predicate `vector1` will contain the result of multiplying the matrix stored in predicate `matrix1` by a vector stored in predicate `vector2`. The combination operation `*` is simple real-valued multiplication.

Aggregation operations can include `+=`, `*=`, `max=`, and `min=`. The combination, on the other hand, can be any numeric formula – although some may be specially optimized during compilation.

Like rule-based systems, WLP systems are computationally tractable while being very expressive. However, where rule-based systems excel only at symbolic reasoning, WLP is suited for a much larger range of problems. WLP naturally expresses dynamic programming algorithms and message-passing algorithms. A wide variety of AI algorithms can be expressed easily in this form (see [9]).

2.3.1 Halide

Although WLP is a concept taken specifically from the Dyna language, another language provides some evidence for the effectiveness of this as a distinguished layer as well: the Halide language [31]. Halide is a language for image processing, which formally separates the *algorithm* and the *schedule*. The algorithm is specified abstractly, using a WLP-like language which specifies only the nature of the values to be computed. A separate scheduling language indicates *how* those values will be computed. The algorithm language uses a concept of *reduction* which is distinct from WLP's concept of aggregation, but serves a similar role. The schedule language specifies how this computation will be implemented at the lower level: the order in which things will be computed, and the location where they will be computed (including CPU or GPU, and distribution across cores). This division allows Halide to beat the best hand-tuned C code in practice. How to write highly optimized code is rarely obvious before writing, because optimization requires tuning a number of trade-offs. In Halide, schedules can test trade-offs very rapidly without changing the core logic of the algorithm, whereas optimizing C code is often a matter of changing the critical inner loops. As a result, Halide allows the user to find non-obvious schedules by trial and error much more rapidly than in C. This also allows the same *algorithm* code to be easily ported to entirely different hardware, only re-writing the schedule part of the code.

Splitting a computation into two parts like this brings to mind the slogan "algorithm = logic + control" from [18] (which discusses the thesis in relation to Prolog). The notion of control information also appears in the problem space hypothesis, which has influenced cognitive architectures for some time (especially Soar and Sigma) [37].

Like WLP, Halide allows formulas which require multiple iterations to reach fixed-point; however, this is very restricted, since Halide is more focused on speed. This kind of iteration is not allowed to span across predicates (or functions, rather, as Halide calls them) and is strictly bounded.

Although no scheduling language has been explored in the present work, a combination of WLP with Halide-like schedules seems like an exciting prospect to be explored. The important issues of control information will be almost ignored in this thesis. Signia instead relies on a fixed WLP solver with its own simple control heuristics (described in Section 3.4).

2.3.2 Hybrid Reasoning in WLP?

As discussed in the introduction, the goal of the Signia system is to unite diverse types of reasoning, including symbolic, nonsymbolic, statistical, relational, continuous, and discrete.

By generalizing predicates to store real-valued information instead of merely Boolean, WLP makes significant progress in this direction. However, although the *ranges* are now continuous, the *domains* are still discrete. As a result, many problems would have to be discretized, or represented in clever special-purpose ways. Signia addresses this by making the domains continuous as well. This means values can no longer be stored in a simple table. Instead, a spatial tree format, discussed in Section 3.3, is used.

The foundation of Dyna is relational, being based on relational rule-systems. However, Dyna actually has no support for lifted reasoning; again, the values are stored in tables, which means every instance of a relation is explicitly represented. So, Signia faces the challenge of incorporating lifted reasoning with WLP.

2.4 Lifted Reasoning

In many domains, a graphical model has repeated structure which we would like to represent abstractly. For example, suppose we would like to reason about the health of a collection of people, each with the same probabilistic structure linking symptoms and diseases, but additionally having relationships with each other determining possible disease transmission. Rather than specifying every variable and factor in the system, we would like to use something like a template. (Furthermore, during learning we want to share parameters across this template.) This can be further complicated by multiple entity types, uncertainty about the number of entities, and so on. *Statistical relational learning* studies probabilistic models for such cases [41]. Here, we will collectively refer to these as *lifted models*.

At first, lifted models needed to be completely instantiated (a step known as propositionalization) so that standard inference procedures could be used. The resulting model could become very large. To avoid this, a variety of lifted inference algorithms were invented to reason on lifted structures more directly.

Lifted versions of a variety of probabilistic inference algorithms have been investigated. For our purposes here, it is most relevant to review the literature related to lifted belief propagation. The first attempt at this was [13], which performed lifted belief propagation on relational Markov networks in the case where the network is not conditioned on any evidence. This worked based on the high degree of symmetry in the no-evidence case; nodes of a particular repeated type will have identical incoming and outgoing messages as each other. Although very limited, this approach was useful for weight learning, which involves performing no-evidence inference. [38] took this idea and applied it to the case with evidence, by constructing a *minimal lifted network*: a network which groups nodes together by like messages when possible. The result is the *lifted belief propagation* algorithm. *Counting* belief propagation was introduced in [15]. It improved on lifted belief propagation by interpreting lifting as *purely* exploitation of symmetry in the (fully instantiated) network. This approach allows nodes to be *grouped together*, as opposed to only thinking of splitting apart relations minimally. It also applies to problems which were never given lifted models to begin with.

While lifting can produce large speedups, it can be very expensive. [16] noted that in some cases, the lifting step alone was taking longer than plain belief propagation would take. [39] propose a hypercube datastructure to reduce the cost, and also present *approximate* lifting algorithms which terminate the lifting step early. This results in a network which is smaller than the minimal lifted network, grouping some elements together despite different states of evidence. This can be done in a way that bounds the resulting error. Along similar lines, [16] observed that lifted networks tend to be very pessimistic, splitting apart all the messages that *could* differ. In a real network, long-distance dependencies can decay quite quickly so that messages become indistinguishable even for cases which lack perfect symmetry. *Informed* lifted belief propagation was introduced to address these issues. It interleaves lifting and message-passing, allowing messages to be grouped together based on actual similarity rather than pessimistic estimates, and reducing the overhead associated with lifting. Yet another approach to reducing the cost of lifting is presented in [25], which addresses the case of online inference. In this setting, new evidence is presented to the network regularly, and inference must be re-run. It would be very expensive to fully reconstruct the lifted network given each new state of evidence. ΔLNC is introduced, providing a way to track evidence changes and minimally update the lifted network in response.

Sigma does not attempt to perform lifted belief propagation in the sense of the above algorithms, but it does have many similarities. It uses something similar to the hypercube representation in [38] to group similar messages together. (More specifically, it utilizes piecewise linear representations, with a grid of orthotopes holding linear functions [33].) Like the approach in [16], it constructs this representation dynamically as message-passing progresses, breaking hypercubes as necessary and grouping them back together when they are indistinguishable. Sigma also saves work during on-line inference by keeping messages which have not changed, as in [25]. So, if *what Sigma was computing* using these tricks matched lifted belief propagation, its lifting algorithm would be combining several of the published optimizations. See Section 4.2 for a more detailed discussion of why Sigma cannot be said to be doing lifted reasoning in the sense intended here.

Lifted belief propagation algorithms guarantee that the results of inference are exactly the results of non-lifted inference, or a very good approximation. Sigma does not try to make lifted inference match or approximate a fully-instantiated version of the same model. This is due to the way Sigma's graphical layer generalizes factor graphs; while providing extended functionality beyond what can be achieved in a factor graph, it does not facilitate a close correspondence between lifted and fully instantiated reasoning. Details of Sigma's departure from lifted belief propagation are discussed in [34].

Signia, on the other hand, provides an even broader generalization of factor-graph behavior while also maintaining a guarantee that lifted reasoning closely matches the results of fully-instantiated reasoning. See sections 3.2.2.4 and 4.2.

Chapter 3

Design

3.1 Overview

In Chapter 1, WLP was identified as a candidate for an approach-agnostic interface layer. However, several problems with WLP were discussed in Chapter 1 and in Section 2.3.2:

- WLP cannot represent algorithms, such as belief propagation, in a general way which can apply to arbitrarily many predicates. It can only state the propagation rules for specific predicates, which would necessitate re-stating general rules of inference over and over.

- If predicate values are stored in simple tables[1], then predicates cannot take real-valued variables as arguments.

[1]By "table", I mean either an array, or a sparse data-representation such as a linked list containing the nonzero values.

- Storing the values in tables also means we are not doing lifted reasoning; every single instance must be represented.[2]

Signia addresses the first problem through *second-order WLP*. Section 3.2 describes Signia's WLP representation. In order to explain things gradually, the explanation starts with first-order WLP in Section 3.2.1, and presenting the second-order system in Section 3.2.2.

Addressing the second and third problem both involved switching to a spatial tree representation, described in Section 3.3. Like the language section, this is also split into a first-order part in Section 3.3.1, and a second-order part in Section 3.3.2.

Signia, like Dyna, attempts to solve the given fixed-point equations by a simple iteration method.[3] The values in all predicates are initialized to some very simple "guess", and then equations are re-computed (improving the estimated values for some predicate based on the stored values in other predicates). If this process reaches a point where nothing is changing any more, then a fixed-point must have been reached. This generalizes methods such as Dijkstra's algorithm and value iteration. Section 3.4 details how this works in Signia.

The descriptions in all these sections are high-level, in that they focus on getting the critical ideas across, and so do not provide quite enough information to re-write

[2]Or, in the case of sparse representations, every single *nonzero* (or more generally *non-default*) value must be represented.

[3]This is not guaranteed to converge. Other algorithms for solving fixed-point equations exist, which can converge in more cases. However, Dyna showed that a broad range of AI algorithms can already be represented in this way.

Signia from scratch. Section 3.5 provides many of the further details which would have been distractions if presented in earlier sections.

Finally, Section 3.6 discusses some mistakes which were made, and alternative design choices which may have been better.

3.2 The Signia Language

Signia is implemented as a domain-specific language within the Julia language.

The syntax represents a middle ground between two objectives: it aims to be as close to the simple equational specification of those rules as possible, while being easily compiled to Julia for execution.

It will be easier to explain the syntax in two parts, discussing the pure first-order fragment first and then explaining the second-order features which enable a kind of meta-programming in Signia. These are closely analogous to first-order logic and second-order logic.

3.2.1 First-Order Syntax

A 'program' in Signia is rule system, represented in Julia by the '`RuleSystem`' datatype.[4] Each `RuleSystem` is a stand-alone collection of rules, with its own working memory (where it stores the contents of predicates) and other important

[4]This is one of many datatypes defined by the Signia library, not a build-in Julia datatype.

data structures (see Section 3.4.1); any number of `RuleSystem`s can be defined, but they do not interact with one another.

A rule system is created with the `@rules` macro. For example:

```
@rules begin
  f(x,z) = @max f(x,y)*f(y,z)
  f(x,y) = @max [1<=x<2]*[3<=y<4] + [3<=x<4]*[5<=y<6]
end
```

Don't worry about what these rules mean for now; we will return to them later in this section.

Julia treats everything between `begin` and `end` as a block, which gets parsed into Julia syntax trees, but not compiled into Julia code. This is then fed to the `@rules` macro, which compiles the syntax tree into an instance of the `RuleSystem` data structure, and returns it. (For details of the compilation process and `RuleSystem` data structure, see Section 3.4.1.)

Each line in the block given to `@rules` is a rule. All rules follow the format

```
  <head> = <summary operation> <body>
```

where `<body>` can be an arbitrary expression, `<head>` is a predicate pattern (to be defined soon), and `<summary operation>` is anything from the list `@max`, `@min`, `@sum`, `@prod`, `@just`.

As discussed in the overview of WLP in Section 2.3, the summary operation is applied to any variables in the rule body which are not present in the rule head.

Most of these are obvious: `@max` takes the maximum value over any such variables; `@min` takes the minimum; `@sum` integrates variables out. `@prod` takes the product-integral, which is the product-based analog of the usual sum-based integral. `@just` is a special summary operation which throws an error if it is called, for use when no summary operation is desired.[5]

In the first-order fragment, a predicate pattern has the format `<predicate name>(<argument list>)`; for example, `f(x,y)`. The argument list is a comma-separated list of variables, possibly empty. Constant arguments, e.g. `f(x,1)`, are not currently supported.[6]

Both predicate names and first-order variables must start with a lowercase character. The list of predicate names is determined by looking at the names actually used in rule heads; for example, in the rule system

```
@rules begin
  a() = @just 1
  b() = @just a() + c()
end
```

[5]When a rule is written which has all the same variables in head and body, no summary operation will be applied during that rule's computation, regardless of what summary operation is specified; there is nothing to summarize! So, there is really no need for an explicit `@just` operation; such rules can use `@sum`, `@max`, etc. equally well. However, it is disorienting to see a rule which a spurious `@sum` which the user must infer is never used. Furthermore, explicitly adding `@just` has the advantage of enforcing one's *intention* to write a rule in which no summarization occurs – one is better off seeing an error than seeing erroneous results with no explanation as to where they came from.

[6]It wouldn't be difficult to support constant arguments, but, constraints allow essentially the same functionality.

the named predicates would be `a` and `b`. As a result, the occurrence of `a()` in the body of the second rule is interpreted as a predicate pattern, which means that in execution it is evaluated as a query to working memory (see Section 3.4). The occurrence of `c()`, on the other hand, is interpreted as a Julia function call instead, since `c` is not a named predicate.

A predicate may be *overloaded*, using a variable number of arguments – `f(x)`, `f(x,y)`, etc. The two-argument version should be thought of as a distinct predicate, with its own devoted place in working memory.

Although the rule body is arbitrary Julia code, certain items are detected and replaced to change the behavior of the code: predicate patterns, and constraints.

Predicate patterns in the rule body are replaced with queries to working memory, retrieving any contents which match[7] the particular predicate pattern (see Section 3.4 for details). Constraints, on the other hand, allow data to be constructed more directly.

To see how constraints work, let's consider again the rule system from earlier:

```
@rules begin
  f(x,z) = @max f(x,y)*f(y,z)
  f(x,y) = @max [1<=x<2]*[3<=y<4] + [3<=x<4]*[5<=y<6]
end
```

[7]The term "match" will be used to refer both to matching working memory for *retrieval* – taking things from working memory to perform a computation – and *storage* – matching working memory locations in order to over-write them with new results.

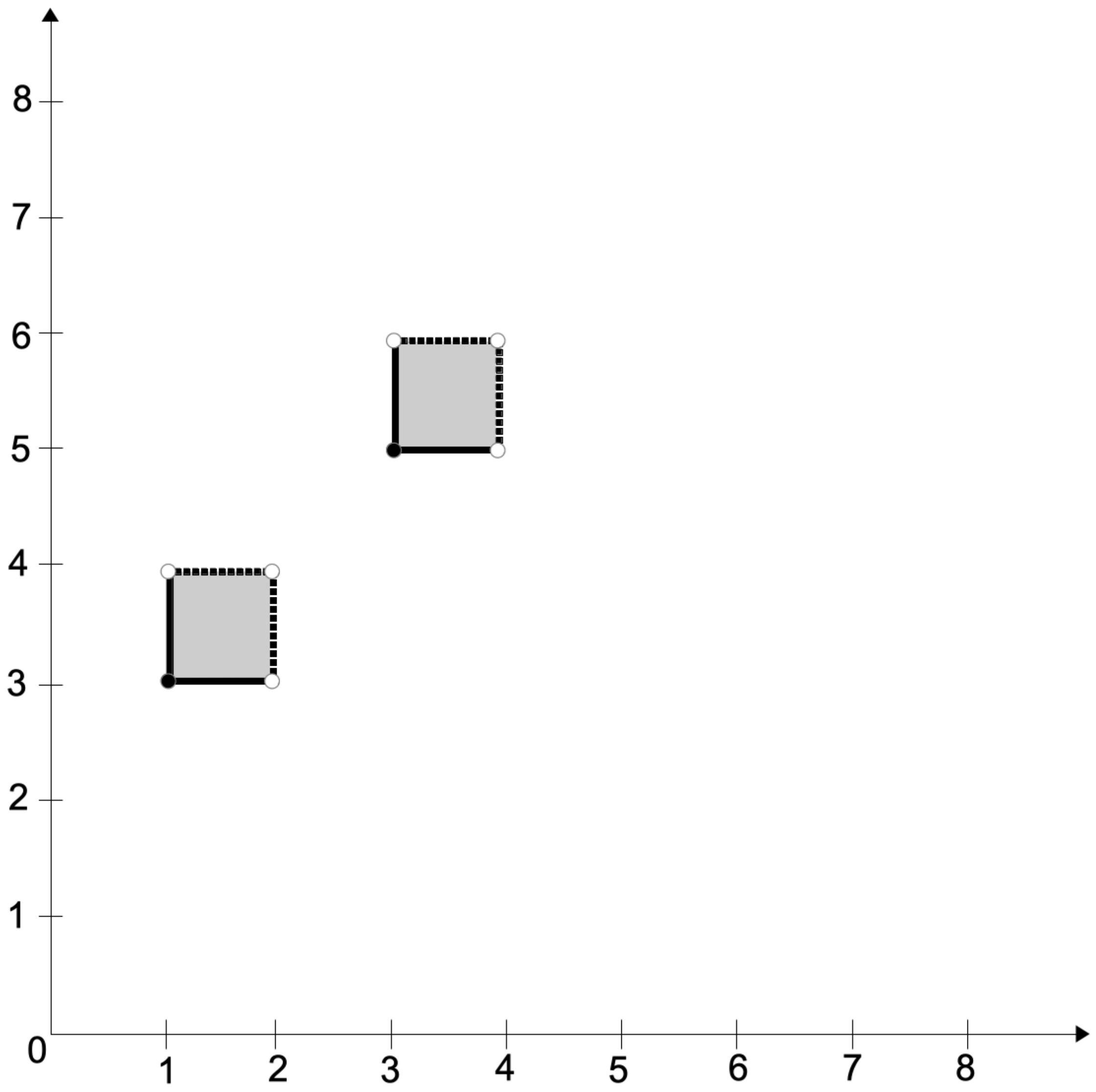
The first rule uses predicate patterns in its body, which will be computed as queries to working memory. It enforces transitivity for the predicate `f`.

The second rule specifies data for the first rule to work with. The syntax in the body represents constraints. First-order constraints are written within square brackets, and can involve arbitrary linear equations and inequalities. The relationships `<`, `>`, `<=`, `>=`, and `==` are allowed. These can be chained between two or more linear expressions. A linear expression can be a term or a sum of terms. A term can be a number, a variable, or a number times a variable.

Inequalities (`<`, `>`, `<=`, `>=`) are replaced by functions which have the value 1.0 where the constraint is true, and 0.0 where false. Equational constraints (`==`), on the other hand, are replaced by Dirac delta functions: infinite when true, 0.0 everywhere else. See Section 3.3 for details on how Dirac delta functions are represented in Signia. There aren't any in this example, so we can understand everything with normal functions.

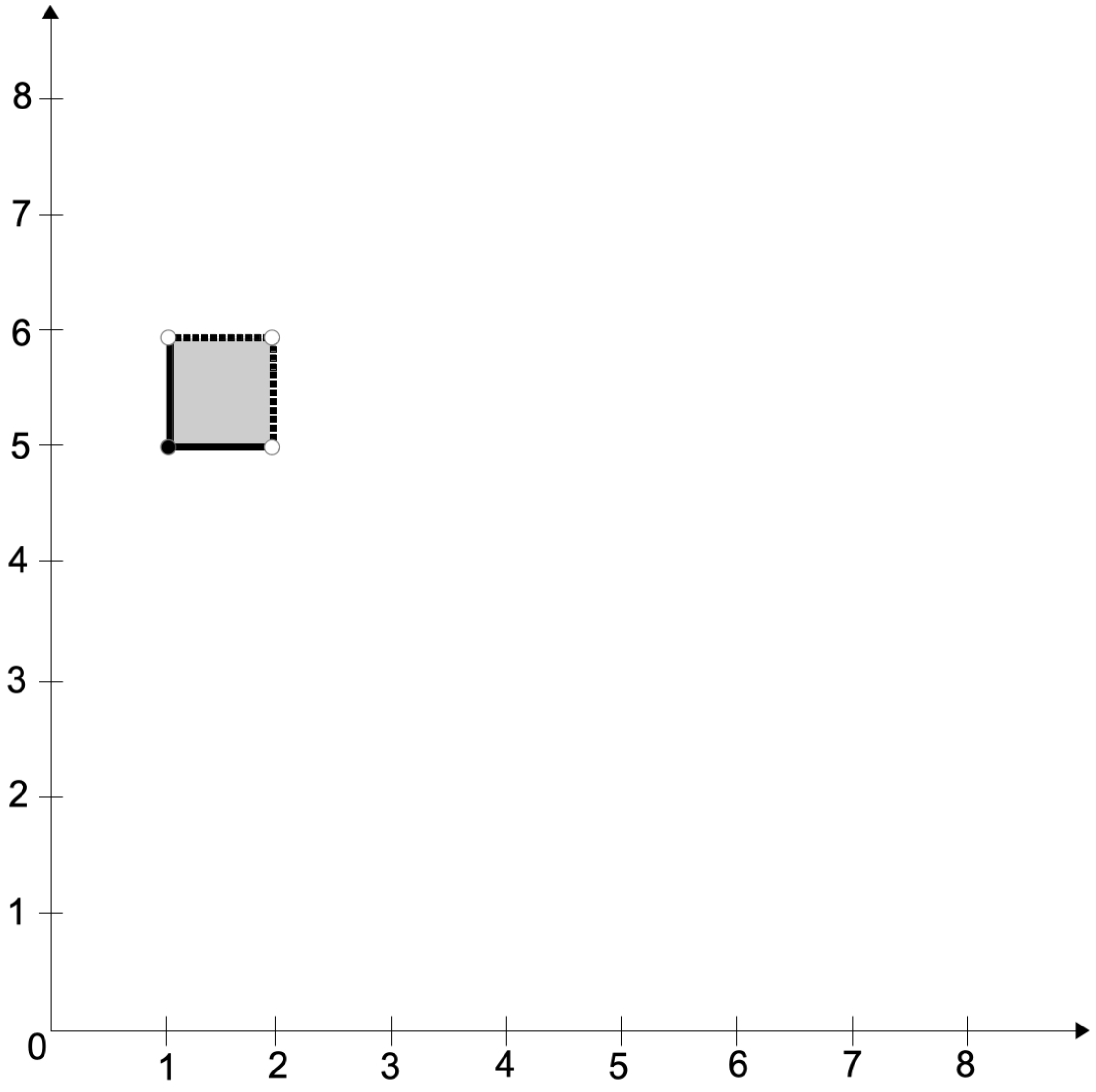
Figure 3.1 illustrates the meaning of these constraints. Each square is the product of a constraint on x and a constraint on y. The two squares are summed together, resulting in the function illustrated in the figure. Since the rule head contains the same variables as the rule body, `@max` doesn't actually summarize any variables, so this figure also illustrates the rule output.

The first rule, which is a form of transitivity, then has data to act on. The result (the rule output) is shown in Figure 3.2.



The output of the second rule. The white background has the value zero. The filled-in grey region has value one. Dotted lines and white circles indicate that the value at that point is the background value, zero. Bold lines and black points indicate boundary points with the same value as the filled-in region.

Figure 3.1: Constraints Illustrated



The output of the first rule.

Figure 3.2: Transitivity Output

Notice that the initial data disappeared; we only get the new square which was derived from the two old squares. If this was simply written to memory, then so-called transitivity rules would delete their initial data, rather than just adding new conclusions.[8] To avoid this, Signia "merges" results with working memory, rather than simply over-writing morking memory with the newest results.

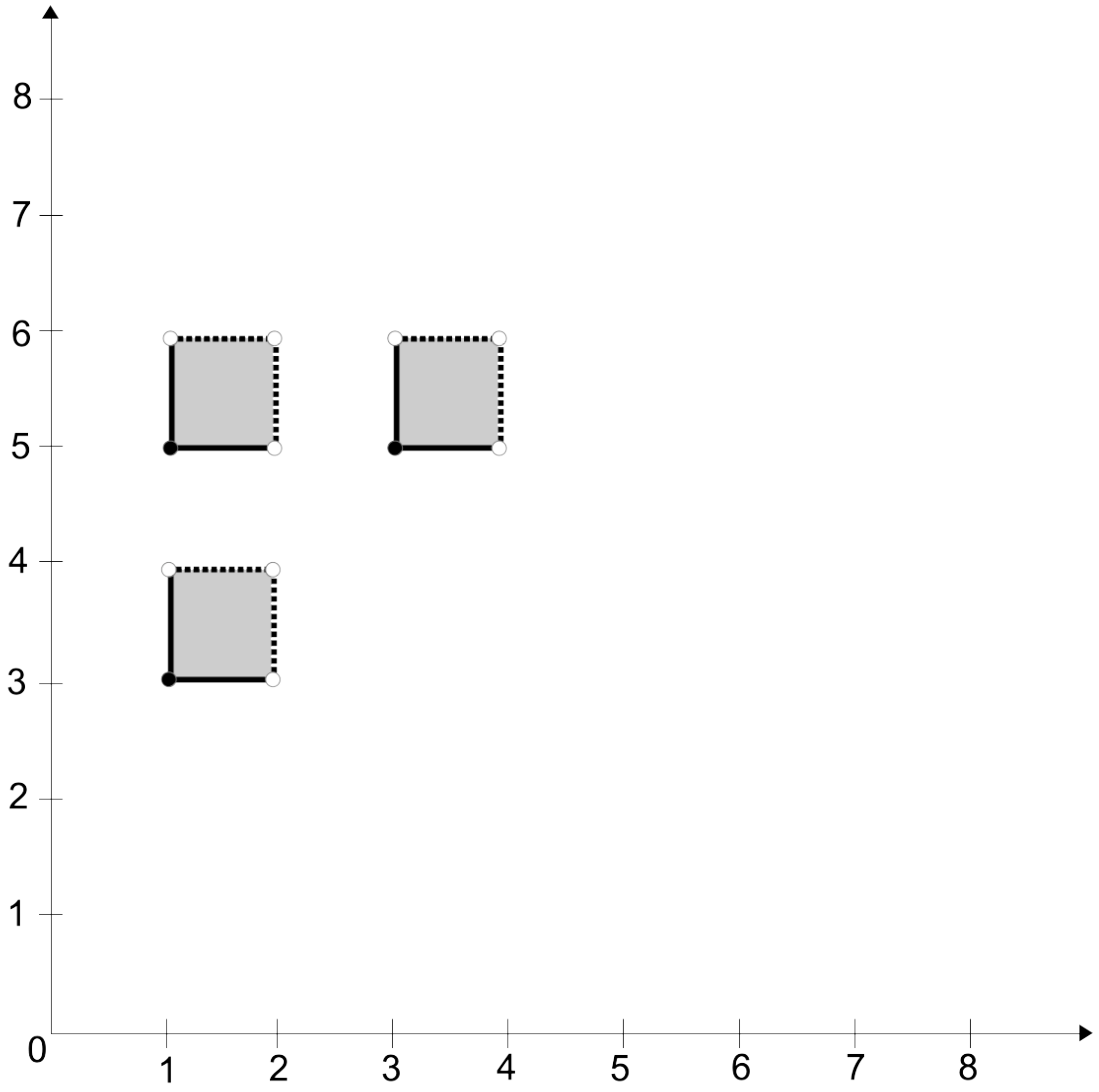
The output of a rule is combined with the output of other rules targeting the same region of working memory, using a combination operation based on the summary operation declared for the rule. `@sum` rules are combined into working memory with `+`; product-integration with `*`; etc. In the example at hand, `max` will be used. The result is illustrated in Figure 3.3; this is what the second rule writes to working memory, *not* the actual rule output from Figure 3.2.

This allows, among other things, the contents of predicates to be effectively *initialized* by a rule. Rules which don't use any predicate patterns in their bodies, such as

```
f(x,y) = @max [1<=x<2]*[3<=y<4] + [3<=x<4]*[5<=y<6],
```

will only be fired once – there is nothing to recompute as working memory changes. The constraints in the body are effectively the initial data for the predicate `f`. So, without the built-in feature to merge results from different rules, it would be much more difficult to provide initial data. (See Section 3.4.2 for details on how working memory is initialized.)

[8]Furthermore, as the rule was iterated, it would eventually remove everything.



Working memory after merging the results in Figure 3.1 and 3.2.

Figure 3.3: Merged Result

This also explains why the example uses `@max` for *both* rules, even though the second rule doesn't summarize any variables. We want to make sure that the result stored in the working memory location for `f(x,y)` is the `max` of the output of the two rules.[9] Typically, if two rules have matching heads, it means they target overlapping regions of working memory, and should have matching summary types. Otherwise, the two rules might combine into the same working memory region in different ways, which can create a large dependence on rule ordering (and generally, confusing and counter-intuitive results). Signia produces a warning message if overlapping rules have different summary types, for this reason.[10]

For example, what is the combined effect of these two rules?

```
f(x) = @max 1

f(x) = @sum 2
```

Should they set the contents of `f` to two or three? It would apparently depend on the order in which they are applied. As a result, the system has no solution, and the rules would iterate infinitely without making any progress.

[9]To spell this out: the summary operator, `@max` in this case, serves *both* to summarize over the free variables in the body – in this case, the variable `y` – *and* to specify how results should be combined into working memory. For example, a result of 1 from this rule would overwrite a pre-existing value of 0, but would not overwrite a value of 2. On the other hand, the `@max` in the second rule *only* serves this second purpose; there are no variables to summarize over, so it just tells us how to combine results into working memory.

[10]An earlier version of Signia threw an error in this situation. However, this was found to be overly restrictive. The use of predicate variables and other second-order features can lead to situations where rules with different summary operations could potentially write to the same memory locations, but, no true ambiguity arises unless the *data* provided to the rules creates a problem. This is difficult to detect at compile time, since the question of whether an ambiguity is ever created may depend on what happens during execution.

An alternative design could be to declare *at the predicate level* how each predicate merges results, rather than using the summary types of each rule to determine how that rule merges. However, the current design takes advantage of the fact that there is a firm correspondence between summary type and the appropriate combination operation for result merging. Also, the second-order syntax will allow us to write rules which can produce output for more than one predicate; so, the current design generalizes more readily to the second-order case.

3.2.2 Second-Order Language

3.2.2.1 Second-Order Variables

As previously discussed, first-order WLP is not sufficient for the cognitive-architucture use-cases considered in the present work, because first-order WLP only allows rules to be written for specific predicates. In this section, a proposal for second-order WLP is considered, which addresses this problem.

The full Signia language has four types of variable:

First-order variables: The variables we saw in the previous section. These variables range over real numbers. A first-order variable name can be any Julia identifier which has a lower-case character as its first symbol; for example, `x`, `var1`, `num_counted`. Predicate variables can occur only as arguments in predicate patterns.

Predicate variables: As the name suggests, these variables range over predicates. A predicate variable must begin with a capital letter, for example, `P`, or `Pred_1`. This distinguishes predicate variables from predicate names, and from first-order variables. A predicate variable can occur either in the predicate position of a predicate pattern, as in `P(x,y)`, or in the argument position, as in `f(P,Q)`.

Tuple variables: Tuple variables range over tuples. They begin and end with an underscore; for example, `_tuple_var_1_`. They can occur either as arguments, or in argument-capture form, `f(_t_...)`. Argument-capture occurrences allow us to write rules applying to predicates even when we don't know how many arguments they take or the argument types; for example, the pattern `F(_t_...)` matches with any predicate and any number of arguments. The rule `F(_t_...)  = @sum 1` would add 1 to the entire working memory. When a tuple variable occurs in argument-capture form, no other arguments are allowed in the argument-list.[11]

Universal Variables: Universal variables are written lowercase, but with an exclimation point at the end; for example, `u!`. These special variables allow predicates to be split into more instances; as such, they are in some ways a second type of predicate variable. For example, suppose that we have a predicate `grade(class)` which takes the first-order variable `class` and returns the

[11]This means we can't specify an initial few variables and then capture the rest in a tuple variable. Such a feature may be useful, but wasn't necessary for this work.

student's grade. Now suppose we want to use this predicate for many students, rather than just one, but we do not want to re-write every single rule involving this predicate, replacing `grade(class)` with the multi-argument `grade(class,student)`; doing so could be a headache. We could instead use the universal variable `student!` to split things into many instances, *without* re-writing everything. This would be written `grade{student!}(class)`. Unlike added arguments, universal variables pass "silently" through rules which do not use them: the rules still work as intended, but now, apply their computation to every instance. Universal variables also play a central role in the form of lifted reasoning which Signia provides.

Predicate and tuple variables work together to allow 'meta-rules' to be stated: rules effectively create a larger number of first-order propagation rules for us, rather than requiring such rules to be re-stated for each case.

Predicate variables range only over named predicates – there is no way for rules to instantiate new predicates in the current system (although universal variables come close to this). So, the effect of a rule involving predicate variables can be visualized by imagining that each predicate variable is replaced by a named predicate, in every possible combination. (However, this is not necessarily done; messages can be attributed to a range of predicates at once. See Section 3.3.2.)

3.2.2.2 Second-Order Arguments

In the first-order fragment of the Signia language, the only major variation in argument lists was their length. In the second-order language, the three types of variable available make this more complicated. As in the first-order fragment, using the same predicate name in different ways leads to overloading. So, for example, `p(x)` (with `x` a first-order variable) refers to a mutually exclusive set of locations in working memory from `p(X)`. The latter is a predicate *on predicates*, storing a value for each predicate name in the rule system (including `p`). Predicates like this will be called *second-order predicates*, although there isn't a real type distinction (in particular, there is no need for a concept of third-order predicates, etc).

Although first-order variables do not have a corresponding notation for constants of the same type, all of the second-order variable types are provided with one:

Predicate literals: If you use a predicate name within an argument list, it is recognized as the predicate in question. For example, `a(a) = @sum 1` would add 1 to the predicate `a`'s evaluation of itself.

Tuple literals: A tuple literal is another argument list, occurring as an item within a larger argument list; for example, `f(a,b,(x,y))`. Tuple literals are allowed to contain more tuple literals recursively. The whole argument-list of a predicate pattern can also be thought of as a tuple literal (except when a tuple variable occurs in argument-capture form, in which case we can think of the argument list as being the tuple variable).

Universal Literals: Rather than writing a universal variable, we can write a numerical range, which specifies one or more instances. For example, the pattern `grade{3:5}(class)` matches only the third, fourth, and fifth student, whereas `grade{student!}(class)` matches the entire range of students. See subsection 3.2.2.4 for more details.

3.2.2.3 Second-Order Constraints

Second-order variables are provided with a notation for constraints, as with first-order variables. Second-order constraints are significantly less complicated than their first-order counterparts: there is no corresponding notion of 'linear expression' involving predicates or tuples. All one can do is constrain particular variables to particular literals.

Predicate constraint: Predicate constraints use the notation `[[P == p]]`, constraining the given predicate variable to equal the given predicate name.

Tuple constraint: Tuple constraints are written similarly; for example, `[[_t_ == (x,y)]]`.

Unfortunately, only a single tuple constraint can be used per rule in the current implementation. For details on the issue, see Section 3.5.4.2.

If second-order constraints don't do anything but set a second-order variable equal to a second-order literal, why use them at all? Why not just eliminate said variable, using the literal directly?

Partly, constraints were implemented just to give more options. However, they are not exactly equivalent to literals, so it's quite possible to have a stronger reason to prefer them. Like first-order constraints, second-order constraints have value one where true, and zero elsewhere. These values can be manipulated within the rule body to produce other functions. For example:

```
f(P) = @sum 0.5 - ([[P == pred1]] + [[P == pred2]])
```

This rule adds 0.5 to the second-order predicate `f` for all predicate names except `pred1` and `pred2`; for those, it subtracts 0.5. Accomplishing this with literals alone would require several rules.

3.2.2.4 Universal Variables

Universal variables permit us to split predicates into more instances without adding more arguments. But, how do they work?

Predicates can be given universal arguments in curly-brackets, before their regular arguments. Suppose we have a predicate `chessboard(row,column,piece)`. This predicate encodes which chesspieces are present at which board positions. Now, suppose that we would like to represent *multiple* chess games. We could modify our predicate to add an argument "game"; however, if we have written many rules relating to chess, this would require us to re-write every single rule which mentions the `chessboard` predicate.

Instead, we can use `chessboardgame!(row,column,piece)` to create many instances of the `chessboard` predicate, while ensuring that (in rules that don't mention the universal variable) each one will be treated exactly as we expect by existing rules. In particular, we could initialize different chessboards to different configurations using universal literals, like `chessboard{3:3}(row,column,piece)`, which would access game number three (since the range given only contains one number). If this was the only use we made of universal variables, then each of these chessboards would work exactly like the single chessboard predicate worked before. Note that universal variables are discrete, like predicates, rather than real-valued, like first-order variables.[12]

So, in rules which don't mention universal variables, then we have two possible situations:

- If the stored memories for the predicates in the rule body involve universal variables, then these universal variables are said to occur *implicitly*. In this case, the rule will act on the many instances "in parallel". For example, if a `games!` universal variable was used, then the rule is effectively calculated for each game. Lifted reasoning is used, so that many identical games can be computed almost as efficiently as a single game.

- If the stored memories don't involve any universal variables, then the rule will act just as one would normally expect. (However, the result could still be

[12]Universal variables piggyback on the code for predicate variables in many ways, so that they act very much like additional predicate-variable dimensions.

split into many instances during result merging, if other rules with matching rule heads have produced outputs split by universal variables.)

In rules which mention universal variables *only in the body*, those universal variables must be summarized out, just as with any other variable which is mentioned in a rule body but not the rule head. However, other universal variables may be present implicitly; those variables would still split things into many instances without being summarized.

Mentioning a universal variable in the head only doesn't accomplish very much; the result would be stored in a way which registers the presence of a universal variable, but it wouldn't really split things into multiple instances. On the other hand, using a universal *literal* in a rule head restricts the rule to the given range of instances, which can be quite useful, EG for initialization.

This concludes the explanation of the syntax of the Signia language. The interested reader probably still has some confusions about exactly what all this syntax means. For example, what does it mean to say that a constraint is replaced with a 'function', or that a predicate pattern in the rule body acts as a 'query to working memory'? Unfortunately, it is impossible to explain everything all at once. The next section describes the data-structure of the memories which rules manipulate, so that we can understand what kind of data is really being passed around. Finally, Section 3.4.1 resolves the question of how rules give rise to computations, with its discussion of compilation and execution.

3.3 Spatial Trees

The major data-structure manipulated by Signia computations is a *variable-dimensional* spatial tree: a tree which can organize spatial data of varying types and varying dimensionality.

For the sake of readability, this section will only give an overview, focusing on the high-level design decisions and the motivations behind them. The goal is to understand enough for the discussion of execution in Section 3.4 to be grounded in a sense of what's being computed. More technical details have been deferred to Section 3.5.

As with the syntax, it will be helpful to divide things into first-order and second-order. The first-order trees represent fixed-dimensional spatial data, using a mostly-standard binary space partitioning tree (BSP tree) to organize spatial data of matching dimensionality; this is described in Section 3.3.1. An overarching structure organizes *those*, enabling the generality of second-order reasoning, in which dimensionality need not be pinned down. This second-order structure is described in Section 3.3.2.

3.3.1 BSP Trees: Representing Hyperreal Functions

The first-order spatial trees, given by the `BSPTree` data-structure, have a few major goals: efficiently represent sparse cases without being too inefficient for dense data;

allow delta functions and linear inequalities to be represented; and, support the simulation of discrete data as a special case of continuous data.

3.3.1.1 Design Considerations

A classical rule system represents data in a fairly direct way, via instances, much like entries in a relational database. The optimizations (such as RETE networks) are focused mostly on making the right instances available at the right time, so that one has to do as little work as possible, in order to check whether a rule fires. Some of those optimizations could be adapted to WLP, but the current work did not focus on that, for two reasons.

First, many of the applications of WLP tend to have non-default[13] values everywhere (at least, within the range where anything happens at all): the propagation rule has to deal with the entire function, so there is little benefit to optimizations aimed at minimizing the cost of finding the instances which must be dealt with.

Second, a major objective of this work was to facilitate lifted reasoning. The optimizations which classical rule-systems focus on are mostly orthogonal to that.

[13]In a classical rule-system, with some exceptions, one only needs to explicitly represent instances with value “true”, letting “false” values be represented implicitly by their absence. This representation makes sense mainly because there are typically more “false” instances than “true”, and also because “true” values are more often the ones which rules need to react to. Altogether, false is a good choice of default. Similarly, in numerical contexts, there may be sensible default values, most often zero. The claim, then, is that cases of interest for WLP may lack useful notions of default in this sense. Note that this does not contradict the usefulness of sparse representations (there may be large regions of identical value), nor the importance of identity values as discussed in Section 3.4.2.

Like Sigma, Signia utilizes piecewise-continuous functions. These are only piecewise-constant at present. Although it would not be difficult to represent piecewise-linear functions (as is done in Sigma), those are not critical for the applications considered here.

The piecewise functions are represented as BSP trees (Binary Space Partitioning), a representation from computational geometry, often employed in 3D graphics. This organizes an n-dimensional space into a tree by iteratively splitting in half, and representing the constant-value regions at the leaves. (It can be thought of as a decision tree, or more precisely, a regression tree.) This choice is very close to the decision-tree arithmetic of [30]. BSP trees are a variety of spatial trees, the usual representation of choice for multidimensional geometry in such applications as 3D graphics and spatial databases. [36]

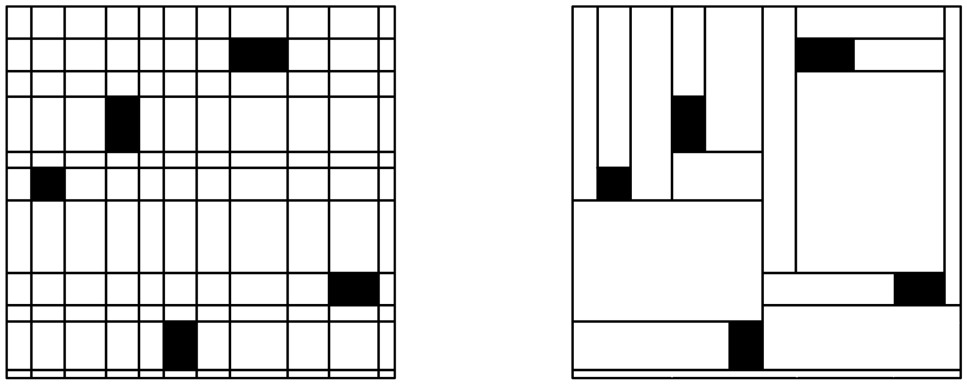
Although Sigma initially used similar representations, they were replaced with a doubly-linked region structure [33]. This was eventually replaced with a structure based on arrays of regions. The original motivation for the doubly-linked region representation in Sigma was to allow functions to be edited in-place more easily, as with doubly-linked lists. However, destructive modification turned out to be uncommon in Sigma. The region-array approach optimizes the more common case by reducing the amount of time spent traversing pointers to find desired regions.

The choice to move back to tree structures in Signia was based on experience with sparse region representations in Sigma. If a function is mostly zero, with only a few non-zero regions, then it's much more efficient to use a sparse representation which only represents those non-zero regions. An implementation was developed which had speed benefits for those cases. However, integration of the new representation into Sigma took quite some time, and by the time a side-by-side comparison was possible, operations on the doubly-linked representation had undergone quite a bit of optimization and were faster for nearly all cases. Focused optimization effort made the sparse representation outperform the doubly-linked representation through the introduction of spatial trees and hash tables to replace linked list structures within parts of the sparse algorithm. However, the resulting algorithm was overly complex as a result of not being based purely on spatial trees. The idea of replacing the doubly-linked structure with a region-array structure was under consideration at that time, and appeared more promising for all but the sparsest use-cases. The sparse representation was eventually abandoned.

Signia's design stuck closer to standard data-structures in computational geometry and spatial indexing, in the hopes to finally provide a sparse representation which outperforms what is currently used in Sigma. As reported in Section 4.1.1, the current implementation has not succeeded in this goal in most cases.

There are some reasons to expect spatial trees to be more efficient than Sigma's region-array representation. For example, the size of the region array for sparse examples grows faster than the size of a spatial tree: n regions in a d-dimensional grid cost $O(n^d)$ for a region-array representation, but can be represented in $O(d^2n)$

with a BSP-tree.[14] This can make quite a difference even in only two dimensions. Figure 3.4 illustrates the idea, although simply counting the apparent number of regions in such depictions is not a perfect comparison.[15]



Five small regions with nonzero values are represented by solid black fill; the remaining regions contain zeroes. A region-array representation requires an 11-by-11 array, making 121 cells (left). A BSP-tree representation can do with only 21 leaf nodes (right). The difference becomes more extreme with more dimensions.

Figure 3.4: Example of Region-Array Inefficiency

[14]The $O(n^d)$ cost for a region array is due to needing $O(n)$ distinct indices in each of the d dimensions. The $O(d^2 n)$ cost for BSP trees can be understood by imagining that we insert the regions one at a time. Each rectangular region needs $2 \cdot d$ "walls", plus one leaf node storing its nonzero value. Other than the one new leaf node, the BSP nodes can all point to the old root node (from before the new insertion). The leaf has constant cost, but the new splits each have cost $O(d)$, making the overall cost for new nodes $O(d^2)$ per insertion.

[15]In the illustration, the region-array appears to represent 121 regions, whereas the BPS tree represents 21. However, the graphic only shows leaf nodes. A BSP tree also requires internal nodes; for n leaf nodes, $n - 1$ internal nodes are needed, bringing the total up to 41. Also note that Sigma doesn't fill every cell of the array with a fully-represented region; only the non-default-value regions are fully represented. (Still, memory must be allocated for the array, and empty cells may still have to be traversed.)

3.3.1.2 Representation Details

The representation is that of a typical BSP tree: a binary tree which iteratively cuts space in half, storing spatial data in the leaves. An internal node stores a separating hyperplane, and two children. A leaf node stores the value of the function in the region. A null node, `None()`, may take the place of a leaf node to represent undefined regions. See Figure 3.5.

Because BSP trees are being used to hold multidimensional functions, rather than objects with locations, the basic operations supported by Signia are somewhat different than what one might find in a textbook BSP tree implementation. However, most of the algorithms are relatively simple in concept, involving recursive descent of trees. Section 3.5 discusses the major algorithms needed.

BSP trees, along with many other data-types in Signia, are immutable: their contents cannot be edited after creation. The benefit of this approach is that one does not have to worry about multiple references to data, which means identical sub-structures can be shared rather than copied.

As an example, suppose we start with a multidimensional function in x and y, which is defined on the square $0 \leq x \leq 2$, $0 \leq y \leq 2$, and takes value 0.0 for $x < 1$, and 1.0 otherwise. This is represented by a `BSPTree` with one `SplitNode` and two `LeafNode`s. Now, suppose that we want to insert a region of value 0.5, at $0.5 \leq y < 1.5$. Because we can share structure, we can do this without duplicating

`BSPTree`

> **Boundary:** A `Region`. Gives the area where this `BSPTree` defines its values.
>
> **Root:** A `BSPNode`.

`Region`

> **Bounds:** A vector of linear inequalities, telling us where the boundaries are (and which are open vs closed).
>
> **Dimensionality:** An `Int`, indicating the number of dimensions this region lives in.

`BSPNode` Either a `SplitNode`, a `LeafNode`, or `None`.

`SplitNode`

> **Split:** A linear equation.
>
> **Negative child:** A `BSPNode`. This is what's on the open side of the split.
>
> **Positive child:** A `BSPNode`. The closed side of the split.

`LeafNode`

> **Value:** A floating-point number, or a `Hyper`.

`None` *(No fields.)*

`Hyper`

> **Mantissa:** A floating-point number.
>
> **Exponent:** An integer, giving the power of infinity.

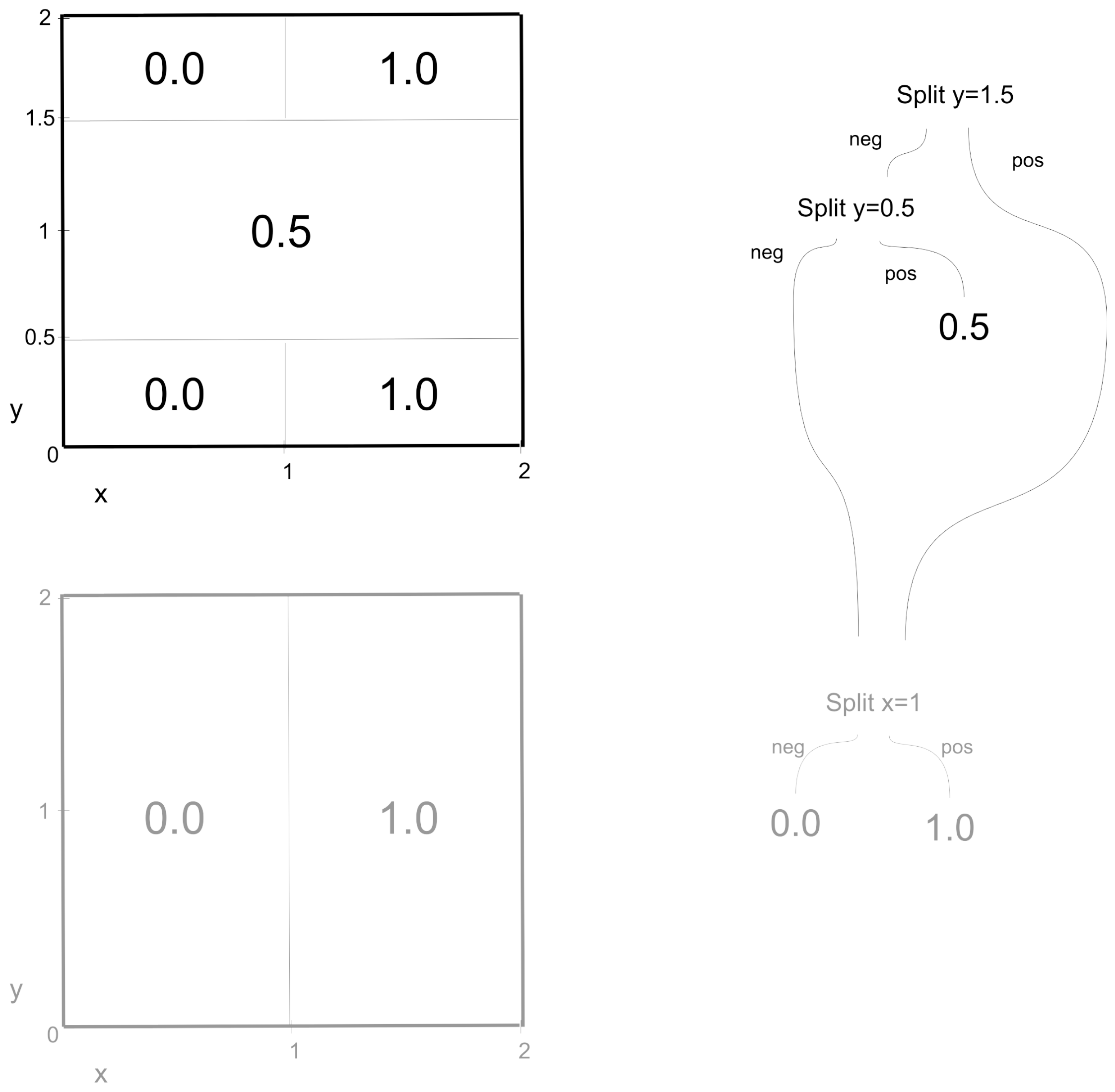
Figure 3.5: `BSPTree` and Associated Data-Structures

the old tree; we only need one new `LeafNode`, to hold the new value, and two new `SplitNode`s, one of which will become the new root. See Figure 3.6.

As mentioned in Section 3.2.1, Signia can also represent Dirac delta functions, enabling the explicit representation of linear equations as functions which can be passed around and manipulated. This is accomplished via two major representation decisions:

- The two children of a BSP node are not treated symmetrically. A BSP node has an open side and a closed side, so we know which side things fall on when they are exactly on the separating hyperplane. This allows non-negligible regions of zero area to be defined.
- A system of infinite values is used which distinguishes between multiples of infinity, as well as powers of infinity. Representing multiples of infinity enables Dirac delta functions of different values; representing different powers of infinity enables constraints in higher dimensions (but see Section 3.6).

Infinite numbers are represented via the `Hyper` type (short for hyperreal, a system of infinite and infinitesimal numbers extending the reals). A `Hyper` consists of a floating-point mantissa a and an integer exponent b, representing the quantity $a \cdot \infty^b$. This can be thought of as a truncated representation of hyperreal polynomials, giving only the coefficient of the largest infinity involved. If we try to calculate $(2 + \infty) - \infty$, we get back zero, since the two gets rounded off. This truncation suffices for the use of infinities in Signia.



A more complex tree sharing a simpler structure between two branches. The simpler function and its tree are illustrated in grey; the more complex, in black. Note that the distinction between $<$ and $\leq$ is not illustrated here.

Figure 3.6: Sharing Structure

Extending all the numerical operations used in Signia to support the `Hyper` type is relatively straightforward. However, dealing with open-vs-closed constraints was somewhat complicated. Signia uses a pre-existing linear programming library for operations such as checking whether a region is empty and checking whether two regions intersect. Strict inequalities are not supported. This means empty regions may be declared non-empty erroneously, and disjoint regions may be thought to overlap. Extra steps must be taken to verify these results.

Specifically, a region with no interior (all points intersect with cutting hyperplanes) might contain some points or no points at all depending on whether inequalities are strict. To check this, we can look at each strict inequality (every open bound in the region's list of bounds), and try to "optimize away" from it: optimize inward to the region, in the direction of the normal vector of the boundary. If we can't get more than `epsilon` away from any one of a region's open bounds, the region is considered empty. This adds significantly to the amount of linear programming which needs to be done, but is necessary for correctness.

Checking region intersections and nonemptiness is a frequent operation in Signia. To mitigate this cost, ortholinear splits (splits along one dimension) are handled as a special case. So long as all of the boundaries defining a region are ortholinear, a region can be checked for nonemptiness by checking in each dimension individually. This reduces cost significantly.

The BSP tree representation forms the foundation for the representation of multidimensional functions in Signia. Loosely speaking, one can imagine that each predicate has its own BSP tree to store its associated data, which has the same number of dimensions as the predicate has arguments. (As mentioned previously, overloading predicates creates additional memory locations per-predicate.) This would serve the purposes of purely first-order Signia rule-systems. What's needed for second-order systems is much more complicated, as we will see in the next section.

3.3.2 Supporting Variable Dimensionality

The second-order tree structure in Signia is `MemTree`. A `MemTree` serves as the working memory of a rule-system. In purely first-order settings, the `MemTree` primarily serves to associate a BSPTree with each named predicate.

The type of a predicate pattern in Signia is called its *signature*.[16] The main goal of the `MemTree` data-structure is to index BSPTrees by signature, in a way which also allows a high degree of flexibility to store data which is common between multiple predicates or signatures (facilitating lifted reasoning).[17]

[16]This term was chosen mainly to avoid terminological conflict with Julia types, when programming in the context of Julia – otherwise a signature might have simply been a 'type'.

[17]This way of framing the goal of the `MemTree` data-structure pre-supposes that BSP trees are kept as a distinct structure, rather than combined into one general-purpose tree-structure. The current implementation takes this approach, but it isn't obvious whether this is desirable; see Section 3.6.

What regions are to the first-order representation, signatures are to the second-order. So, it is important to get a clear picture of them in order to see how MemTrees work.

I will use the following notation to talk about signatures. This notation is not part of the Signia language proper,[18] but is convenient for discussing what is going on:

The signature of a first-order variable is written `:v`; a predicate variable, `:p`; a tuple variable, `:t`. A tuple's signature is written as a tuple of the signatures of its contents; for example, the tuple literal `(a,P,(x,y,z),_t_)` has signature `(:v,:p,(:v,:v,:v,),:t)`. Universal variables are represented more indirectly, as they are held within predicate signatures, which we are simply writing as `:p` here; more on this soon.

The signature of a predicate pattern is given as if it were the two-element tuple consisting of the head and the body: `F(x,y)`'s signature is written `(:p,(:v,:v))`.

There is an inclusion relationship for signatures: a tuple signature `:t` is less specific than any written-out tuple signature. So, for example, although the signature of `(x,y)` is `(:v,:v)`, the signature `:t` also *accurately describes* `(x,y)`; it merely provides less than full detail. A query involving a tuple variable, such as `f(_t_...)`, will retrieve all data stored under more specific signatures, not only those stored under a tuple variable.

[18]The Signia code-base *does* include functions for representing `Signature` data-structures in a format close to this; but, this is for convenience when working with the data structures at a lower level, and never appears in the Signia rule-systems themselves.

The most general signature associated with items in working memory is `(:p,:t)`, since predicate patterns are written like this, and everything has to be associated with *some* predicate pattern.

The signature of a tuple variable in argument-capture form is `:t`, but "one level higher" – for example, the signature of `f(_t_...)` would be `({f},:t)`, whereas the signature of `f(_t_)` would be `({f},(:t))`. This is what allows the argument-capturing form to match with an entire argument list.

A signature can also further specify the values of predicates. The signature `:p` can be thought of as representing the entire set of named predicates. I'll use the notation {.} to represent a set of predicates; for example, the signature `({f, g},:t)` is more specific than `(:p,:t)`, but less specific than `({f},:t)`.[19]

As hinted previously, universal variables are stored within the corresponding predicate signature. For example, `p{x!}(y)` has the signature `({p}, (:v))`, but the {p} carries some extra information to indicate the presence of a universal variable. We can write this as $\{\texttt{p}\}_{\texttt{:u}}$. This is treated as more specific than the version without the subscript; in other words, a predicate which we don't specify as being split into instances by a universal variable always "might be split", we just don't know yet.[20] These variables can also be range-restricted; so, for example, `p{1:5}(y)` restricts the universal variable to the range one through five (inclusive).

[19] Signia assigns numbers to named predicates, and only represents signatures in which the predicate-sets are consecutive ranges of numbers. That representation is obviously more efficient for long lists of consecutive predicates. However, rendering the sets here as lists of predicate names is more comprehensible.

[20] This allows universal variables to propagate through rules which don't mention them, since they can match with non-universalized patterns.

We can write the signature of this as `({p}`$_{\texttt{1:5}}$`, (:v))`. When multiple universal variables are present, some can be specified while others are not; for example, `p{u!, 5:10, v!}(y)` has the signature `({p}`$_{\texttt{:u,5:10,:u}}$`, (:v))`.

As mentioned previously, a signature should be thought of as the second-order notion of "region". A region is a notion of convex set in a fixed-dimensional space; a signature can be thought of as our notion of convex set for our space of *possible* fixed-dimensional spaces. So, how do we use trees to index the space of possible signatures?

Just as BSPTrees cut up a fixed-dimensional space into ever-narrower regions, `MemTree`s cut up variable-dimensional space into increasingly specific signatures. Each node makes an additional decision about the signature. As such, there are three node types:

Predicate split: Splits the space of possible signatures by restricting predicate possibilities. For example, the signature `({f,g,h,i,j},:v)` could be split into `({f,g},:v)` and `({h,i,j},:v)` by checking what is ¡ or $\geq$ to `h` in alphabetical order. The two new signatures each represent the restricted possibilities in one of the two children of a predicate split. A predicate split is a node which always has two children, based on a ¡ test.[21] Predicate splits are also used to split universal variables, in exactly the same way.

[21]Recall that predicates are actually given numbers in the low-level representation, so we don't really use alphabetical order here.

Tuple split: Gives definition to a tuple variable. Tuple splits can have arbitrarily many children, since there are infinite possibilities for tuple signatures. In addition to these children, tuple splits also have a child covering the rest of the space (any possible refinement of the tuple which isn't yet reflected as an explicit child).[22]

Leaf: Finally, the leaf nodes hold the BSPTrees, along with an index of how the dimensions in the BSPTree line up to the first-order variables in the signature.

Whereas the `BSPTree` type consists of a root node together with a `Region`, the `MemTree` type consists of a root node together with a ***`Domain`***. A domain includes both a region (specifying domain-of-definition for any first-order variables) and a signature.

The algorithms for handling `MemTree` are mostly analogous to their first-order counterparts; see Section 3.5.

This section and the previous have described the primary data-structures which Signia manipulates. Now that we have some idea of both the language for specifying rule-systems and Signia's representation of the values which rules manipulate, we can further address the question of what a rule in Signia actually means by discussing how rules are compiled and executed.

[22]See Section 3.5.1 for more on how this works.

3.4 Compilation, Initialization, and Execution

Section 3.2 discussed the syntax of the Signia language, but with only a very shallow explanation of what rules actually do. Signia's rules can be interpreted as fixed-point equations, which Signia seeks to find a fixed-point to. Yet, Signia will not find a solution to arbitrary fixed-point equations; its execution strategy relies on an assumption that the equations given can simply be iterated to hone in on a solution. Because of this, and for other reasons, it will be helpful to understand Signia's execution cycle in order to grasp the full meaning of a rule in Signia. However, in order to understand the execution cycle, it will be helpful to first understand compilation and initialization.

3.4.1 Compilation

The goal of compilation is to convert the `@rules` statements, described in Section 3.2, into the `RuleSystem` data-structure, which is detailed in Figure 3.7. The purpose of a `RuleSystem` is primarily to be a set of rules; Figure 3.8 details the `Rule` data-structure.

In order to create a `RuleSystem`, the list of named predicates is extracted by looking at the head of each rule, and each predicate is assigned a number for use in the dimensional tree representation. Then, `Rule` structures are created for each line of code in the rule system.

Rules: A vector of all the `Rules` in the `RuleSystem`.

Patterns: All the predicate patterns occurring in the body of any rule are stored here, together with the number of the rule which they occur in, and the pattern number counting from within that rule's body. This is for the purpose of checking which rules might be triggered by any change to working memory.

Output Buffers: One for each rule. These hold the most up-to-date output values for a rule.

Working Memory: The joint store of information for all rules.

Priority Queues: Each rule has its own priority queue of changes to working memory which need to be processed. Whenever a change is made to working memory, the patterns are checked, and for any matching pattern, the change is added to the queues of the relevant rules.

Predicate Numbers & Names: A mapping from predicate numbers to names and vice versa.

Merged Output Buffers: A unified output buffer for each merge type (= each summary type), used for debugging purposes.

Local Defaults: Stores the appropriate "default value" for each predicate. This plays a role in result merging in ambiguous cases where different merge types overlap. (Recall that this is discouraged.)

Control Information: To aid in selecting the next rule to fire, we store the number of the rule last fired.

Figure 3.7: The `RuleSystem` Data-structure

Rule-space: This holds the signature of the 'rule-space', which is the (possibly variable-dimensional) space in which the inputs to a rule (the 'sources') are combined. This space needs to have a dimension for every variable mentioned anywhere in the rule body or rule head.

Sources: The predicate-pattern signatures from the rule body, which function as 'inputs' to the rule.

Source maps: Each source must have its own mapping from the common working-memory space to the rule-space. Source maps determine how variables from different patterns match when placed in the rule-space. For example, a rule body like `f(x)+g(x)` would give rise to source maps which map the arguments of `f` and `g` into the *same* dimension in the rule-space; on the other hand, the rule body `f(x)+g(y)` would map them into *different* dimensions.

Target: The signature of the rule head, which provides the working memory location where the output of the rule calculation is to go.

Combination function: The combination function, derived from the rule body. The body can be an arbitrarily complex Julia expression, so long as it has the correct type. In practice, it is always a compound of the combination operators which have been defined to handle `MemTree`s.

Summary function: The function to be applied to remove unwanted dimensions, derived from the summary operator observed in the rule syntax.

Merge function The function used to merge results with rules having overlapping targets. As mentioned in Section 3.2.1, these are derived from a rule's summary operation; overlapping rules are required to have identical merge functions, and rules whose summary operator is `@just` should not overlap with anything at all.

Overlapping rules: The rule-numbers of rules with overlapping targets, for purposes of merging results.

Rule number The number of the rule, used to index it in e.g. overlapping rules.

Figure 3.8: The `Rule` Data-structure

The compilation of individual rules involves turning the body into Julia code which can then be compiled to get the rule combination function. Constraints are substituted for appropriate functions, whereas predicate patterns become occurrences of the arguments to the combination function – appropriate working memory queries are given as argument at run-time. The predicate patterns (including the head pattern) must also be examined for variable matches, producing the rule-space and the source maps to ensure that variables which are shared between patterns line up correctly.

Finally, another pass along all the rules is made to find the overlapping rules. This can only be done after the target domains have been computed; we check for intersection between target domains, keeping a list of those which intersect. This completes the compilation of the `Rule` structures.

3.4.2 Initialization

The priority queues are initialized with a change for each rule (asserting that the entire working memory has "changed"), the pattern list is set up, and the memories are initialized. Memory initialization can be thought of as pretending each rule has already produced an output, but, that output is the identity value of whatever merge function the rule uses: a `@sum` rule gets 0.0, a `@max` rule gets -Inf, and so on. These values are written to both the rule's output buffer and to working memory as if the rule had produced them. Initializing to identity values in this way is necessary for merging to work, since if the initial values were `None()`, the output of a rule would be erased (rather than unmodified) when merging with initial (no-output-yet) values of overlapping rules.

In cases where overlapping rules have different merge-types, the *order* of this initialization matters; for example, a predicate might be initialized to 1.0 or 0.0 depending on whether `@sum` or `@prod` rules are initialized first. Currently, the order is: `max`, `min`, `*`, `+`. (This ordering is fairly arbitrary.) Later in the order means lower priority. At this stage, the contents of working memory are also saved to the "local defaults" field; this comes into play when merging such ambiguous cases. See Section 3.5.6. Recall that merge-type overlaps like this should be avoided when possible.

The role which these data-structures play in the execution cycle is illustrated in Figure 3.9.

3.4.3 The Execution Cycle

At a high level, the execution cycle works as follows: One change/rule pair is processed at a time (flowing through each step of Figure 3.9).[23] This may add

[23]Note that the output buffers allows us to compute one rule at a time, rather than computing a whole set of overlapping rules at once and then combining their results.

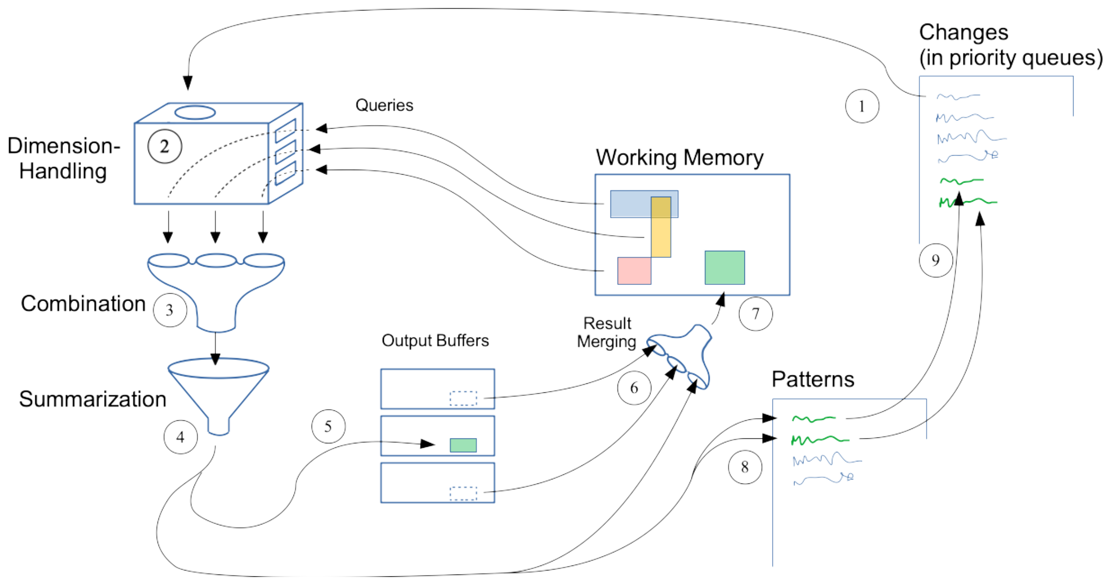


Figure 3.9: The Execution Cycle

several more changes to queues. We continue to process changes (in some order)[24] until no further changes remain to be processed.

Referencing the numbered steps in Figure 3.9: a single iteration of execution begins at (1), when a single change is removed from the list of changes. A change includes a rule which has been activated by a modification to working memory, and information about what area of working memory was modified in order to trigger the rule. This information is converted into queries to working memory (2).

The source-maps are used to compute the region of working memory which needs to be queried. Note that this is not just the original region. Each source may need to query a different region of working memory. If a change to working

[24]In the simplest case, order doesn't matter except for efficiency of convergence. However, order can matter in examples of interest. Currently, Signia repeatedly loops over the rules in order, skipping those with empty queues. The prioritization of changes within a rule's queue is based simply on how recently they were added (making the priority queue act as a stack).

memory matching source #1 is what triggered the rule, then the area to be queried for source #1 is exactly that area.[25] However, if the rule also has a second source, then the relevant region of source #2 must also be queried; this query may be different, depending on the two patterns. All of the queries are then mapped into rule-space. They are then fed into the combination function (3) which has been compiled from the rule body. Extra dimensions are then summarized out (4). The rule-space is constructed such that results after summarization are in the correct dimension ordering for working memory, so that no more dimension reordering is needed.

The result of the rule calculation is checked against its output buffer, to see whether a significant[26] change has been made. If so, then goes to three places:

- It is stored in the output buffer for this rule (5). The output buffers allow us to keep around the raw unmerged results of a rule, so that they may be merged with other rule outputs later.

- It is combined, via its merge function, with the output-buffer contents for any overlapping rules (6), and the result is stored in working memory (7), over-writing the previous contents of that region of working memory.

[25] An earlier design held the working-memory modifications themselves in 'Changes', along with the number of the source which matched, to avoid the cost of querying for this particular source. However, working memory modifications may get stale while sitting in the queue, so it is best to re-query anyway.

[26] A tolerance of 0.0000001 is used by default. See Section 3.5.2 for details.

- The working memory region of the result is compared to the pattern list (8). As described in Figure 3.7, the pattern list holds triples: patterns (from rule bodies), rules (indicating which rule the pattern occurred in), and integers (indicating the source number of that pattern in that rule). This information is used to translate the domain of the output into the rule-space of the newly activated rule (via the source-map of the relevant source). This information is then placed onto the list of changes (9), seeding further iterations.

The overall effect of merging results with overlapping rules and then overwriting what was in working memory is to simulate "combining results with working memory" – which is what we want. However, combining results *directly* with working memory would be troublesome.

Why do we want to simulate combining results with working memory? Why not just write rule outputs directly to working memory? In the absence of overlapping rules, this should amount to the same thing. In fact, whatever other behavior we want can be simulated in the simple write-to-memory version.[27] So, why complicate things?

[27]It would be possible to always avoid writing rules with overlapping target domains, instead explicitly creating intermediate predicates (serving much the same role as output buffers) and explicit rules for combining the intermediate results.

3.4.4 Why Merge Results?

There are a few reasons why we want to combine results with working memory rather than over-writing working memory. In cases where we *do* want the result-merging behavior which Signia implements, hand-coding intermediate buffers and merge rules would be tedious.[28] Furthermore, it seems like the result-merging behavior is what we usually want. Two rules with overlapping target domains should not cancel out each other's results each time they fire. Most importantly, however, the result-merging behavior allows the user to initialize predicate contents with rules.

Working memory starts out empty (or, to put it a different way, it starts out holding the value `None()`). So, in order to get anything started, a user-specified rule system must somehow introduce values into predicates.

One way to do this is to write rules like the following:

```
r(x,y) = @max [1<=x<2]*[3<=y<4] + [3<=x<4]*[5<=y<6]
```

These rules have no patterns in their bodies, so they are never triggered by edits to working memory; they only get added to the change list once, during initialization, so that they can be fired and place their output into working memory.

[28]An earlier design provided an automatic facility which would create these intermediate predicates and merging rules for the user. However, the present design with truly separate buffers proved more natural. Expanding the list of predicates and rules during compilation makes the system harder to interpret and debug. Also, a full rule computation for result-merging is wasteful, since no dimension-handling or summarization is necessary (although the extra cost can be made small). Finally, it over-complicated compilation in comparison with the current solution.

In this example, two non-overlapping squares get set to one, and everything else gets set to zero (because an inequality is represented by a function which is one where the inequality is true and zero everywhere else). Constraints can be multiplied by coefficients to store values other than zero or one.

However, if rules *overwrite* working memory rather than combining with what's there, rules like this won't always serve their intended purpose. The initialized data will be erased when other rules do their work.

Consider combining the initialization rule given earlier with a transitivity rule:

```
r(x,z) = @max r(x,y)*r(y,z)
```

What we want this rule to do is enforce the transitive closure of the relation `r`, by adding new locations with value one wherever they are implied by the existing nonzero values. If rules overwrote working memory with their results, this rule would instead delete the initial nonzero value the first time it fired, writing `[1<=x<2]*[5<=y<6]` to working memory. In the next iteration after that, it would zero out the contents of the predicate `r` entirely, since the rule's output would not contain any nonzero values at all. This would become the end result, since another iteration changes nothing, and after that no further rules would be triggered.

See Section 3.5.6 for more details on how the merging algorithm works.

3.4.5 Why Not Combine Directly?

One might try to combine results with working memory directly, rather than using output buffers to hold un-combined results and re-combining them every time a rule fires. However, this gets complicated.

It is important that re-firing a rule doesn't combine results repeatedly. Although we would like to avoid unnecessary rule computation, we don't want the end result to depend on the number of times we fire a rule.

For example, `f(x) = @sum 1` initializes `f` to have the value one everywhere, but we would not want it to repeatedly add 1. This wouldn't happen, since rules like this only get fired once, but that should be an optimization rather than an essential part of how execution works.

To give an example where a rule *could* actually fire more than once, consider something like `f(x) = @sum 1 + g(x,y)`. If this is the only rule which has `f` as its target, the result should be to set `f` to one greater than the summation $\sum_y$ `g(x,y)`, *not* to add one to everything whenever any instance of `g(x,y)` changes.

To solve this problem, one would have to combine *changes* into working memory, rather than literal results. For example, `@sum` predicates would have to combine differences into working memory. This means we would need output buffers anyway, in order to keep the previous output of a rule for comparison.

This could work for `@sum` rules, but would not work in other cases. A `@min` rule doesn't have an appropriate notion of difference. If some output values have decreased, either leaving things unchanged or achieving a new minimum, that's easy to handle; however, if outputs increase (particularly if outputs which were previously minimal), then there's no way to know how the values in working memory should be changed. So, at least for those cases, it is necessary to re-compute the minimum across all rules with overlapping targets.

The current system takes the re-computation approach in all cases, for simplicity. The more incremental approach applied where possible, such as for `@sum`, could be a beneficial optimization in the future (particularly for examples with large numbers of overlapping rules).

3.5 More Design Details

While Section 3.4.1 gave a high-level picture of how Signia programs are executed, many important implementation details were excluded in service of clarity. This section fills in those details for the interested reader.

3.5.1 Anti-Signatures

Several algorithms which recursively descend `MemTree`s need to refine `Domain`s as a tree is descended, in order to track the increasingly restricted space based on which branch is followed. For the most part, this is simple. However, `TSplit`s present a problem.

Recall that `TSplit`s hold a list of children, each with a specified signature refining the tuple variable being split. These branches can be descended straightforwardly, using the specified signature to refine the targeted variable.

However, each `TSplit` additionally has a *default* branch, which contains "everything else" – anything not fitting into one of the refinements listed explicitly. Its domain is defined negatively with respect to the other domains; it can be visualized as a region with holes punched in it for each oft he explicitly given branches.

It would overcomplicate matters to fully support negations of domains. Domains already need to be closed under intersection. Supporting negation would make them into a Boolean algebra, which would make checking for intersection NP-complete, and checking for equality between domains co-NP complete. It would be better

`SpecifiedTSig`:

Specification: A vector of `Signature`s, giving us required signatures for each element of the tuple.

Anti-Signatures: A `Set` of vectors of signatures. Each vector of signatures here represents a combination of signatures we *don't* want to allow. These can be more specific than the positive requirements in the specification.

`UspecifiedTSig`:

Anti-Signatures: A `Set` of vectors of signatures, as above.

Figure 3.10: The tuple signature data-types. The type `TSig` is a union of the two types above.

to avoid this, if possible. This is analogous to the way we prefer to keep regions convex in the first-order representation.

Fortunately, the added complexity is not necessary. Negation only occurs in this very specific context, which means it only applies to tuple signatures. So, tuple signatures (specified or unspecified) contain a list of "anti-signatures": a list of vectors of signatures. See Figure 3.10. Each vector is interpreted as a specification of a tuple signature; after all, a vector of signatures is like a `SpecifiedTSig` except it doesn't have any anti-signatures of its own. This helps prevent the full complexity of Boolean algebras.

Technically, it is still possible to construct a disjunctive domain, because tuple signatures may exist recursively within the anti-signatures, and these may contain anti-signatures of their own. However, such problematic cases are never created by Signia.

Anti-signatures need to be handled in a variety of basic functions: equality, intersection, specialization, and of course in the original motivating cases – functions which track domains as they recursively descend trees.

3.5.2 Comparison

It is necessary to compare spatial trees at various points, to check whether they represent the same function (that is, whether they hold the same distribution of values). There are two types of equality in Signia:

- '`==`': Relatively fast comparison, very close to a syntactic equality check. Recursively descends the spatial trees and related data-structures and checks whether values are identical.
- '`epsilonequal`': Relatively expensive "semantic" equality check, which identifies corresponding regions in the two trees, and checks whether they hold values within epsilon of each other. (A value of epsilon = 0.0000001 is used unless the user specifies something else.)

The main difference between the two notions of equality is that the tree-structure needs to match in the first case, but not in the second case. For example, the Figure 3.11 illustrates two BSP trees which would be epsilon-equal, but not ==.

Because `epsilonequal` is a more expensive operation, it is preferable to check `==` when possible. However, `epsilonequal` is often needed.

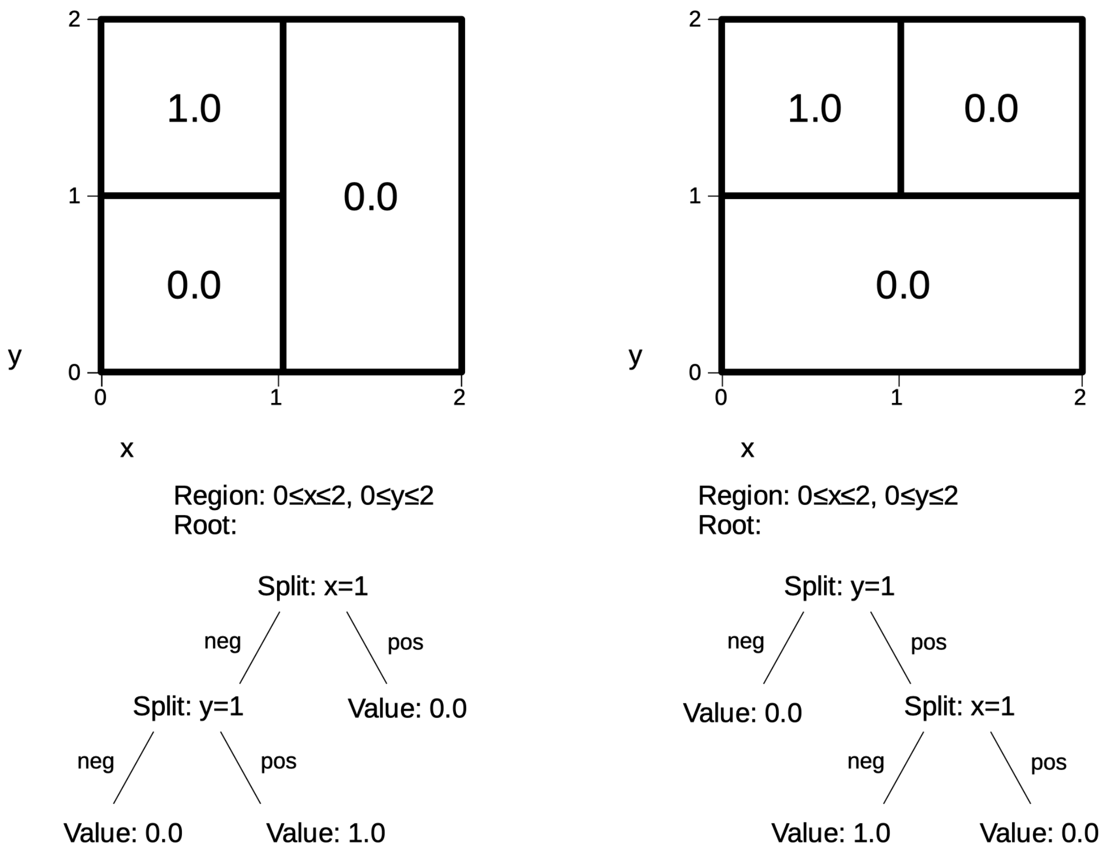


Figure 3.11: Two `BSPTree`s which are `epsilonequal`, but not `==`.

The major use of `epsilonequal` in Signia is in the execution cycle (see Section 3.4): a result is checked against the contents of the execution buffer, to see whether there is any change in value. If not, no further action is taken to propagate that result. It's important to check `epsilonequal` in this case, since true changes in value are what matter.

Other than that, `epsilonequal` is useful within summary calculations; see Section 3.5.4.

Algorithm 1 Epsilon-equality for `BSPTrees`.

```
 1: function EPSILONEQUAL(t1::BSPTree, t1::BSPTree, epsilon::Float)
 2:     if not(intersect(t1.boundary, t2.boundary) then
 3:         return true          ▷ If they have no intersection, they trivially agree.
 4:     else if t1.root is a SplitNode then
 5:         lefteq ← epsilonequal(left(t1), t2, epsilon)
 6:         righteq ← epsilonequal(right(t1), t2, epsilon)
 7:         return lefteq & righteq
 8:     else if t2.root is a SplitNode then
 9:         lefteq ← epsilonequal(left(t2), t1, epsilon)
10:         righteq ← epsilonequal(right(t2), t1, epsilon)
11:         return lefteq & righteq
12:     else if t1.root and t2.root are both None() then
13:         return true
14:     else if t1.root or t2.root is None() then
15:         return false                          ▷ None() must match None().
16:     else
17:         return epsilonequal(t1.root.value, t2.root.value, epsilon)
18:     end if
19: end function
```

Algorithm 1 gives the pseudocode for `epsilonequal` on `BSPTrees`. The 'argument::Type' notation indicates argument type. (Types and type constructors are capitalized.) The 'left' and 'right' functions (lines 5, 6, 9, and 10) follow one branch or the other of a tree. Recall from Section 3.5 that a `BSPTree` includes a region, giving the boundary over which the `BSPTree` is defined. So, 'left' and 'right' are not just `t.root.neg` and `t.root.pos`; they also have to cut the boundary region in half using `t.root.split`. This does not involve any expensive linear algebra, however; in only involves adding one more constraint to the list.

Note that the function only compares data within the intersection of the boundaries of the two `BSPTree`s; if the two boundaries do not overlap, they are considered trivially epsilon-equal, rather than inequal (lines 2-3). On the other hand, `None()` values in one tree must be matched with `None()` values in the other tree. (Otherwise, results which place a non-`None()` where a `None()` previously resided would be ignored!) This means that `None()` and out-of-bounds are very different.

The 'intersect' check (line 2) is a significant expense, because it requires checking whether all the linear constraints are collectively consistent.

In the last case (lines 16-17), the recursion must have reached two `LeafNode`s, by process of elimination. Their values are compared by calling `epsilonequal` on numeric values (ie, `Hyper`s or floating-point numbers). This may appear to continue the recursion, but actually, it calls a different definition of `epsilonequal` than the one shown (`epsilonequal` being an overloaded function, with several definitions) – the definition being `abs(x-y) < epsilon`. This acts as the base case for `BSPTree` comparison.

`epsilonequal` also has a few other definitions, not shown here. Most importantly, `epsilonequal` is also defined for `MemTree`s. The algorithm follows a similar idea to that of Algorithm 1, but with cases for each different type of split. Where the `BSPTree` algorithm bottoms out in numerical comparison, the `MemTree` algorithm bottoms out in `BSPTree` comparison.

3.5.3 Combination

The term "combination function" has been used to refer both to basic binary operations such as sum and product, and to the functions which are compiled from the full rule body. The present section will illuminate the connection between these two.

Section 3.2.1 described the rule body as consisting of arbitrary Julia code. Although this is true, the Julia code needs to be able to manipulate Signia's spatial trees. This means that arbitrary numerical operations, such as addition, subtraction, multiplication, and division, won't work unless these functions have been overloaded to handle `MemTree`s (and, in service of this, `BSPTree`s).

So, for the most part,[29] the rule body is a complex combination function built up of more basic combination functions. These more basic functions need to be given as Julia code.

However, there is a good deal of commonality between these basic combination functions. Re-writing the same tree-manipulation code for each new binary combination function would be tedious and error-prone. Instead, a meta-programming approach is used: Signia creates and compiles appropriate code based on a little

[29]Setting aside constraints and predicate patterns.

information about each binary combination function. This makes it quite easy to extend the set of supported combination functions.

There are two such functions: `treearithmetic` for the `BSPTree` data-type, and `memarithmetic` for the `MemTree` data-type. In both cases, the compilation function takes the following arguments:

Operator: The binary operation to be generalized via overloading. For example, `+` or `*`. This need not be commutative.

Base Type: The type which the operation is already defined on, to use in the base case for recursive definitions. Usually `Number`, Julia's general supertype for any numeric type.

Algorithm 2 sketches all the new definitions which `treearithmetic` adds for the given operator. The general idea is the same for `memarithmetic`, but more complicated due to the larger number of cases in the `MemTree` type.

With these meta-programming functions in hand, declaring new combination operators is very simple. Currently, the declared operators are: `+`, `*`, `max`, `min`, `/`, `-`, and `^`.

As mentioned briefly in Section 3.4.1, the combination function for a rule is compiled from the rule body by (1) replacing occurrences of predicate patterns with variables, which become argument names when the whole body is converted into a function; (2) replacing all constraint expressions with the required `MemTree`

Algorithm 2 Generic combination definitions for `BSPTree`s, given an `operator` and `base_type`.

```
function OPERATOR(n1::BSPNode, n2::base_type)
    if n1 is a LeafNode then
        return LeafNode(operator(n1.value, n2))
    else if n1 is None() then
        return LeafNode(n2)
    else if n1 is a SplitNode then
        return SplitNode(n1.split, operator(n1.neg, n2), operator(n1.pos, n2))
    end if
end function
function OPERATOR(n1::base_type, n2::BSPNode)
    if n2 is a LeafNode then
        return LeafNode(operator(n1, n2.value))
    else if n2 is None() then
        LeafNode(n1)
    else if n2 is a SplitNode then
        return SplitNode(n2.split, operator(n1, n2.neg), operator(n1, n2.pos))
    end if
end function
function OPERATOR(t1::BSPTree, t2::BSPTree)
    i ← intersection(t1.boundary, t2.boundary)
    if t1.root is a LeafNode then
        if i is nonempty then
            return BSPTree(i, None())
        end if
        return BSPTree(i, operator(t1.root.value, trim(t2, i).root))
    else if t1.root is a SplitNode then
        l ← operator(left(t1), t2).root
        r ← operator(right(t1), t2).root
        return BSPTree(i, SplitNode(t1.root.split, l, r))
    else if t1.root is None() then
        return t2
    else if t2.root is None() then
        return t1
    end if
end function
function OPERATOR(t1::BSPTree, t2::base_type)
    return BSPTree(t1.boundary, operator(t1.root, t2))
end function
function OPERATOR(t1::base_type, t2::BSPTree)
    return BSPTree(t2.boundary, operator(t1, t2.root))
end function
```

representing the constraint; (3) creating a domain for the output `MemTree` which collapses all matched dimensions together.

3.5.4 Summarization

Like combination operations, summary operations share quite a bit of structure, so meta-programming is an appealing option. The function `treesummary` compiles summary code for `BSPTree`s, while `memsummary` handles `MemTree`s.

Both `treesummary` and `memsummary` take the following arguments:

Name: The name for the new summary operation. (Unlike combination operations, summary operations don't overload existing functions; they must be given their own name.)

Leaf Summary: The base case of the recursive definition: how does the summary op handle leaf values?

Combination Operation: This determines how results are combined when summarization "smushes" everything together along a dimension. Integration uses `+`, maximization uses `max`, and so on.

Identity Value: This should be the identity element for the combination operation just given. Among other things, the new summary operation will tread `None()` values as if they were this value. This allows predicates with partially-defined values to have meaningful partial summaries.

The generic summary code for `BSPTree`s is fairly simple, as shown in Algorithm 3. The idea is to recursively build up a summary by projecting and combining regions, with two base cases: (1) replace empty regions with identity values (reflecting the intuition that `None()` regions should be 'ignored'); (2) apply the leaf summary operation to non-empty leaf regions.

Algorithm 3 Generic `BSPTree` summary algorithm, given a name, leaf_summary, comb_op, and identity.

```
1: function NAME(tree::BSPTree, dim::Int)
2:     if tree.root is None() then
3:         return BSPTree(project(tree.boundary, dim), LeafNode(identity))
4:     else if tree.root is a LeafNode then
5:         return leaf_summary(tree, dim)
6:     else
7:         b ← project(tree.boundary, dim)
8:         l ← name(left(tree), dim)
9:         l ← BSPTree(b,unfoldboundary(l, LeafNode(identity)))
10:        r ← name(right(tree), dim)
11:        r ← BSPTree(b,unfoldboundary(r, LeafNode(identity)))
12:        return comb_op(l, r)
13:    end if
14: end function
```

Projection (lines 3, 7) means taking a region defined by a system of linear constraints, and finding its projection onto the subspace resulting from removing a single dimension. More explicitly, we have a system of constraints on n variables $x_1, x_2, ...x_n$, and we are seeking a new system of constraints on $n-1$ variables, $x_1, x_2, ..., x_{i-1}, x_{i+1}, ..., x_n$ (removing some x_i), *such that* a set of assignments to the $n-1$ variables constitutes a solution to our new constraints if and only if there exists *some* value for x_i which would make it a solution to the old constraints.

This is accomplished via a slight modification of the Fourier-Motzkin elimination method [3]. Recall from Section 3.3.1 that the notion of linear constraint in Signia can be open or closed. This required an elaboration of the usual Fourier-Motzkin method to track whether bounds are open or closed.

The `unfoldboundary` function (appearing on lines 9 and 11) takes a `BSPTree` and a `BSPNode`. The boundary constraints of the `BSPTree` are converted into splits for new nodes; the root of the tree is placed in the region corresponding to what was the boundary, while the second argument is placed everywhere else. Figure 3.12 clarifies this with an illustration.

In Algorithm 3.12, the point of 3.12 is to convert things which are out-of-bounds into identity values. This allows the two branches to be combined correctly, since the newly-summarized branches may now have overlapping parts (due to the removal of one of the dimensions).

Four integration functions are defined via the metaprogramming function: `integrate`, `pintegrate`, `maxintegrate`, and `mintegrate`. The function `integrate` implements standard continuous-sum integration. Its combination operation is `+`, and its identity value is `0`. The function `pintegrate` implements product-integration (also known as the geometric integral), which is the exponential of the integral of the log:

$$\rho f(x)^{dx} = \exp\left(\int \ln(f(x))dx\right)$$

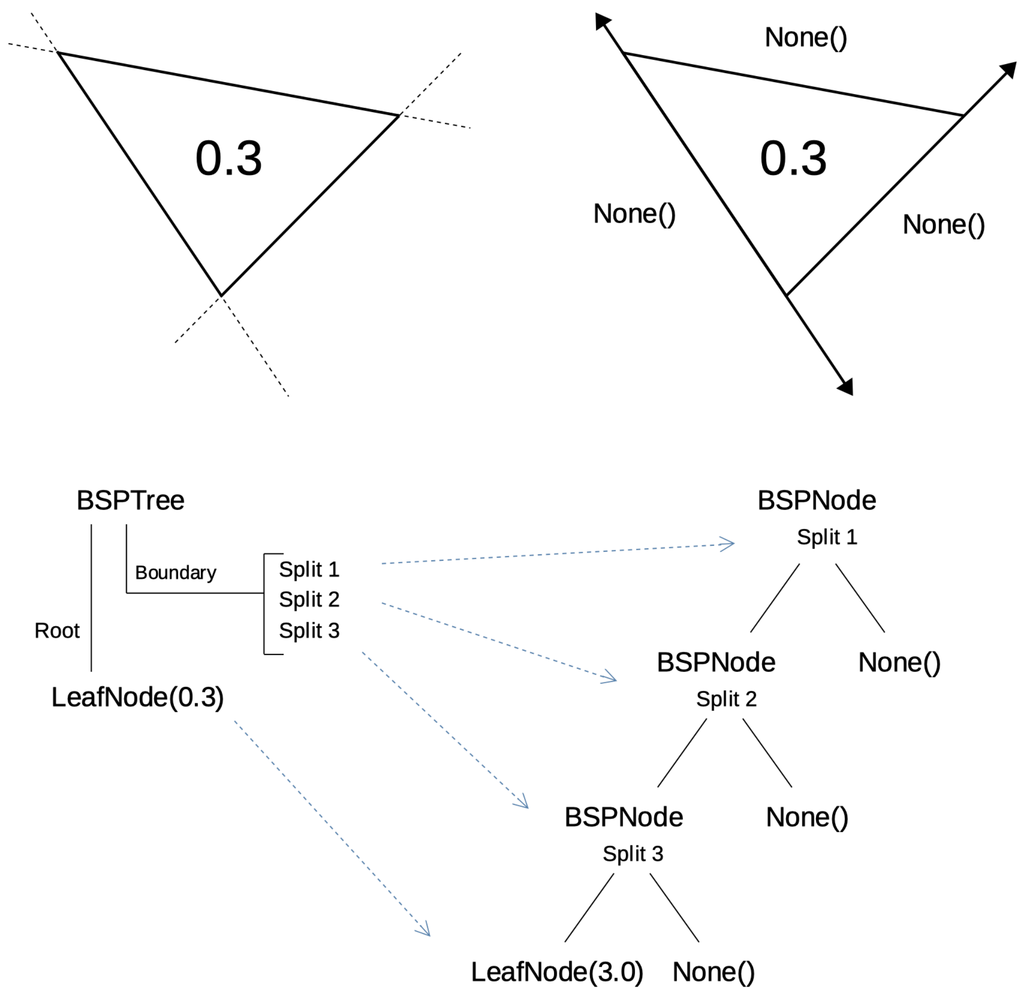


Figure 3.12: Left: a `BSPTree`. Right: the result of applying `unfoldboundary` to the `BSPTree`, with `None()` as the second argument. Light blue lines indicate where items were sent.

(Here, a large ρ is being used to represent the product integral; notation for product integrals is not standardized.) We get a function for this out of our metaprogramming code by giving the combination operation `*`, and identity value `1`.

The function `maxintegrate` takes the maximum value over the dimension being removed. Its combination operation is `max`, and its identity is `-Inf`. Similarly, `mintegrate` takes the minimum, with combination `min` and identity `Inf`.

So much for names, combination operations, and identity values. What about the leaf summaries?

3.5.4.1 Leaf Summaries

While all four summary functions require a leaf summary, the version for `maxintegrate` and `mintegrate` is fairly simple. Since the current system only handles piecewise constant functions, there isn't anything to max/min at a leaf; the leaf summary only has to project the boundary region.

Integration and product-integration, on the other hand, have complicated leaf-integration routines. The integral of a constant function over a region defined by linear constraints can, in general, bu a piecewise linear function. This means that the result needs to be (1) piecewise: a single value isn't enough, unlike max/min summary; (2) approximate: the linear functions need to be approximated via piecewise-constant functions.[30]

[30]The obvious class of functions to avoid a need for approximation would be piecewise-polynomial, but it seems likely that this would be overkill for most applications, and that the cost would be too high.

Here is an outline of how Signia currently handles things:

1. Find the faces of a leaf region. To test each constraint to see whether it constitutes a face: replace that constraint with an equality constraint on the same hyperplane; check whether the resulting system is feasible. If so, add the new system to the list of faces.[31]

2. Check whether a face is a "lower face" or an "upper face" in terms of the dimension being summarized out. In other words, if we made a vector point "inwards" (into the leaf region) from the face, would the vector have a positive dot product with the dimension we're summarizing out, or negative? (If neither, we can safely discard.)

3. Make two modified `BSPTree`s holding the upper fares and lower faces. Imagine it this way. If the original space had dimensions x, y, the new trees are in a space with only x, but store the y-values of the upper (/lower) surface *as values* in the leaves. So, the new trees represent the upper and lower surfaces as functions. These are not legitimate Signia `BSPTree`s because they are piecewise-linear rather than piecewise-constant.

4. Subtract the lower surface from the upper surface.

[31]This method of finding faces is imperfect. It is possible that "faces" of lower dimensionality (such as edges or corners) will be included by mistake. This does not create any errors in the output, but a more accurate test would be better so long as it was not more computationally expensive.

5. Transform the result into a legitimate `BSPTree` by taking a constant approximation of each linear function. The midpoint between the extreme values is used.

The resulting `BSPTree` approximates the differences between upper and lower faces. This can then be used by `integrate` (multiply the leaf value by the differences) or `pintegrate` (exponentiate the leaf value by the differences).

3.5.4.2 Problems Integrating TSplits

The details so far have mainly been about the first-order summary functions. As with combinations, the second-order version is more complicated, but follows the same general outline: recursively descend the tree, removing the dimension which is being summarized out, combining things via the given combination operation.

However, there is one particularly problematic case which warrants extra discussion: the integration of `TSplit`s.

As a reminder, a `TSplit` has the following fields:

Variable: Gives the tuple variable being split, via its `VarLoc` (the location in the signature where the variable lives).

Branches: Gives a set of branches; each branch for a different way of specifying the tuple.

Default: Any tuple can be defined in an infinite number of ways, so, the possibilities can never be fully covered by the explicitly given branches. The default value covers the rest.

When summarizing a tuple variable, there are *effectively* an infinite number of branches, since there are an infinite number of ways the default value could turn into a branch. This is absolutely fine for max-summarization and min-summarization. However, it causes a big problem for integration and product-integration.

For integration, the appropriate operation would apparently be an infinite sum over all possible tuples (besides those already in the branches). However, this sum would produce an infinite value greater than any infinity currently represented in Signia, because *each possible tuple has to be summarized as well*, and we can construct a tuple which integrates to any finite power of infinity.[32]

This isn't actually a problem for min/max-integration, since the min or max of the default branch's contents over infinitely many possible signatures is still just the same thing.

It also isn't a problem for integration if the default branch contains nothing but zero. Similarly, it isn't a problem for product-integration if the default branch contains nothing but one. Both of these cases are, in fact, very common; it's difficult for default branches to get non-default values, since if we specifically add something

[32]Without any bounds, the tuple signature `(:v)` integrates to `Hyper(1,1)` times whatever value is stored in the default branch of the `TSplit`; `(:v,:v)` would give a coefficient of `Hyper(1,2)` (infinity squared); and so on. The "correct" result would seem to be the sum of all these.

to a tree, we generally give it a specific signature (meaning it should end up in a non-default branch, rather than landing in the default branch).

However, the problematic case *can occur*, especially when there are two different `TSplit`s on two different tuple variables. This is why Signia cannot currently handle two different tuple constraints in one rule. There is no general way to avoid putting one `TSplit` into the default value of another.[33]

Currently, all summary operators handle this confusing case by simply returning the default value, summarized to ensure that the dimension being summarized out does not occur in the output (eg, a second `TSplit` on the same tuple variable). This is the correct thing to do in cases where there is a correct thing to do.[34]

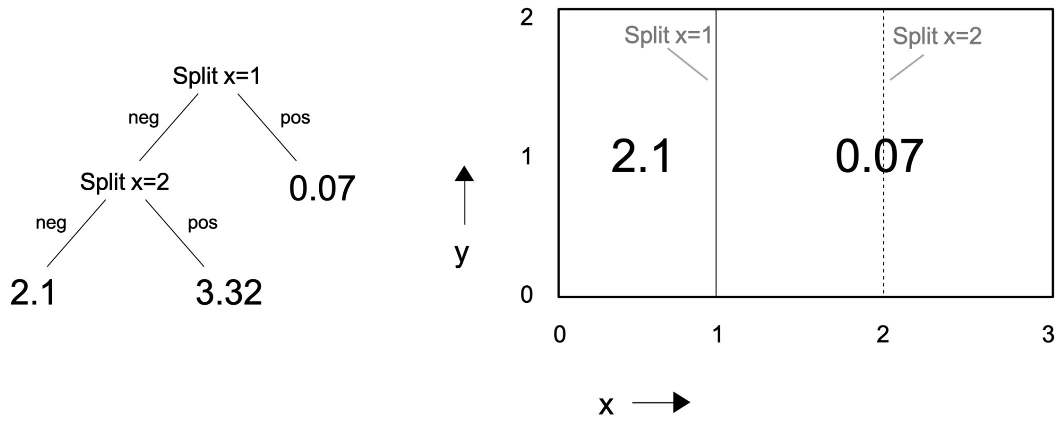
3.5.5 Trimming

Both `BSPTrees` and `MemTrees` can contain useless branches. Figure 3.13 illustrates an example. The $x = 2$ split has been placed in a region where x never equals 2; namely, the region $x < 1$. Therefore, only one side of the split gives a meaningful value.

It would be nice if such aberrations were never introduced into trees, since larger trees take up more memory and take more time to process. However, checking the

[33]It doesn't do to require all `TSplits` to carry the appropriate identity value in their default branches, because locally, a tree doesn't know what summary operation is going to be applied to it; so, the value would have to change, possibly leading to confusing and incorrect results. Furthermore, such a restriction would prevent us from applying arbitrary functions to trees, such as adding 1. It would also deprive us of the simple correct answers we can give in the case of min or max summarization.

[34]A technically correct solution might be achieved by introducing larger infinite values, but even this is rather unclear. Even if so, is such a higher infinity ever *what we really wanted to compute*?



The positive branch of the $x = 2$ `SplitNode` doesn't represent a nonempty region. The tree could be trimmed by making 2.1 the negative child of the $x = 1$ `SplitNode`, eliminating the other split and the leaf value 3.32.

Figure 3.13: A sketch of a `BSPTree` containing an inconsistent branch.

consistency of a branch is somewhat costly, because we have to check the consistency of a linear program.[35] So, it doesn't make sense to ensure consistency throughout every computation. Instead, many operations can result in untrimmed trees, but trees are trimmed at various points to avoid letting them become too large.[36]

The pseudocode for first-order trimming is shown in Algorithm 4. Note that "valid/invalid" refers to a consistency check on the linear equations stored in the tree boundaries, *not* to the contents of the roots. So, each validity check is relatively expensive. The `epsilonequal` check is also relatively expensive, but the benefit in reduced tree sizes is worth the cost.

[35]Specifically, we have to check the consistency of all the linear constraints implied by the path we've traveled down the tree so far.

[36]Furthermore, beyond trimming, no systematic effort is made to optimize trees. Existing tree optimization techniques, such as [26], might provide significant speed benefits – although the use-case of BSP trees here is different from the usual.

Algorithm 4 Trimming for `BSPTree`s.

```
 1: function TRIM(tree::BSPTree)
 2:     if tree is invalid then
 3:         return BSPTree(tree.boundary, None())
 4:     else if tree.root is a SplitNode then
 5:         pos ← takepos(tree)
 6:         neg ← takeneg(tree)
 7:         if both pos and neg are valid then
 8:             trimmed_pos ← trim(pos).root
 9:             trimmed_neg ← trim(neg).root
10:             neg_compare ← BSPTree(tree.boundary, trimmed_neg)
11:             pos_compare ← BSPTree(tree.boundary, trimmed_pos)
12:             if epsilonequal(pos_compare, neg_compare) then
13:                 return BSPTree(tree.boundary, trimmed_pos)
14:             else
15:                 snode ← SplitNode(tree.root.split, trimmed_neg, trimmed_pos)
16:                 return BSPTree(tree.boundary, snode)
17:             end if
18:         else if pos is valid then
19:             return BSPTree(tree.boundary, trim(pos_tree).root)
20:         else                                    ▷ neg must be valid in this case
21:             return BSPTree(tree.boundary, trim(neg_tree).root)
22:         end if
23:     else                            ▷ root is None() or a LeafNode; nothing to trim
24:         return tree
25:     end if
26: end function
```

The idea behind second-order trimming is very similar: descend the various types of `MemTree` nodes, eliminating redundant or useless branches. Figure 5 shows abbreviated pseudocode for the procedure.

Algorithm 5 Trimming for `MemTree`s.

```
function TRIM(tree::MemTree)
    if tree.domain.sig is ESig() then
        return an empty tree
    else if tree.domain.sig is ruled out by its own antispec then
        return an empty tree
    else if tree.root is a TSplit then
        if tree.domain.sig falls inside one of the TSplit branches then
            trimmed ← that branch, trimmed
            return MemTree(tree.domain, trimmed)
        end if
        Remove branches inconsistent with tree signature.
        Trim all branches (including the default branch).
        Remove branches epsilonequal to the default branch.
        if any branches remain then
            return tree
        else
            return BSPTree(tree.domain, root.default)
        end if
    else if tree.root is a PSplit then Trim very similarly to first-order splits.
    else                                ▷ root is a MemLeaf, which holds a BSPTree
        return the BSPTree, trimmed.
    end if
end function
```

3.5.6 Merging Ambiguous Cases

As discussed in Section 3.4, if rules output to overlapping areas of working memory, the results must be *merged*; that is, combined together using a combination operation which is selected based on the summary operation used by the rules. In the ideal case, the summary operations of the overlapping rules match. When this does not happen, the intention behind the rule system is ambiguous; however, Signia must decide how to handle it somehow.

Section 3.4.2 discussed how summary operations also determine how working memory is initialized. Each working memory region is initialized to the identity element of the combination version of the summary operation. For example, `@sum` rules initialize their predicates to 0.0, while `@prod` rules initialize their predicates to 1.0. This fits with the idea that `@sum` rules add their results to working memory, while `@prod` rules multiply their results in. However, the ambiguous cases create a problem here, as well; that section discussed how different summary operations are considered in a precedent ordering, when initializing working memory.

Section 3.4.2 also mentioned that the initialized working memory is saved as `local_defaults`, and plays a role in how ambiguous cases are handled. The current section describes how this works.

First, a rule's output is merged with overlapping outputs of rules *of the same summary type*. This is placed in the appropriate merge buffer. Then, if there are any *other* overlaps, then the overlapping content *in all merge buffers* is combined *in priority order*, but starting with the initial value saved from initialization. Algorithm 6 sketches how this works. This can still produce strange/unexpected results, but is useful enough for the ambiguous cases considered in the current work.

Algorithm 6 Procedure for storing rule outputs in working memory.

```
1: function STORE_RESULT(rule_output::MemTree, rule::Rule, rs::RuleSystem)
2:     insert(rule_output, rs.output_buffers[rule.number])
3:     result ← merge_same_type(rs, rule_output, rule)
4:     insert(result, rs.merge_buffers[rule.merge_type])
5:     if rules of other merge types overlap then
6:         result ← query(rs.local_defaults, result.domain)
7:         for merge_op in merge_priority_order do
8:             result ← merge_op(result, rs.merge_buffers[merge_op])
9:         end for
10:    end if
11:    insert(merged, rs.wm)
12: end function
```

3.6 Design Alternatives

This chapter has reviewed many design decisions which went into the current implementation of Signia. However, there are many alternative design ideas to explore. Some of them are likely much better. The present section discusses some potential modifications to the design.

3.6.1 Hyperreals & Constraints

Section 3.2.2 and 3.3.1 discussed Signia's first-order constraints, and the use of Dirac delta functions to represent them. This implementation leverages the spatial-tree representation in an interesting way. However, although workable, there is at least one major downside to this representation: the representation of a constraint depends on what summary operation the data will be interacting with.

For `@sum` rules, the hyperreal number which must be held in the delta function in order for it to behave like a constraint as expected depends on the number of dimensions being summed out, and on the *angle* of the constraint hyperplane with respect to those dimensions.

For example, suppose the predicate `f(x,y)` contains some data, but we want to extract the data satisfying the constraint $x = y$. We might try to do this with a rule like:

```
g(x) = @sum f(x,y) * [x=y]
```

If only integer values of x and y were being considered, this would do exactly what we want. However, because Signia uses a continuous domain, the results will be roughly 1.414 times what's desired. The values are not merely projected onto the x-axis, but rather *integrated*. The diagonal line $x = y$ has length $\sqrt{2}$ for every length-one interval of x, so the data gets multiplied by this amount.

Although one could argue that these are correct results, they do not fit the intuition of restricting instances of a rule to only cases where $x = y$.

We can compensate for this by multiplying by $1/\sqrt{2}$ in the rule. Other line angles get you other adjustments, which require basic trigonometry to calculate.

Further adjustments may be needed in other cases. Signia places `Hyper(1,1)` inside the constraint by default, so that the result of integration in the rule given earlier is not infinitesimal. However, this won't be right for a three-dimensional case where we want to extract the data at $x = y = z$; the results would be infinitesimal in this case, so to avoid that, we would have to multiply by infinity again in the rule.

For `@max` and `@min`, we don't need to multiply by a hyperreal number or by a trigonometric adjustment. This is nicer, but *different*, meaning that the user has to remember when such an adjustment needs to be made. The use of value 0.0 for areas outside of the given constraint is often inappropriate as well, depending on exactly what one is doing.

Overall, representing constraints as Dirac delta functions made the most sense for the `@sum` case, and even there, is not as straightforward as it could be.

In principle, most or all of these adjustments can be made automatic.[37] (The implementation as of this writing doesn't do very much for you; it places `hyper(1,1)` in all constraints.) However, the choice to represent constraints as Dirac delta functions still seems questionable. On the other hand, it isn't clear what should be done instead. One might possibly do away with the idea of representing Dirac delta functions explicitly in the BSP tree function representation. Constraints would be handled during rule computation, but could not be manipulated as functions or stored as values in working memory. This would unfortunately make the "link table" used in the belief propagation example (Section 5) impossible. In fact, supporting this functionality was a primary consideration in the current design.

[37]At least, the appropriate adjustments to make a constraint interact as expected *with the summary operation of its native rule* could be automated. Because delta functions can be stored in working memory and pass from rule to rule, one must worry about how they interact with *other* rules, possibly of different summary types, as well.

Another route would be to abandon the rigorous treatment of Dirac delta functions, instead opting for intuitively reasonable behavior. The current implementation *explicitly* multiplies results by $\sqrt{2}$ when integrating constraint functions like $x = y$. In the same way, it *explicitly* divides by (hyperreal) infinity when integrating regions of zero width. At the cost of mathematical rigor, one could instead simply do nothing. Regions of smaller and smaller width would amount to less and less in the results of integration, until a region's width fell below `epsilon`; at that point, the contents of that region would be faithfully added to the end result without any adjustment, just as if the width of the region (along the dimension being integrated) had jumped back up to one.

This would result in a system which passed around constraint "functions" which are mathematically much more exotic than Dirac delta functions, but, whose behavior would be more intuitive.

A problem which remains, in that proposal, is that 0.0 is not always the appropriate value for the out-of-constraint areas of a constraint function.

3.6.2 Hyperreals vs. Generalized Functions

The present work represents Dirac delta functions and other generalized functions by representing infinite values, as discussed in Section 3.3.1. An alternative (if still using such functions at all, despite the concerns raised in the previous section) would be for regions to contain a floating-point "mass". The current representation interprets the value held in a leaf node as the value which the function evaluates to, at any point within the relevant region. A “mass” would instead tell us *what the whole region would integrate to*, if we were to integrate out all the dimensions. This number is potentially more useful, since the internal calculations never simply evaluate a function at a point.

Splitting a region would involve calculating what percentage of the mass ends up in which half, which makes combination-type calculations more complex. On the other hand, integration becomes simpler.

This representation change might fix one of the issues in the previous section: a multi-dimensional constraint's weight would not need to be multiplied by infinity for each additional dimension. However, it could also create other problems; for example, what mass can we give a constraint of infinite extent, like $x = y$?

3.6.3 Eliminating Spatial Trees

The experiments in Chapter 4 reveal that spatial trees do not compare favorably to other representations in terms of efficiency. If the efficiency of the current implementation cannot be significantly improved, this suggests that they should either be abandoned, or used only in cases where the generality is really required, using more specialized and optimized representations for most computations. Chapter 5 will discuss some options.

Chapter 4

Analysis

We have now introduced Signia in some detail. How well does it achieve the design goals which were discussed in Chapter 1? The current chapter will examine what Signia has achieved and failed to achieve. Qualitative analysis will be intertwined with quantitative results.

4.1 Stating Algorithms Generically

As discussed in Chapter 1, the main purpose of Signia's second-order extension to WLP is to enable algorithms to be stated in a "generic" way, where they can apply to new predicates readily. This section will walk through the example of belief propagation (BP), to show how this works. Examples will be tested, showing that this generality is achieved at a significant cost of speed.

Recall that one of the major motivations for selecting WLP as our interface layer was how close it is to BP. BP is an instance of the summary-product algorithm,

```
@rules begin
  f(x,y) = @prod constraints specifying a function
  v1(x) = @sum f(x,y)
  v2(y) = @sum f(x,y)
  f(x,y) = @prod v1(x) * v2(y)
end
```

Figure 4.1: BP in WLP, attempt #1. The constraints which would appear in the first rule are not shown.

which is a highly general algorithm (as discussed in [19]). WLP seems to generalize things even further. The semiring concept of the general summary-product algorithm closely mirrors the concept of summary (aka aggregation) and combination in WLP. Furthermore, WLP's idea of mixing different summary and combination operations mirrors Sigma's free intermingling of the summary-product semirings. So: is this only a resemblance, or can we make this idea precise? To show that BP is a special case of WLP, we need to *find* that special case, producing a WLP program which implements BP.

In particular, WLP's convention of summarizing out any variables which occur in a rule body but not the rule head seems like a good fit for the summary-product idea of summarizing out all variables not present in a variable node, to compute the message heading to that variable node. This idea leads us to BP-in-WLP attempt number 1, in Figure 4.1.

The second and third equations capture the idea of `v1` and `v2` marginalizing `f`. However, the messages back to `f` are not correct. The variables are sending back the full marginals, ie, the product of all their incoming messages. (In this case, there is only one incoming message for each variable, but in general there can be many.) In general, the message from a node a to a node b involves only the product of the *other* messages into a, *not* including the message coming from a. (In this case, since there is only the one message *to* each variable node, this means sending back all 1s.) This has a surprisingly intuitive interpretation: it prevents circular reasoning. If an initial bias in beliefs were added to Figure 4.1, this could easily spiral into extreme beliefs for the two variables (depending on the function we put in `f`). In this particular case, the rule system would correctly compute marginals if we simply removed the messages back to `f`. However, this would not allow for the possibility of adding an initial bias to the variables, or extending the model with more variables or factors.

One reason we can't easily extend the rules in Figure 4.1 to larger networks is that variable nodes are supposed to *multiply* all their incoming messages. If we simply add more factors connecting to these variables in the way `f` does, the incoming messages would be *added*, thanks to how result-merging works in Signia.

Unfortunately, this means the elegant rules of Figure 4.1 won't do. The connection between WLP and BP is not quite so direct. We have to use the markedly more complex rules in Figure 4.2.

Here, we divide each message by the message heading in the other direction. This is equivalent to forming a node's outgoing message along a link using only the

```
@rules begin
  f(x,y) = @prod constraints specifying a function
  f_message_to_v1(x) = @sum f(x,y) / v1_message_to_f(x)
  f_message_to_v2(y) = @sum f(x,y) / v2_message_to_f(y)
  v1(x) = @prod f_message_to_v1(x)
  v2(y) = @prod f_message_to_v2(x)
  v1_message_to_f(x) = @sum v1(x) / f_message_to_v1(x)
  v2_message_to_f(y) = @sum v2(y) / f_message_to_v2(y)
  f(x,y) = @prod v1_message_to_f(x) * v2_message_to_f(y)
end
```

Figure 4.2: BP in WLP, attempt #2. Since this is still completely written in first-order WLP, the BP propagation rules must be re-stated in each place they are applied.

incoming messages along *other* links; we are removing the undesired message by dividing.[1]

With this version, we can already start to see how programming in this way would be a pain. We need to re-state each BP rule twice, once for `v1` and again for `v2`. This just keeps getting worse for larger examples; all the rules need to be re-written for each link in the network.

This brings us to the version of BP shown in Figure 4.3. This is a visual mess in comparison with the previous versions, but it's a mess we only need to write once, and can then re-use again and again. There are a lot of things going on here:

- The `link` predicate holds the actual connections; that is, it specifies which variables in the variable nodes correspond to variables in function nodes. No

[1]This is more convenient to state in WLP, but it is also more efficient. For a variable node with n links, computing each outgoing message individually requires multiplying $n-1$ messages together (so $n-2$ multiplications), n times; so, $n(n-2)$ `BSPTree` multiplications. Multiplying everything together first, and then dividing out, requires only $n-1$ multiplications and one division.

```
bp = @rules begin
  link(Fn, Var, _args_, val) = @max max(0, link(Fn, Var, _args_, val))
  is_link(Fn, Var) = @max min(1, link(Fn, Var, _args_, val))
  is_vnode(Var) = @max is_link(Fn, Var)
  fnode_output(Fn, Var, val) = @sum Fn(_args_...)  *
    link(Fn, Var, _args_, val)
  message(Fn, Var, val) = @prod normalize_fo(safe_div(fnode_output(
    Fn, Var, val), message(Var, Fn, val))) ^ is_link(Fn,
    Var)
  var_incoming(Var,val) = @prod message(Fn, Var, val) ^ is_link(Fn,
    Var)
  Var(val) = @prod normalize_fo(var_incoming(Var,val)) ^ is_vnode(Var)
  message(Var, Fn, val) = @prod normalize_fo(safe_div(Var(val),
    message(Fn, Var, val))) ^ is_link(Fn, Var)
  args_message(Var, Fn, _args_) = @sum message(Var, Fn, val) *
    link(Fn, Var, _args_, val)
  Fn(_args_...)  = @prod args_message(Var, Fn,
    _args_) ^ is_link(Fn, Var)
end
```

Figure 4.3: BP in WLP, attempt #3. Here, second-order rules are used to make BP generic, so that it can be applied to arbitrary examples without repeating the propagation rules.

links are given in the rules we see here; rather, the user is expected to specify links (see Figure 4.4.)

- The first rule ensures that non-existant links have value zero. Otherwise, the default value for `@max` predicates, `-Inf`, would take over.[2]

- The predicate `is_link` summarizes `link` into a value of `1.0` if there is any link, and `0.0` otherwise. This gives a computationally more efficient way to check *whether* there is a link between two predicates, in rules where we don't also need to know *what* the link is exactly.

[2]This could be accomplished by defining `link` as a `@sum` node, instead; but the version shown here affords the user greater flexibility: if the user wanted to write rules which logically infer where links should exist, it would probably be via `@max` rules.

- The third rule defines variable nodes as those predicates which are on the right side of a link. (Factor nodes could be similarly defined as predicates occurring on the left side of a link, but we don't need an `is_fnode` predicate for any purpose.)

- `fnode_output` sums up the fnode contents for delivery to a specific message. Keeping this apart from the actual message computation has efficiency benefits; the message computation only has to deal with the contents of `fnode_output` and the `is_link` predicate, rather than the higher-dimensional contents of the node itself and the `link` predicate.

- The predicates `f_message_to_v1`, `v1_message_to_f`, and so on have all been merged into a `message` predicate which takes source and destination as arguments, rather than specifying them in the name.

- The message predicate is `@prod` rather than `@sum`. This is only possible because we already performed the summation in the `fnode_output` computation. This means that the messages are initialized to a value of 1.0, which is standard for BP.[3]

[3]This is not important for the model shown here to produce correct results, because the effects of message initialization have been minimized via rule-ordering; however, initializing the messages to zero can easily cause problems if the rules are placed in a different order.

- The function `normalize_fo` sums out all first-order variables, and then divides the original function by the result. In other words, it normalizes the function, but treating each combination of second-order variables independently.[4]

- A minor problem with the approach in Figure 4.2 is that it divides by zero if one of the messages contains a zero. Although many models will run just fine, one of Signia's primary goals is to incorporate crisp logical reasoning as a special case of real-valued and probabilistic reasoning, so it would be disappointing if BP in Signia could not allow zero as a probability. This can be fixed by adding a second division function to Signia, called `safe_div`, which works like regular division except that it returns the numerator if the denominator is zero. This might seem a bit odd, but it never causes BP to produce incorrect results.

- `var_incoming` is where we multiply all the incoming messages together for a specific variable node. Making this into a separate predicate allows us to normalize the marginals held in the variable nodes.

- `args_message` is where we convert the message to the correct dimensionality to multiply back into the factor node, replacing the variable node's `var` with the factor node's `_args_`.

[4]Use of a special normalization function is not strictly necessary; summing up and dividing can be spelled out with one or two extra rules. A normalization function was used instead in order to decrease the dependence on rule ordering, and reduce the number of rules in the example.

```
model = @rules begin
  f(x,y) = @prod [0<=x<1]*[0<=y<3]*(1/3) + [1<=x<2]*[1<=y<2] +
    [2<=x<3]*[2<=y<3]
  v1(x) = @prod 1
  v2(y) = @prod 1
  link(Fn, Var, (x,y), x) = @max [[Fn == f]]*[[Var == v1]] *
    hyper(1/√2,1)
  link(Fn, Var, (x,y), y) = @max [[Fn == f]]*[[Var == v2]] *
    hyper(1/√2,1)
  using bp
end
```

Figure 4.4: Specifying a BP model which applies the generic BP rules from Figure 4.3 to a simple factor graph.

In order to actually use these rules, we have to specify the graphical model *as data* by initializing the link table, variable nodes, and factor nodes. An example of what this looks like is given in Figure 4.4. (The keyword `using` incorporates rules from another rule-system into the current one.) As discussed in Section 3.6, when we use Dirac delta functions as constraints, the geometry often compels us to adjust by factors such as `hyper(1/√2,1)`, which is what we're seeing here in the `link` initialization.

We've finished setting up a toy example of belief propagation in Signia. How well does it run?

4.1.1 Speed Evaluation

Figure 4.5 shows the simple conditional probability function which was used in Figure 4.4, and which will be used in several models to come. This implies a factor

$y \in (0,1]$	$1/3$	0.0	0.0
$y \in (1,2]$	$1/3$	1.0	0.0
$y \in (2,3]$	$1/3$	0.0	1.0
	$x \in (0,1]$	$x \in (1,3]$	$x \in (2,3]$

Figure 4.5: A simple example of a probability function of y given x.

graph with just three nodes: a variable node for x and y, and a factor node for the conditional probability linking them (as was spelled out in Figure 4.4).

The Julia language uses just-in-time compilation, so the first run of every test in this section is discarded, since it will reflect compilation time as well as execution time. After that first run, the model is run ten times. Here is data from ten runs of the model in Figure 4.4.[5]

[5]Unless mentioned otherwise, speed tests are performed on an early 2015 MacBook Pro with a 2.7 GHz dual-core Intel Core i5 processor.

Run 1	90.531762 seconds
Run 2	95.146118 seconds
Run 3	95.231269 seconds
Run 4	89.677308 seconds
Run 5	88.508172 seconds
Run 6	95.794620 seconds
Run 7	88.758148 seconds
Run 8	95.211901 seconds
Run 9	87.290601 seconds
Run 10	93.533839 seconds
Average	91.9683738 seconds
Min	87.290601 seconds
Max	95.79462 seconds

For comparison, Figure 4.1.1 shows an entirely first-order set of rules which mimics the same computation closely (much closer than the rules in Figure 4.2). Here are the corresponding runtimes:

```
model = @rules begin
  transition_fn(x,y) = @sum [0<=x<1]*[0<=y<3]*(1/3) +
    [1<=x<2]*[1<=y<2] + [2<=x<3]*[2<=y<3]
  v1_incoming(x) = @prod m_t1_v1(x)
  v1(x) = @prod normalize(v1_incoming(x))
  m_v1_t1(x) = @prod normalize(safe_div(v1(x), m_t1_v1(x)))
  t1(x,y) = @prod m_v1_t1(x) * m_v2_t1(y) * transition_fn(x,y)
  t1_out_v1(x) = @sum t1(x,y)
  m_t1_v1(x) = @prod normalize(safe_div(t1_out_v1(x), m_v1_t1(x)))
  t1_out_v2(y) = @sum t1(x,y)
  m_t1_v2(y) = @prod normalize(safe_div(t1_out_v2(y), m_v2_t1(y)))
  v2_incoming(x) = @prod m_t1_v2(x)
  v2(x) = @prod normalize(v2_incoming(x))
  m_v2_t1(x) = @prod normalize(safe_div(v2(x), m_t1_v2(x)))
end
```

Figure 4.6: First-order rules implementing a model similar to Figure 4.4.

Run 1	1.165280 seconds
Run 2	0.958351 seconds
Run 3	0.989410 seconds
Run 4	0.956020 seconds
Run 5	0.995209 seconds
Run 6	0.954515 seconds
Run 7	0.953537 seconds
Run 8	0.960232 seconds
Run 9	0.949194 seconds
Run 10	0.982939 seconds
Average	0.9864687 seconds
Min	0.949194 seconds
Max	1.16528 seconds

As the reader can see, the first-order version is about 100 times faster. We could interpret this optimistically as an upper bound for what might be achieved by optimizing the performance of second-order WLP. This might be achieved by further optimization of second-order inference, or by a compiler which attempts to replace second-order rules by all their first-order instances (when this can be inferred at compile time). However, clearly the current implementation fails to establish that second-order WLP can be as efficient as first-order WLP.

Given that second-order WLP is one of the main contributions of this work, it is necessary to ask whether this drawback is acceptable. Clearly, in the current implementation, second-order WLP will not be acceptable where speed is a critical concern. However, high-level languages often suffer from large slowdowns while remaining popular. Matlab can be hundreds of times slower than C++ [1]. So, the slowdown we see for second-order inference might be considered within the acceptable range, *in comparison to first-order inference.*

What about a non-WLP implementation of the same basic computation? Since the model in figures 4.4 and 4.1.1 use only axis-aligned integer-value splits, it is possible to write a simple array-based implementation, which applies each propagation equation in order with no priority queues or quiescence testing. This was implemented in Julia, and tested with the following run-times:

Run 1	0.000005 seconds
Run 2	0.000018 seconds
Run 3	0.000010 seconds
Run 4	0.000008 seconds
Run 5	0.000019 seconds
Run 6	0.000025 seconds
Run 7	0.000016 seconds
Run 8	0.000022 seconds
Run 9	0.000019 seconds
Run 10	0.000019 seconds
Average	0.000016 seconds
Min	0.000005 seconds
Max	0.000025 seconds

For this model in particular, we see that using Signia at all slows us down by about five orders of magnitude; using Signia's second-order system slowed us down by an additional two orders of magnitude.

Signia's five-to-seven order-of-magnitude slowdown does not compare favorably to the two-to-three cited earlier for Matlab. It should be emphasized that this is not an entirely fair comparison. In comparing Signia to dense array operations, we are putting it in its worst light. While they may not be optimal for this case, the spatial trees can handle much richer geometry than these simple grid structures. Also, note that the array-based implementation doesn't have any of the overhead entailed by running a rule-system. Still: paying five to seven orders of magnitude for the extra flexibility of a rule-system and the possibility of non-grid geometry is a large cost.

In any case, Matlab may not be the best comparison point to use. Signia was designed for use in cognitive architecture work. How does Signia compare to another cognitive architecture?

Since the present work arose in the context of the Sigma cognitive architecture, it is the most natural to compare Signia to. Sigma is only able to time itself down to milliseconds, and clocked in at zero milliseconds when running the example factor graph this section has used so far. To get meaningful time measurements, a larger 10x10 factor function with random entries was used, rather than the 3x3 used so far. Sigma was run on a different machine,[6] so the results are not as directly comparable as they otherwise would be. Still, this gives a general idea of the comparison. The following table gives times for Sigma, along with Signia, all with the same 10x10 factor in the three-node factor graph example (all times given in seconds):

[6]Sigma was run on an iMac (Retina 5K, 27-inch, Late 2014) with a 4 GHz quad-core Intel Core i7. All other results in the table are run on the same machine mentioned earlier, an early 2015 MacBook Pro with a 2.7 GHz dual-core Intel Core i5 processor.

	Sigma	Signia second-order	Signia first-order	Arrays
Run 1	.001	1579.966202	7.531246	1.1e-5
Run 2	.000	1569.655337	6.648404	9.5e-5
Run 3	.002	1571.205184	6.759159	2.5e-5
Run 4	.000	1579.643062	7.259834	2.5e-5
Run 5	.000	1574.594549	6.877367	2.1e-5
Run 6	.001	1606.218968	6.892305	8.5e-5
Run 7	.001	1626.799511	6.958766	2.8e-5
Run 8	.000	1619.644875	6.68005	6.7e-5
Run 9	.001	1580.632998	6.902068	4.9e-5
Run 10	.000	1586.123941	6.763537	2.6e-5
Average	.0006	1589.448463	6.927274	4.3e-5
Min	.000	1569.655337	6.648404	1.1e-5
Max	.002	1626.799511	7.531246	9.5e-5

Although we should be cautious about the comparison, Sigma is clearly significantly better. As before, the first-order implementation is roughly 5 orders of magnitude slower than the array implementation. The second-order version is a little over 200 times slower than that, in this example.[7]

For another comparison point: in [10], Dyna is reported to be about 5 times slower than well-optimized C++ code.

The examples discussed here are only small toy examples, but given the results, it is not worth evaluating performance on larger examples at the present time.

[7]This is somewhat unexpected, since the second-order part of the model is exactly the same as in the earlier example. If the first-order calculations are only about 6 seconds longer in the 10x10 example, then one might think the whole calculation would only take about 6 seconds longer.

So, Signia still has a long way to go even compared to its direct design inspirations. Perhaps more optimization of the spatial tree data-structures can bring it into a tolerable range; or, perhaps Signia would do better to convert to an array-based representation, as Sigma has done.

However, let us not lose sight of the positive accomplishment here: a generic statement of belief propagation. While revealing performance issues with the implementation, the implementation of belief propagation in second-order WLP is a success for the second-order WLP language. A brief statement of belief propagation has been provided which can apply to a broad variety of examples. As previously argued, this capability of second-order WLP is critical for its viability as an interface layer for cognitive architecture. It allows us to write rules of reasoning which apply to whole categories of things, such as all variable nodes and all factor nodes. Without this feature, changing such rules would have to be a matter of re-writing each individual case.

4.2 Lifting & Grouping

As discussed in Section 3, a major design goal of Signia was to support lifted reasoning. In order to analyze how successful the design is in that objective, however, we have to be careful about what "lifted reasoning" means.

The present section will discuss five different notions of lifting:

- Grouping like things together to reduce computational cost.

- Grouping across predicates.
- Explicitly representing generalizations.
- Grouping across explicit generalizations.
- Automatically deriving lifted versions of algorithms.

The present section will discuss each of these, examine what Signia accomplishes in each, and compare to what Sigma accomplishes.

4.2.1 Grouping Like Things Together

One of the most basic notions of lifted reasoning is *to group similar things together and reason about them in one stroke, rather than one at a time.* We can apply this to Signia (and, similarly, to Sigma) by looking at predicates as a collection of instances – that is, a predicate is a kind of "table" which stores values for each possible combination of argument values. Lifting is understood, then, to involve grouping these instances together in order to reason about many at once. Both Sigma and Signia implement this kind of lifting. However, as discussed in Section 3.3.1, Signia's `BSPTree` representation is able to group together significantly more than Sigma's representation. In this sense, Signia is better at lifting.

	N=100	N=10000	N=50000
Arrays	0.0273858	2.3581797	182.7543697
Signia	2.0661051	2.133878	2.0025498
Sigma	0.001	0.0005	0.0008

The first-order Signia model from Section 4.1.1 is used, along with the array based version from that same section. The factor function of those examples was replaced with an N by N matrix with five 1.0 values placed randomly, the rest being 0.0. The average time of ten runs is shown. Times given in seconds, with machine information as in Section 4.1.1.

Figure 4.7: Performance of Signia vs arrays on very sparse data, in three different conditions.

However, this analysis is not entirely satisfactory. The capability described is really more like a generalization of "sparse representation", as opposed to true lifting.[8]

How do Signia's BSP trees perform as a sparse representation? Figure 4.8 compares Signia to a dense-array representation for a sparse belief propagation task.

As expected, BSP trees eventually outperform dense matrices. However, the break-even point appears to be around N=10000, which implies an array with one hundred million floating-point numbers. Signia is able to group those numbers into just 19 floating-point numbers. Yet, the speed advantage of flat arrays over Signia's more complex tree-structured data is so large that those 19 numbers take about as long to work with as one hundred million numbers stored in an array.

[8]The usual concept of sparse representation assumes that we can easily define a notion of "empty", such as values of zero, or some kind of null value. The empty values are effectively grouped together for easy computation, but other values are not grouped together at all. So, the idea of grouping like values together more generally is similar in spirit, and provides similar performance improvements.

	D=4	D=5	D=6	D=7
Signia	12.7166047	33.9938344	29.0928937	670.2778312
Sigma	0.1192	1.7868	27.0809	894.982

Signia vs Sigma for sparse, higher-dimensional data. The N=100 case from Figure 4.8 was modified to use D variables, instead of two, with a correspondingly larger factor graph connecting them. As before, five nonzero regions were placed randomly. Times given in seconds, with machine information as in Section 4.1.1.

Figure 4.8: Comparison of Sigma and Signia on sparse data of higher dimensionality.

This means our data has to be quite sparse indeed before Signia has the advantage. Dense arrays appear to be preferable to BSP trees in most cases.

Sigma's representation does even better, outperforming the other two in every case. Yet, as mentioned previously, Signia can do more grouping, especially as the number of dimensions increases. Does this ever become a speed advantage?

Since Sigma is running on a faster machine, the data is compatible with a high-dimension advantage for Signia. However, the advantage appears small, and Sigma's advantage on lower-dimensional cases is large. So, although Signia can group more things together, it appears this does not translate to faster computations in most cases.

As already discussed, the grouping we are seeing here is not what is typically called lifted inference. In the context of belief propagation, lifting is associated with abstracting *across variable nodes* in order to reason about many at once. In

Sigma, and Signia, variable nodes are represented as predicates.[9] Therefore, lifted reasoning in the usual sense requires grouping across predicates.

4.2.2 Grouping Across Predicates

As discussed in the previous section, to compare the lifting capabilities of Signia and Sigma to lifted belief propagation algorithms, we must decide how the concepts of graphical models (variable nodes, factor nodes, messages, and so on) are compared with the concepts in Sigma and Signia. One possibility is to think of predicates as generalized variable nodes.

Although Sigma creates many more variable nodes than predicates[10], from a user perspective, you'll often be creating one predicate per variable node. Therefore, it makes some sense to think of a predicate as the sensible analogue of a variable node. (Another possibility will be explored in subsequent sections.)

Judging by this standard, Sigma does not support any lifted reasoning, but Signia does. The second-order trees of Section 3.3.2 support generalization across predicates in essentially the same way that the BSP-trees of Section 3.3.1 support generalization over instances within a single predicate. However, this means that

[9]This oversimplifies how things work in Sigma, but not in a way that changes the analysis here. In reality, Sigma creates many variable nodes, not just one variable node per predicate. However, it is true that each predicate is represented by a variable node.

[10]This is not strictly true, due to Sigma's loose adherence to the summary-product algorithm. Calling Sigma's graph a "factor graph" with "variable nodes" and "factor nodes" is only an analogy, though a close one. See Section 2.2, or for more details, [34].

only consecutive ranges of predicates can be abstracted; two predicates whose behavior is precisely the same but which happen to appear with a third predicate between them cannot be grouped together for the sake of reasoning.[11]

The amount of representation-sharing which will actually happen between two adjacent predicates is limited by the behavior of the trimming function, and by the form taken by rules involving those predicates. If two predicates are acted on by separate rules which are identical but for the substitution of one predicate name for the other, Sigma has no facility for recognizing the common elements of the two rules and abstracting over them. This may block lifted reasoning which would otherwise occur. Even if the two predicates are numerically adjacent and so can be grouped together, even if they end up holding identical results at all times during execution, and even if Signia's `trim` function recognizes the identical contents of the predicates and unifies their representation within working memory, the two separate rules will split out different copies of the working-memory content every time and perform redundant calculations.

So, in order to have a good chance of significant benefits from Signia's ability to abstract across different predicates in the `MemTree` representation, it's necessary to make use of second-order rules which reflect the regularity which you want to make use of. Fortunately, this is often already something the user wants to do, since you don't want to write essentially the same rule many times if you don't have

[11]The numerical index given to a predicate is determined entirely by the order in which predicates are introduced in the code.

to. Signia's second-order language allows algorithms to be stated in a predicate-agnostic way so that the fundamental dynamics governing a system (such as belief propagation) can be stated succinctly; so, in Signia's intended usage, the most important rules would already be written in a manner suited to lifted inference.

However, Sigma *also* has a way to express generalizations. Unlike Signia's second-order system, it does not allow the user to generalize across predicates. However, it allows one predicate to play the role of many. This suggests a different way of thinking about lifting capabilities.

4.2.3 Explicit Generalization

This section discusses the ability to state *lifted models*, rather than *lifted reasoning.* Understanding the ability of Sigma and Signia to state lifted models is necessary to make a finer-grained comparison of their lifted reasoning capabilities.

Signia facilitates lifted models in two different ways: second-order rules, and universal variables. Sigma only possesses one of these two: universal variables. So, let's start by comparing Sigma and Signia's universal variables.

Universal Variables

The basic idea of the so-called "universal variables" in Sigma, and their correlate in Signia, is to split a predicate into many instances. The universal variable indexes all of these instances. This allows us to easily write rules which handle many such instances, which would otherwise be a pain. The explicit instance variable also allows us to write rules which summarize across instances. Furthermore, several universal variables can be used, organizing instances across multiple dimensions.

In Sigma, universal variables must be defined as an explicit argument of the predicate. (This is done in the predicate *declaration*, a concept which doesn't exist in Signia). This extra argument must be mentioned whenever a predicate is used in a rule.

Suppose a user wants to repeat a certain sort of predicate many times. For concreteness, suppose we have a predicate `height_prob(h)` which is a probability density function over possible heights. We now want to introduce many people into our model, representing the height of each person.

In Sigma, a universal argument would be added to the predicate; if we did it that way, it would look like `height_prob(h, person!)`.[12] This represents something similar to repeating the `height_prob(h)` predicate for each person. Declaring the variable to be universal associates it with specific types of reasoning, which reflect the idea of repeated instances.

Signia doesn't presuppose anything about the propagation rules which will be used for reasoning, so this wouldn't make very much sense. Adding *any* additional argument to a predicate is *already* creating new instances of the existing structure for each possible value of the new variable.

[12]The notation in Sigma is opposite from that chosen here, though, in that *unique* variables, not universal ones, are notated with '!'.

One of Signia's innovations is to instead allow universal variables to occur in an implicit way. This means we can write rules without thinking about predicate instances, and later decide to use a universal variable to split a predicate into many instances. The previously-written rules will all work fine, handling all instances in the same way they previously handled the whole predicate.

So, as described in Section 3.2.2, we can write something like `height_prob{person!}(h)` when we need to explicitly deal with the universal variable, but still write `height_prob(h)` at other times.

This gives significance to the concept of universal variable in Signia: far from just another argument, this is a way to create many instances of a structure without changing the existing rules for that structure. Rules which don't mention the universal variable at all will simply act as expected, but now applying to all instances. Rules which mention the universal variable in their body, but not in their head, will also behave just as if we'd created many instances of that same rule, each explicitly referencing one instance of the predicate (but without splitting up the predicate in the head). This is an important guarantee which will come up again.

While this is definitely useful, it doesn't greatly change the complexity of a fully-written program; we can always manually add the extra arguments for universal variables. Therefore, the overall ability of Signia and Simga to compactly represent larger models using universal variables is similar.

Second-Order Rules

On the other hand, Signia possesses another facility for explicit generalization, which Sigma completely lacks: predicate and tuple variables, which allow the Signia user to write rules which explicitly generalize over many predicates. Just like universal variables, this allows one rule to accomplish what would otherwise require a great number of nearly identical rules. Unlike universal variables, it allows generalization across explicitly-defined predicates, including across predicates with very different signatures (ie, argument structures).

This is critical for the treatment of belief propagation in Signia, as well as any other cognitive algorithms which are meant to apply to arbitrarily many predicates.

Sigma totally lacks such a capability, but also, largely does not need it: such cognitive algorithms are supposed to be fixed at the architectural level, rather than user-defined via rules.

Nonetheless, this is a sense in which Signia's ability to state lifted models greatly outpaces Sigma.

As stated in Chapter 1, second-order rules do not work well with universal variables in the current implementation. This is an important limitation on both types of lifted representation.

4.2.4 Grouping Across Explicit Generalizations

Having discussed the types of explicit generalization available in the two architectures, we can now ask about lifted inference across these generalizations.

Lifted reasoning across second-order rules was already addressed in Section 4.2.2. Second-order rules introduce explicit generalizations across predicates. With such abstractions introduced, Signia is capable of grouping identical data together, to compute things once rather than repeatedly. Sigma lacks this type of generalization.

The situation with universal variables is more complex.

Sigma determines what summary function to use based on variable types, rather than associating summary types with rules. Universal variables in Sigma are summarized out via max. This provides a real-valued generalization of classical rule-system reasoning, where a rule will fire if it conditions are satisfied by any instance.

However, this generalization is at best very approximate when it comes to probabilistic reasoning across repeated structure in factor graphs and other graphical models. The max-summarization can be thought of as an approximate probabilistic "or"; but lifted belief propagation combines messages with approximate "and", by multiplying them together. (In Signia, this would be product-integration, implemented by the `@prod` summary type.)

In terms of *efficiency*, Sigma is still able to group like data together across universal variables. Signia may be able to do so *more*, for the same reason Signia's first-order reasoning can group more things together. But the basic optimization is similar, and will achieve roughly the same performance benefit in cases where only one universal variable is used.

As previously stated, the benefit of universal variables in Signia is their ability to make many instances of a structure without the user needing to explicitly duplicate it. When a rule doesn't mention a universal predicate at all, then any universal variables in the relevant memory locations will automatically split the rule into many instances. The predicate in the head pattern will also get split up into instances, to accommodate the many conclusions of the different instances. (And when these conclusions are identical, they can be grouped together in the representation.) On the other hand, when a universal variable appears in the body but not in the head, the rule still acts like many instances, but this time all pointing to the same predicate – summarizing out the universal variable from the body plays the role that *result merging* would have played for the many instances.

This provides a kind of correctness guarantee. Both Sigma and Signia can create lifted models which briefly state what would otherwise require a great deal of structure duplication; however, Signia's reasoning over these lifted models is guaranteed to be exactly the same as if we wrote out the larger model;[13] Sigma's use of max-summarization for universal variables does not guarantee this, and in particular, does not provide this for belief propagation.

One of Signia's original goals was to leverage this guarantee to provide lifted reasoning for any algorithm which could be stated in Signia's WLP. Unfortunately, in the current system, this capability is quite limited due to the way universal

[13]Modulo ordering issues: if the result for a model depends on the order in which rules are computed, then grouping like things together could change the result.

variables interact with second-order rules. The following section will consider this in more detail.

4.2.5 Automatically Lifting Algorithms

Having understood how second-order rules allow cognitive algorithms to be stated generically, and having understood how universal variables allow lifted reasoning over duplicated structure, one might naturally think that these two can be combined to turn any algorithm which can be generically stated in WLP into a corresponding lifted algorithm. Unfortunately, Signia doesn't currently support this.

The first thing to note is that if the rules implementing an algorithm don't mention universal variables, then adding a universal variable to the data they operate on will only split *everything* up. We can use this to create multiple instances of a structure, but they won't interact with each other in any way unless we add more rules to make it happen.

Nontrivial lifted reasoning requires more than just duplicated structure which can be efficiently reasoned about; we need to be able to duplicate only *some* of the structure.

For example, consider a simple factor graph in which two variable nodes are connected by a function node. Suppose we want to duplicate the second variable many times, and also duplicate the function node connecting back to the first variable. We can put a universal variable in the rule providing initial data for this variable node. We can even provide different data to different instances of the node. However, all this will only result in many instances of the whole graph; as the universal variables propagate through the system, they will split everything they touch into many instances.

However, there could be some hope: the link table design pattern introduced in Section 5 allows us to, *in effect*, use data to determine which variables are or aren't present, and how they match. Shouldn't this save us?

To keep the first variable at only one copy, we need to product-summarize over the universal variable in the incoming messages for that node. This means the link between the factor and the first variable needs to establish the presence of a universal variable for one predicate but not the other, so that the variable gets summed out.

Such a feature seems possible. However, it isn't clear how to design it. One possibility would be to allow argument-capture in the curly-bracket list, so that a pattern like `P{_u_...}(_v_...)` would not only capture a predicate's arguments in the tuple-variable `_v_`, but *also* capture the tuple's *universal* arguments in `_u_`. This could enable the critical data to be placed in the link table and used. However, it would *also* require the generically specified algorithm to explicitly match with `_u_` in order to use it. So, this doesn't quite accomplish automatic lifting. Variations of this idea might possibly work, though.

In any case, Signia at present implements no features of this sort, so, is unable to include constraints on universal variables in link tables. This unfortunately stops us from using universal variables and generically-specified algorithms together, except in limited ways.

Chapter 5

Conclusions & Future Prospects

In Chapter 1, we began by identifying the development-time problem for cognitive architecture. Cognitive architectures often take decades to develop, leading to very slow feedback on their fundamental assumptions. The possibility of a broadly-utilized *interface layer* has previously been raised [7, 6, 32]. This could accelerate development of cognitive architectures, by alleviating a significant portion of the programming burden such architectures entail.

However, it is critical that such an interface layer be as approach-agnostic as possible. Otherwise, the problem of slow feedback on architectural assumptions is not alleviated: all architectures using a particular interface layer will share the interface layer's assumptions, which will be difficult to abandon without creating a new interface layer. The previous proposals for interface layers, while quite general, rest on graphical models as an architectural assumption.

This work set out to investigate weighted logic programming (WLP) as an interface layer for cognitive architecture. Previous work on WLP demonstrated its capacity to represent a broad range of existing AI algorithms [10, 11, 8, 9]. This makes it quite promising as an interface layer.

However, while existing WLP (which we refer to as first-order WLP) is quite capable of specifying individual models, it can't easily serve as the language in which a cognitive architecture is implemented. General architectural principles such as belief propagation can only be stated as they apply to individual cases. This makes first-order WLP inadequate as a lower layer, because cognitive architectures could not be implemented on top of it; at best, the architectural assumptions of a cognitive architecture could be seen as "design patterns" for WLP programs.

The present work extended WLP in several ways:

- *Second-order WLP* was introduced, providing several new features in support of the *generic* statement of algorithms; that is, stating an algorithm without making any commitment as to which data-tables / predicates it will be applied. Second-order WLP also introduces a notion of *universal variable* inspired by that in the Sigma cognitive architecture, to enable lifted reasoning.

- First-order WLP was also extended to continuous domains, to facilitate a broader class of hybrid reasoning.

In order to implement these extensions, a form of spatial trees was proposed which indexes spaces of variable dimensionality and dimension types.

An implementation of these ideas, Signia, was developed. Use of spatial trees enabled a broad variety of capabilities to be implemented in a relatively unified way:

- Continuous-domain and discrete-domain cases were handled in a uniform way, supporting piecewise-constant functions with arbitrary-angle linear boundaries.
- Linear constraints were handled within the same representation, by introducing infinite values to help represent generalized functions such as Dirac delta functions.
- First-order and second-order data were both represented as spatial trees, allowing very similar algorithmic ideas to be applied in their processing.

In Section , it was demonstrated that this implementation could indeed enable the generic statement of factor-graph belief propagation, as desired. Since factor graphs are themselves an extremely broad and versatile foundation, this is a good start for an approach-agnostic interface layer. It suggests that the broad variety of AI algorithms which first-order WLP is already known to support may also be stated generically, enabling cognitive principles to be stated in a high-level manner which makes them easy to apply to new problems without re-writing their propagation rules.

However, in and , we saw that the spatial trees are often several orders of magnitude slower than alternative methods. While there are cases where spatial trees have a speed advantage, those cases are quite extreme. It is possible that further optimizations to the spatial trees would bring performance within an acceptable range, but the evidence at present speaks against use of these data-structures for this purpose.

It is therefore imperative that further work in this direction address speed issues. There might be many possibilities to consider, but two clearly present themselves. First, the BSP-tree representation used for first-order inference could be replaced with something resembling Sigma's region-array representation. These cannot represent arbitrary-angle boundaries or delta functions, but, Sigma itself solves this problem by using special-case representations of such functions; some conceptual unity is sacrificed for a large gain in speed. Secondarily, the speed issues with second-order inference might be addressed by making most second-order inference happen at compile time. A second-order WLP program would be converted to a first-order program, by instantiating each second-order rule as many first-order rules. If a compilation method could produce results similar to the hand-written example of translating second-order into first-order from Section 5, this would cut several orders of magnitude from inference time without sacrificing the ability to write algorithms generically.

This doesn't necessarily have to come at the cost of lifted inference; universal variables could be exempt from this process, if they could be efficiently represented at runtime.

Other interesting directions for improving performance include Halide-style control information [31], and alternate fixed-point finding strategies.

Another direction of further work would be to make universal variables and generic algorithms work together more effectively. Universal variables allow the *structure* of a model, such as a Bayes net, to be compressed. Generic algorithms allow the *propagation rules* to be compressed. However, as discussed in Section 4.2.5, the current design doesn't allow these two capabilities to be used at once, except in very limited ways. Combining these two capabilities could enable a form of "automated lifting": generically stated algorithms would, when applied to data containing universal variables, become lifted algorithms as well.

Beyond this, in order to truly provide an interface layer for cognitive architecture, it is necessary to do cognitive architecture work in WLP and see where it leads. The present work demonstrated that belief propagation can be stated generically with second-order WLP. What other algorithms can be written in this way? What algorithms can't be, or have serious drawbacks when written in this manner? When several algorithms are written in second-order WLP, is it easier, or harder, to get those algorithms to work together productively than it would be in a different development style?

Bibliography

[1] Tyler Andrews. Computation time comparison between matlab and c++ using launch windows. 2012.

[2] Nick Cammarata, Shan Carter, Gabriel Goh, Chris Olah, Michael Petrov, Ludwig Schubert, Chelsea Voss, Ben Egan, and Swee Kiat Lim. Thread: Circuits. *Distill*, 2020. https://distill.pub/2020/circuits.

[3] George B Dantzig. Fourier-motzkin elimination and its dual. Technical report, STANFORD UNIV CA DEPT OF OPERATIONS RESEARCH, 1972.

[4] Rodrigo de Salvo Braz, Sriraam Natarajan, Hung Bui, Jude Shavlik, and Stuart Russell. Anytime lifted belief propagation. *Proc. SRL*, 9, 2009.

[5] Abram Demski. Expression graphs. In *Artificial General Intelligence*, pages 241–250. Springer, 2015.

[6] Pedro Domingos. What's missing in ai: The interface layer. *Artificial Intelligence: The First Hundred Years. AAAI Press, to appear*, 2006.

[7] Pedro Domingos and Daniel Lowd. Markov logic: An interface layer for artificial intelligence. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 3(1):1–155, 2009.

[8] Jason Eisner. Dyna: A non-probabilistic programming language for probabilistic ai. In *Extended abstract for talk at the NIPS* 2008 Workshop on Probabilistic Programming*, 2008.

[9] Jason Eisner and Nathaniel W Filardo. Dyna: Extending datalog for modern ai. In *Datalog Reloaded*, pages 181–220. Springer, 2011.

[10] Jason Eisner, Eric Goldlust, and Noah A Smith. Dyna: A declarative language for implementing dynamic programs. In *Proceedings of the ACL 2004 on Interactive poster and demonstration sessions*, pages 32–es, 2004.

[11] Jason Eisner, Eric Goldlust, and Noah A Smith. Compiling comp ling: Practical weighted dynamic programming and the Dyna language. In *Proceedings of the conference on Human Language Technology and Empirical Methods in*

Natural Language Processing, pages 281–290. Association for Computational Linguistics, 2005.

[12] Ben Goertzel. Artificial general intelligence: Concept, state of the art, and future prospects. *Journal of Artificial General Intelligence*, 5(1):1–48, 2014.

[13] Ariel Jaimovich, Ofer Meshi, and Nir Friedman. Template based inference in symmetric relational markov random fields. In *Proceedings of the Twenty-Third Conference on Uncertainty in Artificial Intelligence*, pages 191–199, 2007.

[14] Himanshu Joshi, Paul S Rosenbloom, and Volkan Ustun. Isolated word recognition in the Sigma cognitive architecture. *Biologically Inspired Cognitive Architectures*, 10:1–9, 2014.

[15] Kristian Kersting, Babak Ahmadi, and Sriraam Natarajan. Counting belief propagation. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, pages 277–284. AUAI Press, 2009.

[16] Kristian Kersting, Youssef El Massaoudi, Fabian Hadiji, and Babak Ahmadi. Informed lifting for message-passing. In *AAAI*, 2010.

[17] Boicho N Kokinov. The dual cognitive architecture: A hybrid multi-agent approach. In *ECAI*, pages 203–207, 1994.

[18] Robert Kowalski. Algorithm= logic+ control. *Communications of the ACM*, 22(7):424–436, 1979.

[19] Frank R Kschischang, Brendan J Frey, and H-A Loeliger. Factor graphs and the sum-product algorithm. *Information Theory, IEEE Transactions on*, 47(2):498–519, 2001.

[20] John E Laird. Extending the soar cognitive architecture. *Frontiers in Artificial Intelligence and Applications*, 171:224, 2008.

[21] John E Laird, Christian Lebiere, and Paul S Rosenbloom. A standard model of the mind: Toward a common computational framework across artificial intelligence, cognitive science, neuroscience, and robotics. *Ai Magazine*, 38(4), 2017.

[22] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[23] Chris Malcolm and Tim Smithers. Symbol grounding via a hybrid architecture in an autonomous assembly system. *Robotics and Autonomous systems*, 6(1):123–144, 1990.

[24] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[25] Aniruddh Nath and Pedro Domingos. Efficient lifting for online probabilistic inference. In *Statistical Relational Artificial Intelligence*, 2010.

[26] Bruce Naylor. Constructing good partitioning trees. In *Graphics Interface*, pages 181–181. Canadian Information Processing Society, 1993.

[27] Allen Newell. Physical symbol systems. *Cognitive science*, 4(2):135–183, 1980.

[28] Allen Newell and Herbert A Simon. Computer science as empirical inquiry: Symbols and search. *Communications of the ACM*, 19(3):113–126, 1976.

[29] Nils J Nilsson. *The physical symbol system hypothesis: status and prospects.* Springer, 2007.

[30] David V Pynadath and Stacy C Marsella. Fitting and compilation of multiagent models through piecewise linear functions. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 3*, pages 1197–1204. IEEE Computer Society, 2004.

[31] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[32] Paul S Rosenbloom. Towards a new cognitive hourglass. In *Proceedings of the 9th International Conference on Cognitive Modeling*, 2009.

[33] Paul S Rosenbloom. The Sigma cognitive architecture and system. *AISB Quarterly*, 136:4–13, 2013.

[34] Paul S Rosenbloom, Abram Demski, and Volkan Ustun. Rethinking sigma's graphical architecture: An extension to neural networks. In *International Conference on Artificial General Intelligence*, pages 84–94. Springer, 2016.

[35] Paul S Rosenbloom, Abram Demski, and Volkan Ustun. The sigma cognitive architecture and system: Towards functionally elegant grand unification. *Journal of Artificial General Intelligence*, 7(1):1–103, 2016.

[36] Hanan Samet. *Foundations of multidimensional and metric data structures.* Morgan Kaufmann, 2006.

[37] Herbert A Simon and Allen Newell. Human problem solving: The state of the theory in 1970. *American Psychologist*, 26(2):145, 1971.

[38] Parag Singla and Pedro Domingos. Lifted first-order belief propagation. In *AAAI*, volume 8, pages 1094–1099, 2008.

[39] Parag Singla, Aniruddh Nath, and Pedro Domingos. Approximate lifted belief propagation. In *Statistical Relational Artificial Intelligence*, 2010.

[40] Ron Sun. The clarion cognitive architecture: Extending cognitive modeling to social simulation. *Cognition and multi-agent interaction*, pages 79–99, 2006.

[41] Ben Taskar and Lise Getoor. Introduction to statistical relational learning, 2007.

[42] Volkan Ustun, Paul S Rosenbloom, Kenji Sagae, and Abram Demski. Distributed vector representations of words in the sigma cognitive architecture. In *Proceedings of the 7th conference on Artificial General Intelligence*, 2014.