Derivation of an Efficient Rule System Pattern Matcher

by

Jeremy M. Wertheimer

B.E.E.E., Cooper Union (1982)

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

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1989

Signature of Author .....
Department of Electrical Engineering and Computer Science
February 28, 1989

Certified by ..
.....
Charles Rich
Principal Research Scientist
Thesis Supervisor

Accepted by.....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Derivation of an Efficient Rule System Pattern Matcher

by

Jeremy M. Wertheimer

Submitted to the Department of Electrical Engineering and Computer Science
on February 28, 1989, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

This thesis presents a derivation of an efficient rule system pattern matcher. The matcher efficiently computes all matches between a set of rules and a database. The rules may have multiple patterns. The matcher incrementally updates the set of matches as changes are made to the database. This matcher is modeled on the Rete matcher used in the popular OPS5 production system.

The representations used in the matcher are modeled on the structures used in the model–theoretic semantics of first–order logic. This thesis demonstrates the correspondence between these structures and the data structures used in the Rete matcher. A new structure, the lattice of disjunctive substitutions, is introduced to capture the semantics of the rule–system matching computation. An element of this lattice represents the set of all matches between a rule and the terms in a database.

The derivation is implemented using program transformations. First, a formal specification is developed. Then transformations are applied to this specification to derive an initial implementation. Finally, other transformations are applied to derive more efficient implementations from the initial implementation. The main technique used for improving efficiency is finite differencing. This optimization can be shown to arise from distributive laws involving operations on the disjunctive substitution lattice.

The derivation has been implemented using a wide-spectrum language and an interactive program transformation system.

This work is presented as a contribution towards the construction of a library of programming knowledge to facilitate software reuse and automatic programming. In particular, future directions are described for research towards a library of programming knowledge for implementing rule systems.

Thesis Supervisor: Charles Rich
Title: Principal Research Scientist

Acknowledgements

I would like to thank Charles Rich, my thesis advisor, for his support and patience over the years. Part of my research at MIT was funded by a National Science Foundation graduate fellowship, for which I am very grateful. I would also like to thank Reid Smith for supporting me during my visits to Schlumberger Palo Alto Research, and Doug Smith and Cordell Green for supporting me during my visit to Kestrel Institute. I would like to thank Doug Smith, Tom Pressburger, Peter Ladkin, and Lee Blaine, all of Kestrel Institute, for many fruitful discussions about this work. I would also like to thank Marvin Minsky and David Barstow for their interest and encouragement. I would like to thank Jeff Siskind for the camaraderie as we both worked to complete our respective Master's theses. Finally, I would like to thank Joyce Nachimson for her continual support and encouragement.

For my parents,

and for the rest of my teachers

Contents

List of Figures

Chapter 1

Introduction

The first section of this chapter presents background and motivation for this research. The second section presents a brief overview of the thesis.

1.1 Background

Programs are still written, and rewritten, in a manual, ad–hoc manner. A goal of automatic programming research is to foster an alternative approach to programming. This approach involves gathering, organizing, formalizing and implementing programming knowledge. Researchers have begun to use this approach to automate the development of small programs.[1]

We would like to build a library of programming knowledge that embodies the common collection of algorithms, data structures, and techniques that are the basis of programming. Our main objective is to use this library to build automatic programming systems. A second benefit that accrues from this approach is that we develop sharper understandings of current algorithms and techniques. This thesis is intended as a contribution towards both of these goals. As such, it should be of interest both to readers

[1] I emphasize that this approach currently works with *small* programs because it does not directly address the complexity management issues that arise, and dominate, programming–in–the–large. However, automating programming–in–the–small could be of enormous value to all programmers.

interested in automatic programming, and to readers interested in the particular application domain that I have studied.

The application domain that I have focused on is Artificial Intelligence programming. Specifically, I have focused on rule systems, which are one of the most important types of programs in AI. I use the term rule system to encompass all systems derived from the paradigm of logical inference. These include production systems, theorem proving systems, and deductive databases.

I consider the basic task of efficiently implementing a rule system. The core of this task involves efficiently finding all matches between the rules and the data. The algorithm derived is modeled on the Rete matcher [11] used in the OPS5 Production System [5]. This matcher efficiently finds all matches for a rulebase and a database, and then incrementally updates this set of matches as the database is modified.

The heart of the derivation is a mathematical model of the information computed and manipulated in performing this task. The representations used in the final program are derived directly from this model. The structures in this model are similar to the structures used in the model–theoretic semantics of first–order logic. One of the contributions of this thesis is an explicit description of this connection between the structures computed in the Rete network and the valuation structures of model theory.

There have been several papers published on formal models of rule systems, e.g. [27]. However, my attempt to formalize the structures in the Rete matcher has led me to introduce an extension to the published formal models. Specifically, I introduce *disjunctive substitutions* to represent the information obtained from matching a pattern against several possible data in a database. The formal derivation of the matcher is based on a homomorphism from the matcher specification to a lattice formed from these disjunctive substitutions.

This mathematical model leads to an initial algorithm that satisfies the functional specification for the matcher, but does not satisfy the performance requirements. However, by application of the general purpose techniques of *finite differencing* [19] and *partial*

evaluation [15], this initial implementation can be transformed into an algorithm similar to the Rete matcher. The result of this work can therefore be expressed schematically as

$$\begin{aligned} \text{Rete} &= \text{Formal Specification} \\ &+ \text{Lattice Construction based on Homomorphism to Specification} \\ &+ \text{Finite Differencing based on Distributive Laws} \\ &+ \text{Partial Evaluation.} \end{aligned}$$

Though the heart of this thesis is a formal derivation, I also present an implementation of the derivation. The technology used for this implementation consists of a wide–spectrum specification language, and an interactive transformational development system. Specifically, I have used the Refine[2] language[21], and the Kestrel Interactive Development System (KIDS)[31]. However, the derivation is independent of these systems, and could easily be re–implemented in another system.

This thesis also discusses some future directions for program transformation systems, and some future directions towards implementing a comprehensive library of programming knowledge for implementing rule systems.

1.2 Thesis Roadmap

This section outlines the contents of each of the remaining chapters.

Chapter 2 introduces rule systems, and describes the Rete algorithm in detail. It also describes the simplifications made in this thesis, i.e. the differences between the Rete matcher and the matcher derived in this thesis.

Chapter 3 presents the derivation of the initial (unoptimized) version of the matcher. The core of this derivation involves an algebraic construction of a lattice of sets of substitutions representing the matches between patterns and sets of data.

[2]Refine is a trademark of Reasoning Systems, Inc., Palo Alto, CA

Chapter 4 describes the correspondence between the matcher derived in Chapter 3 and the Rete matcher, and describes the optimizations that improve the efficiency of the initial matcher so that it is comparable to the Rete matcher.

Chapter 5 describes the (partial) implementation of this derivation using the Refine wide–spectrum language and the KIDS transformational development system.

Chapter 6 presents a discussion of the derivation. It explores the relationship between the structures used in the derivation and the structures used in model–theoretic semantics, discusses the status of the implementation, outlines directions for future research, surveys the related literature and explains the contribution of this thesis to the literature, and summarizes the conclusions of the thesis.

The appendix contains brief tutorial material on algebraic structures, lattice theory, and model–theoretic semantics.

Chapter 2

Rule Systems and Rete

This chapter presents an informal description of rule systems in general, and of the Rete algorithm in particular. It also describes the simplifications that have been made in this thesis, i.e. the features of the Rete matcher that have been left out of the matcher derived in Chapters 3 and 4. Chapter 3 begins by formalizing the rule system description, and proceeds with the formal derivation. Chapter 4 returns to this description of the Rete algorithm in order to show the correspondence between the algorithm derived in Chapters 3 and 4, and the Rete algorithm.

2.1 Rule System Introduction

A rule system contains two main data structures: a *database* (*db*), and a *rulebase* (*rb*). For our purposes, the data in the database can be considered to be arbitrary Lisp s-expressions. The rules in the rulebase are structures consisting of two fields: the Left Hand Side (LHS), and the Right Hand Side (RHS). The rules are modeled on the inference rules in a logical system. The LHS of a rule consists of a set of *patterns*, which are s-expressions containing variables. A pattern from an LHS can be *matched against a datum in the database* by finding a *substitution* that replaces the variables in the pattern by terms so that the resulting pattern is equal to the datum. The LHS of a rule can

be *matched against the database* by finding a single substitution under which each of the patterns in the LHS matches some datum in the database. If such a substitution exists, the rule is *applicable* to the database. For a given database and rulebase, several rules might be applicable, and the same rule might be applicable with several different substitutions. The set of pairs of applicable rules and corresponding substitutions is called the *conflict set.*[1]

A rule can be *applied in the forward direction* by matching the LHS of the rule against the database, and then *instantiating* the RHS of the rule with the substitution that resulted from the matching, and adding the result to the database. On each cycle of a rule system, the system computes the conflict set, invokes the *conflict resolution procedure* to select a single rule–substitution pair from the conflict set, and applies that rule to the database. The interpreter repeats this cycle of operations until a specified termination condition is satisfied. In this thesis I am not concerned with the termination condition, so I will simply assume that the rule system continues until the conflict set is empty.

This rule–system operation is summarized by the code in Figure 2-1. This specification describes the operation of a *forward–chaining rule system.* It is written in an informal notation that is intended to combine the features of an Algol-like high level programming language, with additional mathematical notation not usually present in a programming language. (One example of the added notation is a *set–former*, e.g. $\{x \mid P(x)\}$, which represents the set of all elements x that satisfy the predicate P.) In Chapter 5 this specification is implemented in the Refine wide–spectrum language.

In this specification there are three data structures: *db* (the database), *rb* (the rulebase), and *cs* (the conflict set). Lines 2 and 6 specify that the code on lines 3–5 are repeated on each cycle of the interpreter. Lines 3 and 9 state that *cs* is assigned the

[1]The terms *conflict set* and *conflict resolution* are taken from the domain of production systems. (The Rete algorithm was first implemented in the OPS5 production system.) In these systems the order in which rules are run is critical to the proper operation of the system. If several rules are applicable in a given state, the rules are said to be in conflict, and a special procedure is called to resolve this conflict and select which rule to run.

```
1  function rule-system(db, rb) =
2      repeat
3          cs ← {⟨r, σ⟩ | r ∈ rb ∧ σ ∈ match-rule(r, db)}
4          ⟨r, σ⟩ ← conflict-resolution(cs)
5          db ← db ∪ {(rhs(r))^σ}
6      until cs = ∅
7      return(db)
8  end function

9  function match-rule(r, db) = Rep[{σ | ∀_{p∈ lhs(r)} ∃_{d∈db} p^σ = d}]
10 end function
```

Figure 2-1: Rule System Specification.

set of all rule–and–substitution tuples, such that the rule is in the rulebase, and, for all patterns in the LHS of the rule, there exists some datum in the database such that the pattern, when instantiated with the substitution, is equal to the datum. That is, the conflict set contains all rules that match against the database, with the corresponding substitutions. (Note that a rule can appear in the conflict set several times, with several different substitutions.) On line 4 the conflict-resolution procedure is used to select one rule–and–substitution tuple from the conflict set. Line 5 instantiates the RHS of the selected rule with the selected substitution, and adds the result to the database.

Since the set of substitutions in line 9 is an infinite set, a representation function (Rep) is used to denote a concrete representation of this set. Chapter 3 contains a more detailed discussion of the formal model underlying this specification.

2.2 Rule System Finite Differencing

A direct implementation of the interpreter shown in Figure 2-1 would be correct, but inefficient. The major inefficiency arises from the recomputation of the conflict set on each cycle of the rule interpreter. The computation of the conflict set involves examining

all of the data in the database and all of the rules in the rulebase. It is usually the case that the database is relatively large and, since only one new element is added to it on each cycle, the recomputation of the conflict set on each cycle is mostly unnecessary. A more efficient approach would be to compute the conflict set for the initial database and rulebase, and thereafter to incrementally update the conflict set as changes are made to the database.[2] This is the approach followed in the Rete algorithm [11] [5]. The central feature of this algorithm is that it maps incremental changes to the database into incremental changes to the conflict set.

2.3 Rete Description

The main idea behind the Rete network is to compile the rulebase into a token-passing dataflow network that incrementally accepts changes to the database, and produces corresponding changes to the conflict set. This section describes the simplified Rete network that we will be considering. The simplifications are described in Section 2.4.

Since the parts of the Rete network generated by different rules are basically independent we will concentrate on the Rete network for a single rule.

Figure 2-2 shows a Rete network for the rule

$$\langle\ \{(f\ ?x), (g\ ?y), (h\ ?x\ ?y)\} \Rightarrow (p\ ?x\ ?y)\ \rangle.$$

A Rete network is composed of three types of nodes:

- Match nodes (called 1-input nodes in Rete),
- Combine nodes (called 2-input nodes in Rete), and
- Rule nodes (called Terminal nodes in Rete).

[2] We assume the rulebase remains fixed during the operation of the rule system.

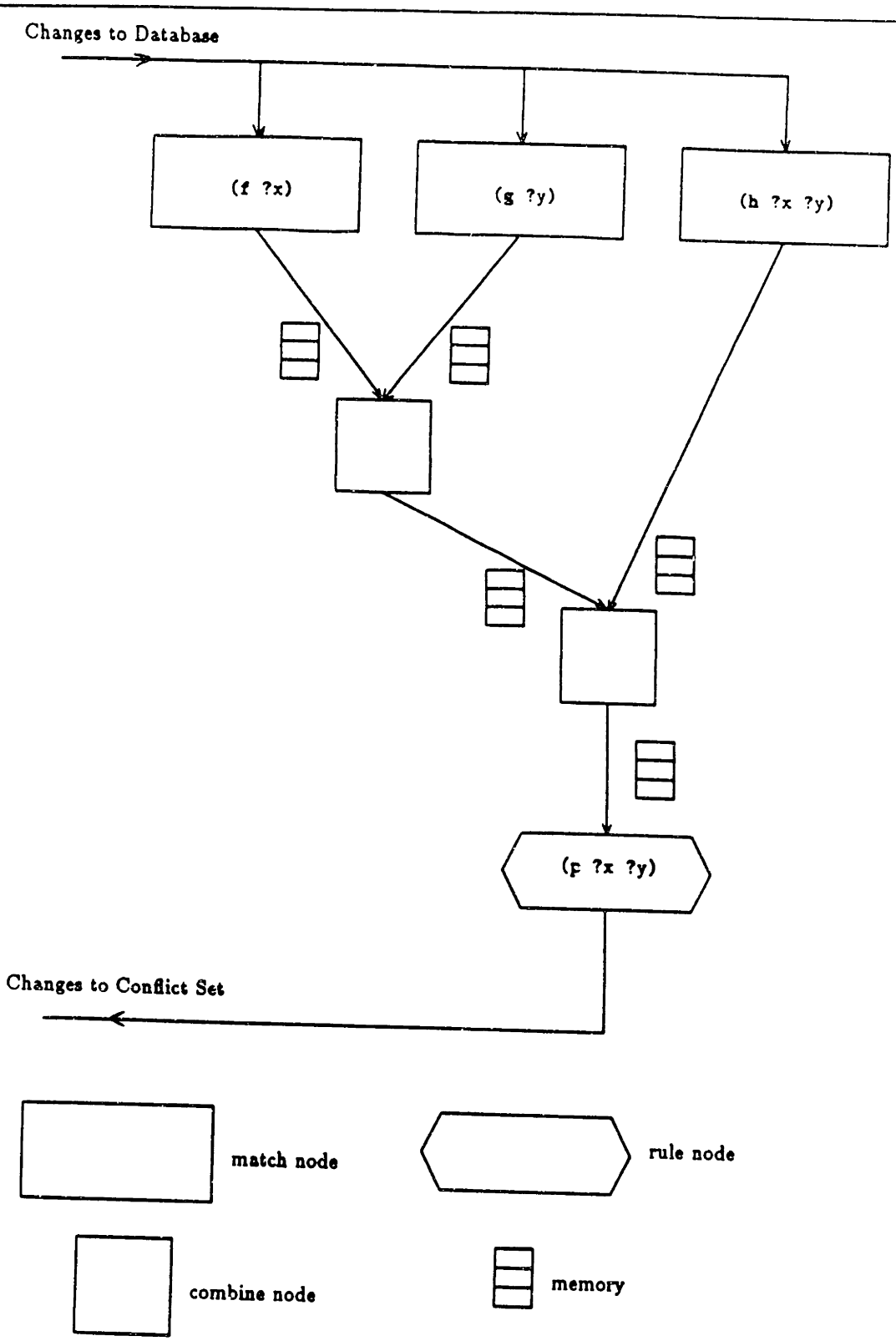


Figure 2-2: A Rete Network.

There are three types of tokens passed between the nodes:

- Database–Change Tokens,
- Substitution Tokens, and
- Rule-Substitution Match Tokens.

The rest of this section is a description of the operation of the Rete network. This description is organized by tracing the progression of a token in the network. (Please refer to Figure 2-2.)

The task of the matcher is to accept database–change tokens, and to output the corresponding changes to the conflict set. The changes to the database are input to the network on the bus at the top of the figure.

Match Nodes

The nodes at the top of the figure, labelled **(f ?x)**, **(g ?y)**, and **(h ?x ?y)**, represent matching nodes for the various patterns in the rule. There is one matching node for each pattern in the rule. (In the full Rete network there would be one matching node for each pattern in the rulebase.) A match node has a single input port and a single output port.

Copies of all new data tokens entering on the top bus are distributed to all match nodes in the network. A match node processes a new data token by matching it against the node's pattern. If the datum and the pattern match, a token containing the resulting substitution is passed out of the output node of the match node. If the datum and pattern do not match, no token is passed out of the match node.

Combine Nodes

Tokens passed from the output ports of match nodes are sent to the input ports of combine nodes. A combine node has two input ports referred to as the left input and the

right input.[3] Each of the input ports of a combine node has a memory associated with it. This memory holds all of the tokens that have been received at that port since the network was initialized. A combine node has a single output port.

The role of a combine node is to combine the substitution tokens from its left and right parent nodes and to generate a stream of substitution tokens the contain substitution that are consistent with the substitutions received at its left and right input nodes. Each output substitution must be consistent with some substitution received at the left input, and with some substitution received at the right input.

The combine node functions as follows. When a new token is received at the left input port of a combine node, it is inserted into the left input memory, and combined with all of the elements in the right input memory of the node. If the results of any of these combinations are valid substitutions, these results are passed out of the output port of the combine node.

The corresponding process is performed when a token is received at the right input port of a combine node. The token is inserted into the right input memory, and combined with all of the tokens in the left input memory. Any valid combinations are passed out the output port.

Rule Nodes

A fan-in tree of combine nodes is built up for all of the patterns in the LHS of a rule, as shown in Figure 2-2. The output port of the final combine node is connected to a rule node. This node accumulates all matches for this rule in the current database. The collection of all of the substitution tokens stored in all of the rule nodes in a network, with each substitution token paired with the corresponding rule, constitutes the conflict set for the current database and rulebase.

[3]The ports are functionally symmetrical; these terms are introduced for explanatory purposes.

2.4 Simplifications

In this thesis I am interested in concentrating on the central feature of the Rete algorithm: the computation of the conflict set, and the incremental recomputation of this set as the database is modified. The matcher derived in this thesis has the same basic structure as the Rete algorithm, and performs the same incremental update of the conflict set as does the Rete algorithm. However, it is a much simplified version of the Rete algorithm. The following features of Rete are not addressed in the derivation:

- In Rete rules can have negated patterns which match if the corresponding positive pattern *does not* match against a datum in the database. The matcher developed in this thesis does not handle negated patterns.
- In Rete rules can delete data from the database, as well as adding data. In this thesis we only consider rules that add data to the database.
- In Rete, the matching for a pattern that appears in several rules is shared between rules. This capability is not implemented in this thesis.

This section briefly discusses these features.

The Rete algorithm allows rules to have patterns marked with negative signs. A rule containing a negative pattern can only run if *none* of the data in the database match against that pattern. These patterns are usually taken to represent negation. This technique of interpreting an absence of data matching a pattern as a "match" for the negation of the pattern is known as the *closed world assumption* [22].

In logical deduction, the only result of applying an inference rule is to deduce a new statement. In formalization of commonsense reasoning and the engineering of rule systems, it has been found useful to also allow rules to retract deductions, and to remove terms from the database. The OPS5 system provides this facility. When a term is removed from the database, the Rete network removes all entries from the conflict set that involved matching against that term. This facility is not present in the matcher

derived in this thesis. It could be added to the matcher, within the framework developed here.

In most rule systems, there are likely to be a number of patterns that appear in several rules. It is inefficient to repeat, for each rule, the computations of the matches to these patterns. A better scheme is to share results of matching a pattern among all of the rules that include that pattern. The Rete network performs this sharing.

In this thesis I focus on deriving the central architecture and features of the Rete matcher: the division of the matching work among a network of nodes, and the incremental update of the conflict set as changes are made to the database. For consideration and illustration of these features, it suffices to analyze the matching network for a single rule (that itself contains several patterns). This network can easily be extended to a Rete network for a set of rules. Chapter 6 discusses future directions for extending the derivation to include these features.

Chapter 3

Specification and Derivation

This chapter presents a derivation of an implementation for the rule matcher. Section 1 formalizes the abstract specification

$$\text{match-rule}(r, db) = \text{Rep}[\{\sigma \mid \forall_{p \in \text{ lhs}(r)} \exists_{d \in db}\ p^{\sigma} = d\}]$$

presented in Chapter 2. In Section 2 the conjunction and disjunction in this specification are expressed as operations on infinite sets of substitutions. In Section 3 a natural ordering on substitutions is used to implement these infinite sets. In Section 4 this implementation is generalized to properly handle disjunctions of substitututions. This chapter concludes with an initial implementation for the rule matcher. Chapter 4 deals with optimizing this implementation, and showing its correspondence to the Rete matcher.

The derivation in this chapter involves the construction of lattices and semilattices. The reader may wish to review Appendix A.1 which briefly summarizes some standard algebraic definitions.

3.1 Formal Specification

In this section we develop a formal specification for the rule system pattern matcher.

For our mathematical model of a rule system, we consider the objects in the database to be terms in a first–order language, and the rules in the rulebase to be inference rules

in a first–order language.[1]

First we need a formal model for the objects in the database. Let V, the set of variables, and C, the set of constants, be two disjoint sets. Let T, the set of terms, be the free semigroup generated by $V \cup C$. (If the operation for this semigroup is thought of as the *cons* function, T can be thought of as the set of s–expressions that can be constructed from the atoms in $V \cup C$.) Let G, the set of ground terms, be the set of all elements in T that do not contain any variables. Using these definitions, the database in a rule system, *db*, is represented as a subset of G.

Next we need a formal model for the rules in the rulebase, and for the process of applying rules to data. A rule contains two components: a Left Hand Side (LHS), and a Right Hand Side (RHS). The LHS contains a set of terms that can be matched against data in the database.[2] The RHS contains a term that can be instantiated and added to the database. The set of rules, R, is therefore the set $2^T \times T$, and the rulebase in a rule system, rb, is a subset of R.

A substitution is a partial function from V to G.[3] Let E denote the set of all substitutions. We will use the Greek letters σ, τ and υ to denote substitutions.

A term p can be *instantiated* with a substitution σ by replacing all of the variables in p with their images in σ. If any of the variables in p are not in the domain of σ, the result is undefined. We use the notation p^σ to denote this instantiation, and refer to the term p as a pattern. The inverse of instantiation is *matching*. The result of matching a pattern p and a datum d is a substitution σ such that $p^\sigma = d$. A full description of instantiation and matching is given in Section 5.2.

The matching computation performed by match-rule is the central part of the rule

[1] In Chapter 5 we present a concrete implementation in terms of Lisp data types.

[2] In many rule systems implemented and used in real–world applications, the order of the patterns in the LHS's of the rules have been carefully hand–tuned by the programmers, in an attempt to improve the performance of the system [29]. To accurately model these systems we would need to represent the LHS of a rule as a sequence. We choose to retain the semantics of logic, where the elements in a conjunct are unordered.

[3] If f is a function, we will use the notation $f(x)$ to denote the value of the function f at x, and dom(f) to denote the domain of f.

system. The rest of this chapter concentrates on implementing this specification. Chapter 4 focuses on incrementally updating the value of match-rule as changes are made to the database.

3.2 The Representation Problem

This section motivates the representation design carried out in Sections 3.3 and 3.4. In order to synthesize code for the specification

$$\text{match-rule}(r, db) = \text{Rep}[\{\sigma \mid \forall_{p \in \text{ lhs}(r)} \exists_{d \in db}\ p^\sigma = d\}], \tag{3.1}$$

we first put this expression into a form where it can be decomposed into several subparts. This can be accomplished by rephrasing the specification in terms of operations on sets, i.e.[4]

$$\text{match-rule}(r, db) = \text{Rep}[\bigcap_{p \in \text{ lhs}(r)} \bigcup_{d \in db} \{\sigma \mid p^\sigma = d\}]. \tag{3.3}$$

This expression can be progressively constructed from the following subexpressions:

$$\text{match}(p, d) = \text{Rep}[\{\sigma \mid p^\sigma = d\}] \tag{3.4}$$

$$\text{match-elt-set}(p, db) = \text{Rep}[\bigcup_{d \in db} \{\sigma \mid p^\sigma = d\}] \tag{3.5}$$

$$\text{match-rule}(r, db) = \text{Rep}[\bigcap_{p \in \text{ lhs}(r)} \bigcup_{d \in db} \{\sigma \mid p^\sigma = d\}]. \tag{3.6}$$

To aid in understanding this specification and its decomposition, let us analyze a small example. Consider matching a rule r, with a LHS given by

$$\text{lhs}(r) = \{p_1, p_2\} = \{(f\ ?x), (g\ ?y)\},$$

[4]The correspondence might be easier to see if we write the quantifiers as logical conjunction and disjunction, i.e.

$$\text{match-rule}(r, db) = \text{Rep}[\{\sigma \mid \bigwedge_{p \in \text{ lhs}(r)} \bigvee_{d \in db} p^\sigma = d\}]. \tag{3.2}$$

against the database given by

$$db = \{d_1, d_2, d_3, d_4\} = \{(f\ 1), (f\ 2), (g\ 3), (g\ 4)\}.$$

A diagram of the values of the subexpressions in Equations 3.4 to 3.6 is shown in Figure 3-1.

This figure demonstrates the key idea in the thesis: *the explicit representations of sets of substitutions for the* conjunctive *match problem that arises from matching all of the patterns in a rule, and for the* disjunctive *match problem that arises from matching a pattern against all of the terms in a database.*

It is important to note that the sets of substitutions in Equations 3.1–3.6 and Figure 3-1 are infinite sets. For example, if a substitution σ is in one of these sets, than any extension of σ (i.e. any substitution τ such that $\forall_x\ (\sigma(x) = \tau(x) \vee \sigma(x) = \omega)$)[5] is also in that set. If $\sigma = \{\ x \mapsto 1\ \}$ is in one of these sets of substitutions, then $\tau = \{\ x \mapsto 1, y \mapsto 2\ \}$ is also in the set. The unboundedness of these sets is necessary in order for the set representing a conjunctive match of two patterns p and q to be equal to the intersection of the sets representing the matches to p and q.

These infinite sets of substitutions cannot be directly stored in an implementation. What is needed to implement this scheme is a finite representation that captures the information in these sets. The next two sections present the derivation of such a representation.

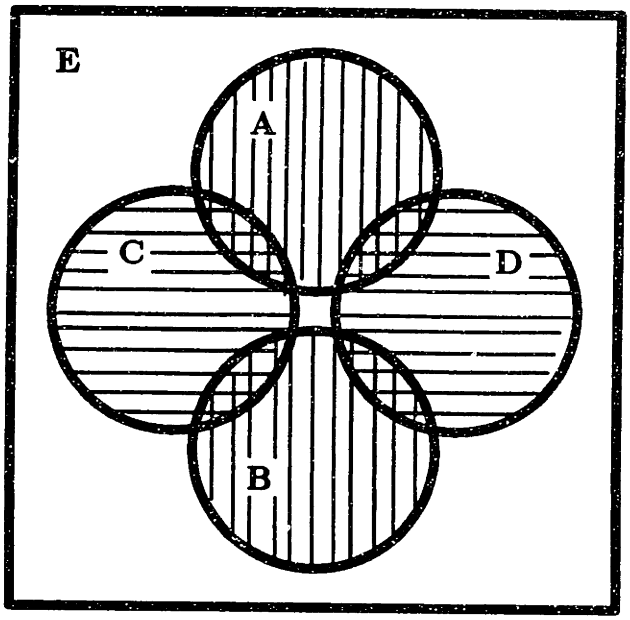
3.3 Representation using Maximal Elements

We might implement the infinite set of substitutions given by

$$S = \{\sigma \mid x_1^\sigma = y_1 \wedge x_2^\sigma = y_2 \wedge \ldots \wedge x_k^\sigma = y_k\}$$

by the single substitution

[5]The symbol ω is used for undefined, since the symbol $\perp$ is used for the bottom element in the lattice introduced in Section 3.3.



$\text{lhs}(r) = \{p_1, p_2\} = \{(f\ ?x), (g\ ?y)\}$

$db = \{d_1, d_2, d_3, d_4\} = \{(f\ 1), (f\ 2), (g\ 3), (g\ 4)\}$

$\text{Abs}[\text{match}(p_1, d_1)] = \{\sigma \mid p_1^\sigma = d_1\} = \{\sigma \mid (f\ ?x)^\sigma = (f\ 1)\} = \{\sigma \mid ?x^\sigma = 1\} = A$

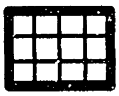
$\text{Abs}[\text{match}(p_1, d_2)] = \{\sigma \mid p_1^\sigma = d_2\} = \{\sigma \mid (f\ ?x)^\sigma = (f\ 2)\} = \{\sigma \mid ?x^\sigma = 2\} = B$

$\text{Abs}[\text{match}(p_2, d_3)] = \{\sigma \mid p_2^\sigma = d_3\} = \{\sigma \mid (g\ ?y)^\sigma = (g\ 3)\} = \{\sigma \mid ?y^\sigma = 3\} = C$

$\text{Abs}[\text{match}(p_2, d_4)] = \{\sigma \mid p_2^\sigma = d_4\} = \{\sigma \mid (g\ ?y)^\sigma = (g\ 4)\} = \{\sigma \mid ?y^\sigma = 4\} = D$

$\text{Abs}[\text{match-elt-set}(p_1, db)] = \bigcup_{d \in db} \{\sigma \mid p_1^\sigma = d\} = A \cup B$

$\text{Abs}[\text{match-elt-set}(p_2, db)] = \bigcup_{d \in db} \{\sigma \mid p_2^\sigma = d\} = C \cup D$



$\text{Abs}[\text{match-rule}(r, db)] = \bigcap_{p \in \text{lhs}(r)} \bigcup_{d \in db} \{\sigma \mid p^\sigma = d\} = (A \cup B) \cap (C \cup D)$

Note. $\text{Abs}[\text{match}_{(p_1,d_3)}] = \text{Abs}[\text{match}_{(p_1,d_4)}] = \text{Abs}[\text{match}_{(p_2,d_1)}] = \text{Abs}[\text{match}_{(p_2,d_2)}] = \emptyset$.

Figure 3-1: Venn diagram illustrating Conjunctive Pattern Matching.

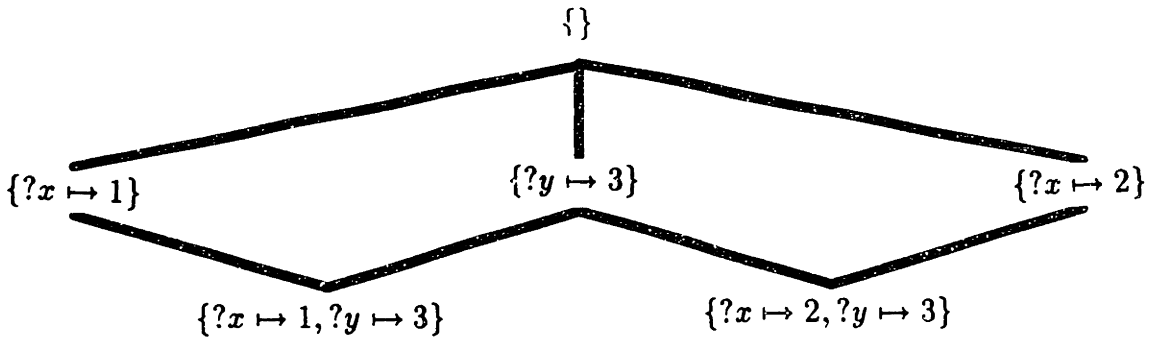


Figure 3-2: Semi-Lattice of Substitutions.

$$Rep[S] = \{x_1 \mapsto y_1, x_2 \mapsto y_2, \ldots, x_k \mapsto y_k\}.$$

That is, represent the set of all mappings that take x_1 to y_1, and x_2 to y_2, etc., by the mapping that takes x_1 to y_1, and x_2 to y_2, etc.

To formalize this representation, we can order the set E using the ordering[6]

$$\sigma \preceq \tau \leftrightarrow \forall_{x \in V} \; (x^\sigma = x^\tau \vee x^\tau = \omega) \tag{3.7}$$

and then represent a set by its maximal element under this ordering.

In this ordering, a substititution σ is $\prec$ a substitution τ iff it agrees with τ on all of the variables on which τ is defined, and is defined on some additional variables.

The structure consisting of the set of substitutions E with the ordering given in Equation 3.7 forms a *semi-lattice.* See Figure 3-2. To complete this semi–lattice, we can introduce a bottom element $\perp$, such that $\forall_{\sigma \in E} \perp \preceq \sigma$. We will refer to this semi–lattice as the Substitution Semi–Lattice (SSL).[7]

[6]Note that this derivation will involve two different lattices. The ordering in this first lattice, SSL, will be denoted by $\sigma \preceq \tau$. The ordering in the second lattice, DSL (introduced in the next section), will be denoted by $\Upsilon \sqsubseteq \Phi$.

[7]The ordering defined for SSL in Equation 3.7 is based on the considering the substitutions as mapping. Alternatively, this ordering could be expressed in terms of the operation of the substitutions in instantiating patterns. The alternative expression is

$$\sigma \preceq \tau \leftrightarrow \forall_p \, \forall_d \; (p^\tau = d \rightarrow p^\sigma = d). \tag{3.8}$$

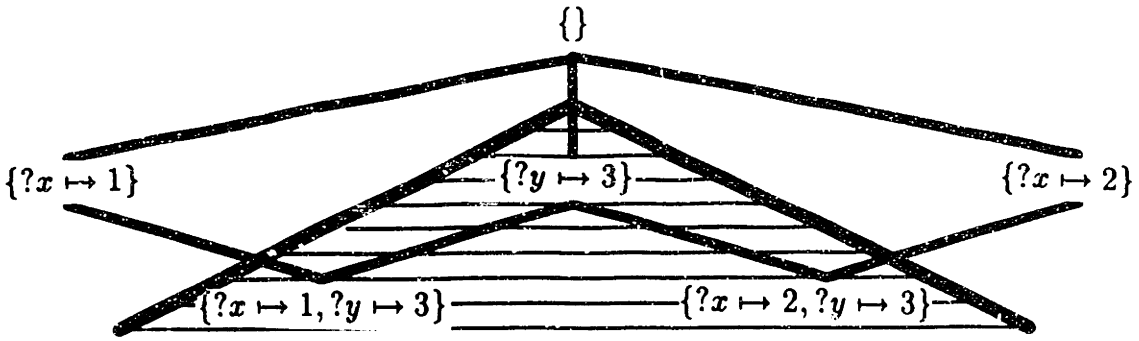


Figure 3-3: A Principal Ideal in the Substitution Semi-Lattice.

The greatest lower bound of two elements in this semi-lattice is given by [8]

$$\sigma \sqcap^* \tau = \begin{cases} \lambda x.(\text{if } x^\sigma = \omega \text{ then } x^\tau \text{ else } x^\sigma) & \text{if } \sigma \doteq \tau \\ \bot & \text{otherwise.} \end{cases} \tag{3.9}$$

To be precise about the representation and the object being represented, we can define the following representation and abstraction functions[9]

$$\begin{aligned} \text{Rep}[S \subseteq E] &= \sigma \mid \forall_{\tau \in S}\, \sigma \succeq \tau = \overset{*}{\bigsqcap}\ S \\ \text{Abs}[\tau \in E] &= \{\sigma \mid \sigma \preceq \tau\}. \end{aligned}$$

A portion of SSL, with a set of substitutions marked, is shown in Figure 3-3. In this diagram we can see that $\text{Abs}(\sigma)$ is the "cone" of elements below σ in the semi-lattice. That is, $\text{Abs}(\sigma)$ is *the principal ideal in SSL generated by* σ.

In this representation, the result of matching a single pattern and datum can be represented by

$$\text{match}(p, d) = \overset{*}{\bigsqcap}\ \{\sigma \mid p^\sigma = d\}. \tag{3.10}$$

This allows us to implement the specification of Equation 3.4 which denotes the result of matching a pattern and a datum. In order to use this representation to implement the

[8]The symbol $\doteq$ denotes weak equality, i.e. $\sigma \doteq \tau \leftrightarrow \forall_x\, (\sigma(x) = \tau(x) \vee \sigma(x) = \omega \vee \tau(x) = \omega)$.
[9]For a description of the use of abstraction functions, see [16].

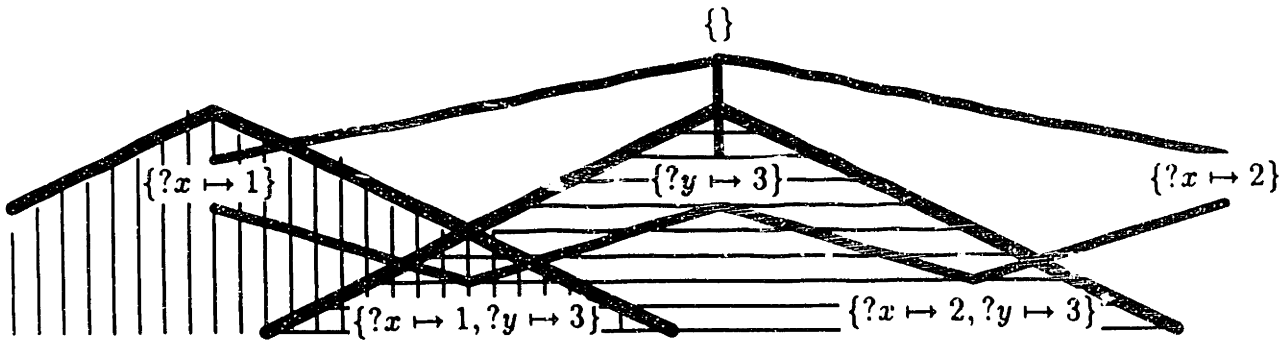


Figure 3-4: Intersection of two Ideals in the Substitution Semi-Lattice.

match-rule specification, we must also be able to represent the unions and intersections of sets of substitutions, as specified in Equations 3.5 and 3.6.

We can represent intersection of substitution sets, as required by Equation 3.6, by the greatest lower bound operation in the lattice:[10]

$$\begin{aligned} Rep[Abs[\sigma] \cap Abs[\tau]] &= Rep[\{\upsilon \mid \upsilon \preceq \sigma\} \cap \{\upsilon \mid \upsilon \preceq \tau\}] \\ &= Rep[\{\upsilon \mid \upsilon \preceq \sigma \wedge \upsilon \preceq \tau\}] \\ &= Rep[\{\upsilon \mid \upsilon \preceq \sigma \sqcap^* \tau\}] \\ &= Rep[Abs[\sigma \sqcap^* \tau]] \\ &= \sigma \sqcap^* \tau. \end{aligned}$$

This is illustrated in Figure 3-4 by the intersection of the two ideals generated by the elements $\{x \mapsto 1\}$ and $\{y \mapsto 3\}$.

The fact that the greatest lower bound in the lattice corresponds to intersection of the sets of substitutions can be stated algebraically as: *the representation function is a (lattice) homomorphism from* $\langle 2^E, \subseteq, \cap \rangle$ *to* $\langle E, \preceq, \sqcap^* \rangle$.[11] (Since, as stated in Appendix A.2, any powerset forms a lattice under the subset relation.)

[10] assuming that these sets are principal ideals generated by some element. This limitation is removed in the representation introduced in Section 3.4.

[11] again, restricting the elements in 2^E to principal ideals.

Unfortunately, this representation cannot represent unions of sets of substitutions as required by Equation 3.5. For example, for $\sigma = \{?x \mapsto 1\}$ and $\tau = \{?x \mapsto 2\}$, there does not exists a v such that $\sigma \preceq v$ and $\tau \preceq v$. Stated algebraically, *this structure is a semi-lattice, and not a lattice, because there is no operation* $\sqcup^*$ *such that the Representaton function is a homomorphism from* $\langle 2^E, \subseteq, \cup \rangle$ *to* $\langle E, \preceq, \sqcup^* \rangle$

3.4 Representation using Sets of Maximal Elements

This problem can be remedied by changing the representation. Instead of representing a set of substitutions S by a single maximal element, we can represent it by a set of maximal elements $Rep[S]$, so that for every element σ in S, there is some element τ in $Rep[S]$, such that $\sigma \preceq \tau$. We will use the capital Greek letters Υ, Φ, Ψ and Γ to denote sets of maximal elements in E.

These sets of maximal elements can be ordered by the following extension to the ordering used in Section 3.3. Let a set of substitutions Υ be less than ($\sqsubseteq$) a set of substitutions Φ, iff every substitution in Υ is less than ($\preceq$) *some* substitution in Φ, i.e.

$$\Upsilon \sqsubseteq \Phi \leftrightarrow \forall_{\sigma \in \Upsilon} \exists_{\tau \in \Phi} \sigma \preceq \tau. \tag{3.11}$$

The result of this set and ordering is a new lattice (see Figure 3-5), which will be referred to as the Disjunctive Substitution Lattice (DSL).[12]

Unlike the semi–lattice described in Section 3.3, DSL is a lattice, with both a greatest

[12]The ordering given in Equation 3.11 can also be expressed, as we did for $\preceq$ in the previous section, in terms of the operation of the substitutions in instantiating patterns. This alternative definition is given by

$$\Upsilon \sqsubseteq \Phi \leftrightarrow \forall_p \forall_d [(\exists_{\sigma \in \Upsilon} p^\sigma = d) \rightarrow (\exists_{\tau \in \Phi} p^\tau = d)]. \tag{3.12}$$

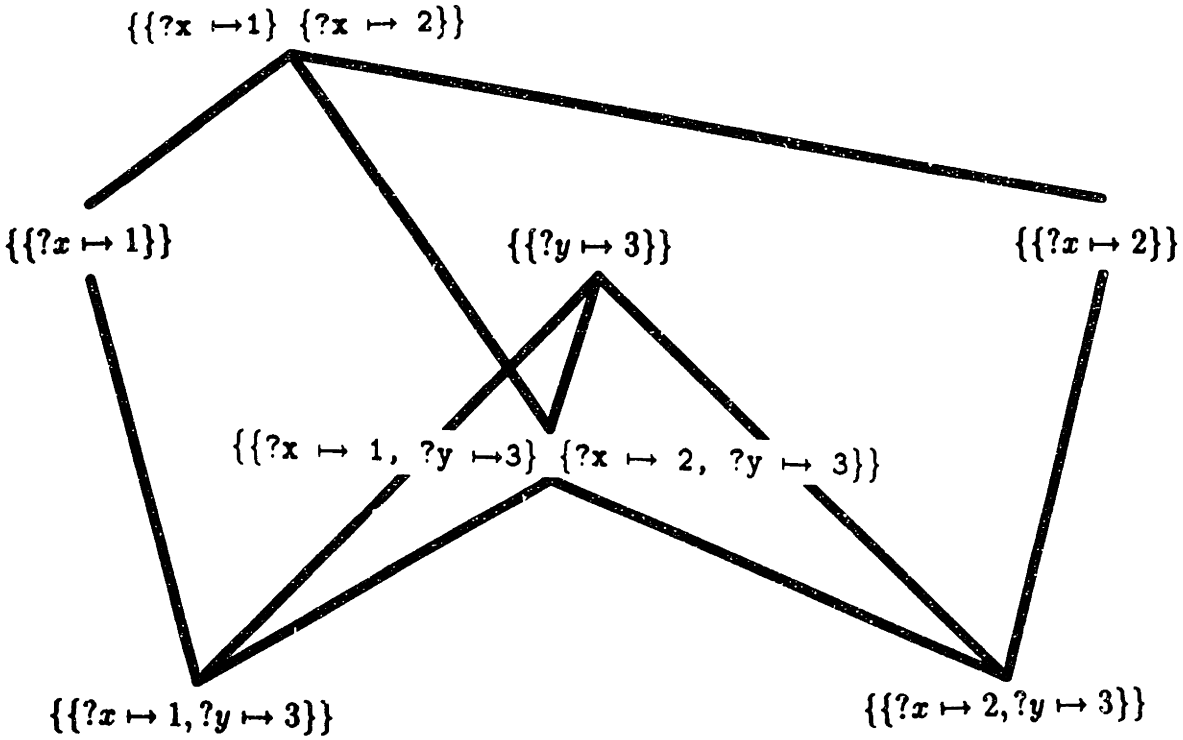


Figure 3-5: Lattice of Disjunctive Substitutions.

lower bound and a least upper bound. These operations are given by[13]

$$\Upsilon \sqcup \Phi = \Upsilon \cup \Phi \tag{3.13}$$

$$\Upsilon \sqcap \Phi = \{\sigma \mid \exists_{\tau \in \Upsilon} \exists_{\upsilon \in \Phi} \sigma = \tau \sqcap^* \upsilon\}. \tag{3.14}$$

For this representation, the representation and abstraction functions are given by

$$\text{Rep}[S \subseteq E] = \bigsqcap S$$

[13]Equation 3.13 is a slight oversimplification. In cases where

$$\exists_{\sigma \in \Upsilon} \exists_{\tau \in \Phi} [(\sigma \prec \tau) \vee (\tau \prec \sigma)],$$

the value of $\Upsilon \sqcup \Phi$ can be simplified using identities. For example,

$$\{x \mapsto 1\} \sqcup \{x \mapsto 1, y \mapsto 2\} = \{x \mapsto 1\}.$$

These cases do not arise in any of the programs in this thesis, since we only apply $\sqcup$ to sets of disjunctive substitutions that represent alternative matches for the same pattern. Consequently, these disjunctive substitutions have the same domains, and their least upper bounds cannot be reduced using identitities. In these cases, Equation 3.13 holds.

$$\text{Abs}[\Upsilon] \;=\; \{\sigma \mid \exists_{\tau \in \Upsilon}\, \sigma \preceq \tau\}.$$

Note that whereas elements in SSL could only represent principal ideals of elements in SSL (i.e. "cones" of elements in the diagrams), any ideal of elements in SSL can be represented by an element in DSL (even those that are represented by *sets* of cones in the diagram). It can therefore be shown that all unions and intersections of sets of substitutions can be represented in DSL, as required by the specifications in Equations 3.5 and 3.6. These values are given by

$$\text{Rep}[S1 \cup S2] = \text{Rep}[S1] \sqcup \text{Rep}[S2] \tag{3.15}$$

$$\text{and} \qquad \text{Rep}[S1 \cap S2] = \text{Rep}[S1] \sqcap \text{Rep}[S2]. \tag{3.16}$$

The specification for match-rule in Equation 3.3 can now be implemented by using representations in DSL. This implementation is given by

$$\text{match-rule}(r, db) = \bigsqcap_{p \in \text{ lhs}(r)} \bigsqcup_{d \in db} \{\sigma \mid p^\sigma = d\}. \tag{3.17}$$

Note that this implementation is isomorphic to the specifications shown in Equations 3.1 and 3.3. That is, *the specification in Equation 3.3 and the implementation in Equation 3.17 represent equivalent expressions in two homomorphic lattices: the set algebra of the specification, and the Disjunctive Substitution Lattice (DSL) of the implementation.*

Finally, we can finish the implementation by replacing the expression for the result of matching a single pattern and datum with the match function. To be consistent with the format of the elements in DSL, the specification for match given in Equation 3.10 can be slightly modified so that it returns a (singleton) set of substitutions, i.e.

$$\text{match}(p, d) = \{\overset{*}{\bigsqcap} \{\sigma \mid p^\sigma = d\}\}. \tag{3.18}$$

Substituting this equation into equation 3.17 and simplifying yields

$$\text{match-rule}(r, db) \;=\; \bigsqcap_{p \in \text{ lhs}(r)} \bigsqcup_{d \in db} \{\sigma \mid p^\sigma = d\}$$

$$= \bigsqcap_{p \in \text{lhs}(r)} \bigsqcup_{d \in db} \sqcap^*\{\sigma \mid p^\sigma = d\}$$

$$= \bigsqcap_{p \in \text{lhs}(r)} \bigsqcup_{d \in db} \text{match}(p, d). \tag{3.19}$$

Equation 3.19 is the final form of our initial implementation of match-rule. Note that this equation constitutes an executable program. For example, in Lisp this equation could be implemented as

```
(defun match-rule (r db)
  (reduce #'⊓
          (mapcar (lambda (p) (reduce #'⊔
                                      (mapcar (lambda (d) (match p d))
                                              db)))
                  (lhs r))))
```

with $\sqcap$ and $\sqcup$ as defined above. (Note that the definitions presented above for $\sqcap$, $\sqcup$, and $\sqcup^*$ also constitute executable programs, as shown in Figures 5-4 and 5-5.)

The next chapter starts with this program, optimizes it, and show its correspondence to the Rete algorithm. Chapter 5 presents an implementation, using program transformations, of this derivation, and of the optimizations presented in Chapter 4.

Chapter 4

Correspondence to Rete and Optimization

This chapter demonstrates the correspondence between the matcher developed in Chapter 3 and the Rete network, and optimizes this matcher using finite differencing and partial evaluation.

4.1 Correspondence to the Rete Matcher

In the last chapter we derived the following program for match-rule (Equation 3.19):

$$\text{match-rule}(r, db) = \prod_{p \in lhs(r)} \bigsqcup_{d \in db} \text{match}(p, d)$$

We can now show the correspondence between this formulation of match-rule, and a Rete network (for a single rule). Let us consider a decomposition of this program analogous to the decomposition of the specification in Equations 3.4 to 3.6. Assume that a rule r has k patterns, and label them $p_1, p_2, \ldots, p_k$. If we define a vector R (corresponding to the right memories of the nodes in a Rete network) as

$$R_i = \bigsqcup_{d \in db} \text{match}(p_i, d) \quad \text{for } 1 \leq i \leq k \tag{4.1}$$

then this expression for match–rule in can be written as

```
1 function match-rule(r, db)
2     let k = length(lhs(r))
3     for i = 1, k
4         R_i ← ⊔_{d∈db} match((lhs(r))_i, d)
5         L_1 ← if i = 1 then R_1
6                         else L_{i-1} ⊓ R_i end if
7     end for
8     return L_k
9 end function
```

Figure 4-1: Match–Rule Implementation.

$$\text{match-rule}(r, db) = R_1 \sqcap R_2 \sqcap R_3 \sqcap \ldots \sqcap R_k. \tag{4.2}$$

Since $\sqcap$ is associative, this expression can be parenthesized as

$$\text{match-rule}(r, db) = (\ldots((R_1 \sqcap R_2) \sqcap R_3)\ldots) \sqcap R_k. \tag{4.3}$$

If we label these parenthesized expressions as elements of a vector L (corresponding to the left memories of nodes in a Rete network), i.e.

$$\begin{aligned} L_1 &= R_1 \\ \text{and } L_i &= L_{i-1} \sqcap R_i \text{ for } 2 \leq i \leq k \end{aligned} \tag{4.4}$$

then the expression for match rule can be written as

$$\text{match-rule}(db, r) = L_k. \tag{4.5}$$

The formulation in Equations 4.1, 4.4 and 4.5 is collected in Figure 4-1 into an implementation of match-rule.

The correspondence between the implementation of match-rule in Figure 4-1, and the Rete network described in Chapter 2, is illustrated in Figure 4-2. We can see the correspondence by identifying the L_i and R_i values in match-rule with the contents of the Left and Right input memories of the combine nodes in the Rete network.

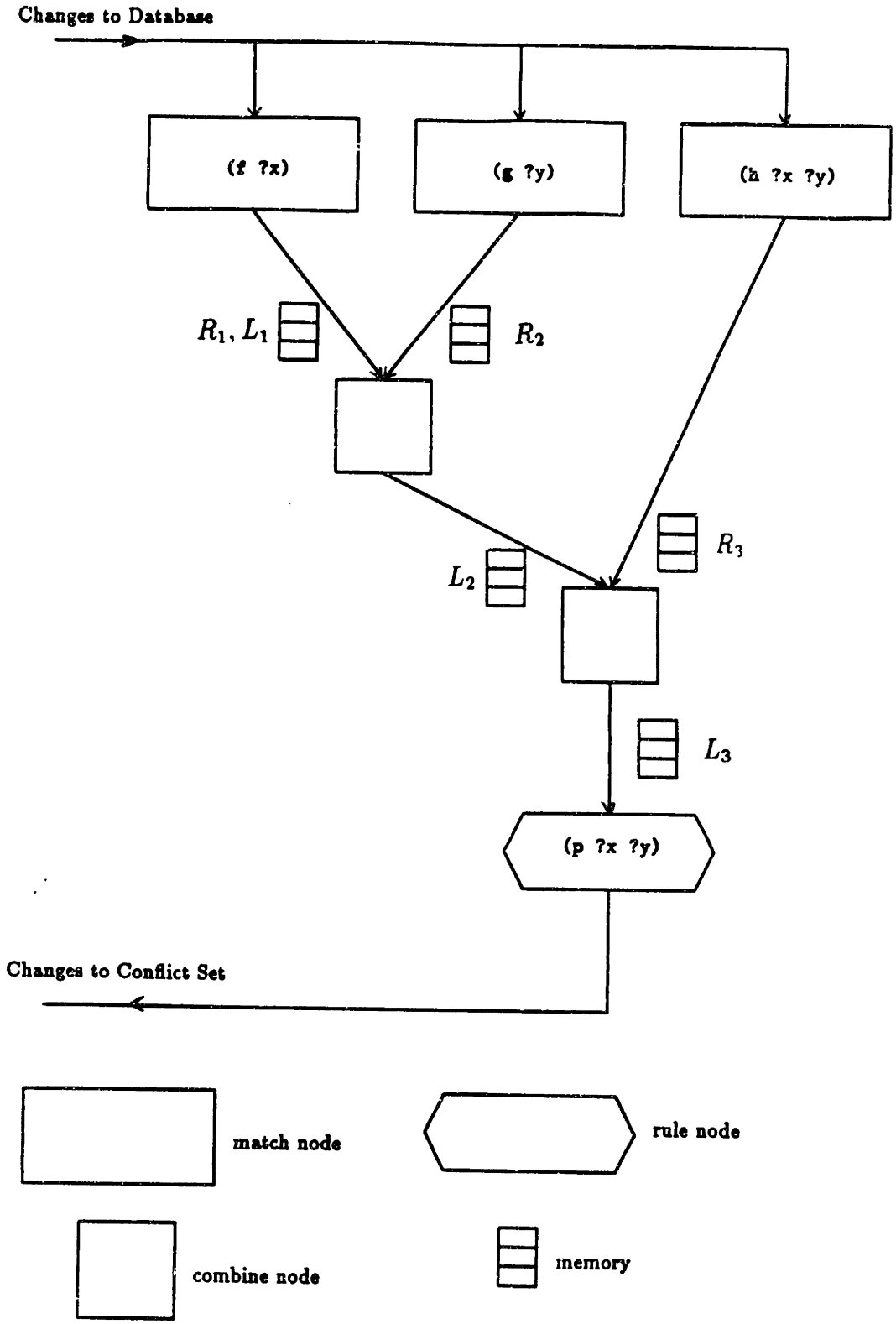


Figure 4-2: Correspondence between Match-Rule and the Rete Network.

Note that this correspondence holds for the topology of the network, but not for its behavior over time. Whereas `match-rule` is a functional program[1], the Rete network incrementally updates the contents of the node memories. The correspondence between the derived program and the Rete network will be extended to include the incremental behavior in the next section, and the remaining differences will be discussed in Sections 5.4 and 6.2.

4.2 Optimization

The major source of efficiency in the Rete matcher is its incremental update of the conflict set as the database is modified. This section will present a derivation of this optimization, and apply it to the program under development.

Let us assume that the database is large, and therefore changes relatively slowly, since only one element is added to it on each cycle. Let us further assume that the computation of match-rule is expensive. This situation suggests that the performance of the system could be improved by finite differencing [19] the computation of match-rule with respect to the updates to the database.

This finite-differencing can be carried out as follows. Consider the operation of the rule system interpreter over two cycles. Let db be the input data to the first cycle; let R_i and L_i be the values computed during this first cycle; let $\dot{db}$ be the new data generated during this cycle, and let

$$\overline{db} = db \cup \dot{db}$$

be the resulting database carried over to the next cycle of the interpreter. Let the values of $\overline{R_i}$ and $\overline{L_i}$ be the values computed during the second cycle, and let $\dot{R}_i$ and $\dot{L}_i$ be the changes to R and L from the first cycle to the second, i.e.

$$\overline{R_i} = R_i \cup \dot{R}_i$$

[1]The program in Figure 4-1 is a single-assignment program.

$$\overline{L_i} = L_i \cup \dot{L}_i.$$

Now, using the associative law[2] for $\sqcup$

$$\bigsqcup_{d \in db1 \cup db2} \{\sigma \mid p^\sigma = d\} = \bigsqcup_{d \in db1} \{\sigma \mid p^\sigma = d\} \sqcup \bigsqcup_{d \in db2} \{\sigma \mid p^\sigma = d\} \tag{4.6}$$

and the distributive law[3] for $\sqcap$ over $\sqcup$

$$\begin{aligned} &(\Upsilon \sqcup \Phi) \sqcap (\Psi \sqcup \Gamma) \\ &= (\Upsilon \sqcap \Psi) \sqcup (\Upsilon \sqcap \Gamma) \sqcup (\Phi \sqcap \Psi) \sqcup (\Phi \sqcap \Gamma) \end{aligned} \tag{4.7}$$

we can derive that

$$\begin{aligned} \overline{R_i} &= \bigsqcup_{d \in db \cup \dot{db}} \{\sigma \mid p^\sigma = d\} && (4.8) \\ &= \bigsqcup_{d \in db} \{\sigma \mid p^\sigma = d\} \sqcup \bigsqcup_{d \in \dot{db}} \{\sigma \mid p^\sigma = d\} \\ &= R_i \sqcup \bigsqcup_{d \in \dot{db}} \{\sigma \mid p^\sigma = d\} && (4.9) \end{aligned}$$

and (for $i \geq 2$)

---

[2]Let $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_m\}$. Then

$$\begin{aligned} \bigsqcup_{x \in A \cup B} f(x) &= f(a_1) \sqcup \ldots \sqcup f(a_n) \sqcup f(b_1) \sqcup \ldots \sqcup f(b_m) \\ &= (f(a_1) \sqcup \ldots \sqcup f(a_n)) \sqcup (f(b_1) \sqcup \ldots \sqcup f(b_m)) \\ &= (\bigsqcup_{x \in A} f(x)) \sqcup (\bigsqcup_{x \in B} f(x)). \end{aligned}$$

[3]Since $\Upsilon \sqcup \Phi = \Upsilon \cup \Phi$, therefore

$$\begin{aligned} (\Upsilon \sqcup \Phi) \sqcap \Psi &= \{\sigma \mid \exists_{\tau \in \Upsilon \sqcup \Phi} \exists_{\upsilon \in \Psi} \sigma = \tau \sqcap^* \upsilon\} \\ &= \{\sigma \mid \exists_{\tau \in \Upsilon \cup \Phi} \exists_{\upsilon \in \Psi} \sigma = \tau \sqcap^* \upsilon\} \\ &= \{\sigma \mid \exists_{\tau \in \Upsilon} \exists_{\upsilon \in \Psi} \sigma = \tau \sqcap^* \upsilon\} \cup \{\sigma \mid \exists_{\tau \in \Phi} \exists_{\upsilon \in \Psi} \sigma = \tau \sqcap^* \upsilon\} \\ &= \{\sigma \mid \exists_{\tau \in \Upsilon} \exists_{\upsilon \in \Psi} \sigma = \tau \sqcap^* \upsilon\} \sqcup \{\sigma \mid \exists_{\tau \in \Phi} \exists_{\upsilon \in \Psi} \sigma = \tau \sqcap^* \upsilon\} \\ &= (\Upsilon \sqcap \Phi) \sqcup (\Phi \sqcap \Psi). \end{aligned}$$

$$
\begin{aligned}
\overline{L_i} &= \overline{L_{i-1}} \sqcap \overline{R_i} \\
&= (L_{i-1} \cup \dot{L_{i-1}}) \sqcap (R_i \cup \dot{R_i}) \qquad (4.10) \\
&= (L_{i-1} \sqcap R_i) \cup (\dot{L_{i-1}} \sqcap R_i) \cup (L_{i-1} \sqcap \dot{R_i}) \cup (\dot{L_{i-1}} \sqcap \dot{R_i}) \\
&= L_i \cup (\dot{L_{i-1}} \sqcap R_i) \cup (L_{i-1} \sqcap \dot{R_i}) \cup (\dot{L_{i-1}} \sqcap \dot{R_i}). \qquad (4.11)
\end{aligned}
$$

If we assume that

$$| db | \gg | \dot{db} |,$$

then the computations in Equations 4.9 and 4.11 are less expensive than the computations in Equations 4.8 and 4.10, since the sets being manipulated are much smaller.

Figure 4-3 incorporates the match-rule code from Figure 4-1 into the original interpreter of Figure 2-1, and Figure 4-4 transforms this interpreter, using the optimizations presented in this section, into an incremental interpreter.

Lines 3–6 in Figure 4-4 perform the initial computation of the L and R values. Line 8 generates the conflict-set from L_k. Lines 10–12 select a rule and substitution from the conflict set, instantiate the rule with the substitution to obtain a new datum, and add the new datum to the conflict set. Line 14 computes the update to the R values resulting from the addition of the new data, and lines 15–16 compute the updates to the L values that result from the updates to the R values. Lines 18–20 add the new elements computed in lines 14–16 to the L and R values.

Figure 4-5 presents the rule interpreter after partial evaluation. The partial evaluation consists of unrolling the loops in Figure 4-4 into the single assignment statements in Figure 4-5. Each of the expressions in Figure 4-5 represents (according to the correspondence shown in Figure 4-2) a value computed by a node in the Rete network, and the variable references and assignments represent the dataflow between the nodes.

This concludes the formal algorithm derivation. The next chapter will present an implementation of this derivation using a program transformation system.

```
1  function single-rule-system (r, db)
2      let k = size(lhs(r))
3          repeat
4              for i = 1, k
5                  R_i ← ⊔_{d∈db} match((lhs(r))_i, d)
6                  L_1 ← if i = 1 then R_1
7                                 else L_{i-1} ⊓ R_i end if
8              end for
9              cs ← {⟨r, σ⟩ | σ ∈ L_k}
10             if cs = ∅ then return db end if
11             ⟨r, σ⟩ ← conflict-resolution(cs)
12             dȧb ← {rhs(r)^σ}
13             db ← db ∪ dȧb
14         end repeat
15 end function
```

Figure 4-3: Rule System for a Single Rule.

```
1  function single-rule-system (r, db)
2      let k = size(lhs(r))
3          for i = 1, k
4              R_i ← ⊔_{d∈db} match((lhs(r))_i, d)
5              L_1 ← if i = 1 then R_1
6                                 else L_{i-1} ⊓ R_i end if
7          end for
8          cs ← {⟨r, σ⟩ | σ ∈ L_k}
9          while cs ≠ ∅
10             ⟨r, σ⟩ ← conflict-resolution(cs)
11             ḋb ← {rhs(r)^σ}
12             db ← db ∪ ḋb
13             for i = 1, k
14                 Ṙ_i ← ⊔_{d∈ḋb} match(p_i, d)
15                 L̇_i ← if i = 1 then Ṙ_1
16                                 else (L̇_{i-1} ⊓ R_i) ∪ (L_{i-1} ⊓ Ṙ_i) ∪ (L̇_{i-1} ⊓ Ṙ_i)end if
17             end for
18             for i = 1, k
19                 R_i ← R_i ∪ Ṙ_i
20                 L_i ← L_i ∪ L̇_i
21             end for
22         end while
23         return db
24 end function
```

Figure 4-4: Incremental Rule System.

```
1  function single-rule-system (db, r)
2      let lhs(r) = ⟨p1, p2, p3⟩

3          R1 ← ⊔_{d∈db} match(p1, d)
4          R2 ← ⊔_{d∈db} match(p2, d)
5          R3 ← ⊔_{d∈db} match(p3, d)

6          L1 ← R1
7          L2 ← L1 ⊓ R2
8          L3 ← L2 ⊓ R3

9          repeat
10             cs ← {⟨r, σ⟩ | σ ∈ L3}
11             ⟨r, σ⟩ ← conflict-resolution(cs)
12             ḋb ← rhs(r)^σ}
13             db ← db ∪ ḋb

14             Ṙ1 ← ⊔_{d∈ḋb} match(p1, d)
15             Ṙ2 ← ⊔_{d∈ḋb} match(p2, d)
16             Ṙ3 ← ⊔_{d∈ḋb} match(p3, d)

17             L̇1 ← Ṙ1
18             L̇2 ← (L̇1 ⊓ R2) ∪ (L1 ⊓ Ṙ2) ∪ (L̇1 ⊓ Ṙ2)
19             L̇3 ← (L̇2 ⊓ R3) ∪ (L2 ⊓ Ṙ3) ∪ (L̇2 ⊓ Ṙ3)

20             R1 ← R1 ∪ Ṙ1
21             R2 ← R2 ∪ Ṙ2
22             R3 ← R3 ∪ Ṙ3
23             L1 ← L1 ∪ L̇1
24             L2 ← L2 ∪ L̇2
25             L3 ← L3 ∪ L̇3
26         until cs = ∅
27         return db
28 end function
```

Figure 4-5: Rule System after Partial Evaluation.

Chapter 5

Implementation using Transformations

This chapter describes an implementation, using a program transformation system, of the derivations described in Chapters 3 and 4. A first set of transformations is used to transform the specification into an initial implementation by transforming the universal and existential quantifiers in the specification, first into set operations, and then into lattice operations. A second set of transformations is used to optimize the initial implementation by performing finite differencing and partial evaluation.

Sections 5.1 and 5.2 introduce transformation systems in general, and specifically the Refine language used in this implementation. The next three sections describe the implementation. Section 5.3 presents the preliminaries for the development. These include the types used in system, some sample data and rules, the match and instantiate functions, and the lattice operations. Section 5.4 presents the rule system specifications, the transformations used in the initial implementation, and the resulting code. Section 5.5 presents the transformations used for finite differencing and partial evaluation, and presents the resulting optimized code. Section 5.6 describes the current status of the implementation, and describes further work to be done, such as automatically synthesizing the match, instantiate, and lattice operations, and automating the control of the

transformation applications.

5.1 Transformation Systems

This section describes transformation systems in general, and the Refine language in particular.

A transformational development begins with a complete formal specification for the target program. Correctness–preserving tranformations are then applied to this specification to derive the program. These transformations embody fragments of programming knowledge. Specifically, a transformation is a rewrite rule that replaces a fragment of source text with an equivalent fragment (that hopefully represents a step towards the final implementation).

Note that transformations are local, i.e. they apply to a local piece of the specification and generate a local piece of code. The paradigm of transformational programming relies on the specification and the resulting program having the same locality. Some attempts have been made to remove this restriction by having transformation systems gather information from other parts of the specification, e.g. from the contexts of a source expression within the syntax tree [31].

Note also that transformational programing can be either semi–automatic, where the user specifies which transformations to run and which nodes of the syntax tree to run them on, or fully automatic, where the system makes these decisions. The implementation presented in this chapter is semi–automatic. The final section of this chapter discusses possibilities for completely automating the derivation.

A transformation system requires a language for specifications and a language for implementations. A wide–spectrum language is a single language that can be used for expressing both specifications and implementations. Its constructs range from high–level, and possibly non–executable constructs such as unbounded quantifers, to the usual low–level constructs found in programming languages.

Refine [21] is a strongly–typed language, containing Algol–like program constructs, e.g. `if...then...else`, along with higher–level mathematical constructs, e.g. set–formers such as $\{x \mid (x)x \in S \wedge x > 1\}$. Refine is built on an object database. All objects in the system—types, variables, functions, transformation rules—are stored in a single database.

Refine has the standard primitive data types such as numbers, symbols, etc., similar to the datatypes in Lisp. Refine allows the definition of compound data types. The following are the type declarations, syntax for literal values, and syntax for general expressions, for the data types used in this derivation. Note that both the set–former and the map–former contain, after the "|", a list of variables local to the expression.

datatype	type declaration	literal expression	former expression
set	set(T)	$\{1,2,3\}$	$\{x \mid (x)\ p(x)\}$
map	map(T1, T2)	$\{\mid 1 \rightarrow 3, 2 \rightarrow 5 \mid\}$	$\{\mid x \rightarrow y \mid (x,y)\ p(x,y) \mid\}$
sequence	seq(T)	$[1,2,3]$	
tuple	tuple(f1:T1, f2:T2)	$\langle 1,2,3 \rangle$	

The set operations that we will use are union ($\cup$), intersection ($\cap$), adjoin an element to a set (S with x), construct the image of a function on a set (image(f, S)), and reduce a set using a binary operation (reduce(op, S)). For maps we will use $dom(m)$ which returns the domain of the map m, and $m(x)$ which returns the image of x in map m.

5.2 Types and Operations

This section will present preliminaries for the development, including type declarations and sample data, the `match` and `instantiate` functions, and the lattice operations. In this implementation these primitive functions have been written manually. Section 5.5 describes how these functions might be automatically synthesized.

Figure 5-1 shows the declarations of the types used in the system.[1] Variables and

[1]Note that for the implementation, the prefix character for variables has been changed from "?" to

constants are both symbols, and they are differentiated by the variablep and constantp predicates shown. Database terms are s-expressions containing variables and constants. This type should be a union of variables, constants, and, recursively, pairs of terms. Since Refine lacks union types, an escape from the type system, any-type, was used. Rules are represented as tuples of the LHS and RHS. The lattice elements are represented as described in Chapter 3. A substitution (SUBST) is a map from variables to terms, and a disjunctive substitution (DSUBST) is a set of substitutions. Finally, the figure includes a sample database and rulebase.

```
(defobject TERM* type = any-type)
(defobject VARIABLE* type = symbol)
(defobject CONSTANT* type = symbol)
(defobject RULE* type = tuple(lhs:set(term*), rhs:term*))
(defobject SUBST* type = map(variable*, term*))
(defobject DSUBST* type = set(subst*))

(defobject VARIABLEP* function (p: term*)
  : boolean =
  if symbolp(p)
  then (symbol-to-string(p))(1) = #\-
  else false)

(defobject CONSTANTP* function (p: term*)
  : boolean =
  if symbolp(p) then ¬variablep*(p) else false)

(defobject *DB* var:set(term*) = {['f,'a], ['f,'b], ['f, 'c], ['g, 'a],
                                  ['g, 'b], ['h, 'a, 'b],
                                  ['if, 'q, 'r], 'q})

(defobject *RB* var:set(rule*) =
          {<{['f, '-x], ['g, '-y], ['h, '-x, '-y]},
            ['p, '-x, '-y]>,
           <{['if, '-x, '-y], '-x},
            '-y>})
```

Figure 5-1: Implementation Data Types.

"-". This was necessary because the Refine grammar does not allow symbols to begin with "?". Also, to avoid conflicting with predefined types in Refine, an asterisk was appended to the names of all types defined in this derivation.

The code for instantiate and for match is shown in Figures 5-2 and 5-3.

```
(defobject INSTANTIATE function (p: term*, s: subst*)
   : term* =
   if constantp*(p)
   then p
   else if variablep*(p)
        then if p ∈ domain(s)
             then s(p)
             else undefined
        else cons(instantiate(car(p), s), instantiate(cdr(p), s)))
```

Figure 5-2: Instantiate Implementation.

```
(defobject MATCH function (p: term*, d: term*)
   : dsubst* =
   if constantp*(p)
   then if p=d
        then { {| |} }
        else {}
   else if variablep*(p)
        then { {| p ↦ d |} }
        else if constantp*(d)
             then {}
             else dsubst-meet(match(car(p), car(d)),
                              match(cdr(p), cdr(d))))
```

Figure 5-3: Match Implementation.

Instantiate takes as arguments a term and a substitution, and returns the term that results from replacing all of the variables in the term with their images in the substitution. It is implemented using structural ("car-cdr") recursion on the term. Match takes as arguments two terms, a pattern and a datum, and returns a disjunctive substitution (DSUBST*) under which the two are equivalent. Match recurses down the two structures in parallel. If a variable is encountered in the pattern, it creates a substitution binding that variable to the corresponding structure in the datum. If the corresponding parts of the pattern and datum differ in any other way, it returns failure (an empty DSUBST*).

The code for the lattice operations is shown in Figures 5-4 and 5-5. These operations are straightforward translations of the definitions in Chapter 3.

```
(defobject SUBST-MEET function (s1: subst*,
                                s2: subst*)
   : subst* =
   let (dom = domain(s1) ∪ domain(s2))
     if ∃(x) (x ∈ dom ∧ d1 = s1(x) ∧ d2 = s2(x)
                ∧ defined?(d1) ∧ defined?(d2)
                ∧ d1 ≠ d2)
       then undefined
       else
       {| x ↦ if defined?(s1(x)) then s1(x) else s2(x)
            | (x) x ∈ dom |} )
```

Figure 5-4: Substitution Semi-Lattice Meet.

```
(defobject DSUBST-MEET function (ds1: dsubst*, ds2: dsubst*)
   : dsubst* =
   { s | (s1, s2, s) s1 ∈ ds1 ∧ s2 ∈ ds2
         ∧ s = subst-meet(s1, s2)
         ∧ defined?(s) })
(defobject DSUBST-JOIN function (ds1: dsubst*, ds2: dsubst*)
   : dsubst* =
   ds1 ∪ ds2)
```

Figure 5-5: Disjunctive Substitution Lattice Operations.

5.3 Initial Derivation

This section presents the rule system specifications. The specification shown in Figure 5-6 is a straightforward Refine translation of the specification in Figure 2-1. Since we are only concerned with the matching part of the rule system, a stub has been inserted for the conflict–resolution procedure.[2]

[2]A stub has also been inserted for the representation operation (Rep), to satisfy the requirement of the Refine/KIDS system that all portions of the specification syntax tree be defined.

```
(defobject FC-SPEC function (db:set(term*), rb:set(rule*)) : set(term*) =
  for* (cs, db = db, cs-elt, r, e)
    cs ← { <r, e> | (r, e) r ∈ rb ∧ e ∈ match-rule(r, db) }
    while cs ≠ {},
    cs-elt ← conflict-resolution(cs),
    r ← cs-elt.1, e ← cs-elt.2,
    db ← db with instantiate(r.rhs, e)
    returns db)

(defobject MATCH-RULE function (db:set(term*), r:rule*) : dsubst* =
  rep({e | (e) ∀(p) (p ∈ r.lhs =>
                     (∃(d) (d ∈ db ∧ instantiate(p, e) = d)))}))

(defobject CONFLICT-RESOLUTION function (cs: set(any-type)) : any-type =
  arb(cs))

(defobject REP function (es:set(subst*)) : dsubst*)
```

Figure 5-6: Rule System Specification (Iterative Version).

The iterative "for*" construct used in Figure 5-6 is part of the language used in KIDS, but is not currently implemented in Refine. Therefore, I have reformulated the specification tail recursively. See Figure 5-7.

As in Chapters 3 and 4, this chapter will focus on the implementation of `match-rule`. (This is also the only portion of the specification in Figure 5-7 that cannot be simply translated into Lisp by the Refine compiler.)

The transformations used in the initial implementation are shown in Figure 5-8. A

```
(defobject FC-SPEC function (db:set(term*), rb:set(rule*)) : set(term*) =
  let (cs = { <r, e> | (r, e) r ∈ rb ∧ e ∈ match-rule(r, db) })
    if cs ≠ {}
    then let (cs-elt = conflict-resolution(cs))
           let (r = cs-elt.1, e = cs-elt.2)
             fc-spec(db with instantiate(r.rhs, e), rb)
    else db)
```

Figure 5-7: Rule System Specification (Tail Recursive Version).

```
(defobject UNIVERSAL-TO-INTERSECTION* rule (a)
  a = '{ @expr | ($vars1) ∀(y) (y ∈ @S => @formula) }'
  -->
  a = 'Intersection*( { {@expr | ($vars1) (@formula)} | (y) y ∈ @S } )' )
(defobject EXISTENTIAL-TO-UNION* rule (a)
  a = '{ @expr | ($vars1) ∃(y) (y ∈ @S ∧ $other-cjs) }'
  -->
  a = 'Union*( { {@expr | ($vars1) ∧($other-cjs)} | (y) y ∈ @S } )' )
(defobject INTERSECTION*-TO-REDUCE-INTERSECTION rule (a)
  a = 'Intersection*(@S)'
  -->
  a = 'Reduce('∩, @S)')
(defobject UNION*-TO-REDUCE-UNION rule (a)
  a = 'Union*(@S)'
  -->
  A = 'Reduce('∪, @S)')
(defobject INSTANTIATE-TO-MATCH rule (a)
  a = 'rep({ e | (e) instantiate(@p, e) = @d })'
  -->
  a = 'match(@p, @d)')
(defobject REP-UNION-TO-DSUBST-JOIN-REP rule (a)
  a = 'rep(@S1 ∪ @S2)'
  -->
  a = 'DSUBST-JOIN(rep(@S1), rep(@S2)))
(defobject REP-INTERSECTION-TO-DSUBST-MEET-REP rule (a)
  a = 'rep(@S1 ∩ @S2)'
  -->
  a = 'DSUBST-MEET(rep(@S1), rep(@S2)))
```

Figure 5-8: Transformations for Initial Implementation.

transformation is implemented as an object of type rule.[3] Its single argument is a node in a Refine syntax tree. The syntax ... --> ... specifies that if the value on the left side of the arrow is true, then the expression on the situation described on the right side of the arrow is actualized.[4] In Refine's rule pattern syntax, symbols beginning with "@"

[3]Refine's **rule**, not the **rule*** used in the program being derived here

[4]Obviously this can only be accomplished for a restricted set of right side conditions. Refine can handle conditions that require modifying a stored value, e.g. destructively modifying a structure in memory to contain a specified value. Here the condition a = ... is achieved by destructively modifying the portion of the syntax tree on which the rule was invoked.

are variables that match a single program term, and symbols beginning with "$" are variables that match a sequence of program terms.

The first two rules in this figure transform expressions involving universal and existential quantifiers into equivalent expressions involving unary intersection and union. (These transformations can be viewed as homomorphisms from Boolean algebra to set algebra.) The next two rules implement unary intersection and union using reductions of the corresponding binary operations. These first four rules are domain independent. The next three rules are specific for this problem domain. The first expresses the specification for the primitive match function. The last two rules express the homomorphism of the representation, from unions and intersection of sets of substitutions, to operations in the disjunctive substitution lattice.

The result of applying these transformations to the match-rule specification in Figure 5-6 is the implementation shown in Figure 5-9, which corresponds to the implementation derived in Chapter 3.

In the next section this initial implementation will be optimized using finite difference and partial evaluation transformations. Before performing these transformations, it will be convenient to perform a program folding step.[5] The rest of this section describes this step.

One problem that arises in many symbol manipulation systems, such as program transformation systems and computer algebra systems, is intermediate–expression blow–

[5]The transformation that replaces a call to a function by the inline expansion of the program body is known as *unfolding*. The inverse transformation is known as *folding*.

```
(defobject MATCH-RULE function (db: set(term*), r: rule)
  : dsubst* =
  reduce('dsubst-meet,
         image((λ(p) reduce('dsubst-join, image((λ(d) match(p,d)),
                                                db))),
               r.lhs)))
```

Figure 5-9: Initial Implementation of Match–Rule.

up. The intermediate results that arise in these systems, though correct, can become overly complex, taxing the resources of both the computer and the user. Ideally, symbol manipulation systems might be designed to help minimize this problem. For now, manual methods will be used.

A small example of expression blow-up that occurs here can be solved by manual program folding. The initial implementation in Figure 5-9 consists of calls to `dsubst-join` nested inside of calls to **`dsubst-meet`**. It turns out to be convenient to fold the sections of the code containing the calls to **`dsubst-join`**. This will simplify the transformations needed in the next section. The result of this folding is shown in Figure 5-10. The new function is called **`match-elt-set`** because it matches a pattern against all of the elements of a set and returns a disjunctive substitution summarizing the results.

5.4 Optimization

This section presents transformations to (partially) implement the optimizations described in Chapter 4. Note that whereas the programs in Chapter 4 explicitly manipulate the vectors L and R element by element, the functional programs in this section are restricted to manipulating entire sets and sequences. The next two subsections describe the finite differencing and partial evaluation optimizations.

```
(defobject MATCH-RULE function (db: set(term*), r: rule)
  : dsubst* =
  reduce('dsubst-meet,
         image((λ(p) match-elt-set(p, db)),
               r.lhs)))
(defobject MATCH-ELT-SET function (p: term*, s: set(term*))
  : dsubst* =
  reduce('dsubst-join, image((λ(d) match(p,d)),
                             s)))
```

Figure 5-10: Match-Rule after Folding Match-Elt-Set.

5.4.1 Finite Differencing

The main optimization involves incrementally updating the conflict set as changes are made to the database. Assume that the system has just modified the database by adding a new datum, and is now computing the new conflict set by matching all of the rules against the new database. Incremental updating involves reusing the intermediate results from the previous cycle in generating this new conflict set.

The key to this optimization lies in factoring the expression for the new value of the conflict set into (1) terms involving the previous conflict set computation, and (2) terms involving the newly added datum. In addition to this factoring, "bookkeeping" code is needed to store the results from the previous cycle for use in the next cycle. This can be seen in the use of the L(eft) and R(ight) vectors in lines 13–20 in the program in Figure 4-4. In the current implementation, only the first of these tasks, the factoring, has been carried out.[6]

This factoring is accomplished as follows. Assume that the system has just added some new data ($\dot{db}$) to the previous contents of the database (db).[7] The task is now to compute the conflict set for this new database. For a given rule this can be expressed as computing `match-rule`$(db \cup \dot{db})$. (See Figure 5-11.) The requirement that the program be incremental involves separating, as much as possible, the computations involving db from the computations involving $\dot{db}$. In the final program all computations involving $db \cup \dot{db}$ will be separated into computations involving db, and computations involving $\dot{db}$. The motivation is that the values involving db will be saved from the previous cycle of the interpreter, and the computations involving $\dot{db}$ will be relatively fast since the sets manipulated will be small (compared with the sets generated in matching db, which is assumed to be much larger than $\dot{db}$).

[6] Compare this division to the separation of functional portions of code from portions of code involving state in [1].

[7] Though the rule system specification that we have been using only requires adding a single datum to the database at a time, in actuality it is more efficient to add several new data at a time. Therefore, the formulation here has been generalized to deal with adding a set of new data to the database.

```
(defobject FD-PE-SPEC function (db-old:set(term*), db-delta:set(term*))
  : dsubst* =
  match-rule(< { ['f, '-x], ['g, '-y], ['h, '-x, '-y] },
               ['p, '-x, '-y] >,
             db-old ∪ db-delta))
```

Figure 5-11: Specification for Optimized Matcher.

The rules needed for finite differencing involve the distribution of $\sqcup$ over the union of db and $\dot{db}$, and of $\sqcap$ over $\sqcup$.[8] Some of these rules are shown in Figure 5-12.

Performing the source–to–source transformations, from initial implementation to finite–differenced code, requires several low–level rules similar to REP-REDUCE-UNION-TO-REDUCE--DSUBST-JOIN-REP in Figure 5-12. In general, any mathematical property, such as a homo-

[8]Corresponding, respectively, to the Right memories and the Left memories in Chapter 4.

```
(defobject LEMMA-DISTRIBUTE-MATCH-ELT-SET-OVER-UNION rule (a)
  a = 'match-elt-set(@p, @S1 ∪ @S2)'
  ∧ p2 = c-t(p)
  -->
  a = 'match-elt-set(@p, @S1) ∪ match-elt-set(@p2, @S2)')

(defobject DISTRIBUTE-DSUBST-MEET-OVER-DSUBST-JOIN rule (a)
  a = 'dsubst-meet(@S1, dsubst-join(@S2, @S3))'
  -->
  a = 'dsubst-join(dsubst-meet(@S1, @S2), dsubst-meet(@S1, @S3))')

(defobject DISTRIBUTE-SETFORMER-OVER-UNION-OF-DOMAINS rule (a)
  a = '{ @expr | (y, $vars1) y ∈ @S ∪ @R ∧ $other-cjs }'
  ∧ expr2 = c-t(expr)
  -->
  a = '{ @expr | (y, $vars1) y ∈ @S ∧ $other-cjs }
       ∪ { @expr2 | (y, $vars1) y ∈ @R ∧ $other-cjs }')

(defobject REP-REDUCE-UNION-TO-REDUCE-DSUBST-JOIN-REP rule (a)
  a = 'rep( reduce('∪, image((λ ($vars1) @expr), @S)))'
  -->
  a = 'reduce('dsubst-join, image((λ ($vars1) rep(@expr)), @S))')
```

Figure 5-12: Finite Differencing Transformations.

morphism or a distributive law, can be used to generate several different syntactic rules. Ideally, we would like to be able to enter the mathematical properties —the homomorphisms and distributive laws—directly, and have a simple automated theorem proving component synthesize the specific low–level rules needed. For example, the `REP-REDUCE--UNION-TO-REDUCE-DSUBST-JOIN-REP` rule in Figure 5-12 should be easily derivable from the `REP-UNION-TO-DSUBST-JOIN-REP` rule in Figure 5-8.[9] For now, we will enter these rules manually. (And, in the current status of the implementation, some of the transformations have been performed manually.)

As mentioned above, the development can be simplified by folding `match-elt-set` (see Figure 5-10), and expressing the transformations in terms of this function. For example, the distributive property of `dsubst-join` over union, can be expressed as a lemma involving distributing `match-elt-set` over union (shown in Figure 5-12). This greatly reduces the number of steps required in the derivation, and makes the final code (shown in Figure 5-14) more concise.

5.4.2 Partial Evaluation

The Rete matcher involves a dataflow network of matching nodes, as described in Chapters 2 and 4. This network can be automatically generated by partially evaluating the initial implementation in Figure 5-10. This partial evaluation replaces the run–time iteration of the lattice operations over the elements in the left hand side of the rule, with a compile–time expansion of the call tree for each rule.

The partial evaluation is conducted by substituting the fixed value for the LHS of the rule into the call to match rule, and simplifying the resulting program. These simplifications are performed by propagating constant values up through the code. For example, the expression

[9]The automated generation of transformations required here might not be too difficult. For example, the transformation needed here could have been provided if for every rule involving a homomorphisms on associative binary operations, a rule generalizing this homorphism to reductions of these operation on sets of values could be automatically generated.

```
(defobject PARTIAL-EVALUATE-IMAGE-OF-LAMBDA-ON-LITERAL-SET rule (a)
  a = 'image(@f, {$elts})'
  -->
  a = '{ $(image((λ (elt) make-term(c-t(f),[elt])), elts)) }')

(defobject REDUCE-BINOP-OF-LITERAL-SET rule (a)
  a = 'reduce(@binary-op, {@x, $y})'
  -->
  replace a by make-term(c-t(binary-op), [c-t(x),
                                          'reduce(@(c-t(binary-op)),
                                                  {$(c-tset(y))})']))

(defobject REDUCE-SINGLETON rule (a)
  a = 'reduce(@f, @s)' ∧ singleton-literalformer(s)
       ∧ x = the-literal-expr(s)
  --> replace a by c-t(x))
```

Figure 5-13: Partial Evaluation Transformations.

```
image((λ(p) match-elt-set(p, db)), [p1, p2, p3])
```

can be simplified to

```
[match-elt-set(p1, db), match-elt-set(p2, db), match-elt-set(p3, db)]
```

using the partial evaluation rules shown in Figure 5-13. These rules, along with many other simplification rules, are provided by the simplification facility in the KIDS system.

The effect of both finite differencing and partial evaluation is demonstrated by the transformation of the specification shown in Figure 5-11. This specification represents matching a particular rule against a database `db` with an incremental update `db-delta`. The result of applying the finite differencing rules and the partial evaluation rules to this specification is shown in Figure 5-14.

This resulting code corresponds to an expansion, for a 3–pattern rule, of the incremental matcher described in Equation 4.11. The correspondence between the calls to `match-elt-set` in this program and the values of the Right memory vector in Chapter 4 is as follows:

$$R_3 = \texttt{match-elt-set(['f, '-x], db),}$$

```
(defobject MATCH-RULE function (db: set(term*), db-delta: set(term*))
  : set(subst*) =
  dsubst-meet(match-elt-set(['f, '-x], db),
              dsubst-meet(match-elt-set(['g, '-y], db),
                          match-elt-set(['h, '-x, '-y], db)))
  ∪ dsubst-meet(match-elt-set(['f, '-x], db),
                dsubst-meet(match-elt-set(['g, '-y], db-delta),
                            match-elt-set(['h, '-x, '-y], db))
                ∪ dsubst-meet(match-elt-set(['g, '-y], db),
                            match-elt-set(['h, '-x, '-y], db-delta))
                ∪ dsubst-meet(match-elt-set(['g, '-y], db-delta),
                            match-elt-set(['h, '-x, '-y], db-delta)))
  ∪ dsubst-meet(match-elt-set(['f, '-x], db-delta),
              dsubst-meet(match-elt-set(['g, '-y], db),
                          match-elt-set(['h, '-x, '-y], db)))
  ∪ dsubst-meet(match-elt-set(['f, '-x], db-delta),
              dsubst-meet(match-elt-set(['g, '-y], db-delta),
                          match-elt-set(['h, '-x, '-y], db))
              ∪ dsubst-meet(match-elt-set(['g, '-y], db),
                            match-elt-set(['h, '-x, '-y], db-delta))
              ∪ dsubst-meet(match-elt-set(['g, '-y], db-delta),
                            match-elt-set(['h, '-x, '-y], db-delta))))
```

Figure 5-14: Final Match–Rule Implementation.

$$\dot{R}_3 = \texttt{match-elt-set(['f, '-x], db-delta),}$$

$$R_2 = \texttt{match-elt-set(['g, '-y], db),}$$

$$\dot{R}_2 = \texttt{match-elt-set(['g, '-y], db-delta),}$$

$$R_1 = \texttt{match-elt-set(['h, '-x, '-y], db),}$$

$$\dot{R}_1 = \texttt{match-elt-set(['h, '-x, '-y], db-delta).}$$

Using this correspondence, the program in Figure 5-14 can be rewritten as

$$\begin{aligned}
\overline{L_3} &= (R_3 \sqcap (R_2 \sqcap R_1) \\
&\cup (R_3 \sqcap [(\dot{R}_2 \sqcap R_1) \cup (R_2 \sqcap \dot{R}_1) \cup (\dot{R}_2 \sqcap \dot{R}_1)] \\
&\cup (\dot{R}_3 \sqcap (R_2 \sqcap R_1) \\
&\cup (\dot{R}_3 \sqcap [(\dot{R}_2 \sqcap R_1) \cup (R_2 \sqcap \dot{R}_1) \cup (\dot{R}_2 \sqcap \dot{R}_1)],
\end{aligned}$$

which is the same expression obtained by expanding Equation 4.11 for $i = 3$.

The remaining step for converting this program into a Rete network is the addition of the bookkeeping code alluded to above.

5.5 Future Work

This section briefly discusses two directions for extending this implementation.

5.5.1 Automatic Synthesis of Primitive Functions

One possible future direction for this implementation is the automatic synthesis of the primitive functions described in Section 5.2. Simple recursive functions, similar to match and instantiate, have been automatically synthesized using the synthesis component of KIDS [30]. Structural recursions such as these can be fully characterized by their recursive and base cases, as described in Section 5.2. These descriptions contain most of the information necessary to automatically derive these functions. Unfortunately, it was not possible to perform this synthesis in the current version of KIDS, since, at present,

the synthesis component can only handle recursions that terminate in a single base case, whereas the structural recursions for match and instantiate have two base cases: constants, and variables.

The remaining primitive functions are the lattice operations `subst-meet`, `dsubst-meet`, and `dsubst-join`. In Chapter 3, all of the information required to describe these functions is derived from the descriptions of the Substiutution Semi–Lattice and the Disjunctive Substitution Lattice. One possibility for automatically synthesizing is to automatically verify the derivation in Chapter 3 using a symbolic algebra system, perhaps using algebraic techniques similar to those in [4].

5.5.2 Automatic Control of Transformations

Another future direction for this implementation is in the area of automatic control of transformation application. The application of the transformations in this chapter has, for the most part, been manually directed. Though even manually directed transformational programming has advantages over manual program writing—e.g. correctness assurances—ultimately we would like the system to direct the use of the transformations. For example, KIDS currently has *tactics* to direct the synthesis of simple divide and conquer algorithms and simple global search algorithms.

One technique for automating the use of transformations is to design a set of transformations that when exhaustively applied[10] to a given source text will derive the desired result. This should be possible for the transformations in Section 5.3 since the derivation proceeds in a straight progression from terms involving compound logical expressions in set–formers, to terms involving set union and intersection, and finally to terms involving the lattice operations.

Finally, work is currently being conducted on general–purpose finite-differencing and partial–evaluation facilities to enable the automation of the optimizations performed in Section 5.4.

[10] i.e. repeatedly applied until none are applicable

Chapter 6

Discussion

This chapter discusses the lessons learned from this work, describes the progress made, identifies future directions to pursue, and describes the place of this thesis in the context of the related literature.

Section 6.1 discusses the relationship between the structures introduced in the derivation and the structures used in the model–theoretic semantics of first–order logic. This section briefly mentions some advantages of using algebraic techniques for software development. Section 6.2 discusses possible future directions for this research. Section 6.2.1 discusses short–range directions, such as addressing the simplifications in the derivation and implementation. Section 6.2.2 discusses long–range prospects, such as building a comprehensive library of programming knowledge. Section 6.3 discusses the related literature, and describes how this thesis contributes to this literature. Section 6.4 summarizes the conclusions of this thesis.

6.1 Correspondence to Model Theory

This section discusses the correspondence between the structures used in the derivation described in Chapter 3 and the structures used in the model–theoretic semantics of first-order logic. (Appendix A.2 presents a very brief review of basic model theory.)

This direct relationship between the domain structures and the implementation structures yields many advantages. It enables us to explain and (in principle) to verify the features implemented so far, and provides clear directions for implementing extensions.

Let us consider the analogy between a rule system with database *db* and rulebase *rb*, and a first order language L with a model structure whose universe is U. Identify the database *db* with the universe U, and consider the LHS of the rule to be a conjunction of terms in L. In this view, the set of matches for the LHS in *db* consists of the set of valuations σ, with universe U, such that $LHS^{\sigma} = \top$. (Where $\top$ denotes truth.) That is, the problem of finding all matches for a rule in a database can be seen as the problem of finding all possible assignments to the variables in a conjunctive term under which the term has an interpretation in a given universe.

Interpreting the rule system in this way, the following interpretations can be given to the structures used in Chapter 3:

- The semi-lattice SSL consists of valuations; $\preceq$ is an ordering on valuations; and $\sqcap^*$ is a binary operation on valuations.
- The lattice DSL consists of sets of valuations; $\sqsubseteq$ is an ordering on sets of valuations; and $\sqcap$ and $\sqcup$ are binary operations on sets of valuations.
- The procedure *match–rule* takes a LHS, and a universe *db*, and returns the maximal valuation, under $\preceq$, from the set $\{\sigma \mid LHS^{\sigma} = \top\}$.

This correspondence provides us with a clear semantics for the Rete algorithm in terms of the usual model–theoretic semantics of first–order logic.

This correspondence also provides a directions for implementing the extensions outlined in Section 2.4. For example, the facility for handling negated patterns in the Left Hand Side of a rule can be obtained by directly implementing the interpretation for negation in the semantic definition shown in Appendix A.2. In this definition, a predication $P(t_1, \ldots, t_n)$ is true under a valuation σ iff the tuple of its arguments, under the valuation σ, is an element of the relation P^{σ}, i.e.

$$(P(t_1, \ldots, t_n))^\sigma = \begin{cases} \top & \text{if} \quad \langle t_1^\sigma, \ldots, t_n^\sigma \rangle \in P^\sigma \\ \bot & \text{otherwise} \end{cases}$$

The semantic definition also indicates that a negation of a term α is true if and only if the term is false, i.e.

$$(\neg\alpha)^\sigma = \begin{cases} \top & \text{if} \quad \alpha^\sigma = \bot \\ \bot & \text{otherwise} \end{cases}$$

Therefore, a negated predication is true iff the tuple of its arguments is not an element in the corresponding relation in the model, i.e.

$$(\neg P(t_1, \ldots, t_n))^\sigma = \begin{cases} \top & \text{if} \quad \langle t_1^\sigma, \ldots, t_n^\sigma \rangle \notin P^\sigma \\ \bot & \text{otherwise} \end{cases}$$

The direct implementation of this specification consists in matching a negated pattern $\neg P(t_1, \ldots, t_n)$ (which represents a negated predication) iff there does not exist a valuation under which the pattern $P(t_1, \ldots, t_n)$ is true, i.e., iff there does not exits a binding to the variables in $P(t_1, \ldots, t_n)$ under which $P(t_1, \ldots, t_n)$ matches an element in the database. This corresponds exactly to the closed world assumption described in [22].

6.2 Future Work

This section describes possible directions for future research. The first two subsections decribe relatively short–range directions based on extending the derivation to include more of Rete's features, and completing the implementation. The second section describes some longer–range prospects.

6.2.1 Coverage of Features in Rete

Section 2.4 discussed several features of Rete that were not covered in the derivation in this thesis. Section 6.1 has already discussed incorporating one of these features,

matching for negated patterns, into the derivation. This section considers incorporating the remaining features into the derivation.

Updating the conflict set as data are removed from the database could be accomplished by distributing the match computation over set–difference, as well as over set–union. Distributive laws for set–difference could easily be added to the distributive laws for set–union used in the optimizations in Chapter 4.

The complexity in updating the conflict set as data are removed from the database arises from the difference between adding an item to a collection and removing an item previously added to a collection. Adding the matches for new data to the conflict set does not require reference to any past information about the database or conflict set. All of the information required is contained in the Left and Right memories. These encode the information from matches between component patterns and objects currently in the database, and are used to compute resulting conjunctive matches that include the new data being added to the database. When a new datum is added to the database, new substitutions can simply be added to these memories. (Assuming that the set–adjoin function can avoid adding duplicate elements, if the programmer chooses to represent sets as irredundant sequences.)

However, removing an element previously added to a collection requires either (1) maintaining a reference to that object, or (2) searching through the entire collection to remove all elements that match a given description. The second approach is very inefficient, and would cancel the benefits of having an incremental matcher. Therefore, the technique of choice is to maintain information, for each substitution in the network, about which data were matched in the derivation of that substitution. This information can be used by the system to update the network when a datum is retracted. This update is performed by removing from the network all substitutions that were derived using the datum being removed. In Rete, this is implemented by storing, with each token in the network, references to the data used in its generation. This could also be implemented using a dependency maintenance system. This dependency technique is

used in the AMORD rule system [9].

The sharing of computations between rules could be handled by performing straightforward common subexpression removal between the matching code generated for the separate rules.

6.2.2 Implementation

The derivation described in Chapters 3 and 4 was only partially implemented in the KIDS system. This section briefly discusses some future possibilities for this implementation.

It should be possible to synthesize `match` and `instantiate` using Smith's Divide–and–Conquer tactic. This is not possible at present because the tactic can only handle recursions with one base case, whereas the recursions on s–expressions in `match` and `instantiate` have two base cases (variables and constants). If KIDS were extended to handle recursions with multiple base cases, this synthesis could be performed. The two base cases in the representation of s–expressions also proved a problem for Refine, since it does not currently support union types.

The finite–differencing in Chapter 5 was performed using explicit transformations. Work is currently underway to expand the KIDS finite–differencing facility. This may enable the system to perform the finite differencing in Chapter 5, and to handle the finite differencing over deletions to the database that is required to implement retraction.

Many of the transformations used in the derivation could be implemented as theorems in the Rainbow theorem prover[31], (rather than being represented as syntactic transformations). This would allow for more flexible use of these rules.

Finally, the performance of the matcher could be improved by some simple data structure improvements. For example, the performance of `subst-meet` could be vastly improved if it were possible to quickly determine when two substitutions had disjoint domains. This could be achieved by including, in the implementation of a substitution, a bit–vector representation of the substitiution's domain. This would allow determining, in constant time, if two substitutions had any variables in common.

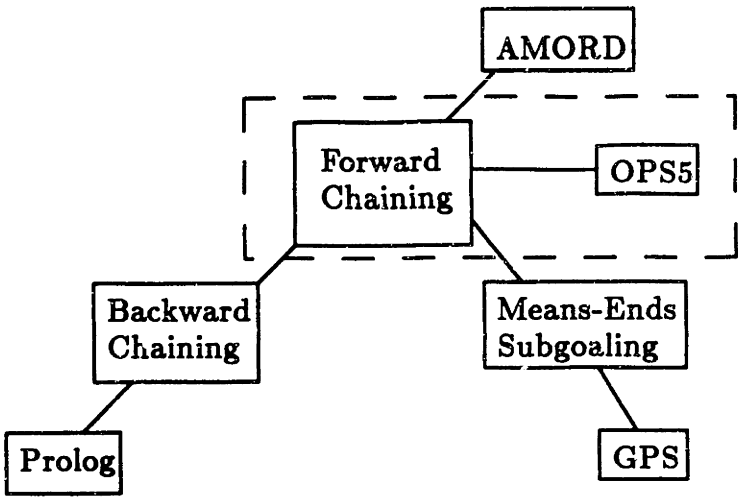


Figure 6-1: Rule System Design Space.

6.2.3 Program Design Spaces

The origin of this thesis was in a more ambitious plan by the author to formalize all the programming knowledge used in implementing rule systems. This thesis represents a detailed formalization of a small part of this design space.

The overall approach involves the following steps: (1) Examine several programs from the literature that belong to a particular application domain, e.g. rule systems. (2) Construct a taxonomy of these programs, based on the different design decisions that they embody. (3) Rederive these programs (or simplified versions of them) using formal specifications, domain models, formally–defined representations, and transformation rules.

The goal of this work is to build up libraries containing: formal domain models, mathematical structures for use in building domain models, and common programming techniques for efficiently implementing these structures.

This work involves an attempt to map out the *design space* of rule systems, i.e. the various design decisions that differentiate the programs in this domain. Figure 6-1 shows a portion of the rule system design space.

The programs in this space share a common domain model, but differ in various

design decisions made. For example, the AMORD system is a forward chaining system like OPS5, and also implements an incremental matching scheme. However, the AMORD system differs from OPS5 in the technique for maintaining the partial matches. AMORD rules cannot directly contain conjunctive patterns in their LHS's. The LHS's are limited to containing a single pattern. To obtain the effect of conjunctive patterns, a technique is used that is isomorphic to the technique of *Currying* in mathematical logic. To represent a conjunct of patterns, a rule is specified that matches the first pattern of the conjunct, and, as a RHS action, adds a new rule to the rulebase that handles the matching of the remaining patterns. This is repeated recursively until all of the patterns in the conjunct have been matched. For example, the OPS5 rule

$$\langle\ \{(f\ ?x), (g\ ?y), (h\ ?x\ ?y)\} \Rightarrow (p\ ?x\ ?y)\ \rangle$$

would be translated into the AMORD rule

$$\langle (f\ ?x) \Rightarrow \langle (g\ ?y) \Rightarrow \langle (h\ ?x\ ?y) \Rightarrow (p\ ?x\ ?y)\ \rangle\rangle\rangle.$$

Another example of differing design decisions involves indexing the patterns in the rulebase. Instead of matching a new datum against all of the rule patterns, the system can first use a quick approximate matcher to filter out rule patterns that cannot possibly match the datum. Alternatives for this quick matcher include testing if the predicate symbols in the pattern and datum are identical, testing if the length, or nesting, of the pattern and datum are identical, or testing if the constant portions of the pattern and datum are identical.

A third example of design decisions involves mechanisms for handling assumptions and retraction. Mechanisms that have been used for this function include context systems and truth–maintenance systems.

Achieving the ultimate goal of this project requires both the high–level mapping of the design space described above, and the detailed formalization and implementation of the pieces of this design space. This thesis presents a portion of this detailed formalization: a beginning for the section dealing with the formal derivation of OPS5.

Though it is very time–consuming, filling in these design spaces with formal pieces such as the one in this thesis appears to be a promising direction towards building up the library of programming knowledge that will allow us to progress to the next level of programming tools.

6.3 Related Work

This section describes work related to this thesis. This includes include references for the Rete network; research on formalizing matching and unification; research on program synthesis, transformations, and automatic programming; and general work on utilizing formal approaches in (manual) software development.

6.3.1 The Rete Matcher

The Rete network was developed by Forgy in 1974, and reported on in [11]. A more complete presentation can be found in Forgy's PhD thesis [12]. The Rete algorithm is used in the widely–used OPS5 Production System language. A textbook for the OPS5 system [5] contains a chapter on the Rete algorithm. However, these sources do not provide a concise formal description of the algorithm, a formal derivation of the algorithm, or a correctness proof.

This thesis has presented a formal derivation of the algorithm, and a partial implementation of this derivation, using correctness-preserving transformations. However, this thesis has dealt with a simplified version of the Rete algorithm, as described in Chapter 2 and Sections 6.2.1. It has focused on the core feature of Rete: the incremental update of the conflict set as the database is modified. The derivation and structures presented here can serve as a framework for a derivation of the complete Rete system.

6.3.2 Matching and Unification

This thesis has studied the problem of determining all possible matches between a set of conjunctive patterns (the LHS of a rule) and a database, and of incrementally updating this information as the database is modified. In this work, the relatively simple function for matching a single pattern and a single datum, shown in Figure 5-3, has been taken as a primitive. An approach for automating the derivation of this single–pattern single–datum matcher has been mentioned in Section 5.5. This section will discuss research on formalizing the single–pattern single–datum matching problem, and a generalization of this problem, unification, that allows both pattern and datum to contain variables.

Unification is a generalization of matching. In matching, only the pattern can contain variables; the datum must be a ground term. Matching requires finding a substitution for the variables in the pattern that make it equivalent to the datum. Unification is the analogous computation for two patterns that both contain variables. As in matching, the goal is to find a single substitution under which both terms are equivalent.

There is a body of literature about techniques for implementing unification. One of the earliest references to unification in the computer science literature was in Robinson's description of resolution theorem proving [26]. A recent summary of the formalization of unification and resolution can be found in [27]. A formalization somewhat similar to the formalization in this thesis, but only covering the single–pattern single–datum case, can be found in [10].

Manna and Waldinger [17] have presented a detailed formal derivation of the unification algorithm. Another derivation of the unification algorithm, published by Rydeheard and Burstall [28], is based on concepts from Category theory.

Recent work on unification has centered on the problem of constructing unification procedures that incorporate certain equational theories. For example, in a rule system dealing with arithmetic, representing each fact about addition requires *two* rules, due to the commutativity of addition. For example, the identity $x + 0 = x$ would be represented as the two rules

$$(+\ ?x\ 0) \Rightarrow ?x$$

$$(+\ 0\ ?x) \Rightarrow ?x.$$

It would be preferable to program the matcher to treat "+" as a commutative operator, and to allow $(+\ ?x\ 0)$ to match, for example, against $(+\ 0\ (f\ 1))$. The specific problem of incorporating commutativity and associativity into a matcher, known as *AC Unification* (Associative-Commutative Unification), has received much attention in the literature [34]. The general problem of adding equational theories to unification algorithms has also been addressed [20].

6.3.3 Automatic Programming and Transformations

The field of automatic programming, and the subfield of program transformations, are well served by survey articles and compilations [24, 13, 14].

Two pioneering efforts in codifying programming knowledge are the PhD theses of Barstow [2] and Rich [23]. Both of these codifications focused on the domain of common data structures and operations.

The derivation in Chapter 3 of this thesis is most closely related to the work on program synthesis, for example Smith's derivations of divide-and-conquer algorithms [30] and global-search algorithms [32].

The early section of the derivation in Chapter 3 is concerned with translating a logical specification into an executable form. The general problem of implementing such logical specifications has been addressed in several systems, for example, the AP5 system developed at ISI [7], and the CHI system developed at Stanford [35]. (The CHI system was a predecessor of the Refine system used in the implementation in Chapter 5.)

The optimization in Chapter 4 is related to the work on program optimization using finite differencing begun in the SETL project at New York University [19]. This technique has also been used in the KIDS system used in the implementation in Chapter 5 [31] [33].

This thesis was conducted as part of the Programmer's Apprentice project at the MIT Artificial Intelligence Laboratory [25]. The goal of this project is to build an intelligent apprentice system to aid programmers in all phases of their activity. I have focused on the problem of creating the library of programming knowledge that is fundamental to the operation of any such system. In order to achieve the level of detail and precision necessary for formalization and automation, this thesis has focused on a single narrow program domain. Hopefully the ideas presented here will be applicable to other efforts in the overall task of building the "complete library of programming knowledge."

6.3.4 Formal Methods in Deriving Programs

In addition to the work in automatic programming, there is a large effort in computer science to increase the use of formal methods in manual software development. This work is often associated with the pioneering work of Dijkstra, Hoare, and many others.

One approach to developing a formal theory of programs is to concentrate on functional programs. In his ACM Turing lecture[1], Backus discussed constructing an algebra of programs using the functional language FP. Functional programs have the advantage of being much easier to reason about than programs with state, since, like mathematical objects, functional expressions have the same value in any context. However, efficiency considerations have dictated that most real–world programming has been done using imperative languages. In [1], Backus presents some thoughts about combining the benefits of functional programming with the efficiency of imperative programming.

An example of current work on formalizing an algebra of (functional) programs is the work by Meertens [18] and Bird [4]. The aim of their work is to produce mathematical theories of common data structures and their operations. For example, [4] presents a portion of a theory of lists. Both the work in [4], and the FP work described in [1], deal with formalizations of common data structures such as lists and sequences. The derivation in Chapters 3 and 4 is offered as an example of applying these algebraic techniques to new compound data structures (SSL and DSL) derived for a particular

application domain.

The derivation in this thesis belongs to a tradition of formal derivations of algorithms, such as those published in the journal *Science of Computer Programming*. It is important to note that almost all of these formal derivations, including the one in this thesis, have been done *after the fact*. The preliminary state of our knowledge, and the exigencies of programming in the real world, do not usually allow the luxury of using formal derivation for writing new and innovative programs. Perhaps this situation will change as we progress in our experience with formal methods. Then we will be able to bring the clarity and precision of our best presentations of programs to our development of new programs. I hope that this thesis has contributed towards this goal.

6.4 Conclusions

This thesis has analyzed the rule system matching problem, and has derived a simple, but efficient, implementation. The core of the development is a mathematical model of the information computed and manipulated in performing this task. The representations used in the implementation are directly derived from this mathematical model. The structures in this model are similar to the valuation structures used in the model–theoretic semantics of first–order logic.

My attempt to formalize this model has led me to introduce *disjunctive substitutions* to represent the information obtained from matching the patterns of a rule against several possible data in a database. The formal derivation of the matcher is based on a homomorphism from the matcher specification to a lattice formed from these disjunctive substitutions.

The initial implementation has been optimized based on distributive properties of the representation. Further optimization has been performed using partial evaluation. The resulting program has been shown to be isomorphic to a simplified Rete network.

This derivation can be summarized schematically as:

Rete = Formal Specification
+ Lattice Construction based on Homomorphism to Specification
+ Finite Differencing based on Distributive Laws
+ Partial Evaluation.

Both the initial derivation and the optimizations have been (partially) implemented using program transformations in the Refine wide–spectrum language and the Kestrel Interactive Development System.

The structures introduced in the above program derivation have been shown to correspond to structures used in developing the model–theoretic semantics of first order logic. This connection provides an explanation and verification of the algorithm, and provides directions for extending it into a more complete implementation.

Though the type of formal derivation and implementation described in this thesis can be extremely time–consuming, I feel that it has the potential for automating a significant portion of programming–in–the–small. Though it does not directly address the complexity management issues that dominate programming–in–the–large, perhaps after rationalizing the development of small software components, we will be in a better position to address the large–scale issues.

Appendix A

Mathematical Definitions

A.1 Lattice Theory

This section presents some standard mathematical definitions for lattices and related structures. Some of these definitions are used in the derivation in Chapter 3. Discussion of this material can be found in many algebra, universal algebra, and model theory texts, e.g. [3] [6] [8].

A.1.1 Sets, Relations, Posets

Given a set A, the *power set* of A, denoted by 2^A, is the set of all subsets of A (including the empty set, and A itself).

The *Cartesian Product* of a finite sequence of sets $A_1, \ldots, A_n$, denoted by $A_1 \times \ldots \times A_n$, is the collection of all n-tuples $\langle a_1, \ldots, a_n \rangle$ with $a_1 \in A_1, \ldots, a_n \in A_n$. If each of the A_i is identical with a fixed set A, we write $A^n = A_1, \ldots, A_n$.

An *n-ary relation* on a set A is a subset of A^n.

A *partial function* from S to T is a binary relation R such that $\langle x, y \rangle \in R$ and $\langle x, z \rangle \in R$ implies $y = z$.

The domain of a function f, denoted by $dom(f)$, is $\{x \mid (\exists y \in T)\langle x, y \rangle \in R\}$.

A *function* from S to T is a partial function with domain S.

An *n-ary operation* on a set S is a function $f : S \times \ldots \times S \rightarrow S$.

An *algebra* is a finite collection of n-ary relations and n-ary operations.

A *poset (partially ordered set)* is a set A with a binary relation $\leq$ such that

$$(\forall x)\ x \leq x$$

$$(\forall x, y)\ (x \leq y \& y \leq x) \Rightarrow x = y$$

$$(\forall x, y, z)\ (x \leq y \& y \leq z) \Rightarrow x \leq z.$$

These equations state that the relation is reflexive, antisymmetric, and transitive.

A.1.2 Semilattices and Lattices

A *semilattice* is a poset $\langle A, \leq \rangle$ with a binary operation $\wedge$ (greatest lower bound) on the set A such that

$$(\forall x, y)\ x \wedge y \leq x$$

$$(\forall x, y)\ x \wedge y \leq y$$

$$(\forall x, y, z)\ (z \leq x)\ \&\ (z \leq y) \Rightarrow z \leq x \wedge y.$$

The first two equations state that $x \wedge y$ is a lower bound. The third equation states that $x \wedge y$ is the greatest lower bound.

A *lattice* is a poset $\langle A, \leq \rangle$ with two binary operations $\wedge$ (greatest lower bound) and $\vee$ (least upper bound) on the set A such that

$$(\forall x, y)\ x \wedge y \leq x$$

$$(\forall x, y)\ x \wedge y \leq y$$

$$(\forall x, y, z)\ (z \leq x)\ \&\ (z \leq y) \Rightarrow z \leq x \wedge y$$

$$(\forall x, y)\ x \vee y \geq x$$

$$(\forall x, y)\ x \vee y \geq y$$

$$(\forall x, y, z)\ (z \geq x)\ \&\ (z \geq y) \Rightarrow z \geq x \vee y.$$

The first three equations state that $x \wedge y$ is the greatest lower bound. The fourth and fifth equations state that $x \vee y$ is an upper bound. The sixth equation states that $x \vee y$ is the least upper bound.

A.1.3 Distributive Lattices and Boolean Algebras

A *distributive lattice* is a lattice in which

$$(\forall x, y, z) x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z).$$

This equation states that $\wedge$ distributes over $\vee$. It can be shown that $\wedge$ distributes over $\vee$ iff $\vee$ distributes over $\wedge$.

An element x in a lattice L is a *least element* if

$$(\forall y \in L) x \leq y;$$

it is a *greatest element* if

$$(\forall y \in L) y \leq x.$$

Let the least element in a lattice be denoted by $\perp$, and the greatest element by $\top$ (if they exist).

A *complemented lattice* is a lattice which has a least element and a greatest element, and in which

$$(\forall x \in L)(\exists y \in L)(x \wedge y = 1\ \&\ x \vee y = \perp).$$

A *Boolean algebra* is a complemented, distributive lattice.

Any powerset, such as 2^A, with the subset ordering $\subseteq$, forms a *power set algebra*, which is a Boolean algebra. In the power set algebra 2^A:

$$(\forall\, x, y \in 2^A)\; x \wedge y = x \cap y,$$

$$(\forall\, x, y \in 2^A)\; x \vee y = x \cup y,$$

$$\bot = \emptyset,$$

$$\top = A.$$

A.1.4 Filters and Ideals

A subset S of a lattice L is a *filter* if the following conditions hold:

$$(\forall x, y \in S) x \wedge y \in S$$

$$(\forall x \in S)(\forall y \in L) x \leq y \Rightarrow y \in S$$

$$\bot \notin S$$

$$S \neq \emptyset.$$

A subset S of a lattice L is an *ideal* if the following conditions hold:

$$(\forall x, y \in S) x \vee y \in S$$

$$(\forall x \in S)(\forall y \in L) y \leq x \Rightarrow y \in S$$

$$\top \notin S.$$

Let L be a lattice or semi-lattice, and let x be an element in L. The *principal ideal* in L *generated by* x is the subset

$$\{y \mid y \in L \wedge y \leq x\}.$$

A.2 Model–Theoretic Semantics

This appendix presents a very brief description of the basic semantic definition used in model–theory. Discussion of this material can be found in most texts on logic and model theory, e.g. [3, Ch. 2].

Consider a first order language L consisting of variable symbols, function symbols (consider constants to be 0-ary functions), predicate symbols, and quantifiers.[1] Consider a model for this language, called a *structure*, consisting of a Universe U, and a mapping from each n-ary function symbol in L to an n-ary operation on U, and from each n-ary predicate symbol in L to an n-ary relation on U.

A *valuation* σ is a structure together with an assignment of a value $x^\sigma \in U$ to each variable x. Given a valuation σ, a value can be assigned to any term in L using the following rules:

$$1.(f(t_1,\ldots,t_n))^\sigma = f^\sigma(t_1^\sigma,\ldots,t_n^\sigma).$$

$$2.(P(t_1,\ldots,t_n))^\sigma = \begin{cases} \top & \text{if } \langle t_1^\sigma,\ldots,t_n^\sigma\rangle \in P^\sigma \\ \bot & \text{otherwise} \end{cases}$$

$$3.(\alpha \wedge \beta)^\sigma = \begin{cases} \top & \text{if } \alpha^\sigma = \top \text{ and } \beta^\sigma = \top \\ \bot & \text{otherwise} \end{cases}$$

$$4.(\alpha \vee \beta)^\sigma = \begin{cases} \top & \text{if } \alpha^\sigma = \top \text{ or } \beta^\sigma = \top \\ \bot & \text{otherwise} \end{cases}$$

$$5.(\neg\alpha)^\sigma = \begin{cases} \top & \text{if } \alpha^\sigma = \bot \\ \bot & \text{otherwise} \end{cases}$$

$$6.(\alpha \rightarrow \beta)^\sigma = \begin{cases} \top & \text{if } \alpha^\sigma = \bot \text{ or } \beta^\sigma = \top \\ \bot & \text{otherwise} \end{cases}$$

This definition is known as Tarski's truth definition, and forms the basis of the model–theoretic semantics of first–order logic.

[1] For the purposes of this thesis, we will only consider quantifier-free formula.

Bibliography

[1] John Backus. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Communications of the ACM*, 21(8):613–641, 1978.

[2] David R. Barstow. *Knowledge-Based Program Construction.* North–Holland, 1979.

[3] Daniel Bell and Moshe Machover. *A Course in Mathematical Logic.* North Holland, 1977.

[4] Richard S. Bird. An Introduction to the Theory of Lists. In Manfred Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 5–42. Springer–Verlag, NATO Advanced Science Institutes Series, Series F, Number 36, 1987.

[5] Lee Brownston, Robert Farrell, Elaine Kant, and Nancy Martin. *Programming Expert Systems in OPS5.* Addison-Wesley, 1985.

[6] Stanley Burris and H. P. Sankappanavar. *A Course in Universal Algebra.* Springer–Verlag, Graduate Texts in Mathematics, 1981.

[7] Donald Cohen. Automatic Compilation of Logical Specifications into Efficient Programs. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 20–25, 1986.

[8] Haskell Curry. *Foundations of Mathematical Logic.* Dover, 1977. (Corrected reprint of the 1963 McGraw-Hill edition).

[9] Johan de Kleer, Jon Doyle, Charles Rich, Guy L. Steele Jr., and Gerald Jay Sussman. AMORD: A Deductive Procedure System. Memo 435, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, January 1978.

[10] Elmar Eder. Properties of Substitutions and Unifications. *Journal of Symbolic Computation*, 1:31–46, 1985.

[11] Charles Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, 19:17–37, 1982.

[12] Charles L. Forgy. *On the Efficient Implementation of Production Systems.* PhD thesis, Carnegie–Mellon University, 1979.

[13] Special Section on Program Transformations. IEEE Transactions on Software Engineering, SE-7 (1), January 1981.

[14] Special Issue on Artificial Intelligence and Software Engineering. IEEE Transactions on Software Engineering, SE-11 (11), November 1985.

[15] K. Kahn. A Partial Evaluator of Lisp Written in Prolog. Technical Report 17, Uppsala University, The Programming Methodology and Artificial Intelligence Laboratory, Uppsala, Sweden, February 1983.

[16] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* MIT Press, 1986.

[17] Zohar Manna and Richard Waldinger. Deductive Synthesis of the Unification Algorithm. *Science of Computer Programming*, 1:5–48, 1981.

[18] L. G. L. T. Meertens. An Abstracto Reader prepared for IFIP WG 2.1. Note CS-N8702, Centrum voor Wiskunde en Informatica (Centre for Mathematics and Computer Science), Amsterdam, April 1987.

[19] Robert Paige and Shaye Koenig. Finite Differencing of Computable Expression. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, July 1982.

[20] G. Plotkin. Building in Equational Theories. *Machine Intelligence*, 7:73–90, 1972.

[21] Reasoning Systems, Inc., Palo Alto, CA. *Refine User's Manual*, 1985.

[22] Raymond Reiter. On Closed World Data Bases. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum, 1978.

[23] Charles Rich. Inspection Methods in Programming. Technical Report 604, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, June 1981.

[24] Charles Rich and Richard C. Waters, editors. *Readings in Artificial Intelligence and Software Engineering.* Morgan Kaufmann, 1986.

[25] Charles Rich and Richard C. Waters. The Programmer's Apprentice: A Research Overview. *Computer*, 21(11):10–25, November 1988.

[26] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[27] J. A. Robinson. Notes on Logic Programming. In Manfred Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 109–144. Springer–Verlag, NATO Advanced Science Institutes Series, Series F, Number 36, 1987.

[28] D. E. Rydeheard and R. M. Burstall. A Categorical Unification Algorithm. In *Category Theory and Computer Programming*, pages 493–505. Springer–Verlag, Lecture Notes in Computer Science, Volume 240, 1985.

[29] David E. Smith and Michael R. Genesereth. Ordering Conjunctive Queries. *Artificial Intelligence*, 26(2):171–215, 1985.

[30] Douglas R. Smith. Top–Down Synthesis of Divide–and–Conquer Algorithms. *Artificial Intelligence*, 27(1):43–96, 1985.

[31] Douglas R. Smith. KIDS — A Knowledge–Based Software Development System. Technical Report KES.U.88.7, Kestrel Institute, Palo Alto, CA, October 1988.

[32] Douglas R. Smith. The Structure and Design of Global Search Algorithms. Technical Report KES.U.87.12, Kestrel Institute, Palo Alto, CA, July 1988.

[33] Douglas R. Smith and Thomas T. Pressburger. Knowledge–Based Software Development Tools. In P. Brereton, editor, *Software Engineering Environments*, pages 79–103. Ellis Horwood Ltd., Chichester, 1988.

[34] M. Stickel. A Unification Algorithm for Associative Commutative Functions. *Journal of the ACM*, 28:423–434, 1981.

[35] Stephen Westfold. *Logic Specifications for Compiling.* PhD thesis, Stanford University, 1984.