

# Dyadic Logic: Austere, Intensional, Predicate Calculus

M. S. Lanus

E. A. Ashcroft

Department of Computer Science and Engineering

Declarative Language / Advanced Architecture Lab

Arizona State University

Tempe, AZ 85287-5406

February 5, 1992

## Abstract

An austere syntax for the predicate calculus is presented consisting of a single operator, the list constructor []. The semantics of this logic is intensional because the meaning of a formula depends on its context. Several equivalence and deductive rules are presented and some interesting properties of the logic are explored.

## 1 Introduction

We have been investigating deductive systems based on negation normal form hoping that they can be made less susceptible to the combinatorial explosion than systems based on conjunctive or disjunctive normal form.

Our first experiments used binary operators for conjunction and disjunction. However, since binary operators do not explicitly support the associative property of conjunction and disjunction, our simplification rules were sensitive to the order of conjuncts and disjuncts within a formula. As an example the rules would simplify

$$(P \wedge Q) \vee (P \wedge \neg Q) \vee (\neg P \wedge Q) \vee (\neg P \wedge \neg Q)$$

to *true*, but would not simplify

$$(P \wedge Q) \vee (\neg P \wedge \neg Q) \vee (P \wedge \neg Q) \vee (\neg P \wedge Q),$$

because they could not find common subexpressions in non-adjacent disjuncts.

To solve this problem we looked into set operators for conjunction and disjunction. Our starting point was the *set of sets* notation for clausal form used in resolution theorem provers. The main difference between clausal form and negation normal form is that clausal form is a *two-level* logic whereas negation normal form is a *multi-level* logic, and so our notation must involve a *set of sets of sets of ...* notation where the levels alternate between conjunction and disjunction.

Negation normal form (nnf) has the interesting property that formulas can be conjoined or disjoined in constant space and time, merely by placing a  $\wedge$  or  $\vee$  operator between them, as nnf does not dictate the distribution of  $\wedge$  over  $\vee$  or vice versa. However, negating an nnf formula requires time and space linear in the size of the formula, as  $\wedge$  operators must be changed to  $\vee$ ,  $\vee$  to  $\wedge$ , and literal occurrences must be complemented. We wanted to find a notation that allows *all* basic operations to be implemented in constant space and time.

Past experience in implementing deductive systems showed that an austere notation has many benefits such as: fewer cases for the humble programmer to consider, fewer rules for the machine to execute, and a smaller, simpler system to optimize.

Another design goal was to find a notation that closely maps to built-in data types, because operations supporting built-in data types are usually efficient. We decided to list the preceeding requirements and search for a notation that satisfies them:

- support multi-level logic
- support set operators for  $\wedge$  and  $\vee$
- implement  $\wedge$ ,  $\vee$ , and  $\neg$  operations in constant space and time
- employ an austere notation
- map to built-in data types

We succeeded in finding a notation that satisfies the requirements and were surprised to discover that it had other interesting properties such as an intensional semantics and an innate self-duality.

## 2 Solution Scheme

We wanted to see how much of the predicate calculus we could encode using a single operator, the list constructor `[]`. This approach seemed promising because: lists can support multi-level logic as they may be nested to any number of levels, lists can support set operators, lists are built-in data types in most very high level languages such as LISP, Prolog, Scheme, and Haskell, and any notation that uses one operator certainly qualifies as austere.

The basic scheme is to treat the list constructor operator as a *\*junction* operator, where *\** is either *con* or *dis* depending upon its level of nesting or *context*. Since there are only two cases to distinguish we only need to distinguish between even and odd levels of nesting. A list constructor at one level represents the conjunction of its elements, at the other level the disjunction of its elements. Thus the list constructor has a context sensitive or *intensional* semantics [5].

This generalization of the set of sets notation used in clausal form systems is similar to notation used in the *connection graph* method of Bibel [8, 7, 6], the *generalized matings* technique of Andrews [3, 1, 2], and the *dissolution* method of Murray and Rosenthal [11, 12].

Choosing a representation for negation was more difficult. The standard approach uses the unary operator  $\neg$ . In [7] Bibel uses two unary operators, a superscript 0 and a superscript 1, to indicate the positive and negative polarity of literals. We rejected these schemes because they complicate the syntax and introduce *extensional* semantics into an otherwise purely intensional system. Then we found a scheme that solves both problems: let the context indicate whether



a proposition symbol denotes a positive or negative literal. A proposition symbol at one level represents a positive literal, a proposition symbol at the next level indicates a negative literal, and so on.

## 2.1 Yin Yang

Working with this intensional logic proved confusing since the meaning of a formula depends on its context. After much thrashing about we realized that we needed a simple rule to map from contexts to meanings. The continuous alternation of conjunction and disjunction, negative and positive, brought to mind the ancient Chinese philosophy of a universe bound together by the dynamic interplay of two opposite yet complementary forces, Yin and Yang. Yang is the positive, active, contractive principle; Yin the negative, passive, expansive one.

It is striking how well logic fits this metaphor. A positive literal corresponds to the positive Yang principle; a negative literal to Yin. Conjunction  $P \wedge Q$  is contractive as the set of interpretations that satisfies  $P \wedge Q$  is the intersection of the set of interpretations that satisfies  $P$  with the set of interpretations that satisfies  $Q$ . Therefore, conjunction corresponds to the contractive Yang principle and a similar argument shows that disjunction corresponds to Yin. Furthermore, since universal quantification  $\forall x p(x)$  is equivalent to the conjunction of  $p(d)$  for all domain elements  $d$ , the universal quantifier is Yang and the existential quantifier Yin.. Table 1 summarizes this metaphor and introduces the context symbols  $\Delta$  and  $\nabla$ .

Table 1: Logic as Yin and Yang

Yin ( $\nabla$ )	Yang ( $\Delta$ )
Negative	Positive
Expansive	Contractive
Disjunctive	Conjunctive
Existential	Universal

In [13] Rucker discusses the number-space or discrete-continuous distinction. He says *Together the pair make up what the Greeks called a "dyad," or pair of opposing concepts*. He cites other examples of dyads such as: on-off, 0-1, one-many, cold-hot, and light-dark. Therefore, it seemed appropriate to name this intensional logic, based on a pair of contexts with opposite yet complementary meanings, the *dyadic logic*.

One aspect of dyadic logic that is confusing is the context of a list. A list has an external context which defines the relationship between the list and its peers and an internal context which defines the relationship between the members of the list. Thus a list encountered in a  $\Delta$  context is conjoined to its peers ( $\Delta$  external context) and defines the disjunction of its members ( $\nabla$  internal context).

Having defined the meanings of the two contexts, the only decision that remained was to specify the outermost context of a formula. We wanted to make the outermost context  $\nabla$ , so that the members of the outermost list would be conjoined, and dyadic logic would resemble clausal form. However, if the outermost context is  $\nabla$ , then an atomic formula at the outermost level would denote a *negative* literal, which is counter intuitive. Also, the functions to translate

between classical and dyadic logic are more complicated when the outermost context is  $\nabla$  than when it is  $\Delta$ . Therefore, we chose to define the outermost context as  $\Delta$ .

## 2.2 Serendipity

Upon investigating dyadic logic we discovered that it possessed interesting properties that we never intended giving it. One such property is that an equivalence rule may fire in either context, but its meaning in one context is the dual of its meaning in the other. That is, the same syntactic rule means  $\mathcal{F} \vee \text{true} \equiv \text{true}$  in a  $\Delta$  context, and  $\mathcal{F} \wedge \text{false} \equiv \text{false}$  in a  $\nabla$  context. Another property is that inference rules may fire in either context, but the direction of the firing depends on the context. That is, the same syntactic rule means  $\mathcal{F} \wedge \mathcal{G} \vdash \mathcal{F}$  when fired from left-to-right in a  $\Delta$  context, and  $\mathcal{F} \vdash \mathcal{F} \vee \mathcal{G}$  when fired from right-to-left in a  $\nabla$  context.

These surprising properties are instances of *serendipity* [5], which Webster's dictionary defines as *the faculty of discovering desirable or valuable things, accidentally or unexpectedly*.

To express these more powerful versions of the inference and equivalence relations we adopted

the notation  $\left\langle \frac{\mathcal{F}}{\mathcal{G}} \right\rangle$  and  $\langle \mathcal{F} \mid \mathcal{G} \rangle$  where:

$$\left\langle \frac{\mathcal{F}}{\mathcal{G}} \right\rangle \text{ means } \begin{cases} \mathcal{G} \vdash \mathcal{F} & \text{if the context is } \Delta, \\ \mathcal{F} \vdash \mathcal{G} & \text{if the context is } \nabla, \text{ and} \end{cases}$$

$$\langle \mathcal{F} \mid \mathcal{G} \rangle \text{ means } \left\langle \frac{\mathcal{F}}{\mathcal{G}} \right\rangle \text{ and } \left\langle \frac{\mathcal{G}}{\mathcal{F}} \right\rangle.$$

Note that if  $\mathcal{F} \vdash \mathcal{G}$  and  $\mathcal{G} \vdash \mathcal{F}$  then  $\mathcal{F} \equiv \mathcal{G}$ , so  $\langle \mathcal{F} \mid \mathcal{G} \rangle$  means  $\mathcal{F} \equiv \mathcal{G}$  in dyadic logic.

## 2.3 Intensionality

The following principles form the intensional foundation of dyadic logic:

- the meaning of a formula depends on its context
- the  $\Delta$  context denotes the Yang principle (positive literal, conjunctive, universal quantifier)
- the  $\nabla$  context denotes the Yin principle (negative literal, disjunctive, existential quantifier)
- equivalence rules may fire in either direction in either context
- inference rules may fire in either direction, depending on the context
- the order of members in a list does not matter



### 3 Propositional Dyadic Logic

#### 3.1 Syntax

The syntax of propositional dyadic logic is exceedingly simple

An *atomic formula* is an uppercase letter from the end of the alphabet  $P, Q, R, \dots$  with or without numeric subscripts and a *formula* is an atomic formula or a list of formulas.

Some examples of propositional dyadic logic formulas are  $P, Q, [P], [Q], [P, Q], [[P, Q]], [],$  and  $[[[]]$ .

#### 3.2 Semantics

In order to assign meaning to formulas we must first define an *interpretation*. An interpretation  $I$  is a mapping from atomic formulas to truth values from the domain  $Truth = \{\text{true}, \text{false}\}$ . The truth value that the interpretation  $I$  assigns to the atomic formula  $\mathcal{A}$  is denoted  $\mathcal{A}_I$ . An interpretation  $I$  is said to be an interpretation for a formula  $\mathcal{F}$  if it maps every atomic formula in  $\mathcal{F}$  to some truth value.

Figure 1 defines the semantic function  $V_I : Interp \rightarrow Dyadic \rightarrow Truth$  which maps from *Interp*, the domain of interpretations, and *Dyadic*, the domain of propositional dyadic logic formulas, to the domain of truth values. We use the convention that the metavariable  $\mathcal{A}$  represents an arbitrary *atomic* formula, and  $\mathcal{L}$  represents an arbitrary *non-atomic* formula or list. Table 2 summarizes the formulas in  $B_2$ , the set of boolean functions of two arguments.

$$\begin{aligned} V_I \mathcal{A} &= \mathcal{A}_I \\ V_I \mathcal{L} &= \begin{cases} \text{true} & \text{if } V_I \mathcal{F} = \text{false for some } \mathcal{F} \in \mathcal{L} \\ \text{false} & \text{if } V_I \mathcal{F} = \text{true for every } \mathcal{F} \in \mathcal{L} \end{cases} \end{aligned}$$

Figure 1: Semantic Function  $V_I : Interp \rightarrow Dyadic \rightarrow Truth$

#### 3.3 Translation Functions

This section defines functions to translate between propositional logic and propositional dyadic logic.

##### 3.3.1 $D : Prop \rightarrow Dyadic$

Figure 2 defines  $D$ , the translation function from propositional logic formulas to formulas in propositional dyadic logic.

##### 3.3.2 $C : Dyadic \rightarrow Prop$

Figure 3 defines  $C$ , the translation function from propositional dyadic logic formulas to formulas in classical propositional logic. The translation function  $C$  is defined in terms of two other

Table 2: Propositional Dyadic Logic Summary of  $B_2$

Propositional Logic	Dyadic Logic
$P$	$P$
$Q$	$Q$
$\neg P$	$[P]$
$\neg Q$	$[Q]$
$P \wedge Q$	$[[P, Q]]$
$P \wedge \neg Q$	$[[P, [Q]]]$
$\neg P \wedge Q$	$[[[P], Q]]$
$\neg P \wedge \neg Q$	$[[[P], [Q]]]$
$P \vee Q$	$[[P], [Q]]$
$P \vee \neg Q$	$[[P], [Q]]$
$\neg P \vee Q$	$[P, [Q]]$
$\neg P \vee \neg Q$	$[P, Q]$
$(P \wedge Q) \vee (\neg P \wedge \neg Q)$	$[[P, Q], [[P], [Q]]]$
$(P \wedge \neg Q) \vee (\neg P \wedge Q)$	$[[P, [Q]], [[P], Q]]$
$true$	$[[[]]]$
$false$	$[]$

$$\begin{aligned}
D \text{ true} &= [[[]]] \\
D \text{ false} &= [] \\
D P &= P \\
D \neg \mathcal{F} &= [D \mathcal{F}] \\
D \mathcal{F} \wedge \mathcal{G} &= [[D \mathcal{F}, D \mathcal{G}]] \\
D \mathcal{F} \vee \mathcal{G} &= [D \overline{\mathcal{F}}, D \overline{\mathcal{G}}] \\
D \mathcal{F} \rightarrow \mathcal{G} &= [D \mathcal{F}, D \overline{\mathcal{G}}] \\
D \mathcal{F} \equiv \mathcal{G} &= [[D \mathcal{F}, D \mathcal{G}], [D \overline{\mathcal{F}}, D \overline{\mathcal{G}}]]
\end{aligned}$$

$$\text{where } \overline{\mathcal{F}} = \begin{cases} \mathcal{G} & \text{if } \mathcal{F} = \neg \mathcal{G} \\ \neg \mathcal{F} & \text{otherwise} \end{cases}$$

Figure 2: Translation Function  $D : Prop \rightarrow Dyadic$

functions  $C^\Delta$  and  $C^\nabla$ . The notation  $C^\Delta$  is meant to suggest the application of  $C$  in a  $\Delta$  context, and  $C^\nabla$  the application of  $C$  in a  $\nabla$  context.

$$\begin{aligned} C\mathcal{F} &= C^\Delta\mathcal{F} \\ C^\Delta\mathcal{A} &= \mathcal{A} \\ C^\Delta\mathcal{L} &= \bigvee_{\mathcal{F} \in \mathcal{L}} C^\nabla\mathcal{F} \\ C^\nabla\mathcal{A} &= \neg\mathcal{A} \\ C^\nabla\mathcal{L} &= \bigwedge_{\mathcal{F} \in \mathcal{L}} C^\Delta\mathcal{F} \end{aligned}$$

Figure 3: Translation Function  $C : Dyadic \rightarrow Prop$

### 3.3.3 $C' : Dyadic \rightarrow Prop$

Nicholas Sterling of Sun Microsystems pointed out to us that the propositional dyadic logic has an *extensional* semantics, where the list constructor is interpreted as an *n-input nand gate*. The 2-input nand gate is known as *alternative denial* [10] or *Sheffer's Stroke* [9], so the list constructor for propositional dyadic logic is the set version of Sheffer's Stroke or *Sheffer's Set*.

Figure 4 defines  $C'$ , the translation function from propositional dyadic logic formulas to formulas in propositional logic based upon this extensional interpretation. Figure 5 defines the same function but so as to generate fewer  $\neg$  symbols in the resulting formula, by using the complementation operator defined in Figure 2.

$$\begin{aligned} C'\mathcal{A} &= \mathcal{A} \\ C'\mathcal{L} &= \neg \bigwedge_{\mathcal{F} \in \mathcal{L}} C'\mathcal{F} \end{aligned}$$

Figure 4: Translation Function  $C' : Dyadic \rightarrow Prop$

$$\begin{aligned} C'\mathcal{A} &= \mathcal{A} \\ C'\mathcal{L} &= \bigvee_{\mathcal{F} \in \mathcal{L}} \overline{C'\mathcal{F}} \end{aligned}$$

Figure 5: Translation Function  $C' : Dyadic \rightarrow Prop$

The extensional semantics for propositional dyadic logic is interesting because it implies that any results we generate can be used by hardware logic designers using n-input nand gates as a universal logic element. (As an illustration of the utility of the nand gate consider the fact that over 95% of the original Cray 1 was built from 2-input and 3-input nand gates.)

### 3.4 Basic Operations

Table 3 summarizes the implementation of  $\wedge$ ,  $\vee$ , and  $\neg$  in dyadic logic. All these operations can be implemented in constant space and time. This is an improvement over negation normal form and shows that we are dealing with a new form, derived from nnf but distinct from it.

Table 3: Basic Logic Operations in Dyadic Logic

Basic Operation	Dyadic Logic Implementation	List Operations
$\neg \mathcal{F}$	$[\mathcal{F}]$	1
$\mathcal{F} \wedge \mathcal{G}$	$[[\mathcal{F}, \mathcal{G}]]$	2
$\mathcal{F} \vee \mathcal{G}$	$[[\mathcal{F}], [\mathcal{G}]]$	3

### 3.5 Equivalence Rules

This section presents some equivalence rules in propositional dyadic logic. Each rule has different meanings in the two contexts.

**Rule 1 ( $\neg$ -Elimination)**  $\langle [[\mathcal{F}]] \mid \mathcal{F} \rangle$   
 $\langle [\mathcal{F}_1, \dots, [[\mathcal{G}_1, \dots, \mathcal{G}_m]], \dots, \mathcal{F}_n] \mid [\mathcal{F}_1, \dots, \mathcal{G}_1, \dots, \mathcal{G}_m, \dots, \mathcal{F}_n] \rangle$

This rule says that you can delete two immediate levels of brackets as long as the result is a formula. This is the dyadic logic version of the  $\neg$ -elimination rule  $\neg\neg\mathcal{F} \equiv \mathcal{F}$ .

**Rule 2 ( $\wedge/\vee$ -Elimination)**  $\langle [\mathcal{F}_1, \dots, \mathcal{G}, \dots, \mathcal{G}, \dots, \mathcal{F}_n] \mid [\mathcal{F}_1, \dots, \mathcal{G}, \dots, \mathcal{F}_n] \rangle$

This rule says that any redundant copies of a formula may be removed from a list. In a  $\Delta$  context it means  $\mathcal{F} \vee \mathcal{F} \equiv \mathcal{F}$ ; in a  $\nabla$  context it means  $\mathcal{F} \wedge \mathcal{F} \equiv \mathcal{F}$ .

**Rule 3 (Complement)**  $\langle [\dots, \mathcal{F}, \dots, [\mathcal{F}], \dots] \mid [[]] \rangle$

This rule says that any list containing a formula and its complement is equivalent to the singleton list containing the empty list. In a  $\Delta$  context it means  $\mathcal{F} \vee \neg\mathcal{F} \equiv \text{true}$ ; in a  $\nabla$  context it means  $\mathcal{F} \wedge \neg\mathcal{F} \equiv \text{false}$ .

**Rule 4 (GLB / LUB I)**  $\langle [\dots, [], \dots] \mid [[]] \rangle$

This rule says that any list containing the empty list is equivalent to the list containing only the empty list. In a  $\Delta$  context it means  $\mathcal{F} \vee \text{true} \equiv \text{true}$ ; in a  $\nabla$  context it means  $\mathcal{F} \wedge \text{false} \equiv \text{false}$ .

**Rule 5 (GLB / LUB II)**  $\langle [\mathcal{F}_1, \dots, [[]], \dots, \mathcal{F}_n] \mid [\mathcal{F}_1, \dots, \mathcal{F}_n] \rangle$

This rule says that you may always remove the singleton list containing the empty list from any list. In a  $\Delta$  context it means  $\mathcal{F} \vee \text{false} \equiv \mathcal{F}$ ; in a  $\nabla$  context it means  $\mathcal{F} \wedge \text{true} \equiv \mathcal{F}$ .



### 3.6 Inference Rules

$$\text{Rule 6 } (\wedge\text{-Elimination}/\vee\text{-Introduction}) \left\langle \frac{[\mathcal{F}_1, \dots, \mathcal{G}, \dots, \mathcal{F}_n]}{[\mathcal{F}_1, \dots, \mathcal{F}_n]} \right\rangle$$

This rule says that, depending on the context, we may remove formulas from a list or add arbitrary formulas to a list. In a  $\Delta$  context it means  $\mathcal{F} \vdash \mathcal{F} \vee \mathcal{G}$ ; in a  $\nabla$  context it means  $\mathcal{F} \wedge \mathcal{G} \vdash \mathcal{F}$ .

### 3.7 Examples

To gain insight into dyadic logic we need to study a few examples. Let's see what DeMorgan's laws look like in dyadic logic. DeMorgan's first law is  $\neg(\mathcal{F} \wedge \mathcal{G}) \equiv \neg\mathcal{F} \vee \neg\mathcal{G}$ ; in dyadic logic this becomes  $\langle [[[\mathcal{F}, \mathcal{G}]]] \mid [\neg\mathcal{F}, \neg\mathcal{G}] \rangle$ . But this is the law of  $\neg$ -elimination. Thus in dyadic logic the  $\neg$ -elimination rule subsumes DeMorgan's first law.

DeMorgan's second law  $\neg(\mathcal{F} \vee \mathcal{G}) \equiv \neg\mathcal{F} \wedge \neg\mathcal{G}$  is  $\langle [[[\neg\mathcal{F}], [\neg\mathcal{G}]]] \mid [[[\mathcal{F}], [\mathcal{G}]]] \rangle$  in dyadic logic. But this is no law at all; it is pure syntactic identity. As a guide for the perplexed let's look at this example in detail.

Since negation is implemented by placing an expression in a list we translate  $\neg(\mathcal{F} \vee \mathcal{G})$  by enclosing the translation of  $(\mathcal{F} \vee \mathcal{G})$  in a list. The translation of  $(\mathcal{F} \vee \mathcal{G})$  is  $[[\mathcal{F}], [\mathcal{G}]]$ . Composing the results yields  $[[[\mathcal{F}], [\mathcal{G}]]]$ .

To translate  $\neg\mathcal{F} \wedge \neg\mathcal{G}$  we see that the principal operation is conjunction which is represented in dyadic logic by placing the conjuncts in a list two levels deep. The translation of  $\neg\mathcal{F}$  is  $[\neg\mathcal{F}]$  and the translation of  $\neg\mathcal{G}$  is  $[\neg\mathcal{G}]$ . Composing the results yields  $[[[\neg\mathcal{F}], [\neg\mathcal{G}]]]$ .

## 4 Predicate Dyadic Logic

The next step in developing dyadic logic was to move from propositional logic to predicate logic by adding functions, predicates, constants, variables, and quantifiers. Functions, predicates, constants, and variables were easy to represent, but it was not so obvious how to represent quantifiers.

We considered using the standard technique of interpreting free variables as universally quantified and replacing existentially quantified variables with skolem functions, but this introduces extensionality into an otherwise intensional system. We also considered using two quantifier symbols  $Q_0$  and  $Q_1$  whose meanings would be relatively dual (one means  $\exists$  when the other means  $\forall$ ) but whose absolute meaning would depend on the context. (ex. In a  $\Delta$  context  $Q_0$  would mean  $\forall$  and  $Q_1$  would mean  $\exists$ ; In a  $\nabla$  context the reverse would be true). However, this solution detracts from the austerity of the notation.

Since representing quantifiers was difficult we decided to eliminate them from the syntax. This is accomplished by allowing variables to appear as members of lists and letting their meaning depend on the context. A variable introduced in a  $\Delta$  context is universally quantified and one introduced in a  $\nabla$  context is existentially quantified.

As an example consider the formula  $[x, [y, p(x, y)]]$ . The variable  $x$  is introduced in a  $\nabla$  context, and so it is existentially quantified; the variable  $y$  is introduced in a  $\Delta$  context, so it is

universally quantified. The context of  $p(x, y)$  is  $\Delta$  so it denotes a positive literal. Therefore, this formula means  $\exists x \forall y p(x, y)$ .

We found this scheme interesting not only because it is intensional but also because the only notation required to represent the quantification of a variable is the variable itself. Also, free variables may be represented as variables that occur in predicates but are not introduced in any encompassing list. This would not be possible using the skolemization technique which interprets all free variables as being universally quantified.

## 4.1 Syntax

The terms and atomic formulas in predicate dyadic logic are identical to those in predicate logic, and the formulas are defined as follows:

A *formula* is an atomic formula or a list of variables and formulas.

Table 4 shows some predicate logic formulas expressed in dyadic logic.

Table 4: Examples of Predicate Dyadic Logic Formulas

Predicate Logic	Dyadic Logic
$\forall x p(x)$	$[[x, p(x)]]$
$\forall x \neg p(x)$	$[[x, [p(x)]]]$
$\exists x p(x)$	$[x, [p(x)]]$
$\exists x \neg p(x)$	$[x, p(x)]$

## 4.2 Semantics

In predicate logic an interpretation  $I$  is a domain of individuals  $D = \{d_1, d_2, \dots\}$  along with a mapping from constant symbols  $a, b, c, a_1, b_1, c_1, \dots$  to elements of the domain  $D$ , denoted by  $a_I$ , from  $n$ -ary predicate symbols  $p, q, r, p_1, q_1, r_1, \dots$  to  $n$ -place relations in  $D$ , denoted by  $p_I$ , and from  $n$ -ary function symbols  $f, g, h, f_1, g_1, h_1, \dots$  to operations over  $D^n \rightarrow D$ , denoted by  $f_I$ . An assignment  $\phi$  is a mapping from variables  $x$  to elements of  $D$ , denoted by  $\phi(x)$ . An assignment  $\phi$  is an assignment for an expression  $\mathcal{E}$  if it maps every free variable occurring in  $\mathcal{E}$  to some domain element.

Predicate dyadic logic syntax allows variables and formulas to appear in any order within a list, which makes it difficult to develop elegant translation functions. We need some method of separating the variables from the formulas within a list. Figure 6 defines two functions  $P$  and  $M$  to accomplish this.  $P\mathcal{L}$  extracts the variables from the list  $\mathcal{L}$  and thus produces the list comprising the *relative prefix* of  $\mathcal{L}$  and  $M\mathcal{L}$  extracts the formulas from  $\mathcal{L}$  and thus generates the list containing the *relative matrix* of  $\mathcal{L}$ . We call the value of  $P\mathcal{L}$  the *relative prefix* because it only extracts the variables from the outermost list, and  $M\mathcal{L}$  the *relative matrix* because there may be variables introduced in internal lists of  $M\mathcal{F}$ .

The definition of the polymorphic semantic function  $V_I^\phi$  follows Andrew's definition in [4]. Given an expression  $\mathcal{E}$ , an interpretation  $I$  for  $\mathcal{E}$ , and an assignment  $\phi$  for the free variables in  $\mathcal{E}$ , we define  $V_I^\phi \mathcal{E}$ , the value of the expression  $\mathcal{E}$  with respect to  $\phi$  in  $I$  as shown in Figure 7.



$$\begin{aligned} P\mathcal{L} &= \{x \mid x \in \mathcal{L} \wedge x \text{ is a variable}\} \\ M\mathcal{L} &= \mathcal{L} - P\mathcal{L} \end{aligned}$$

Figure 6: Definition of  $P\mathcal{L}$  and  $M\mathcal{L}$

$$\begin{aligned} V_I^\phi x &= \phi(x) \\ V_I^\phi a &= a_I \\ V_I^\phi f(t_1, \dots, t_n) &= f_I(V_I^\phi t_1, \dots, V_I^\phi t_n) \\ V_I^\phi p(t_1, \dots, t_n) &= p_I(V_I^\phi t_1, \dots, V_I^\phi t_n) \\ V_I^\phi \mathcal{L} &= \begin{cases} \text{true} & \text{if } V_I^{\phi'} \mathcal{F} = \text{false for some } \mathcal{F} \in M\mathcal{L} \\ & \text{and some } \phi' \text{ that differs from } \phi \\ & \text{by at most the variables in } P\mathcal{L} \\ \text{false} & \text{if } V_I^{\phi'} \mathcal{F} = \text{true for every } \mathcal{F} \in M\mathcal{L} \\ & \text{and every } \phi' \text{ that differs from } \phi \\ & \text{by at most the variables in } P\mathcal{L} \end{cases} \end{aligned}$$

Figure 7: Semantic Function  $V_I^\phi : \text{Assign} \rightarrow \text{Interp} \rightarrow \text{Dyadic} \rightarrow \text{Truth}$

### 4.3 Translation Functions

#### 4.3.1 $D : \text{Pred} \rightarrow \text{Dyadic}$

The translation function  $D$  from predicate logic formulas to formulas in predicate dyadic logic is identical to the translation function defined in Figure 2, with the addition of the rules defined in Figure 8.

$$\begin{aligned} D \forall x \mathcal{F} &= [[x, D \mathcal{F}]] \\ D \exists x \mathcal{F} &= [x, D \overline{\mathcal{F}}] \end{aligned}$$

Figure 8: Translation Function  $D : \text{Pred} \rightarrow \text{Dyadic}$

#### 4.3.2 $C : \text{Dyadic} \rightarrow \text{Pred}$

Figure 9 defines  $C$ , the translation function from predicate dyadic logic formulas to formulas in predicate logic.

#### 4.3.3 $C' : \text{Dyadic} \rightarrow \text{Pred}$

Figure 10 defines  $C'$ , the translation function from predicate dyadic logic formulas to formulas in predicate logic based upon the Sheffer's Set extensional interpretation.



$$\begin{aligned}
C\mathcal{F} &= C^\Delta \mathcal{F} \\
C^\Delta \mathcal{A} &= \mathcal{A} \\
C^\Delta \mathcal{L} &= \bigvee_{x \in P\mathcal{L}} x \bigvee_{\mathcal{F} \in M\mathcal{L}} C^\nabla \mathcal{F} \\
C^\nabla \mathcal{A} &= \neg \mathcal{A} \\
C^\nabla \mathcal{L} &= \bigwedge_{x \in P\mathcal{L}} x \bigwedge_{\mathcal{F} \in M\mathcal{L}} C^\Delta \mathcal{F}
\end{aligned}$$

Figure 9: Translation Function  $C : Dyadic \rightarrow Pred$

$$\begin{aligned}
C' \mathcal{A} &= \mathcal{A} \\
C' \mathcal{L} &= \bigvee_{x \in P\mathcal{L}} x \bigvee_{\mathcal{F} \in M\mathcal{L}} \overline{C' \mathcal{F}}
\end{aligned}$$

Figure 10: Translation Function  $C' : Dyadic \rightarrow Pred$

#### 4.4 Equivalence Rules

##### Rule 7 ( $\neg$ -Elimination)

$$\left\langle [\mathcal{F}_1, \dots, [[\mathcal{G}_1, \dots, \mathcal{G}_m], \dots, \mathcal{F}_n] \mid [\mathcal{F}_1, \dots, \mathcal{G}_1, \dots, \mathcal{G}_m, \dots, \mathcal{F}_n] \right\rangle$$

*if no variable in  $P[\mathcal{G}_1, \dots, \mathcal{G}_m]$  is free in  $M[\mathcal{F}_1, \dots, \mathcal{F}_n]$*

This rule says that any formula enclosed in two levels of nesting is equivalent to itself lifted up two levels. This is the dyadic logic version of the  $\neg$ -elimination rule  $\neg\neg\mathcal{F} \equiv \mathcal{F}$ .

##### Rule 8 ( $\forall/\exists$ -Elimination)

$$\left\langle [\mathcal{F}_1, \dots, x, \dots, \mathcal{F}_n] \mid [\mathcal{F}_1, \dots, \mathcal{F}_n] \right\rangle$$

*if  $x$  is not free in  $M[\mathcal{F}_1, \dots, \mathcal{F}_n]$*

This rule says that you may remove certain variables from a list or add certain variables to a list. In a  $\Delta$  context this rule means  $\exists x \mathcal{F} \equiv \mathcal{F}$ , *if  $x$  is not free in  $\mathcal{F}$* ; in a  $\nabla$  context it means  $\forall x \mathcal{F} \equiv \mathcal{F}$ , *if  $x$  is not free in  $\mathcal{F}$* .

#### 4.5 Inference Rules

##### Rule 9 ( $\forall$ -Elimination/ $\exists$ -Introduction Rule)

$$\left\langle \frac{[\mathcal{F}_1, \dots, x, \dots, \mathcal{F}_n]}{[\mathcal{F}_1, \dots, \mathcal{F}_n]\{x \leftarrow t\}} \right\rangle$$

*if  $t$  is free for  $x$  in  $M[\mathcal{F}_1, \dots, \mathcal{F}_n]$*

This rule says that, depending on the context, we may replace quantified variables with certain terms or terms with certain quantified variables. In a  $\Delta$  context it means  $\mathcal{F}\{x \leftarrow t\} \vdash \exists x \mathcal{F}$ ; if  $t$  is free for  $x$  in  $\mathcal{F}$ . In a  $\nabla$  context it means  $\forall x \mathcal{F} \vdash \mathcal{F}\{x \leftarrow t\}$ , if  $t$  is free for  $x$  in  $\mathcal{F}$ .

## 4.6 Examples

In predicate logic the formulas  $\forall x \exists y p(x, y)$  and  $\exists y \forall x p(x, y)$  are not equivalent. (If the domain of interpretation is the natural numbers and the meaning of  $p(x, y)$  is  $x < y$  the first formula is valid and the second is unsatisfiable.) We were concerned about the correctness of our implementation of quantified variables, and so we translated some predicate logic formulas into dyadic logic to convince ourselves that the notation was sound. These formulas are summarized in Table 5.

Table 5: More Examples of Formulas in Predicate Dyadic Logic

Predicate Logic	Dyadic Logic
$\forall x \exists y p(x, y)$	$[[x, [y, [p(x, y)]]]]$
$\exists y \forall x p(x, y)$	$[y, [x, p(x, y)]]$
$\neg \forall x \exists y p(x, y)$	$[[[x, [y, [p(x, y)]]]]]$
$\exists x \forall y \neg p(x, y)$	$[x, [y, [p(x, y)]]]$
$\neg \exists x \forall y p(x, y)$	$[[x, [y, p(x, y)]]]$
$\forall x \exists y \neg p(x, y)$	$[[x, [y, p(x, y)]]]$
$\forall x \forall y p(x, y)$	$[[x, y, p(x, y)]]$
$\forall x \neg \exists y \neg p(x, y)$	$[[x, [[y, p(x, y)]]]]$

The first two examples show that the dyadic logic notation is indeed robust enough to differentiate between  $\forall x \exists y p(x, y)$  and  $\exists y \forall x p(x, y)$ . The next two examples show that the semantic equivalence of  $\neg \forall x \exists y p(x, y)$  and  $\exists x \forall y \neg p(x, y)$  shows up in dyadic logic as yet another example of the  $\neg$ -elimination rule. The next two examples show that in dyadic logic not only are  $\neg \exists x \forall y p(x, y)$  and  $\forall x \exists y \neg p(x, y)$  semantically equivalent but also syntactically identical. The last two examples demonstrate that the equivalent formulas  $\forall x \forall y p(x, y)$  and  $\forall x \neg \exists y \neg p(x, y)$  would be syntactically identical in dyadic logic after applying the  $\neg$ -elimination rule.

## 5 Summary

- We have introduced the dyadic logic, an encoding of the full predicate calculus that supports multi-level logic, supports set-oriented operators, implements conjunction, disjunction, and negation in constant space and time, employs an austere, machine-oriented notation, and maps directly to the list built-in data type.
- Dyadic logic has both an intensional and extensional semantics:

- The intensional semantics is useful in deductive systems as equivalence rules can fire in either direction in either context, and inference rules can fire in either direction but the direction depends on the context.
- The extensional semantics has applications in silicon compilation where the list constructor maps directly to n-input nand gates.
- The translation function  $D : Pred \rightarrow Dyadic$  is linear in the size of the formula it is applied to, in both time and space.
- Dyadic logic is *not* a normal form of anything – there are no restrictions, anything is allowed.
- Dyadic logic is a new form for logic formulas, derived from negation normal form but distinct from it.

## References

- [1] P. B. Andrews, Refutations by matings, *IEEE Trans. Comput.* C(25) (1976) 801–807.
- [2] P. B. Andrews, General matings, in: *Proc. 4th Workshop on Automated Deduction*, Austin, Texas (1979) 19–25.
- [3] P. B. Andrews, Theorem proving via general matings, *J. ACM* 28(2) (1981) 193–214.
- [4] P. B. Andrews, *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof* (Academic Press Inc., Orlando, Florida, 1986).
- [5] E. Ashcroft, A. Faustini, and R. Jagannathan, An intensional parallel processing language for applications programming, in: B. Szymanski, ed., *Parallel Functional Programming Languages and Environments* (ACM Press, 1990).
- [6] W. Bibel, On matrices with connections, *J. ACM* 28(4) (1981) 633–645.
- [7] W. Bibel, *Automated Theorem Proving* (Friedrich Vieweg & Sohn, Braunschweig/Wiesbaden, 1982).
- [8] W. Bibel, Matings in matrices, *Commun. ACM* 26(11) (1983) 844–852.
- [9] G. S. Brown, *Laws of Form* (The Julin Press, Inc., New York, 1972).
- [10] E. Mendelson, *An Introduction to Mathematical Logic* (Wadsworth & BrooksCole, Monterey, Calif., 3rd ed., 1987).
- [11] N. V. Murray and E. Rosenthal, Inference with path resolution and semantic graphs, *J. ACM* 34(2) (1987) 23–41.
- [12] N. V. Murray and E. Rosenthal, Path dissolution: A strongly complete rule of inference, in: *Proc. 6th National Conference on Artificial Intelligence*, Seattle, Washington (1987) 161–166.
- [13] R. Rucker, *Mind Tools: The Five Levels of Mathematical Reality* (Houghton Mifflin Company, 1987).