A System for Models of First Order Theories

Anwar AbdalBari

Department of Computer Science

Submitted in partial fulfillment
of the requirements for the degree of

Master of Science

Faculty of Mathematics and Science, Brock University
St. Catharines, Ontario

Abstract

If you want to know whether a property is true or not in a specific algebraic structure, you need to test that property on the given structure. This can be done by hand, which can be cumbersome and erroneous. In addition, the time consumed in testing depends on the size of the structure where the property is applied. We present an implementation of a system for finding counterexamples and testing properties of models of first-order theories. This system is supposed to provide a convenient and paperless environment for researchers and students investigating or studying such models and algebraic structures in particular.

To implement a first-order theory in the system, a suitable first-order language and some axioms are required. The components of a language are given by a collection of variables, a set of predicate symbols, and a set of operation symbols. Variables and operation symbols are used to build terms. Terms, predicate symbols, and the usual logical connectives are used to build formulas. A first-order theory now consists of a language together with a set of closed formulas, i.e. formulas without free occurrences of variables. The set of formulas is also called the axioms of the theory.

The system uses several different formats to allow the user to specify languages, to define axioms and theories and to create models. Besides the obvious operations and tests on these structures, we have introduced the notion of a functor between classes of models in order to generate more complex models from given ones automatically. As an example, we will use the system to create several lattices structures starting from a model of the theory of pre-orders.

Acknowledgements

I would like to gratefully thank Prof Michael Winter for his guidance, support, and patience.

I would like to gratefully acknowledge the support of all my family. Without their help, encouragement and support, this project wouldn't be completed.

A.A

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Logic is the study of reasoning. The language of logic consists of rules and techniques dedicated to logical reasoning. In logic, we are interested whether a given statement is true or not. In addition, by using the rules and the principles of the language, we can differentiate a true statement from false statement based on other statements. A Knowledge of logic helps to solve problems in more systematic way, and the solutions of the problems are more certain and easier than the solutions of informal method. There are different types of logic such as propositional logic, first-order logic, second-order logic etc. In this thesis, we are going to be concerned with first-order logic or predicate logic.

First-order model theory deals with the relationships between descriptions in first-order languages and the structures that satisfy these descriptions. Structures in particular and models of first-order theories in general play an important role, not only in mathematics but also in adjacent disciplines such as logic and computer science [2]. For example, certain types of lattices such as Boolean algebras, ortholattices, and pseudo-complemented lattices are essential in qualitative spatial reasoning. This area of artificial intelligence tries to describe and to reason about spatial objects and their relationship in human-like terms instead of abstract topological notions.

An algebraic structure is a model of a theory that is based on operations only, i.e., does not use any relation symbols. Therefore, such a structure consists of a set, closed under one or more operations, satisfying some axioms. No one can deny the simplicity of checking small structures satisfying some axioms with large ones. In this thesis, we present an implementation of a system for finding counterexamples and testing properties of models of first-order theories, algebraic structures in particular. As a

special case of algebraic structures, we use lattices to visualize their structures and to determine if the underlying operations satisfy a certain property true or not.
To illustrate the concept of a first order theory, consider this example of a partially ordered set $(E, \leq)$. The elements of a partially order set are arranged in an ordered fashion where a binary relation is needed to manage this order. A binary relation (a binary predicate) $\leq$ that satisfies the partial order property should be reflexive, antisymmetric, and transitive. A lattice is a partially ordered set in which any two elements have a unique supremum (the elements' least upper bound; called their join) and an infimum (greatest lower bound; called their meet). This concept could be implemented in first-order model theory. In particular first order model theory deals with the relationships between descriptions in first order language and the structures that satisfy these descriptions [3]. Moreover, in first-order model theory the mathematical objects can be represented as models for a language. Hence, in this thesis, first order theory concepts are fully implemented, therefore the following steps were performed to achieve that implementation.
First-order theory consists of a first-order language and some axioms. Hence, the first step is to implement a suitable language. A language is a collection of variables, set of predicate or relation symbols, and set of operation or function symbols. Each of the predicate symbols and the operation symbols has its arity. An operation symbol with zero parameters can be considered as a constant symbol. The collections of first-order language are used to build the formulas of the language in stepwise manner. Hence, variables, constant symbol, and operation symbols are used to build terms. Terms, predicate symbols, and logical connectives are used to build formulas. First-order theory deals with closed formulas. A closed formula is a formula where its variables are bound by a quantifier.
The next step is to provide a suitable model of a language. A model of a language consists of a non-empty set of elements called the universe together with interpretations for the function and predicate symbols in term of functions (or operations) and predicates over the universe. Such a model is a model of a theory if it is a model of the language and all axioms are true in the model. Within the system the underlying universe is always supposed to be finite and given by the integers 0 to $n-1$ where n is the total number of elements. Refer to the lattice example mentioned earlier, the model of the lattice consists of meet and join binary operations. Whereas, less than $(<)$ and greater than $(>)$ denote the predicates of that model.

The creation of models in the system can be done by two methods. The first method is to provide a proper model to the system. The next method is to use the notion of a functor between classes of models in order to generate more complex models from given ones. In this method, various theories like current theory, new theory, a set of current predicates and operations, definitions of new predicates and operations, and a model are applied to a functor to construct a new model.

The next chapter gives introduction to first-order logic.

Chapter 2

First Order Logic

Before we explain first-order logic, we provide a brief summary of a formal language and logic. A language is formal if the syntax of the language is defined with proper rules that are suitable for a computer to check whether any particular sentence belongs to the language. A logic is a formal language together with a notion of trueness and a proof systems for deriving true statements. However, there are many different kinds of logical system where proofs can be constructed. First-order logic is included in these systems.
First-order logic (FOL) is a formal logical system used in mathematics, philosophy, and computer science. First-order logic is distinguished from propositional logic in that it has quantifiers and predicates. In addition, first-order logic is symbolized reasoning in which each sentence or statement is broken down into a subject and a predicate. The predicate modifies or defines the properties of the subject. In first-order logic, a predicate can only refer to a single property, i.e., one cannot quantify over predicates.
The structure of first-order logic simulates the nature of the languages. It assumes the world contains the following:

- Objects e.g. people, cities, colors, numbers, theories, categories, ...
- Relations e.g. blue, has color, inside, greater than, less than, ...
- Functions e.g. mother of, union, intersection, plus, minus, ...

Predicates and functions in first order logic have one or more arguments, e.g. $P(x, y, z)$ or $f(a, b, c, d)$. Based on predicates one can build sentences such as $\forall x$:

$(\forall y : (\forall z : P(x, y, z)))$. This sentence has a value which is either true or false. In the previous statement, the identifiers (x, y, z) represent variables and some values could be assigned to them from the domain of these variables.
Consider this statement in first order logic, $\exists x : (\forall y : (\exists z : P(x, y, z)))$. This means there exists a value of x and no matter what value of y, there will be a value of z which is able to satisfy $P(x, y, z)$. Note that $\exists x : (\forall y : (\exists z : P(x, y, z)))$ is not the same as $\forall y : (\exists x : (\exists z : P(x, y, z)))$, so they are not equivalent.

2.1 Syntax

To build a suitable language in first-order logic, we require the following components:

1. X a set of variables.
2. F a set of function (or operation) symbols. Each symbol has its arity.
3. P a set of predicate (or relation) symbols. Each symbol has its arity.

Functions with zero parameter are called constant symbols. As mentioned earlier predicate symbols are used to build sentences, and function symbols can be used to build terms.

2.1.1 Terms

The objects of the language consist of collections of variables, a set of predicate symbols, and a set of operation symbols. Variables and operation symbols are used to build terms. Hence, the following is the definition of a term in first-order logic.

Definition 1 [10] *The set* Term *of terms is recursively defined by the following*:

1. *Each variable* $x \in X$ *is a term*, i.e., $X \subseteq \text{Term}$.
2. *If* $f \in \mathrm{F}$ *is an n-ary function symbol and* $t_1, \ldots, t_n \in \text{Term}$ *are terms, then* $f(t_1, \ldots, t_n) \in \text{Term}$.
3. *Nothing else is a term.*

From Definition 1, an individual variable is a term. An individual constant, i.e., a 0-ary function symbol is a term. A function symbol with n terms as parameters is a term. Nothing else is a term. Examples of terms are: A, 5, John, a, and $MotherOf(y)$. where A, 5, and John are constant symbols.

2.1.2 Formulas

Terms and predicate symbols are used to build formulas.

Definition 2 [10] *The set FOL of first-order formulas(or formulas) is defined as follows:*

1. *If* $t_1,\ldots,t_n \in$ *Term* are terms, *and* If p $\in P$ is an n-ary predicate symbol, then p$(t_1,\ldots,t_n) \in$ *FOL.*
2. $\perp$ is a formula, i.e, $\perp \in$ *FOL.*
3. *If* $\varphi \in$ *FOL* then $\neg\ \varphi \in$ *FOL.*
4. *If* t_1 and $t_2 \in$ *Term* are terms, then $t_1 = t_2 \in$ *FOL.*
5. *If* φ_1, $\varphi_2 \in$ *FOL* then $\varphi_1 \wedge \varphi_2 \in$ *FOL.*
6. *If* φ_1, $\varphi_2 \in$ *FOL* then $\varphi_1 \vee \varphi_2 \in$ *FOL.*
7. *If* φ_1, $\varphi_2 \in$ *FOL* then $\varphi_1 \rightarrow \varphi_2 \in$ *FOL.*
8. If $\varphi \in$ *FOL* and x $\in X$ then
   (a) $\forall x : \varphi \in$ *FOL* and
   (b) $\exists x : \varphi \in$ *FOL* .
9. *Nothing else is a formula.*

In first-order logic, formulas should be read in a unique way. To ensure that formulas are not ambiguous, rules have been developed about the precedence of the connectives to limit the need to write parentheses in writing formulas. These rules are shown as follows:

- $\neg$ is evaluated first.

- $\wedge$ and $\vee$ are evaluated next.
- $\forall$ and $\exists$ are evaluated next.
- $\rightarrow$ and $\leftrightarrow$ are evaluated last.

From the above rules we see that negation ($\neg$) has the highest precedence. Conjunction ($\wedge$) and disjunction ($\vee$) have the same level of precedence. In addition, they have higher precedence than ($\forall, \exists$) quantifiers. Finally, implication ($\rightarrow$) and equivalence ($\leftrightarrow$) logical symbols have the lowest precedence.

Let us illustrate the precedence of the connectives through an example.

$$\neg(\forall x\ P(x)) \rightarrow \exists x\ (\neg(P(x)))$$

By using the above rules of the precedence of the connectives, the above formula could be written as

$$\neg\forall x\ P(x) \rightarrow \exists x\ \neg P(x)$$

From Definition 2, a predicate symbol with n terms as parameters is a formula. The propositional symbol ($\perp$) is a formula. The equality of two terms is a formula. If φ is a formula, then $\neg\ \varphi$, $\exists x : \varphi$, $\forall x : \varphi$ are formulas, where x is an individual variable. If φ_1 and φ_2 are formulas then, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, and $\varphi_1 \rightarrow \varphi_2$ are formulas.
An atomic formula is a formula of kind 1 or 4. of the previous definition. Hence, it is constructed using a predicate symbol, including =, and a suitable number of term. Such an atomic formula does not have any logical connectives. A complex or a compound formula is a formula that contains atomic formula with logical connectives. Figure 2.1 represent an atomic formula. Notice that "Length" and "Tower" are unary operations. The parameter of these functions maybe variable, function, or constant like in this case.
Figure 2.2 represents a complex formula, which is built from two atomic formulas and a connective symbol. From both figures we can distinguish between a predicate and a function by the fact that, a function returns value such as "Length" and "Tower" functions, while a predicate returns either true or false such as "Sibling" and ">" predicates.

In the following section, we want to introduce the concept of free and bound variables in a formula.

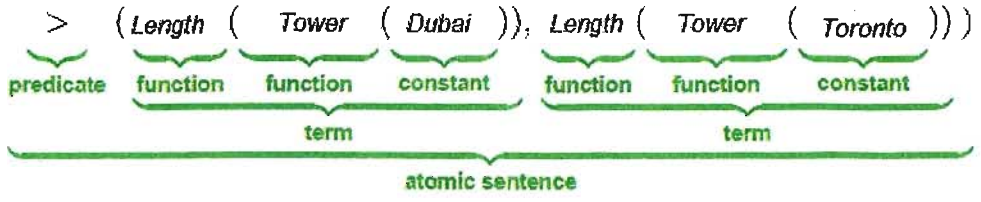


Figure 2.1: Atomic Formula

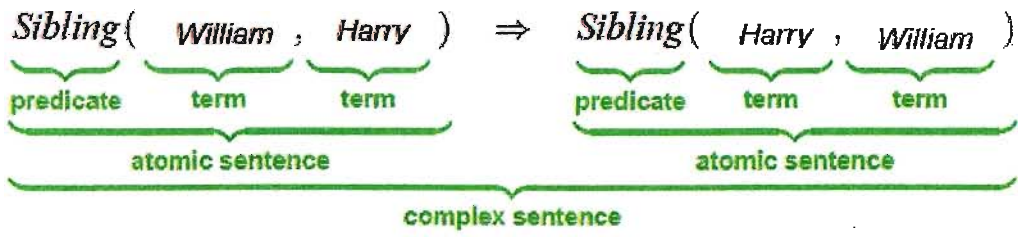


Figure 2.2: Complex Formula

2.1.3 Free and Bound Variables

Any variable may occur free and bounded in the same formula. Hence, an occurrence of variable in a sentence is free if it is not in the scope of any quantifier with the same variable. Otherwise, non-free variable occurrence is bound.

Definition 3 [10] *An occurrence of a variable $x \in X$ in a formula is bounded iff it is in a subformula of the form $\forall x : \varphi$ or $\exists x : \varphi$. An occurrence is free iff it is not bounded.*

According to the last definition, consider this example:

$$\forall x : P(x) \wedge Q(x)$$

Notice that, both occurrences of x are bounded by $\forall x$. In the following example, the first occurrence of x is bound by $\forall x$, while the second occurrence is free.

$$(\forall x : P(x)) \wedge Q(x)$$

Another example is:

$$\forall x : R(x, z) \wedge \exists y : S(x, y)$$

In this example both occurrences of x are bounded by $\forall x$, while the occurrence of y is bound by $\exists y$, and the occurrence of z is free.

To illustrate the difference between free and bounded variables, consider these two examples of formulas.

$$\exists x : (\exists y : (Female(x) \wedge Human(y) \wedge MotherOf(x, y))) \text{ and}$$
$$\exists x : (Female(x) \wedge MotherOf(x, z)).$$

There is an important difference between the two examples. The first might be translated as: There is a female and a human so that the female is the mother of the human. Whereas, the second example might be translated only as incomplete sentence as: There is a female, and that female is a mother of something. We are unable to complete the sentence because we don't have enough information about the term z. We will say that z occurs free in that formula. In contrast, no free variables occur in the first example. In a first-order theory, we will be interested in closed formulas only.

As mentioned earlier, the language of first-order logic contains quantifiers and connectives symbols. A convenient conversion of these symbols are used in the system. Table 2.1 shows the quantifiers and the connectives of the language and their conversion in the system.

Recall the $(E; \wedge, \vee, \leqslant)$ structure mentioned earlier. The structure represents a lattice. The relation $\leq$ is reflexive, transitive and antisymmetric, and every pair of the set elements has meet and join operations. Table 2.2 shows the formulas that satisfied by that structure and their representations in first-order logic. In the case of reflexivity notice that, the word (All) in language of FOL denotes $(\forall)$ symbol, (x) denotes a variable symbol, and $(<)$ denotes a binary predicate symbols of the language.

The system allows the user to specify the notation of the predicate and the operation symbols of the language. Hence, the logical notation could be infix, prefix, or postfix. We use infix logical notation in case of binary predicate symbol. Therefore, instead of writing a reflexive relation in this way $(Allx < (x, x))$, for more convenience we write it as $(Allx\ (x < x))$.

Table 2.1: FOL Symbols Conversion

Symbols in FOL	Symbols in the system
$\neg$	*Not*
$\forall$	*All*
$\exists$	*Exists*
$\wedge$	&
$\vee$	|
$\rightarrow$	*Rightarrow*
$\leftrightarrow$	*Leftrightarrow*
$=$	*Equal*

In the next section we will introduce the interpretation of a language in FOL, and the notion of a model of a theory.

Table 2.2: FOL Formula Conversion

	FOL Notation	The System Notation
Reflexive	$\forall x : R(x, x)$	*All x (x < x)*
Antisymmetric	$\forall x : \forall y : R(x, y) \wedge$ $R(y, x) \rightarrow (x = y)$	*All x (All y ((x < y) & (y < x)) Rightarrow (x = y))*
Transitive	$\forall x : \forall y : \forall z :$ $R(x, y) \wedge R(y, z)$ $\rightarrow R(x, z)$	*All x (All y (All z (x < y & y < z Rightarrow x < z)))*
Meet	$\forall x : \forall y : \exists z :$ $[z \leq x$ *and* $z \leq y, \forall u$ $[u \leq x$ *and* $u \leq y]$ $\rightarrow u \leq z]$	*All x (All y (Exists z ((z < x & z < y & (All u (u < x & u < y) Rightarrow u < z)))))*
Join	$\forall x : \forall y : \exists z :$ $[x \leq z$ *and* $y \leq z, \forall u$ $[x \leq u$ *and* $y \leq u]$ $\rightarrow z \leq u]$	*All x (All y (Exists z ((x < z & y < z & (All u (x < u & y < u) Rightarrow z < u)))))*

2.2 Semantics

We need an interpretation and a model in order to define whether a formula is true or not. A model consists of objects and relations between these objects. The interpretations are based on functions that connect the symbols in the language of first-order logic to elements and sets of objects in a model or a domain. First we have to define the model.

Definition 4 [10] *Let F be a set of function symbols, and P be a set of predicate symbols. A model $\mathcal{M}$ consists of the following data:*

1. $\mid \mathcal{M} \mid$ *a non-empty set, called the domain or the universe.*
2. *For each function symbol $f \in F$ with arity n a n-ary function $f^{\mathcal{M}}$: $\mid \mathcal{M} \mid^n \rightarrow \mid \mathcal{M} \mid$.*
3. *For each predicate symbol $p \in P$ with arity n a subset $p^{\mathcal{M}} \subseteq \mid \mathcal{M} \mid^n$.*

From the above definition, notice that the constant symbols of the language are mapped to elements of the domain. Furthermore, unary predicate symbols are mapped to sets of elements of the domain, while the binary predicate symbols are mapped to sets of pairs of the domain elements.
Because the variables have no real meaning, and the meaning of variables is defined by an assignment, we have to know the semantics of terms and the formulas that contain these variables. In addition, we have to replace these variables by real values.

Definition 5 [10] *Let $\mathcal{M}$ be a model. An environment $\sigma : X \rightarrow |\mathcal{M}|$ is a function from the set of variables X to the universe of the model. For an environment σ, a variable x, and a value $a \in |\mathcal{M}|$ denote by $\sigma[a/x]$ the environment defined by*

$$\sigma[a/x](y) = \begin{cases} \text{a} & \text{iff } x = y, \\ \sigma(y) & \text{iff } x \neq y. \end{cases}$$

A term should denote an element in first-order logic so that the interpretation of a term is an element of the universe.

Definition 6 [10] *Let $\mathcal{M}$ be a model, and σ be an environment. The extension $\bar{\sigma}$: Term $\rightarrow |\mathcal{M}|$ of σ is defined by:*

1. *$\bar{\sigma}(x) = \sigma(x)$ for every $x \in X$.*
2. *$\bar{\sigma}(f(t_1, \ldots, t_n)) = f^{\mathcal{M}}(\bar{\sigma}(t_1), \ldots, \bar{\sigma}(t_n))$.*

The previous definition defines the interpretation of terms. More specifically rule one defines the interpretation of variables terms while rule two defines the interpretation of functional terms. Next, we want to define the validity of formulas.

Definition 7 [10] *Let M be a model, σ be an environment, and φ be formula. The satisfaction relation $\models_M \varphi$ $[\sigma]$ is recursively defined by:*

1. *$\models_{\mathcal{M}} p(t_1, \ldots, t_n)$ $[\sigma]$ iff $(\bar{\sigma}(t_1), \ldots, \bar{\sigma}(t_n)) \in P^{\mathcal{M}}$.*
2. *$\not\models_{\mathcal{M}} \perp$ $[\sigma]$, i.e., not $\models_{\mathcal{M}} \perp$ $[\sigma]$.*
3. *$\models_{\mathcal{M}} \neg\varphi$ $[\sigma]$ iff $\models_{\mathcal{M}} \varphi$ $[\sigma]$.*

4. $\models_{\mathcal{M}} \varphi_1 \wedge \varphi_2\ [\sigma]$ *iff* $\models_{\mathcal{M}} \varphi_1\ [\sigma]$ *and* $\models_{\mathcal{M}} \varphi_2\ [\sigma]$.

5. $\models_{\mathcal{M}} \varphi_1 \vee \varphi_2\ [\sigma]$ *iff* $\models_{\mathcal{M}} \varphi_1\ [\sigma]$ *or* $\models_{\mathcal{M}} \varphi_2\ [\sigma]$.

6. $\models_{\mathcal{M}} \varphi_1 \rightarrow \varphi_2\ [\sigma]$ *iff* $\models_{\mathcal{M}} \varphi_1\ [\sigma]$ *whenever* $\models_{\mathcal{M}} \varphi_2\ [\sigma]$.

7. $\models_{\mathcal{M}} \forall : \varphi\ [\sigma]$ *iff* $\models_{\mathcal{M}} \varphi\ [\sigma[a/x]]$ *for all* $a \in \mid \mathcal{M} \mid$.

8. $\models_{\mathcal{M}} \exists : \varphi\ [\sigma]$ *iff* $\models_{\mathcal{M}} \varphi\ [\sigma[a/x]]$ *for some* $a \in \mid \mathcal{M} \mid$.

Next, we will introduce the satisfiability and validity of formulas in general.

Definition 8 [10] Let Σ be a set of formulas, and φ be a formula.

1. φ *is called valid in the model* $\mathcal{M}$ $(\models_{\mathcal{M}} \varphi)$ *iff* $\models_{\mathcal{M}} \varphi[\sigma]$ *for all environments* σ.

2. φ *is called valid* $(\models \varphi)$ *iff* $\models_{\mathcal{M}} \varphi$ *for all models* $\mathcal{M}$.

3. φ *is called satisfiable iff there is a model* $\mathcal{M}$ *and an environment so that* $\models_{\mathcal{M}} \varphi$ $[\sigma]$.

4. φ *follows from* Σ *in* $\mathcal{M}$ $(\Sigma \models_{\mathcal{M}} \varphi)$ *iff for all environments* σ*, whenever* $\models_{\mathcal{M}} \psi[\sigma]$ *for all* $\psi \in \Sigma$*, then* $\models_{\mathcal{M}} \varphi[\sigma]$.

5. φ *follows from* Σ $(\Sigma \models \varphi)$ *iff* $\Sigma \models_{\mathcal{M}} \varphi$ *for all models* $\mathcal{M}$.

If a formula φ or a set of formulas Σ is valid in a model $\mathcal{M}$, we will call $\mathcal{M}$ a model of φ or Σ, respectively.

2.3 First-order Theory

In this section, we want to introduce a theory of first-order logic. Moreover, we want to introduce the notion of a model of such a theory.

Definition 9 [10] *A theory* $\mathcal{T}$ *is a set of closed formulas in first-order language.*

If we have a theory $\mathcal{T}$ and a model $\mathcal{M}$ such that the model $\mathcal{M}$ satisfies all the sentences of $\mathcal{T}$, we say that the model $\mathcal{M}$ is a model of theory $\mathcal{T}$. Similarly, if a

model $\mathcal{M}$ satisfies a sentence φ we say that, the model $\mathcal{M}$ is a model of φ.
Let us recall the lattice structure $(E; \wedge, \vee, \leq)$ to define its language and the corresponding theory. In this example, we will now provide a model of that theory.

Consider the lattice visualized in Figure 2.3. The sets consists of the following:

predicates: $<, >$; functions: $meet, join$; constant: U

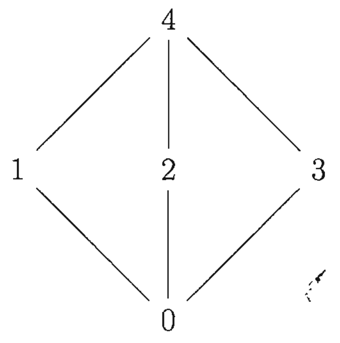


Figure 2.3: Four-elements Lattice

Table 2.3: The Lattice Model

Description	Name	Arity	Set elements
Predicate	<	2	(0,0),(0,1),(0,2),(0,3),(0,4),(1,1), (1,4),(2,2),(2,4),(3,3),(3,4),(4,4)
Predicate	>	2	(0,0),(1,0),(2,0),(3,0),(4,0),(1,1), (4,1),(2,2),(4,2),(3,3),(4,3),(4,4)
Operation	meet	2	(0,0,0),(0,1,0),(0,2,0),(0,3,0),(0,4,0),(1,0,0),(1,1,1), (1,2,0),(1,3,0),(1,4,1),(2,0,0),(2,1,0),(2,2,2),(2,3,0), (2,4,2),(3,0,0),(3,1,0),(3,2,0),(3,3,3),(3,4,3),(4,0,0), (4,1,1),(4,2,2),(4,3,3),(4,4,4)
Operation	join	2	(0,0,0),(0,1,1),(0,2,2),(0,3,3),(0,4,4),(1,0,1),(1,1,1), (1,2,4),(1,3,4),(1,4,4),(2,0,2),(2,1,4),(2,2,2),(2,3,4), (2,4,4),(3,0,3),(3,1,4),(3,2,4),(3,3,3),(3,4,4),(4,0,4), (4,1,4),(4,2,4),(4,3,4),(4,4,4)

Next, we want to construct a finite model or a domain of the theory. The domain of the lattice above contains operations and predicates. In this domain "meet" and "join" denote the domain operations, while "<" and ">" denote the domain predicates.

The model of the lattice visualized in Figure 2.3 is specified by Table 2.3. As we can see from Table 2.3, each predicate and operation consists of a set of elements. The representation of these elements is varying, depending on the arity of the represented predicate or operation. In case of a binary predicate, i.e., "<" and ">", the elements are represented by tuples. Each tuple consists of two elements, these elements satisfy the predicate functionality. While in case of binary operation, i.e., "meet" and "join", the elements are represented by a triple. The first two elements of the triple are inputs to that operation, while the third element is the output. The arity of the predicates and the operations in a model should match the arity of the predicate symbols and operation symbols of a language.

Tables 2.4 and 2.5 show the predicates of that domain.

	0	1	2	3	4
0	T	T	T	T	T
1	F	T	T	T	T
2	F	F	T	T	T
3	F	F	F	T	T
4	F	F	F	T	T

Table 2.4: Less Than Predicate

	0	1	2	3	4
0	T	F	F	F	F
1	T	T	F	F	F
2	T	T	T	F	F
3	T	T	T	T	F
4	T	T	T	T	T

Table 2.5: Greater Than Predicate

Notice that the domain of the predicates is {True,False}. Therefore, if you test any two elements of the set in Table 2.4, the result will be true or false. For example, (1<0)= False is represented by the cell of the table. Similarly, if you test any two elements of the set in Table 2.5, the result will be true or false. For example, (3>2)= True.

	0	1	2	3	4
0	0	0	0	0	0
1	0	1	1	1	1
2	0	1	2	2	2
3	0	1	2	3	3
4	0	1	2	3	4

Table 2.6: Meet Operation

	0	1	2	3	4
0	0	1	2	3	4
1	1	1	4	4	4
2	2	4	2	4	4
3	3	4	4	3	4
4	4	4	4	4	4

Table 2.7: Join Operation

Notice that the domain of the operations is {0,1,2,3,4}. Therefore, the result of 2 meet 0 is 0, which is represented by the cell of the Table 2.6. Similarly, the result of 3 join 2 is 4. In this example, we used the $\wedge$ and $\vee$ symbols as shown in the theory formulas to represent or to map the meet and the join functions of the model.

2.4 Functor

This section gives a brief overview of the important categorical concepts, namely category and functor.

A category is a collection of data that satisfy some particular properties. Furthermore, Category Theory studies objects and morphisms between them.

Definition 10 [1] *A category C is*

1. *a collection Ob_C of objects, denoted by a, b... A, B ....*
2. *a collection Mor_C of morphisms (arrows), denoted by f, g ....*
3. *two operations dom, cod assigning to each arrow f two objects respectively called domain (source) and codomain (target) of f.*
4. *an operation id assigning to each object b a morphism id_b (the identity of b) such that $dom(id_b) = cod(id_b) = b$*
5. *an operation "$\circ$" (composition) assigning to each pair f, g of arrows with dom(f) = cod(g) an arrow $f \circ g$ such that $dom(f \circ g) = dom(g), cod(f \circ g) = cod(f)$*
6. *identity and composition, moreover, must satisfy the following conditions: identity law: for any arrows f, g such that cod(f) = b = dom(g)*

   - $id_b \circ f = f$
   - $g \circ id_b = g$

   associative law: for any arrows f, g, h such that dom(f) = cod(g) and dom(g) = cod(h)

   - $(f \circ g) \circ h = f \circ (g \circ h)$.

We write $f : a \longrightarrow b$ to denote a morphism whose source and target are respectively a and b.
Consider the category in which the objects are categories and the morphisms are mappings between categories. The morphisms in such a category are known as functors.

Definition 11 [1] *Let C and D be categories. A functor $F : C \longrightarrow D$ is a pair of operations $F_{ob} : Ob_C \longrightarrow Ob_D$, $F_{mor} : Mor_C \longrightarrow Mor_D$ such that, for each $f : a \longrightarrow b, g : b \longrightarrow c$ in C,*

- $F_{mor}(f) : F_{ob}(a) \longrightarrow F_{ob}(b)$
- $F_{mor}(g \circ f) = F_{mor}(g) \circ F_{mor}(f)$
- $F_{mor}(id_a) = id_{F_{ob}(a)}$.

From Definition 11, if two categories are given, C and D, a functor $F : C \longrightarrow D$ maps each morphism of C onto a morphism of D, such that: F preserves identities, i.e. if x is a C-identity, then $F(x)$ is a D-identity and F preserves composition, i.e $F(f \circ g) = F(f) \circ F(g)$. From the above, a functor is a special type of mapping between categories. Functors can be thought of as homomorphisms between categories.

In abstract algebra, a homomorphism is a structure-preserving map between two algebraic structures [1]. Next, we will define a homomorphism between models.

Definition 12 *Let $\mathcal{M}$ and $\mathcal{N}$ be models, $h : \mathcal{M} \rightarrow \mathcal{N}$ is a homomorphism, iff*

- $h :\mid \mathcal{M} \mid \rightarrow \mid \mathcal{N} \mid$.
- *for every function symbol f and $a_1, \ldots, a_n \in \mid \mathcal{M} \mid$*
  - $h(f^{\mathcal{M}}(a_1, \ldots, a_n)) = f^{\mathcal{N}}(h(a_1, \ldots, a_n))$
  - $(a_1, \ldots, a_n) \in P^{\mathcal{M}} \Leftrightarrow (h(a_1), \ldots, h(a_n)) \in P^{\mathcal{N}}$.

Lemma 1 *Let $\mathcal{T}$ be a first-order theory. Then the models of $\mathcal{T}$ together with the homomorphism form a category.*

Our functor F with an input language $\mathcal{L}_1$ and an input theory $\mathcal{T}_1$ and an output language $\mathcal{T}_2$ and an output theory $\mathcal{T}_2$ is a functor between the category of those models of $\mathcal{T}_1$ that give a model of $\mathcal{T}_2$ using the definition in F and the models of $\mathcal{T}_1$.

Chapter 3

The System

This chapter will discuss the implementation of first-order logic concepts in the system. As an example, algebraic structures will be used to illustrate the usage of the system. We will focus on structures like order, lower lattice, lattice, bounded lattice, pesudocomplement lattice, quasicomplement lattice, and p-algebra. First, we will introduce the definitions and the properties of these structures. Next, we will show how to translate these structures into the syntax of the system.

3.1 System Overview

The system provides three main functions to the user. These functions are:

- Test a formula.
- Apply a functor to a model.
- Check if the model is a model of a given theory.

By "test a formula" we mean that, if we have a model and a formula loaded in the system, simply by using this function we can check if a selected model satisfies a selected formula or not. Moreover, in case the selected model does not satisfy the selected formula, the system shows the user a counterexample.
"Apply a functor to a model" function is used to generate more complex models from given ones. For example, by using a system functor and an order model as input, other structures like lower lattice, lattice, bounded lattice, pesudocomplement lattice, quasicomplement lattice, and p-algebra could be generated. Additional operations

and predicates are implemented in the functor. The structure generated should satisfy the axioms of the output theory. This is checked automatically.
Finally, "Check if the model is a model of a theory" function, is used to check if the selected model satisfies a specific theory or not.
Formulas are used in various ways in the generation of new models in the system, i.e., formulas are used to create theories as well as functors. Hence, given a formula φ with free variables $x_1, \ldots, x_n, z$ there are three forms of that formula used in the system as the follows:

- $\forall x_1, \ldots, x_n \; \exists z \; \varphi$
- $\forall x_1, \ldots, x_n \; \varphi \; [f(x_1, \ldots, x_n)/x]$
- φ

The first formula is typical for testing the properties of a model, i.e., if we want to check if a model satisfies a specific property, we use that form of formula to represent that property. As an example, consider $\forall x : x < z$ for φ. Then the following formula denotes the top element of a lattice:

$$\exists z : \forall x : x < z$$

The second form used as a component of a theory. As mentioned earlier a theory consists of a language and some formulas. We use that form of formula to represent a formula used to create a theory. The following form states that the constant "1" is the top element of the bounded lattice:

$$\forall x : (x < 1)$$

The third form is typically used when adding a new function using a functor, i.e., it used in the functor by the following format: $f \; x_1, \ldots, x_n \; z \; \varphi$. In this form, "$f$" denotes a function name, $x_1, \ldots, x_n$ represent the free variables in the formula, and z denotes the output of that formula. The following formula denotes the top element of the bounded lattice used in a functor:

$$1 \,\_\, z \; \mathit{All}x \; (x < z)$$

Note that the above formula is a closed formula. Hence, no free variables are found.

3.2 The Example Structure

Figure 3.1 shows the structure we will be using in order to illustrate the system. Initially we will consider this structure as a model of an ordered set. Then we will apply functors stepwise in order to extend this model into a model of the theory of Stonian p-ortholattices.

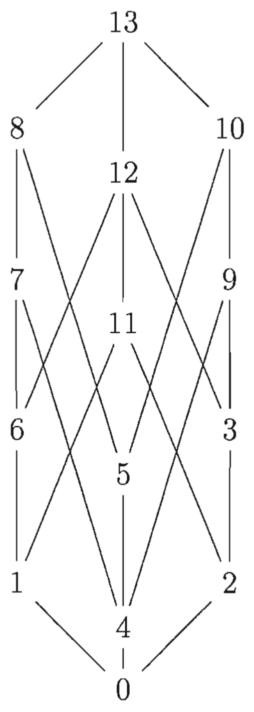


Figure 3.1: The Outer Structure of C_{14}

In each step of this demonstration, we will use the main functions of the system to build a new model, check if the current model satisfies a given formula, and to check if that model is a model of the theory in question. In addition, we will check if the structure below is a Stonian p-ortholattices or not. The structure has 14 elements. For simplicity, we use the elements set from 0 . . . 13, with elements 0 and 13 to represent the bottom and the top of the structure respectively.

3.2.1 Order

Partially ordered set (or poset) generalizes the concept of an ordering. An order is the arrangement of elements in a set in an ordered fashion. Therefore, a binary relation is needed to manage the order of the elements pairs in the set. Hence, a poset $(E, \leq)$ is a tuple that consists of a set and a binary relation. A binary relation in a poset indicates that, for certain pairs of elements in the set, one of the elements precedes the other.

Definition 13 *A binary relation $\leq$ on a set E is called partial order set if it is:*

1. *Reflexive, i.e., $x \leq x$ for all $x \in E$.*
2. *Antisymmetric, i.e., $x \leq y$ and $y \leq x$ implies $x = y$ for all $x, y \in E$.*
3. *Transitive, i.e., $x \leq y$ and $y \leq z$ implies $x \leq z$ for all $x, y, z \in E$.*

The following steps are used to implement Figure 3.2 example in the system.

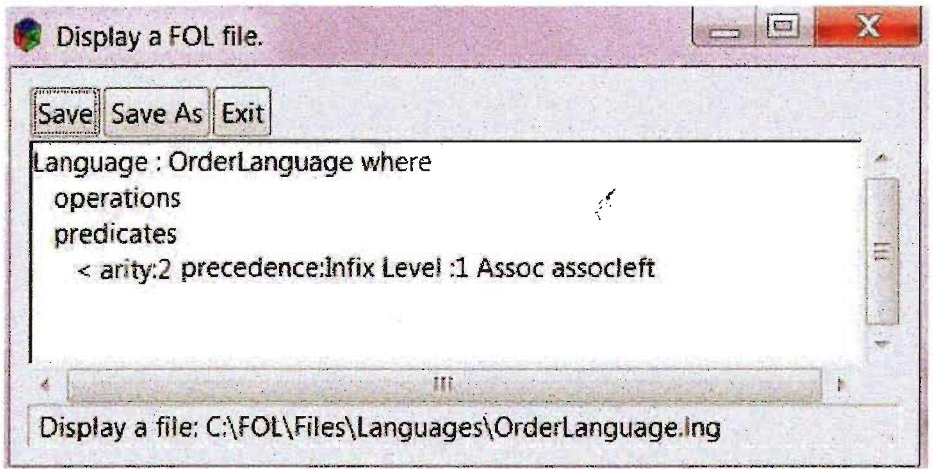
First step, we need a language to be loaded in the system. The language gives information about predicate symbols and function symbols used to create formulas as well as theories. Moreover, the languages files are stored in the operating system with "lng" extension. The structure of the language file is as follows:

Language : (language name) where

> *Operations*
>
> > *(Operation name) arity: n precedence: (Infix, Prefix, or Postfix) Level: n Assoc (AssocNone, AssocLeft, or AssocRight)*
>
> *Predicates*
>
> > *(Predicate name) arity: n precedence: (Infix, Prefix, or Postfix) Level: n*

We tried to make it easy and convenient to the user to define a language. Hence, by using the above structure, a language could be created by specifying a language name, a set of operation symbols, and a set of predicate symbols. An operation symbol takes five arguments to be defined. The first argument is the operator name. The second argument is the arity of the operator. Operator can be Infix, Prefix or Postfix which can be specified by the third argument. The fourth argument is the level of the operator in the operator table. Infix operators also have an associativity: left, right or none, which can be specified by the fifth argument. If operators are right-associative, the operators are applied in right-to-left order. Whereas, if operators are left-associative, the operators are applied in left-to-right order.

In this example of an order implementation, the language contains one binary predicate symbol "<". This predicate symbol is used to denote the order relation of Definition 13. Figure 3.2 shows the language of the order structure example.



```
Language : OrderLanguage where
  operations
  predicates
    < arity:2 precedence:Infix Level :1 Assoc assocleft
```

Display a file: C:\FOL\Files\Languages\OrderLanguage.lng

Figure 3.2: Order Language

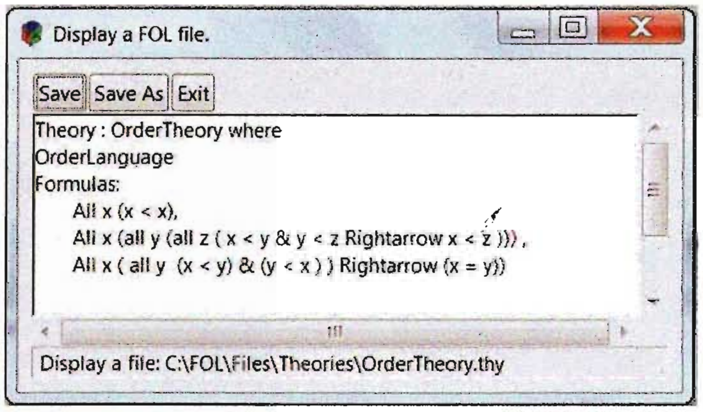
As shown above, after we specified the language name, we defined the predicate of the language. The arity of the "<" predicate is two, which means that, we have predicate symbol "<" with two parameters with infix logical notation. No function symbols are needed to be specified in the language of the order structure. Particulary, we want to define formulas contain one predicate symbol.

The second step is to define the theory of the language of the order structure. A theory consists of a language and a set of axioms. Theories files are stored in the operating system with "thy" extension. The following structure is used to identify a theory in the system.

Theory : (Theory name) where

(Language name)
Formulas:
List of formulas separated by comma

The above structure is based on Definition 9, where a theory is a set of sentences in first-order language. Hence, we used "Theory Name" to denote the name of the theory, "Language name" to refer the a pre-defined language in the system, and "Formulas" to denote a list of closed formulas.



```
Theory : OrderTheory where
OrderLanguage
Formulas:
    All x (x < x),
    All x (all y (all z ( x < y & y < z Rightarrow x < z ))),
    All x ( all y  (x < y) & (y < x ) ) Rightarrow (x = y))
```

Display a file: C:\FOL\Files\Theories\OrderTheory.thy

Figure 3.3: Order Theory

Figure 3.3 shows a snapshot of the theory of the order structure example. From the figure, note that OrderLanguage refers to the language we already defined in the system. Also note, how we represent the reflexive, transitive, and antisymmetric formulas.

The third step is to load the model of the language of the order structure. A model consists of non empty set of elements, called the universe, together with interpretations for the function and predicate symbols in term of functions (or operations) and predicates over the universes. Model files are stored in the operating system with "mdl" extension. The following structure is used to identify a model in the system.

Model : (Model name) where
Elements: n
Operations

List of operations separated by a semicolon

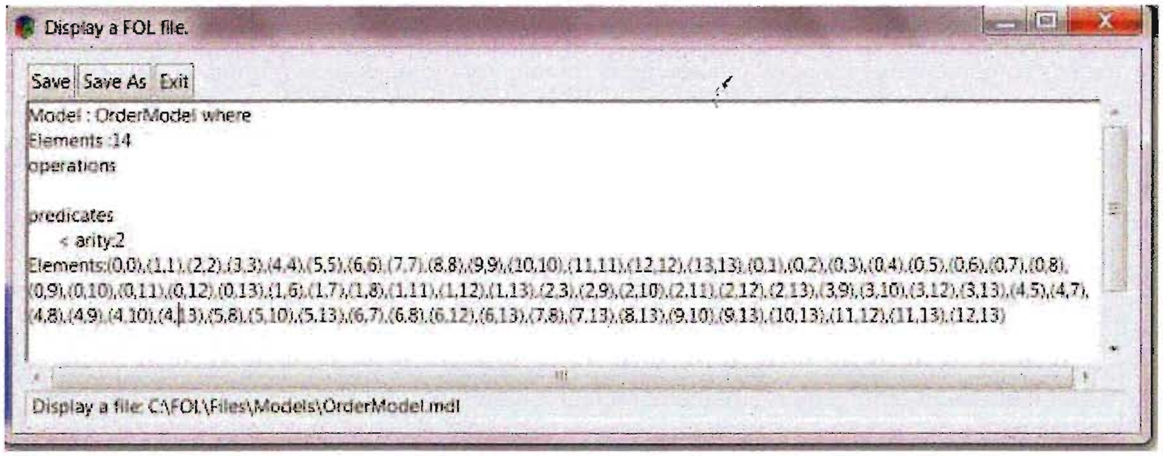
Operations are in the form (The operation name, the operation arity, and a list of the operation elements)

Predicates

List of predicates separated by a semicolon

Predicates are in the form (The predicate name, the predicate arity, and a list of the predicate elements)

For the order example, the system represents a model with an order relation "<" as shown in Figure 3.4. We used an input parser in the system. More details about the parser used in the system are covered in Chapter 4.



```
Model : OrderModel where
Elements :14
operations

predicates
    < arity:2
Elements:(0,0),(1,1),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7),(8,8),(9,9),(10,10),(11,11),(12,12),(13,13),(0,1),(0,2),(0,3),(0,4),(0,5),(0,6),(0,7),(0,8),
(0,9),(0,10),(0,11),(0,12),(0,13),(1,6),(1,7),(1,8),(1,11),(1,12),(1,13),(2,3),(2,9),(2,10),(2,11),(2,12),(2,13),(3,9),(3,10),(3,12),(3,13),(4,5),(4,7),
(4,8),(4,9),(4,10),(4,13),(5,8),(5,10),(5,13),(6,7),(6,8),(6,12),(6,13),(7,8),(7,13),(8,13),(9,10),(9,13),(10,13),(11,12),(11,13),(12,13)
```

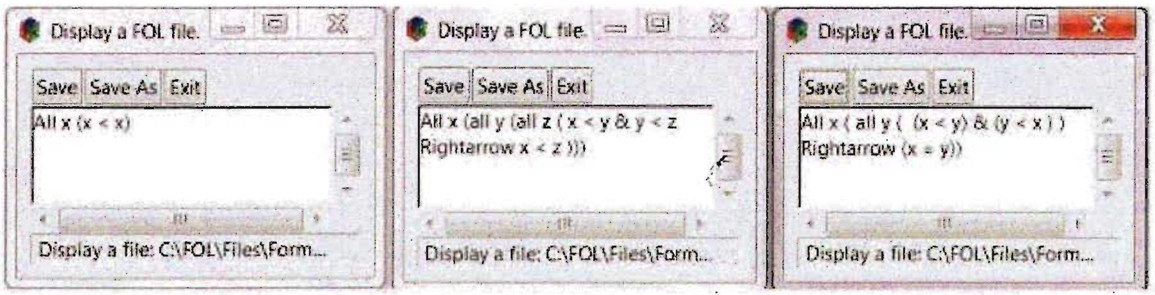
Display a file: C:\FOL\Files\Models\OrderModel.mdl

Figure 3.4: Order Model

In the model class, a new predicate is defined by the following: the name of the predicate, which is denoted by "<" symbol, the arity of the predicate (in our example the arity is two), and the elements of the predicate set. Moreover, the function of the model is defined in the same way.

From the previous figure, note that, the representation of the elements in the predicate set is in order manner, i.e., (2, 13) means that two is less than or equal 13, (6, 12) means that six is less than or equal 12, and (13, 13) means that 13 is less than or equal 13 etc. In addition, note that no functions are defined in the model of order lattice.

To verify if a model satisfies certain formulas, we need to load these formulas. Formulas files are stored in the operating system by "frm" extension. To accomplish the process of loading formulas, we need a proper language to be loaded in the system. The representation of the formula in the system is based on the syntax of first-order logic. Precisely, Definition 2 of Chapter 1.
Figure 3.5 shows the implementation of reflexive, antisymmetric, and transitive formulas in the system. As mentioned earlier, the formulas used the predicate symbol "<" which defined in the language, and the truth-value of this predicate is found in the model of this structure.



```
All x (x < x)
```

Display a file: C:\FOL\Files\Form...

```
All x (all y (all z ( x < y & y < z
Rightarrow x < z )))
```

Display a file: C:\FOL\Files\Form...

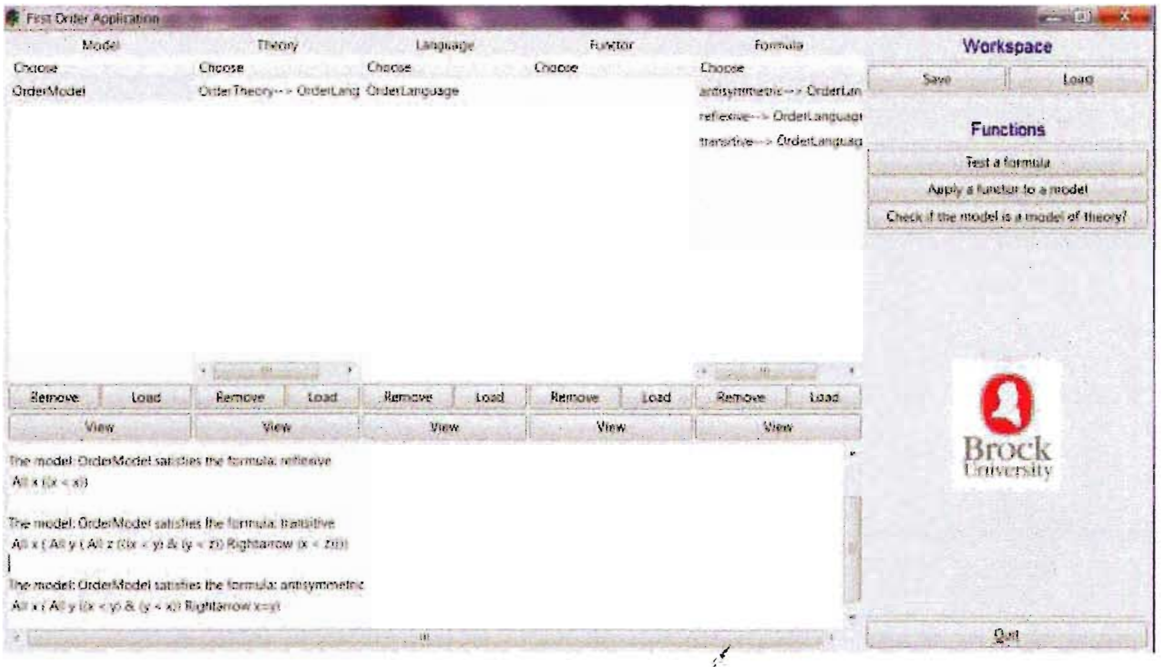
```
All x ( all y (  (x < y) & (y < x ) )
Rightarrow (x = y))
```

Display a file: C:\FOL\Files\Form...

Figure 3.5: Reflexive, antisymmetric, and transitive formulas

Now, we are ready to check if the model of Figure 3.1 structure satisfies the formulas that were mentioned in Definition 13. To perform that check, we simply choose the model and the formula we want to check, then we click on "test formula" button as shown in the following figure.
As we can see in Figure 3.6, the figure shows the loaded model, the loaded language, and the loaded formulas. The result of checking the model against the three formulas respectively, shows that the model of the order structure satisfies these formulas, and the results are shown in the workplace displaying of the system. Therefore, the model of the structure shown in Figure 3.1 is an order structure.

Note that from the formula column in Figure 3.5, we used an arrow beside the formula name to reference the language, which the formula is based on.
As shown in Figure 3.3, we defined the order theory by a predefined order language, and formulas of reflexivity, transitivity, and antisymmetry. After loading the theory, the system is ready to check if a model is a model of a theory. The check can be

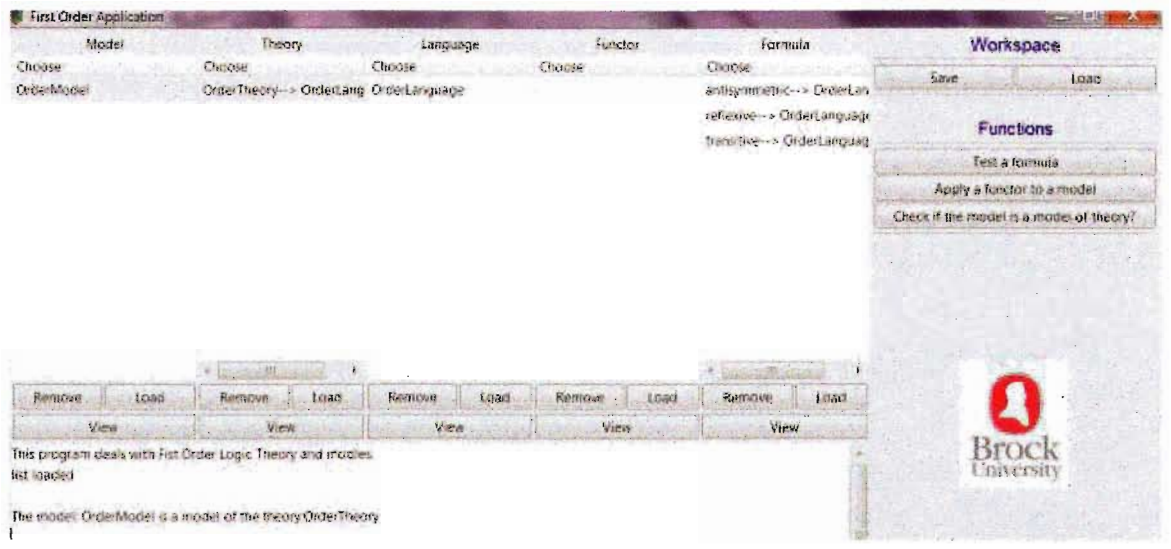


```
The model: OrderModel satisfies the formula: reflexive
 All x ((x < x))

The model: OrderModel satisfies the formula: transitive
 All x ( All y ( All z (((x < y) & (y < z)) Rightarrow (x < z))))

The model: OrderModel satisfies the formula: antisymmetric
 All x ( All y ((x < y) & (y < x)) Rightarrow x=y)
```

Figure 3.6: Snapshot of order workspace

accomplished by selecting a model and a theory, and by clicking the "check if a model is a model of a theory" button, we get the result in the workplace displaying area of the system. Figure 3.7 shows that, the model is a model of theory.
To summarize this example, we created a model containing a binary predicate that is in fact an order relation. Afterwards, we defined and loaded a language. The language contained order predicate symbol. Then we loaded the formulas of reflexivity, transitivity, and antisymmetry. Next, we defined and loaded a theory. The theory contained order formulas. Finally, we checked the model against these formulas as well as against the corresponding theory of orderings.

In the next subsection, we will move forward to the next algebraic structure, which is a lower semilattice. We will show how the system will generate a semilattice model out of the order model defined in this section using a functor.



```
This program deals with First Order Logic Theory and modles
list loaded

The model OrderModel is a model of the theory OrderTheory
```

Figure 3.7: Snapshot of order theory

3.2.2 Lower Semilattice

A lower semilattice is a partially ordered set in which each pair of elements has a greatest lower bound [2]. A lower bound of a subset F of a partially order set $(P, \leq)$ is an element $x \in P$ that element is smaller than or equal to every element of F, i.e., $x \leq y$. An upper bound is defined dually. The greatest lower bound of F is the great element of the lower bounds. Whereas, the least upper bound is the least element of the upper bounds. Notice that greatest lower bounds and least upper bounds do not necessarily exist.

The following formula denotes the greatest lower bound formula of an order set.

$$\forall x : (\forall y :(\exists z : (z < x \land z < y \land (\forall u : (u < x \land u < y) \rightarrow u < z))))$$

To generate a model of lower semi-lattice from an order model, we will use the concept of a functor implemented in the system. In general, a functor is a notion used in category theory. It is a mapping between categories that preserves the structure of the categories. In that sense the functors implemented in the system are functors between categories of models of first-order theories. If one applies a function to a given model of a theory, the newly generated model contains the same functions and predicates as the input model plus new functions or predicates specified by the functor. Therefore, a functor is a functor between the category of models of the input theory providing extra structure to the category of models of the enriched theory.

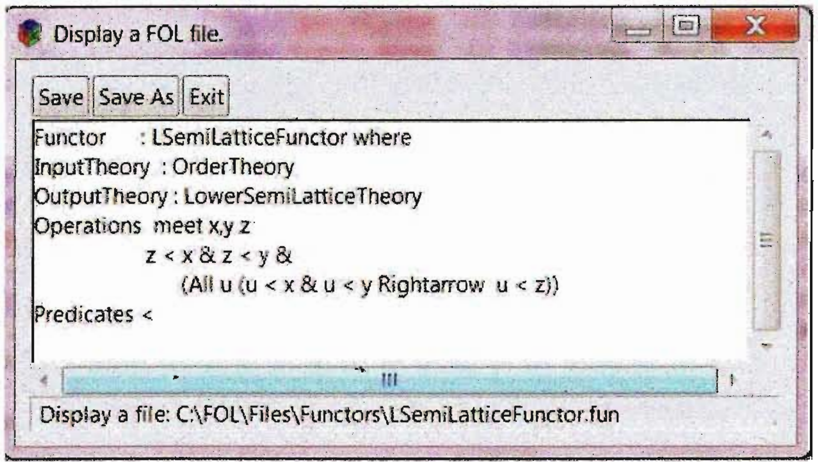
Functors files are stored in the operating system with "fun" extension. The following is the structure of the functor file used in the system.

```
Functor : (Functor name) where
    InputTheory : (The name of input theory)
    OutputTheory : (The name of output theory)
    Operations
        (A list of operations separated by a comma)
          Operations are in the form
            The name of the existence operations in the input model ,
            the name of a new operation in the generated model,
            the free variables of a formula represent that operation,
            the output variable of that formula, and the formula itself
    Predicates
        (A list of predicates separated by a comma)
          Predicates are in the form
            The name of the existence predicates in the input model,
            the name of a new predicate in the generated model,
            the free variables of a formula represent that predicate,
            and the formula itself
```

From the above code, "functor name" denotes the name of the functor. "InputTheory" denotes a theory of a given model. "OutputTheory" denotes a theory of a new model that will be generated by a functor. "Operations" consists of names of the existence operations in the given model separated by a comma. Also, it includes the definition of a new operation. To define a new operation in a functor, the free variables of a formula represent that operation, the output variable of that formula, and the formula itself are needed. In the same way, "predicates" consists of names of the exist predicates in the given model separated by a comma. Also, it includes the definition of a new predicate. To define a new predicate in a functor, the free variables of a formula represent that operation, and the formula itself are needed.

To generate a model of a lower lattice from an order model by a system functor, the given model should satisfy the input theory of the functor and the output model should satisfy the output theory of the functor. Precisely, in addition to the output language, the output theory should contain the order formulas and the greatest lower bound formula.

Figure 3.8 shows a snapshot of a functor used to generate a model of lower semi-lattice from an order model. A new function called meet will be created in the new model. The function details are given in the operations parameter of the functor. The details include the free variables in the formula, the output of the formula, and the formula itself. Note that, the predicates parameter of the functor has "<" symbol, without any new predicates definitions. That means, the elements of the predicate "<" will be copied to the new model without any updates. Also consider the representation of the input theory and the out put theory. The input theory is denoted by "OrderTheory", which is the theory of the order structure model, figure 3.6. Whereas the output theory is denoted by "LowerSemiLatticetheory" theory, which is constructed from the "OrderTheory" theory and the greatest lower bound formula.



```
Functor     : LSemiLatticeFunctor where
InputTheory : OrderTheory
OutputTheory : LowerSemiLatticeTheory
Operations  meet x,y z
            z < x & z < y &
                (All u (u < x & u < y Rightarrow  u < z))
Predicates <
```

Display a file: C:\FOL\Files\Functors\LSemiLatticeFunctor.fun

Figure 3.8: Snapshot of lower semi-lattice functor

The system is ready now to generate the model of lower semilattice. Simply, by selecting the base model, which is the order model in this example, and the functor of lower lattice, a new model will generate by clicking on "Apply functor to a model" function. The new generated model has the same functions and the same predicates of the base model. In addition, a "meet" function will be added to the model functions.

The model of lower semilattice generated by the previous example satisfies the lower semilattice theory just created. In addition, the model satisfies the lower semilattice formula as well as the order formulas.

3.2.3 Lattice

In the previous subsection, we introduced the meet semilattice. Also we found that a meet semilattice is an order set in which every pair of elements has a greatest lower bound. Duality can be developed from the above to produce a join semilattice. A join semillattice is an order set in which every pair of elements has a least upper bound.

Definition 14 *A lattice is an ordered set $(E; \leq)$ which, with respect to its order, is both a meet semilattice and a join semilattice.*

From the above definition a lattice is a poset in which any two elements have a greatest lower bound (meet) and a least upper bound (join); join and meet for any two elements should be unique. The lattice is often denoted by $(E; \wedge, \vee, \leq)$.
It is easy to create a model of a lattice by using the system whenever there is a lower semilattice. In this case, the lattice is a lower semilattice and a join function added to it.
The following steps are used to create a model of a lattice from a lower semilattice. First, we define the language of the lattice. The following script shows this language.

```
Language : LatticeLanguage where
operations
            meet arity: 2 precedence: Infix Level: 1 Assoc assocleft,
            join arity: 2 precedence: Infix Level: 2 Assoc assocleft
predicates
            < arity: 2 precedence: Infix,
            > arity: 2 precedence: Infix
```

Note that, the language is *LatticeLanguage*. Also, we have specified the arity, the precedence, and the associativity for each function of the language. Moreover, we have specified the arity of predicates of the language.

The next step is to create the theory of the lattice. The theory contains the language definition and a set of formulas of an order, meet, and join. We will use the lower semilattice theory which is created in the previous subsection, in addition to the least upper bound formula which is shown below:

$$\forall x, y\ \exists z(\ x < z \wedge y < z \wedge (\forall u\ (x < u \wedge y < u) \rightarrow z < u))$$

The above formula could be implemented in the system and added to the theory of the lattice. The following script shows the join formula implementation in the system.

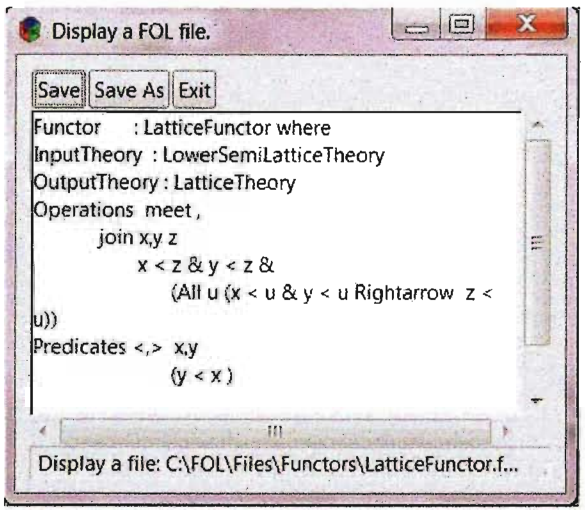
```
All x (
        All y (
                x < (x join y) & y < (x join y) &
                (All u (x < u & y < u Rightarrow (x join y) < u ))
            )
        )
```

In the final step, we use the functor to generate the model of the lattice structure. The functor contains the lower semilattice theory as an input theory, lattice theory as an output theory, and the definitions of the new entities.

Figure 3.9 shows a snapshot of the functor used to generate the model of the lattice. New definition for join function is added to the operations section. Moreover, new definition is added to the predicate section to define the ">" predicate symbol.
The new generated model contains meet and join functions. Moreover, the model contains "<" and ">" as predicates. In addition, the generated model satisfies the lattice theory as well as reflexive, antisymmetric, transitive, meet, and join formulas.

3.2.4 Bounded Lattice

In what follows we introduce the bounded concept in an order set. By way of illustration, if $(E; \leq)$ is an ordered set then by a top element or maximum element of E



```
Functor      : LatticeFunctor where
InputTheory  : LowerSemiLatticeTheory
OutputTheory : LatticeTheory
Operations  meet ,
         join x,y z
               x < z & y < z &
                    (All u (x < u & y < u Rightarrow  z <
u))
Predicates <,> x,y
                    (y < x )
```

Display a file: C:\FOL\Files\Functors\LatticeFunctor.f...

Figure 3.9: Snapshot of a lattice functor

we mean an element $x \in E$ such that $y < x$ for every $y \in E$. The dual notion is that of a bottom element or minimum element, namely an element $z \in E$ such that $z < y$ for every $y \in E$ [2].

A top element and a bottom element formulas are represented by the following:

$$\forall y \; \exists x (y < x)$$
$$\forall y \; \exists z (z < y)$$

To implement a bound lattice in the system based on the structure of Figure 3.1, we will follow the same steps used to generate the lattice structure in the previous subsection.
First, we define the language of the bounded lattice. The following script shows this language.

```
Language : BoundLanguage where
Operations
           meet arity: 2 precedence: Infix Level: 1 Assoc assocleft,
           join arity: 2 precedence: Infix Level: 2 Assoc assocleft
           0 arity: 0,
           1 arity: 0
Predicates
```

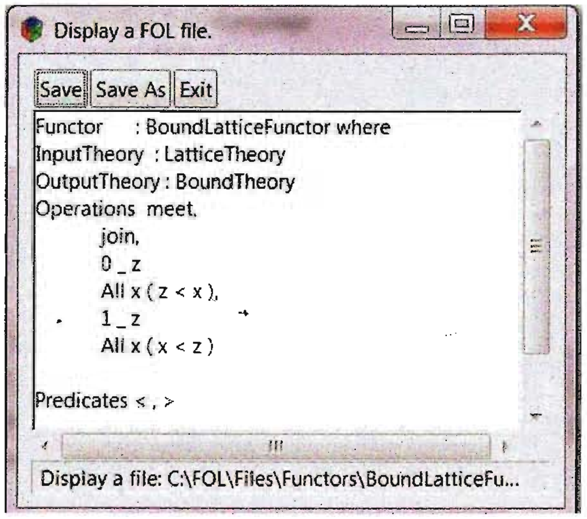
$<$ *arity: 2 precedence: Infix,*
$>$ *arity: 2 precedence: Infix*

From the above script, note that, we keep the same information of meet and join functions. In addition, we add two more functions definitions for the top element and the bottom element of the structure. The arity of the new functions is zero, which means these functions denote constant symbols in the language.
The next step is to create the theory of the bounded lattice. We will reuse the lattice theory file, which is created in the previous subsection to create a new theory file for the bounded lattice and add the top and the bottom formulas to the rest of the new file. The formulas to be added are shown below:

$$\textit{All } x \ (1 > x),$$
$$\textit{All } x \ (0 < x).$$

In the final step, we will use the functor to generate the model of the bounded lattice structure. The functor will contain the lattice theory as an input theory, bounded lattice theory as an output theory, and the definitions of the new entities.

Figure 3.10 shows the functor used to generate a model of bounded lattice. From

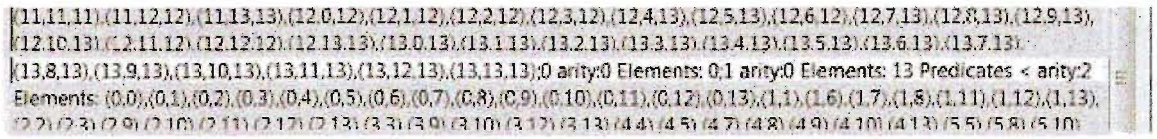


```
Functor      : BoundLatticeFunctor where
InputTheory  : LatticeTheory
OutputTheory : BoundTheory
Operations  meet,
        join,
        0 _ z
        All x ( z < x ),
        1 _ z
        All x ( x < z )

Predicates < , >
```

Display a file: C:\FOL\Files\Functors\BoundLatticeFu...

Figure 3.10: Snapshot of a bounded lattice functror

the figure, note how 0 and 1 functions are defined. Also, note that we did not specify any parameters for these functions. The model of a bounded lattice will be generated by clicking "apply functor to a model" button after selecting the lattice model and the functor which is shown in the above figure.



```
(11,11,11),(11,12,12),(11,13,13),(12,0,12),(12,1,12),(12,2,12),(12,3,12),(12,4,13),(12,5,13),(12,6,12),(12,7,13),(12,8,13),(12,9,13),
(12,10,13),(12,11,12),(12,12,12),(12,13,13),(13,0,13),(13,1,13),(13,2,13),(13,3,13),(13,4,13),(13,5,13),(13,6,13),(13,7,13),
(13,8,13),(13,9,13),(13,10,13),(13,11,13),(13,12,13),(13,13,13);0 arity:0 Elements: 0;1 arity:0 Elements: 13 Predicates < arity:2
Elements: (0,0),(0,1),(0,2),(0,3),(0,4),(0,5),(0,6),(0,7),(0,8),(0,9),(0,10),(0,11),(0,12),(0,13),(1,1),(1,6),(1,7),(1,8),(1,11),(1,12),(1,13),
(2,2),(2,3),(2,9),(2,10),(2,11),(2,12),(2,13),(3,3),(3,9),(3,10),(3,12),(3,13),(4,4),(4,5),(4,7),(4,8),(4,9),(4,10),(4,13),(5,5),(5,8),(5,10)
```

Figure 3.11: Snapshot of a bounded lattice model

Figure 3.11 shows a snapshot of the model generated by the system. Precisely, it shows 0 and 1 constants represented in the model generated. We can check if the bounded model satisfies the bounded theory, by clicking "Check if the model is a model of a theory" button. In addition, we can check if the model satisfies the top and the bottom formulas by clicking "Test a formula" button. The result of performing these functions on the model of the bounded lattice is that, the model satisfies the theory as well as the top and the bottom formulas.

3.2.5 Pseudocomplement Lattice

In what follows we will define a pseudocomplement element in a lattice.

Definition 15 *An element $a^* \in L$ is called a pseudocomplement of a if the following two conditions are satisfied:*

- $a \wedge a^* = 0$;
- $a \wedge x = 0$ $(x \in L)$ *implies* $x \leq a^*$.

Any element of L can have at most one pseudocomplement. We say that L is a pseudocomplemented lattice (or p-algebra) if every element of L has a pseudocomplement, i.e., $a^* = max\{x \in \text{L} | \ a \wedge x = 0\}$. From Definition 15, the following formula denotes a pseudocomplement of an element in a lattice.

$$\forall x \ (\ \exists z : (\forall y : (x \wedge y = 0 \quad \leftrightarrow \quad y < z)))$$

Now we are ready to use the above formula to generate a model of a pseudocomplemented lattice. To implement a pseudocomplemented lattice in the system based on the structure of Figure 3.1, we will follow the same steps used to generate the lattice structure in the previous subsection.

First, we define the language of the pseudocomplemented lattice. The following script shows this language.

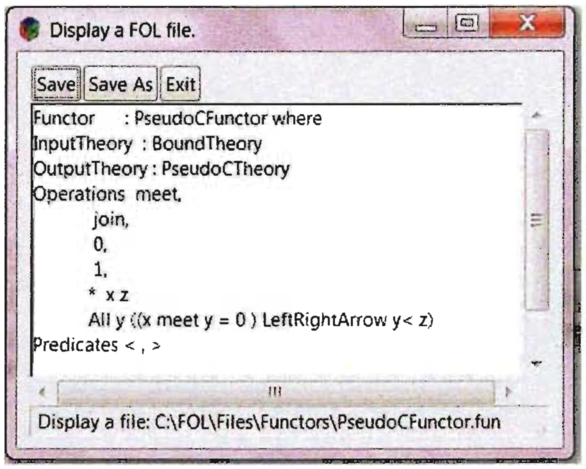
```
Language : PseudoCLanguage where
Operations
            meet arity: 2 precedence: Infix Level: 1 Assoc assocleft,
            join arity: 2 precedence: Infix Level: 2 Assoc assocleft
            0 arity: 0,
            1 arity: 0,
            * arity: 1 precedence : PostFix Level: 3
Predicates
            < arity: 2 precedence: Infix,
            > arity: 2 precedence: Infix
```

From the above script, note that, we keep the same information of meet, join, top, and bottom functions. In addition, we add a definition of an unary function for the pseudocomplement to the structure. The arity of the new functions is one.
The next step is to create the theory of the pseudocomplemented lattice. We will reuse the bounded lattice theory file, which is created in the previous subsection to create a new theory file for the pseudocomplemented lattice and add the pseudocomplement formula to the rest of the new file. The formula to be added is shown below:

$$All\ x\ (\ All\ y\ ((x\ meet\ y = 0\ )\ LeftRightArrow\ y < x^*))$$

In the final step, we will use the functor to generate the model of the pseudocomplemented lattice structure. The functor will contain the bounded lattice theory as an input theory, a pseudocomplemented lattice theory as an output theory, and the definitions of the new entity.

Figure 3.13 shows the functor used to generate a model of pseudocomplemented lattice. From the figure, note how the pseudocomplement function, which is denoted

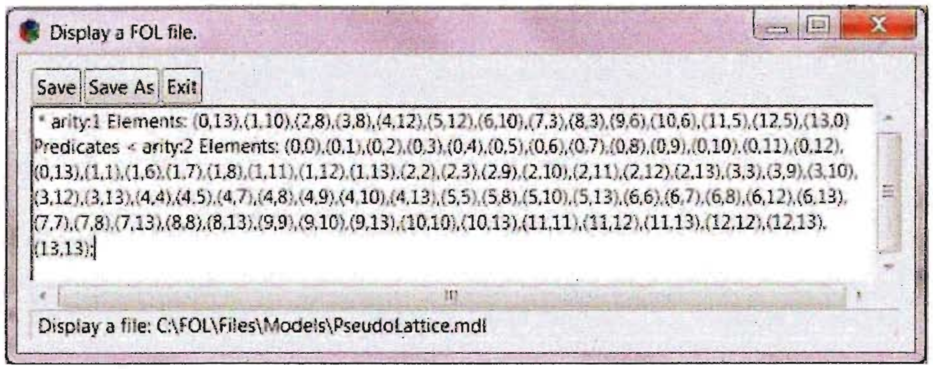


```
Functor     : PseudoCFunctor where
InputTheory : BoundTheory
OutputTheory : PseudoCTheory
Operations meet,
        join,
        0,
        1,
        * x z
        All y ((x meet y = 0 ) LeftRightArrow y< z)
Predicates < , >
```

```
Display a file: C:\FOL\Files\Functors\PseudoCFunctor.fun
```

Figure 3.12: Snapshot of a pseudocomplemented functor

by *, is defined. The model of a pseudocomplemented lattice will be generated by clicking "apply functor to a model" button after selecting the model of lattice and the functor which is shown in the above figure.



```
* arity:1 Elements: (0,13),(1,10),(2,8),(3,8),(4,12),(5,12),(6,10),(7,3),(8,3),(9,6),(10,6),(11,5),(12,5),(13,0)
Predicates < arity:2 Elements: (0,0),(0,1),(0,2),(0,3),(0,4),(0,5),(0,6),(0,7),(0,8),(0,9),(0,10),(0,11),(0,12),
(0,13),(1,1),(1,6),(1,7),(1,8),(1,11),(1,12),(1,13),(2,2),(2,3),(2,9),(2,10),(2,11),(2,12),(2,13),(3,3),(3,9),(3,10),
(3,12),(3,13),(4,4),(4,5),(4,7),(4,8),(4,9),(4,10),(4,13),(5,5),(5,8),(5,10),(5,13),(6,6),(6,7),(6,8),(6,12),(6,13),
(7,7),(7,8),(7,13),(8,8),(8,13),(9,9),(9,10),(9,13),(10,10),(10,13),(11,11),(11,12),(11,13),(12,12),(12,13),
(13,13);
```

```
Display a file: C:\FOL\Files\Models\PseudoLattice.mdl
```

Figure 3.13: Snapshot of a pseudocomplemented lattice model

Figure 3.14 shows snapshot part of the model generated by the system. Precisely, it shows the "*" function which is denotes the unary pseudocomplement function in the generated model.

We can check if the generated model satisfies the pseudocomplement theory, by clicking "Check if the model is a model of a theory" button. In addition, we can check if the model satisfies the formula for pseudocomplements by clicking "Test a formula" button. The result of applying these functions to the model of pseudocomplemented lattice is that, the model satisfies the theory as well as the formula for pseudocomplements.

3.2.6 Quasicomplemented lattice

The notion of a quasicomplement a^+ of a is dual to the notion of a pseudocomplement, i.e. it is characterized by $a^+ \leq x \Leftrightarrow a \vee x = 1$. A quasicomplemented lattice is a lattice in which every element has a quasicomplement, i.e. the dual of a pseudocomplemented lattice [8].

Based on the above, the following formula represents a quasicomplement of an element in a lattice.

$$\forall x : (\exists z : (\forall y : (x \vee y = 1 \quad \Leftrightarrow \quad z < y)))$$

Now we are ready to use the above formula to generate a model of quasicomplement lattice using the system. To implement a quasicomplemented lattice in the system based on the structure of Figure 3.1, we will follow the same steps used to generate the lattice structure in the previous subsection.

First, we define the language of the quasicomplemented lattice. The following script shows this language:

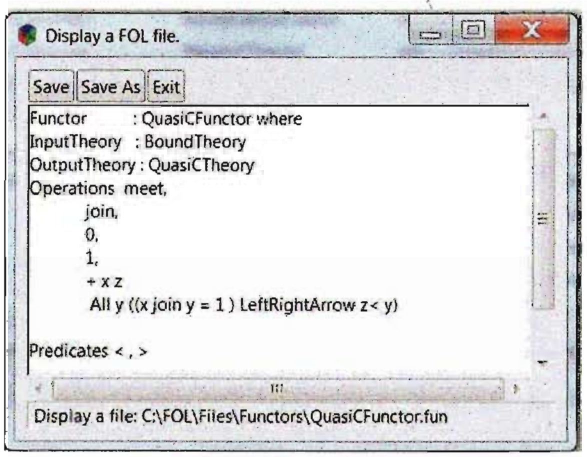
Language : QuasiCLanguage where
Operations
    meet arity: 2 precedence: Infix Level: 1 Assoc assocleft,
    join arity: 2 precedence: Infix Level: 2 Assoc assocleft
    0 arity: 0,
    1 arity: 0,
    + arity: 1 precedence : PreFix Level: 3
Predicates
    < arity: 2 precedence: Infix,
    > arity: 2 precedence: Infix

From the above script, note that we keep the same information of meet, join, top, and bottom functions. In addition, we add one more function definition for the quasicomplement function.

The next step is to create the theory of the quasicomplemented lattice. We will reuse the bounded lattice theory file, which is created in the previous subsection to create a new theory file for the quasicomplemented lattice and add the formula for quasicomplement to the rest of the new file. The formula to be added is shown below:

$$All\ x\ (\ All\ y\ ((x\ join\ y = 1\ )\ LeftRightArrow\ +x < y))$$

In the final step, we will use the functor to generate the model of the quasicomplemented lattice structure. The functor will contain the bounded lattice theory as an input theory, a quasicomplemented lattice theory as an output theory, and the definitions of the new entity.

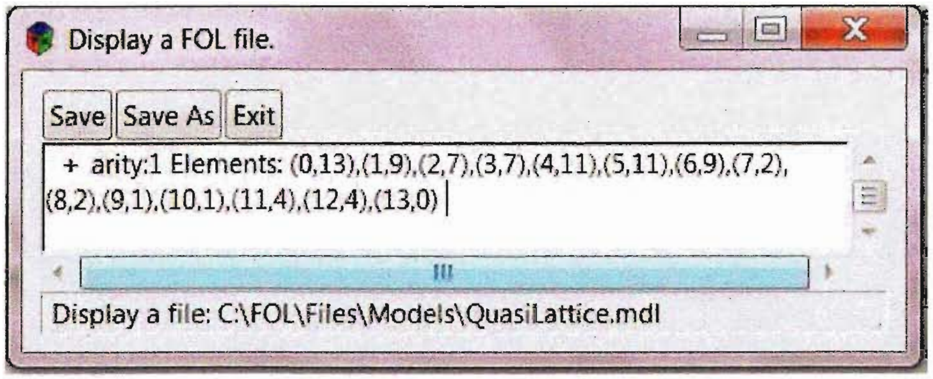


```
Functor       : QuasiCFunctor where
InputTheory   : BoundTheory
OutputTheory : QuasiCTheory
Operations  meet,
          join,
          0,
          1,
          + x z
           All y ((x join y = 1 ) LeftRightArrow z< y)

Predicates < , >
```

Display a file: C:\FOL\Files\Functors\QuasiCFunctor.fun

Figure 3.14: Snapshot of a quasicomplemented functor

Figure 3.15 shows the functor used to generate the model of quasicomplemented lattice. From the figure, note how the quasicomplement function,which is denoted by "+", is defined. The model of a quasicomplemented lattice will be generated by clicking the "Apply functor to a model" function after selecting the model of lattice and the functor which is shown in the above figure.
Figure 3.16 shows a snapshot of the model generated by the system. Precisely, it shows "+" function which denotes the unary quasicomplement function in the generated model. We can check if the generated model satisfies the quasicomplement



+ arity:1 Elements: (0,13),(1,9),(2,7),(3,7),(4,11),(5,11),(6,9),(7,2),
(8,2),(9,1),(10,1),(11,4),(12,4),(13,0)

Display a file: C:\FOL\Files\Models\QuasiLattice.mdl

Figure 3.15: Snapshot of a quasicomplemented lattice model

theory, by clicking the "Check if the model is a model of a theory" button. In addition, we can check if the model satisfies the pseudocomplement formula by clicking "Test a formula" button. The result of applying these functions to the model of quasicomplement lattice is that, the model satisfies the theory as well as the quasicomplement formula.

3.2.7 Orthocomplemented Lattice

An orthocomplementation on a lattice is an involution which is order-reversing and maps each element to a complement.

Definition 16 [8] *An ortholattice (or orthocomplemented lattice) is a structure* $\langle L, \vee, \wedge, ^{\perp}, 0, 1\rangle$ *such that*

1. $\langle L, \vee, \wedge, 0, 1\rangle$ is a bounded lattice,

2. $a^{\perp}$ is an orthocomplement of a, i.e. for all $a, b \in L$ we have

   - $a^{\perp\perp} = a$,
   - $a \wedge a^{\perp} = 0$,
   - $a \leq b$ *implies* $b^{\perp} \leq a^{\perp}$

An orthocomplementation on a bounded lattice is a function that maps each element a to an "orthocomplement" $a^{\perp}$ in such a way that the pervious axioms are satisfied.

Now we are ready to use the above formulas to generate a model of orthocomplemented lattice using the system. To implement an orthocomplemented lattice in the system based on the structure of Figure 3.1, we will follow the same steps used to generate the quasicomplemented lattice structure in the previous subsection.

First, we define the language of the orthocomplemented lattice. The following script shows this language:

```
Language : OrthoComplementLanguage where
Operations
          meet arity: 2 precedence: Infix Level: 1 Assoc assocleft,
          join arity: 2 precedence: Infix Level: 2 Assoc assocleft,
          0 arity: 0,
          1 arity: 0,
          oc arity: 1 precedence : PreFix Level: 4
Predicates
          < arity: 2 precedence: Infix,
          > arity: 2 precedence: Infix
```

From the above script, note that we keep the same information of meet, join, top, and bottom functions. In addition, we add one more function definition for the orthocomplementation. The arity of the new functions is one.

The next step is to create the theory of the orthocomplemented lattice. We will reuse the bounded lattice theory file, which is created in the previous subsection to create a new theory file for the orthocomplemented lattice and add the formulas of orthocomplement to the rest of the new file. The formulas to be added are shown below:

$$
\begin{gathered}
All\ x\ (oc(oc\ x)) = x\ , \\
All\ x\ (All\ y\ (x < y\ Rightarrow\ (oc\ y < oc\ x))), \\
All\ x\ (x \text{ meet } (oc\ x) = 0).
\end{gathered}
$$

Notice that we use a different approach here. In the previous examples all new concepts such as join, meet, least and greatest element, pseudocomplement, and quasicomplement the corresponding elements are unique if they exist. Therefore, the formulas used an existential quantifier. Orthocomplements are not necessarily unique so that the formulas use the function "oc" provided by the user within a model.

Because orthocomplements are not necessarily unique, we will not use a functor to generate the model of orthocomplement lattice. Instead, we will create an orthocomplement model file based on the bounded lattice model file created earlier. The following function will be added to the file:

oc arity:1 elements: (0,13),(1,10),(2,8),(3,7),(4,12),(5,11),(6,9),(7,3),(8,2),(9,6),(10,1),(11,5),(12,4),(13,0)

The above script shows the "oc" function which denotes an unary orthocomplement function in the model.

We can check if the generated model satisfies the orthocomplement theory, by clicking the "Check if the model is a model of a theory" button. In addition, we can check if the model satisfies the formulas of orthocomplement by clicking the "Test a formula" button. The result of performing these functions on the orthocomplemented lattice model is that, the model satisfies the theory as well as the formulas of orthocomplement.

3.2.8 p-Ortholattices

A lattice that is both pseudocomplemented and orthocomplemented is called a p-ortholattice as the following definition declares:

Definition 17 [8]. *A pseudocomplemented ortholattice (or p-ortholattice) is a structure* $\langle L, \vee, \wedge, ^*, ^\perp, 0, 1\rangle$ *such that*

- $\langle L, \vee, \wedge, ^*, 0, 1\rangle$ *is a p-algebra,*
- $\langle L, \vee, \wedge, ^\perp, 0, 1\rangle$ *an ortholattice.*

The creation of a model of p-ortholattices will be based on the model of orthocomplemenet lattice which was generated in the previous subsection. To accomplish this we will follow the same steps used to generate a model in the system.

First, we define the language of the p-ortholattices, the following script shows this language:

```
Language : PorthoLanguage where
Operations
          meet arity: 2 precedence: Infix Level: 1 Assoc assocleft,
          join arity: 2 precedence: Infix Level: 2 Assoc assocleft
          0 arity: 0,
          1 arity: 0,
          oc arity: 1 precedence : PostFix Level: 3
          * arity:1 precedence : PostFix Level:4
Predicates
          < arity: 2 precedence: Infix,
          > arity: 2 precedence: Infix
```

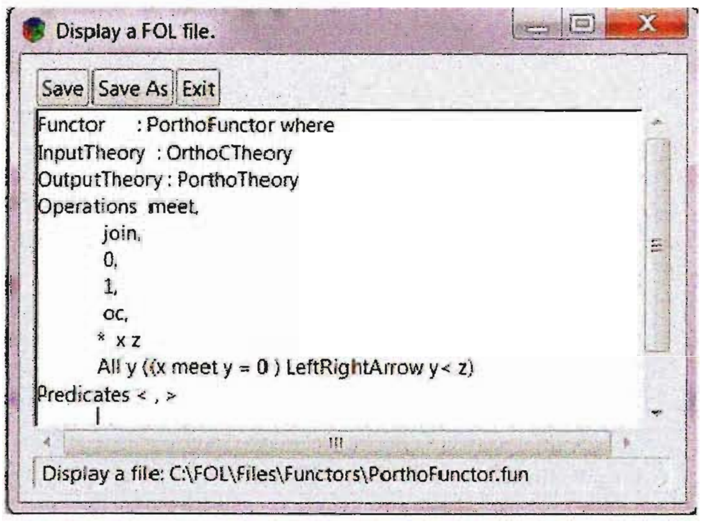
From the above script, note that, we keep the same information of meet, join, top, bottom, and orthocomplement functions. In addition, we add one more function definition for the pseudocomplemented function. As a whole these functions denote the p-ortholattices definition.

The next step is to create the theory of the p-ortholattices. We will reuse the p-orthocomplement lattice theory file, which is created in the previous subsection to create a new theory file for the p-ortholattices and add the pseudocomplemented formula to the rest of the new file. The formula to be added is shown below:

$$All\ x\ (\ All\ y\ ((x\ meet\ y = 0\ )\ LeftRightArrow\ y < x^*))$$

In the final step, we will use a functor to generate the model of the p-ortholattices structure. The functor will contain the orthocomplemented lattice theory as an input theory, a p-ortholattices theory as an output theory, and the definitions of the new entity.

Figure 3.17 shows the functor used to generate a model of p-ortholattices. The model of a p-ortholattices will be generated by clicking the "Apply functor to a model" button after selecting the orthocomplement model and the functor which is shown in the above figure.



```
Functor      : PorthoFunctor where
InputTheory  : OrthoCTheory
OutputTheory : PorthoTheory
Operations  meet,
        join,
        0,
        1,
        oc,
        * x z
        All y ((x meet y = 0 ) LeftRightArrow y< z)
Predicates < , >
```

Display a file: C:\FOL\Files\Functors\PorthoFunctor.fun

Figure 3.16: Snapshot of a p-ortholattices functor

We can check if the model of p-ortholattices satisfies the p-ortholattices theory, by clicking "Check if the model is a model of a theory" button. The result of performing this function to the p-ortholattices model is that, the model satisfies the theory of the p-ortholattices.

3.2.9 A Stonian p-ortholattice

In this section we check if the model of p-ortholattice was created in the previous subsection is a Stonian p-ortholattice or not.

Definition 18 [8] *A Stonian p-ortholattice is a structure* $\langle L, \vee, \wedge, ^*, ^\perp, 0, 1\rangle$ *such that*

- $\langle L, \vee, \wedge, ^*, 0, 1\rangle$ *is a p-algebra,*
- $\langle L, \vee, \wedge, ^\perp, 0, 1\rangle$ *an ortholattice.*
- $(a \wedge b)^* = a^* \vee b^*$ *holds for all* $a, b \in L$.

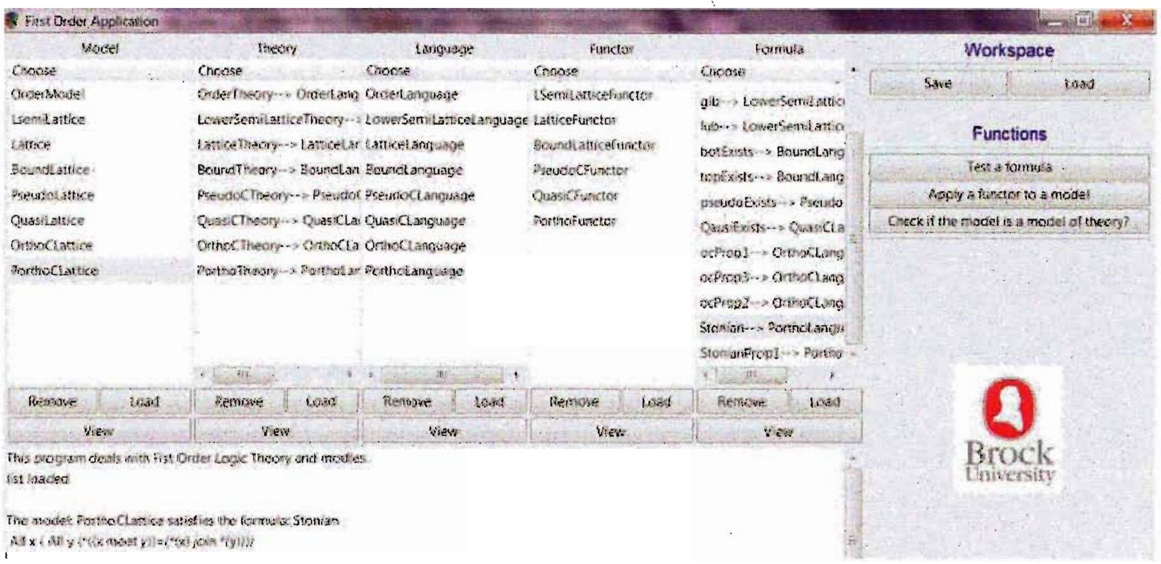
From the first two axioms of Definition 18 note that a Stonian p-ortholattice is a p-ortholattice. In addition, it should satisfy the Stone identity, which is the third axiom of the definition. As mentioned at the beginning of this subsection we want to check if a p-ortholattice is a Stonian p-ortholattice. To accomplish this check, we have to load the Stone identity formula file into the system. The file looks like this:

All x (All y ((x meet y) * = (x *) join (y *)))

The Next step is to check if the model of p-ortholattice satisfies the above formula; this can be done by using "Test a formula" button. The result is shown in Figure 3.18. The result shows that, the p-ortholattice model satisfies the Stone identity formula. Hence, the model of p-ortholattice is a model of Stonian p-ortholattice.

Another property we want to check regarding the model of p-ortholattice is the following:

$$x^{\perp} \leq y = (x \wedge y)^{**} \vee x^{**} \wedge y^{**}$$



```
This program deals with Fist Order Logic Theory and modles.
fist loaded

The model: PorthoCLattice satisfies the formula: Stonian
 All x ( All y (*((x meet y))=(*(x) join *(y))))
```

Figure 3.17: Snapshot of a p-ortholattices workspace

To accomplish this check, we have to load the previous formula file into the system. The file looks like this:

All x (
    All y (
        (oc $x < y$) LeftRightArrow x meet $((y^*)^*)$ join $((x^*)^*)$

```
        ) meet y = (($x^*$)$^*$) meet (($y^*$)$^*$)
        )
    )
```

The next step is to check if the model of p-ortholattice satisfies the above formula; this can be done by using "Test a formula" button. The result shows that, the model of p-ortholattice does not satisfy the above formula.

Chapter 4

The Technical Specification

This chapter gives a brief overview of the main modules of the system. In addition, it gives details on how to use the system.

4.1 Overview of the System

As mentioned before, the purpose of the system is to find counterexamples and testing properties of models of first-order theories. To implement first order theory concept in the system, various modules are used. Some of the important modules are: language module, formula module, term module, theory module, model module, and functor module. In the following, we will discuss the data type of these modules.

In the file "LanguageModule.hs" the following data type is defined:

$$\begin{aligned} \text{data Language} &= \mathit{Lan} \ \{\mathit{langName} :: \mathit{String}, \\ &\qquad \mathit{opLang} :: [(\mathit{String}, \mathit{Int}, \mathit{Int})], \\ &\qquad \mathit{predLang} :: [(\mathit{String}, \mathit{Int}, \mathit{Int})] \\ &\qquad \}\mathit{deriving}(\mathit{Eq}, \mathit{Read}) \end{aligned}$$

The above script shows the language data type defined in the system, where "langName" denotes a language name, "opLang" consists of a list of operations symbols, and "predLang" consists of a list of predicates symbols. Each tuple of the operation symbols list denotes the operation name, the arity of that operation, and the precedence of the operation order. Each tuple of the predicate symbols list consists the predicate name, the arity of that predicate, and the logical notation of the predicate. The logical notation could be infix, prefix, or postfix.

Next, we want to define the formula and the term data type. But before that, consider the following productions for valid FOL formulas used in the system:

$$\begin{array}{lcl}
\text{Formula} & \longrightarrow & AtomicFormula \\
 & | & FormulaConnectives \\
 & | & QuantifierVariable, \ldots Formula \\
 & | & \neg Formula \\
 & | & (Sentence) \\
\text{AtomicFormula} & \longrightarrow & Predicate(Term, \ldots) \\
 & | & Term = Term \\
\text{Term} & \longrightarrow & Function(Term) \\
 & | & Constant \\
 & | & Variable \\
\text{Connective} & \longrightarrow & And, Or, Implies, Equiv \\
\text{Quantifier} & | & \forall, \exists \\
\text{Constant} & \longrightarrow & X \mid Y \mid RED \mid \ldots \\
\text{Varibale} & \longrightarrow & x \mid y \mid z \mid \ldots \\
\text{Predicate} & \longrightarrow & Greaterthan \mid Lessthat \mid isBlue \mid \ldots \\
\text{Function} & \longrightarrow & Father \mid Sibling \mid \ldots
\end{array}$$

In the file "FormulaModule.hs", the following formula data type is defined:

$$\begin{array}{lcl}
\text{data Formula} & = & Equal\ Term\ Term \\
 & | & Pred\ String\ Int\ [Term] \\
 & | & And\ Formula\ Formula \\
 & | & Or\ Formula\ Formula \\
 & | & Implies\ Formula\ Formula \\
 & | & Not\ Formula \\
 & | & All\ String\ Formula \\
 & | & Exists\ String\ Formula
\end{array}$$

The representation of a formula in the system is based on the syntax of first-order logic, in particular, Definition 2. The equality of two terms, as well as predicate symbol with terms as parameters denote a formula. Hence, we want to define a term in the system. In the file "TermModule.hs", the following term data type is defined:

$$\begin{aligned} \text{data Term} &= Var\ String \\ &\mid\ OP\ String\ Int\ Int\ [Term] \end{aligned}$$

A theory is needed to be loaded in the system. In the file "TheoryModule.hs", the following theory data type is defined:

$$\begin{aligned} \text{data Theory} = Thry\ & \{thryName :: String, \\ & thryLang :: Language, \\ & thryFormulas :: [Formula] \\ & \} \end{aligned}$$

The above code is based on Definition 9, where the theory is a set of sentences in first-order language. Hence, we used "thryName" to denote the name of the theory, "thryLang" to reference to a pre-defined language in the system, and "thryFormulas" to reference a list of formulas.

In the file "ModelModule.hs" the following model data type is defined:

$$\begin{aligned} \text{data Model} = Modl\ & \{name :: String, \\ & numElements :: Int, \\ & operations :: [(String, Int, S.Set([Int], Int))], \\ & predicates :: [(String, Int, S.Set([Int]))], \\ & \}\ deriving(Eq, Read) \end{aligned}$$

From the above script, "name" consists a model name, "numElements" consist of the number of elements in a model, "operations" consists of the operations of a model, and "predicates" consists of the predicates of a model.

In the file "FunctorModule.hs" the following model data types is defined:

```
data TFunctor  = Func  {funcName :: String,
                        inputTheory :: Theory,
                        outputTheory :: Theory,
                        outputOperations :: [(String, Maybe
                                             ([String], String, Formula))],
                        outputPredicates :: [(String, Maybe
                                             ([String], Formula))],
                       }
```

4.2 The Parser

We used Parsec, a fast combinator parser written in Haskell in order to parse various inputs within our system. We have implemented a parser for each kind of system file, i.e., for model files, theory files, language files, functors files, and formula files. Some forms of calling those parsers is as follows:

```
langExpr :: GenParser PosToken (M.Map String Theory,M.Map String Language) Language
termExpr :: Language → GenParser PosToken (M.Map String Theory,M.Map String Language) Term
formulaExpr :: Language → GenParser PosToken (M.Map String Theory,M.Map String Language) Formula
```

From the above script, the "lanExpr" function is used to generate a language data type. This function uses the general parser type "GenParser". GenParser token state a is the type of all parsers that parse tokens as an input, utilizing a user defined state, and producing outputs of type a. We use lexical analyzer or scanner to avoid whitespace in the input string, and to convert the input string into a list of tokens. We use "(M.Map String Theory,M.Map String Language)" to define a user state of the parser. A state in Parsec is very useful when you want to keep track of parsed variables. Hence, a user state is defined in the system to keep track of parsed languages and theories in the system. The importance of using a user state in a parser appears in referring the name of a parsed variable instead of parsed it again which lead to improve the efficiency of the system.

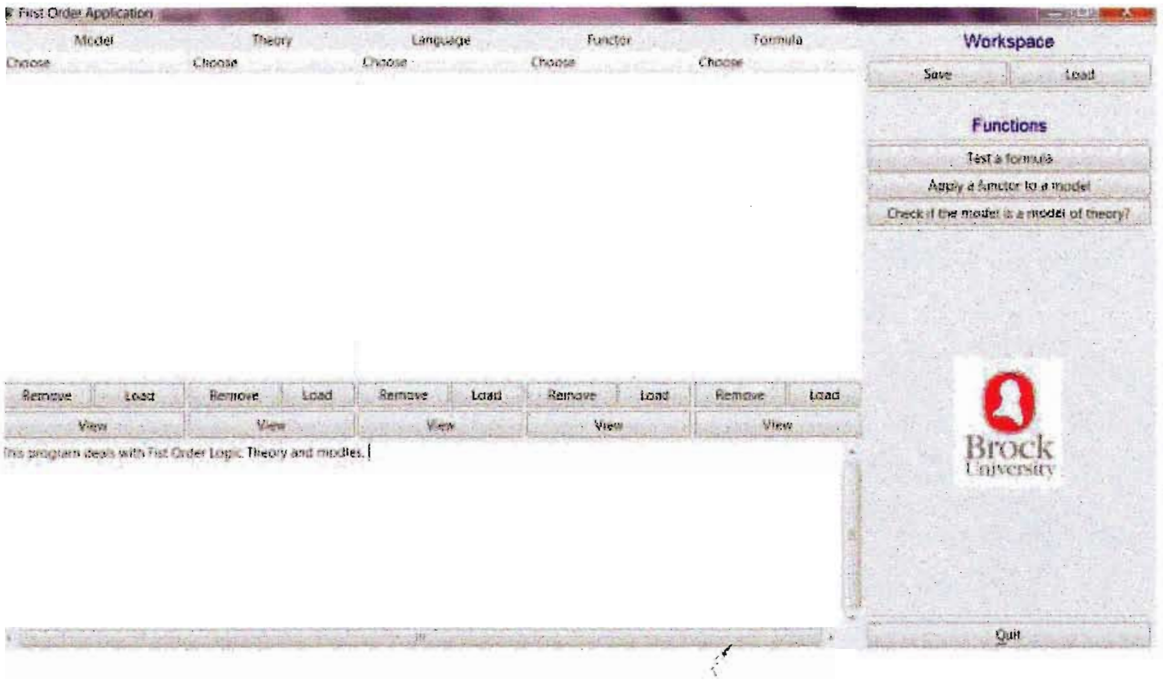
As mentioned before, the theory uses the language data type as one of its data type parameter. In addition, the functor uses the theory data type as one of its data type. Hence, the parser state is used to keep track of a parsed theory and a parsed language

in the system to use them later in the theory parser and in the functor parser.
Also, note that the "termExpr" as well as the "formulaExpr" functions take a language as an input with the "GenParser" type to generate term and formula data types respectively. In addition, both functions use "buildExpressionParser" function to build an expression parser for terms term with operators from an operator table, taking the associativity and precedence specified in that table into account. An operator table is a list of operators list. The list is ordered in descending precedence. In addition, an operator in a table is either binary infix or unary prefix or postfix and all operators in one list have the same precedence (but may have a different associativity). For more detail about the Parsec library refer to [6].

4.3 A Manual for the System

Figure 4.1 shows a snapshot of the system. The system consists of three main parts: The first part consists of the loaded lists of models, theories, languages, functors, and formulas. Each list has a set of function buttons attached to it. The function buttons have the "Remove" button which is used to remove an element from a list. In addition, the "Load" button is used to load an element from the system files into a list, and the "View" button is used to view the contents of an element of a list. The second part consists of the workspace buttons and a set of functions. By workspace we mean the elements of all the loaded lists. The "Save" button in the workspace is used to save the current workspace, while the "Load" button is used to load a previously saved workspace. The set of functions are used to handle the system main functions. For example, "Test a formula" button is used to check if a model satisfies a specific formula. "Apply functor to a model" button is used to generate a new model based on another model. "Check if the model is a model of a theory" button is used to check if a model satisfies a theory or not. The third part displays messages generated by the system to the user.

To load a new file into the system two steps are needed. The first step is to create a file with appropriate extension. This file contains an input string of an element to be loaded. The element could be a model, a theory, a language, a functor, or a formula. A model file has a "mdl" extension, a theory file has a "thy" extension, a language file has a "lng" extension, a functor file has a "fun" extension, and a formula



This program deals with Fist Order Logic Theory and modles.

Figure 4.1: Snapshot of a p-ortholattices workspace

file has a "frm" extension. The next step is to use the load button to load a saved file. For example, to load a language file, we click on the load button in the language list. The language is loaded into the system, precisely, into the language list after the system successfully parse the entire file. If the parser failed to parse a file, an appropriate message is displayed in the messages area of the system.

In the system, we use the concept of dependency. A formula depends on a language. Hence, the system gives us the ability to choose a language before we load a formula. Other forms of dependency are found between a theory and a language, and between a functor and a theory. I.e., to load a theory into the system with a specific language name, that language should be loaded before loading a theory.
If you want to check a formula regarding a model, then every quantifier is basically a loop through all elements of that model. the worst case for exists quantifier . Therefore, the polynomial time where the degree is given by the number of quantifiers exist in a formula.

4.4 Tools Used

The system has been developed by Haskell functional programming language. More specifically, GHCI version 6.10.3. Whereas, the GUI has been developed by Glade version 3.6.1. Glade is a user interface designer for GTK+. GTK+ is a multi-platform toolkit for creating graphical user interfaces.

Chapter 5

Conclusions

In this chapter, we briefly review the content of the thesis. Then we discuss extensions to our work.

5.1 Summary

Counterexample is an example that is used to disprove a statement or a theory. This method is useful for students who do not know the steps to prove a statement or a theory. In addition, this method helps researchers who know the proven methods by saving their time. In this thesis, we present an implementation of a system for finding counterexamples and testing properties of models of first-order theories.

We started by introducing the concept of first-order logic. We provided the syntax and semantics of the first-order language. The language has three main types of components: variables, a set of predicate symbols, and a set of operation symbols. Variables and operation symbols are used to build terms. Terms, predicate symbols, and the usual logical connectives are used to build formulas. Then we introduced a first order theory. A first-order theory consists of a language together with a set of closed formulas, i.e., formulas without free occurrences of variables. We also introduced a model, where it consists of a non-empty set of elements, called the universe, together with interpretations for the components of a language. Then, we introduced the notion of a functor to generate more complex models from given ones. As an example, we showed how the system created several lattice structures and test their properties. Finally we gave a brief overview about types of the system.

The example of Figure 3.1 was an excellent example to test the system due to its size and complexity. The system was capable of visualize the structure of the mentioned example. Moreover, the system was capable of creating several complex structures, and testing these structures against several theories and formulas.

The system has three key features. First, the system uses several different formats to allow the user to specify languages, to define axioms and theories and to create models. Second, the system is capable of saving and loading models, theories, languages, and formulas defined by the user. Third, the whole workspace can be saved and loaded by the user.

5.2 Future Work

In its current state, the system for models of first-order theories is already a useful tool and works as designed. However, there is always room for future work.

An outstanding issue to consider is to provide more features to define a functor. As an example, transitive closure cannot be expressed in first-order logic. Hence, the system cannot create or represent models having transitive closure properties. This can be solved by using second order logic, where the syntax and semantics of first-order logics can be extended by means for quantification over sets of elements in the universe. Another solution is by providing a Haskell function to implement the transitive closure and allow the user to apply such a function to a binary predicate within a functor.

Another issue is to add new logical operators. For example, the uniqueness quantification $\exists!\ x : \varphi$. This means "exists exactly one x such that φ is true". Uniqueness quantification can be expressed in terms of the existential and universal quantifiers of first-order logic by defining the above formula as follows:

$$\exists x : (\varphi \wedge \forall y : \varphi\ [y/x] \rightarrow x = y)$$

Bibliography

[1] Asperti A., Longo G.: *Categories Types and Structures.* Foundation of Computing Series MIT Press (1991).

[2] Blyrh T. S.: *Lattices and Ordered Algebraic Structures.* Springer-Verlag London (2005).

[3] Enderton H.B.: *Mathematical Introduction to Logic (2nd edition).* Hartcourt Academic Press (2001).

[4] Gelbaum B., Olmsled J.: *Counterexamples in Analysis.* Holden-day Inc. (1964).

[5] Hahmann T., Winter M., Gruninger M.: *Stonian p-Ortholattices: A new approach to the mereotopology RT_0.* Artificial Intelligence 173(15), pp. 1424-1440 (2009).

[6] Leijen D.: *Parsec, a fast combinator parser.* University of Utrecht. (2001).

[7] Margaris A.: *First Order Mathematical Logic.* Blaisdell Publishing Company. (1967).

[8] Steen L., Seebach J., Jr.: *Conuterexamples in Topology.* Holt, Rinehart and Winston, Inc. (1970).

[9] "Transitive closure." Wikipedia. 21 March 2011. Wikimedia Foundation. Inc.. 02 July 2011 $\langle http://en.wikipedia.org/wiki/Transitive\_closure\rangle$

[10] Winter M.: *Logic in Computer Science Lecture Material,* Brock University.