RELATIONAL MODEL FOR PROGRAM SEMANTICS

A thesis submitted
to Kent State University in
partial fulfillment of the requirements
for the degree of Master of Science

by
Pradeep Kumar Punnam

December, 2008
From the beginning of the development of computer programming languages, computer scientists and mathematicians have been developing methodologies to describe the programming languages. They use mathematics as a vehicle to describe and understand the properties of a language. Different methods of program semantic representations have been proposed to capture different aspects of programming languages. Each serves a different purpose in the understanding of a programming language. Axiomatic, denotational and operational semantics are the well known and often used methods. Axiomatic semantics is well suited for collecting the requirements for programs, and denotational semantics is useful in collecting the requirements and also proving the correctness of programs. Operational semantics gives meaning to programs by showing how programs are executed in an abstract machine; this gives essential insight into the efficiency of the language.

In this thesis we develop a relational model that uses binary relations to describe computational states and programs. The relational view of specifying the programs gives us an ability to better understand the properties of programming languages using the existing mathematical theory of relations. We define orderings on states and programs using Hoare and Smyth orderings, and we define refinement and non-determinism and give their properties. We also describe some program primitives and operations and prove properties about them using our relational model.
Thesis written by

Pradeep Kumar Punnam

B.Tech., J.N.T. University, 2003

M.S., Kent State University, 2008

Approved by

Dr. Austin Melton, Advisor

Dr. Robert Walker, Chair, Department of Computer Science

Dr. John Stalvey, Dean, College of Arts and Sciences
# TABLE OF CONTENTS

Acknowledgements ................................................................. vi

1 Introduction ........................................................................ 1

2 Background Concepts .......................................................... 3
   2.1 Set Theory ....................................................................... 3
   2.2 Relations ....................................................................... 4
   2.3 Orderings on Domains ..................................................... 6

3 Related Work ........................................................................ 8
   3.1 Syntax and Semantics ..................................................... 8
   3.2 Axiomatic Semantics: ................................................... 11
   3.3 Denotational Semantics ................................................ 11
   3.4 Operational Semantics .................................................. 13

4 Relational Model for Program Semantics ................................. 17
   4.1 Relational Model ........................................................... 17
      4.1.1 Variables and Values ............................................. 18
      4.1.2 States .................................................................. 19
      4.1.3 Programs ............................................................. 21
   4.2 Hoare’s Properties of Programs ....................................... 25
   4.3 Properties in Relational Model ........................................ 27
Acknowledgements

I would like to thank Dr. Austin Melton for his guidance and encouragement in this research work. Without his constant support, this work would not have been possible. I would like to thank my friends and family for their support during the course of my master’s work.
CHAPTER 1

Introduction

Program semantic analysis enables the programming language designers, developers and analysts a systematic method to analyze and develop languages. Describing the properties of a language gives the user a better perspective on language usage. From the beginning of the development of computer programming languages, computer scientists and mathematicians have been developing methodologies to describe the programming languages. They use mathematics as a vehicle to describe and understand the properties of a language.

Different methods of program semantic representations have been proposed to capture different aspects of programming languages. Each serves a different purpose in the understanding of a programming language. Axiomatic, denotational and operational semantics are the well known and often used methods. Axiomatic semantics is well suited for collecting the requirements for programs, and denotational semantics is useful in collecting the requirements and also proving the correctness of programs. Operational semantics gives meaning to programs by showing how programs are executed in an abstract machine; this gives essential insight into the efficiency of the language.

In this thesis we develop a relational model that uses binary relations to describe computational states and programs. The relational view of specifying the programs gives us an ability to better understand the properties of programming languages using the existing mathematical theory of relations.
Extensive research is going on in proving equivalences between different methodologies of specifying the semantics. C.A.R. Hoare et al. [1] tried to show how algebraic style of semantics can be used to derive operational semantics. And Robin Milner [2] showed how we can use operational semantics to get algebraic semantics. We believe that the relational model is perfectly suited to prove the equivalence between different semantics. We define refinement and step relations on programs to give an operational semantics view using relations.

Modeling non-determinism has been a difficult problem for long time, and traditional semantic methodologies are not suitable in specifying and understanding their properties. Different semantics used special notations to represent non determinism. But relations intuitively model non-determinism in them selves. Relations enable us to better describe and understand the non-deterministic properties of the language. K. Rustan M. Leino et al. [3] presents a way of reasoning about the secure information flow using the weakest precondition calculus and relations. Relational model with operational view will give us better understanding of the informational flow to reason about the security in programs.

In the following chapter we are going to introduce some of the background concepts on relations, and in chapter 3 we are going to present introductions on syntax, semantics and three semantic methodologies (axiomatic, denotational and operational semantics). We will introduce the relational model in the chapter 4 by defining variables, states and programs, and provide some of the properties using Hoare and Smyth orderings. We also present some of the primitives and properties of programming languages given by Hoare et al., and prove them in the relational model. At the end we define refinement and non-determinism and provide properties using the relational model.
CHAPTER 2

Background Concepts

2.1 Set Theory

A set can be viewed as a collection of objects, the elements or members of the set. We write \( a \in X \) when \( a \) is an element of the set \( X \).

Types of sets:

- **Empty Set:** A set which has no elements is called the empty set. More formally, the empty set, denoted by \( \emptyset \), is a set that satisfies the following: \( \forall x \ x \not\in \emptyset \).

- **Universal Set:** A set which has all the elements in the universe of discourse is called a universal set. More formally, a *universal set*, denoted by \( U \), is a set that satisfies the following: \( \forall x \ x \in U \).

- **Subset:** A set \( X \) is said to be a subset of a set \( Y \), written \( X \subseteq Y \) iff every element of \( X \) is an element of \( Y \), i.e., \( X \subseteq Y \iff \forall z \in X.z \in Y \).

- **Superset:** A set \( X \) is said to be a superset of a set \( Y \), written \( X \supseteq Y \) iff every element of \( Y \) is an element of \( X \), i.e., \( X \supseteq Y \iff \forall z \in Y.z \in X \).

- **Equality of sets:** Two sets are equal if and only if they have the same elements. More formally, for any sets \( A \) and \( B \), \( A = B \) if and only if \( \forall x [x \in A \leftrightarrow x \in B] \).

**Constructions on sets:**

A new set can be constructed from two or more sets using the following operations.
• *Comprehension:* if $X$ is a set and $P(x)$ is property, we can form the set $\{x \in X \mid P(x)\}$ which is another way of writing $\{x \mid x \in X \land P(x)\}$. This is the subset of $X$ consisting of all elements $x$ of $X$ which satisfy $P(x)$.

• *Powerset:* The set of all subsets of a set $X$ is called the power set of $X$ and denoted by $\text{Pow}(X)$. $\text{Pow}(X) = \{Y \mid Y \subseteq X\}$

• *Union:* The set consisting of the union of two sets has as elements those elements which are either elements of one set or the other set or both. It is written and described by: $X \cup Y = \{a \mid a \in X \lor a \in Y\}$

• *Big Union:* Let $X$ be a set of sets. Their union $\bigcup X = \{a \mid \exists x \in X. a \in x\}$.

• *Intersection:* Elements are in the intersection $X \cap Y$, of two sets $X$ and $Y$, iff they are in both sets, i.e. $X \cap Y = \{a \mid a \in X \land a \in Y\}$

• *Big Intersection:* Let $X$ be a nonempty set of sets, then $\bigcap X = \{a \mid \forall x \in X. a \in x\}$.

2.2 Relations

**Definition 2.2.1** Relation: Let $A$ and $B$ be sets. A binary relation or simply a relation from $A$ to $B$ is subset of $A \times B$. Suppose $R$ is a relation from $A$ to $B$. Then $R$ is a set of ordered pairs where each first element comes from $A$ and each second element comes from $B$; that is, for each pair $a \in A$ and $b \in B$, exactly one of the following is true:

(I) $(a, b) \in R$; we then say “$a$ is $R$-related to $b$”.

(II) $(a, b) \notin R$; we say “$a$ is not $R$-related to $b$”.

If $R$ is a relation from set $A$ to itself, that is, if $R$ is a subset of $A^2 = A \times A$, then we say that $R$ is relation on $A$.
• $R$ is reflexive iff $xRx$ for all $x \in A$.

• $R$ is symmetric iff $xRy$ implies $yRx$ for all $x, y \in A$.

• $R$ is antisymmetric iff $xRy$ and $yRx$ implies $x = y$ for all $x, y \in A$.

• $R$ is transitive iff $xRy$ and $yRz$ implies $xRz$ for all $x, y, z \in A$.

• $R$ is a partial order iff $R$ is reflexive, antisymmetric and transitive.

**Definition 2.2.2** Composition of Binary relations: The composition of binary relations $r$ and $s$ where $r \subseteq X \times Y$ and $s \subseteq Y \times Z$ is defined as follows: $s \circ r = \{(x, z) | \exists y[(x, y) \in r \land (y, z) \in s]\}$

**Definition 2.2.3** Partially Ordered Set: A partially ordered set or a poset is a set taken together with a partial order on it. Formally, a partially ordered set is defined as an ordered pair $P = (X, \leq)$, where $X$ is called the ground set of $P$ and $\leq$ is the partial order of $P$.

**Definition 2.2.4** Join (Least Upper Bound): An upper bound of a subset $B \subseteq A$ of a poset $(A, \leq)$ is an element $a \in A$ such that for all $b \in B$ we have $b \leq a$. A least upper bound (LUB) or join of $B$ is an upper bound $a$ such that for all other upper bounds $a'$ we have $a \leq a'$.

**Definition 2.2.5** Meet (Greatest Lower Bound): A lower bound of a subset $B \subseteq A$ of a poset $(A, \leq)$ is an element $a \in A$ such that for all $b \in B$ we have $a \leq b$. A greatest lower bound (GLB) or meet of $B$ is a lower bound $a$ such that for all other lower bounds $a'$ we have $a' \leq a$. 
**Definition 2.2.6** Monotone Functions (Order-Preserving): Let \((X, \leq)\) and \((Y, \leq)\) be partially ordered sets, and let \(f : X \rightarrow Y\) be a function. \(f\) is monotone or order-preserving if and only if whenever \(x \leq y\) then \(f(x) \leq f(y)\).

2.3 Orderings on Domains

**Definition 2.3.1** Smyth Ordering: Suppose \((X, \leq)\) is a partially ordered set and \(S_1, S_2\) are subsets of \(X\). We say \(S_1 \leq \# S_2\) iff \((\forall d \in S_2) (\exists c \in S_1)\) such that \(c \leq d\).

**Definition 2.3.2** Hoare ordering: Suppose \((X, \leq)\) is a partially ordered set and \(S_1, S_2\) are subsets of \(X\). We say \(S_1 \leq \flat S_2\) iff \((\forall c \in S_1) (\exists d \in S_2)\) such that \(c \leq d\).

**Definition 2.3.3** Directed Partial Order Set: A directed partial order is a poset \((P, \leq)\) that is directed, i.e. every subset of \(P\) has an upper bound in \(P\).

**Definition 2.3.4** Complete Directed Partial Order Set/ Complete Partial Order Set (CPO): A poset \(D\) is complete; if every non empty directed subset \(P \subseteq D\) has a least upper bound.

**Definition 2.3.5** Flat Ordering: Let \(S_{\bot}\) be a set which includes a bottom element \(\bot\) i.e \(S \cup \{\bot\}\). If \(\forall s_1, s_2 \in S\) \(s_1 \leq s_2\) iff \(s_1 = \bot\) or \(s_1 = s_2\) then the ordering is said to be flat order.

For example, if set of natural numbers \((\mathbb{N}_{\bot} = \{\bot, 1, 2, 3, \ldots n\})\) is flat ordered, then Figure-1 shows the ordering on its elements.

**Proposition 2.3.1**: If \(S\) is flat ordered, then \(S\) is a CPO.
**Proof**: Let $P$ be a directed subset of $S$. Then $P$ is $\{\bot\}$ or $\{x\}$ for some $x \in S$ or $P = \{\bot, x\}$ for some $x \in S$. In the first case $\bot$ is the least upper bound of $P$, and in the other two cases $\{x\}$ is the least upper bound. So set $P$ is CPO.

**Definition 2.3.6** Pointwise Ordering : Let $P$ and $Q$ be posets, then the pointwise ordering is defined as the ordering on $P \times Q$ such that $(a, c) \leq (b, d)$ if and only if $a = b$ and $c \leq d$. The pointwise ordering is often extended to functions from $P$ to $Q$. If $f, g : P \to Q$, we say $f \leq g$ if for each $a \in P$, $f(a) \leq g(a)$.

**Definition 2.3.7** Lexicographic Ordering : Let $P$ and $Q$ are posets, then lexicographic ordering is defined as the ordering on $P \times Q$ such that $(a, c) \leq (b, d)$ if and only if either $a \leq b$ or ($a = b$ and $c \leq d$).
3.1 Syntax and Semantics

**Syntax:**

Syntax of a programming language defines how a well-formed sentence or string can be formed from the elements of that language. Syntax deals solely with the form and structure of the symbols in a language without any consideration given to their meaning. BackusNaur form (BNF) is one of the well known ways of representing the context free grammar of a programming language. It consists of a set of terminal and non-terminal symbols and a set of production rules. Production rules define the ways a non-terminal symbol can be formed from the set of terminal and non-terminal symbols. The following example demonstrates how a well formed natural number can be derived from set of digits.

\[
\text{\textless NaturalNumber \textgreater} ::= \textless digit \textgreater \mid \textless digit \textgreater \textless NaturalNumber \textgreater
\]

\[
\text{\textless digit \textgreater} ::= 0|1|2|3|4|5|6|7|8|9
\]

All the digits are terminal symbols, and \textless NaturalNumber \textgreater, and \textless digit \textgreater are non-terminal symbols. Using the rules shown above, we can derive all the natural numbers from the terminal and non-terminal symbols. These well formed syntactic entities have no meaning until we define some semantics to them. Actually our syntactic rules only gives us numerals, and we do not get numbers until we define semantics for our numerals.
Abstract Syntax:

The BNF representation of syntax for a programming language consists of information that is useful for analyzing the validity of program statements, which are useful in lexical and syntactic analyzers to create a list of tokens and derivation trees from the list of tokens. This form of syntax is referred as concrete syntax. But in the context of giving semantic definitions to a programming language most of the information is unnecessary except for the basic structure of the syntax. Abstract syntax gives the basic structure of the syntax, simplifying information for semantic analysis. The following figure shows a simple abstract syntax tree for an expression.

![Abstract syntax tree for an expression](image)

Figure 2: Abstract syntax tree for an expression.

Semantics:

Semantics of a programming language gives meaning to a syntactically valid program
statement. Semantics defines how a given entity (expression, sentence, or phrase) in a language is interpreted and the behavior of the machine when the program statement is executed. We can give the behavior of the machine by giving the relation between the input and output of a program or by a step-by-step explanation of how a program will execute on a real or an abstract machine. The input-output explanation is a relational semantics explanation, and the step-by-step explanation is an operational semantics explanation.

Let’s give some semantics to the natural number example.

\[ \text{digit} \in D = \{0, 1, 2, 3...9\} \]

\[ \text{NaturalNumber} \in N_0 = \{0, 1, 2...\} \]

This gives a meaning to \( \text{digit} \) and \( \text{NaturalNumber} \) that, they belong to the set of single digit numbers and the set of natural numbers, respectively.

Production rules:

\[ \text{NaturalNumber} : D \times N_0 \rightarrow N_0 \]

\[ \text{digit} ::= [D] \]

\[ \text{NaturalNumber} ::= [DN_0] \]

This gives the production rule a meaning that natural numbers can be formed by digits itself or by combining digits and the natural numbers.

There are a few different semantic methodologies that we can use to give meaning to a programming language namely axiomatic, denotational and operational.
3.2 Axiomatic Semantics:

Axiomatic semantics defines the meaning of a program command by giving assertions on that command execution. Assertions are logical statements, i.e., predicates with variables. Each program can be reduced to a sequence of commands, and then we can give the axiomatic semantics to the program by placing assertions on each of the command. And by combining the assertions for all the statements into an assertion for the program. Assertions are always valid when the control of the program reaches the points of assertions. We place an assertion before the command called precondition and a post condition after the command. \{\text{PRE}\}C\{\text{POST}\} Therefore the meaning of command \(C\) can be viewed as the ordered pair \(<\text{PRE}, \text{POST}>\), called a specification of \(C\). If the precondition of the command is true before the execution of the command, and the post condition of the command is true after the halting of the program execution, then the command is correct with respect to the specification. By giving these assertions to all the commands in a program, we give the semantics of a program in a language. From the Hoare’s logic rules [4], we can represent an assignment as \{A[x/e]\} \(x := e\{A\}\), where \(A[e/x]\) denotes that every free occurrence of variable \(x\) has been replaced with the expression \(e\). and \(x := e\) is the command where \(x\) is assigned the value of expression \(e\). If \{A[x/e]\} is true before the assignment command is true, then \{A\} will be true i.e. \{x + 1 = 2\} \(y := x + 1\{y = 2\}\).

3.3 Denotational Semantics

Denotational semantics [5,6] associates an appropriate mathematical object, such as a number, a tuple, or a function, with each phrase of a program or a language. The
phrase is said to denote the mathematical object, and the object is called the denotation of the phrase. Traditionally, denotational definitions use special brackets, the emphatic brackets \[[ \]\], to separate the syntactic world from the semantic world. If \( p \) is a syntactic phrase in a programming language, then a denotational specification of the language will define a “meaning” mapping , so that meaning \([[p]]\) is the denotation of \( p \), namely, an abstract mathematical entity that models the semantics of \( p \). A denotational definition of a language consists of an abstract syntax definition of the language, the semantic algebra and a valuation function. A semantic algebra consists of a semantic domain and an algebra defined on that domain. A semantic function or valuation function gives the meaning of a program by connecting both the syntactic domain and the semantic algebra.

We can explain the denotational semantic with a simple binary operation example.

Abstract syntax:

\[
\begin{align*}
B & \in \text{Binary-numeral} \\
D & \in \text{Binary-Digit} \\
B & ::= BD|D \\
D & ::= 0|1
\end{align*}
\]

Semantic algebra:

- Natural number Domain \( \text{Nat}:\mathbb{N} \)
- Operations
  \[
  \begin{align*}
  \text{zero, one, two, ...} & : \text{Nat} \\
  \text{plus, times} & : \text{Nat} \times \text{Nat} \to \text{Nat}
  \end{align*}
  \]

Valuation function:
\[ B : \text{Binary-Numeral} \rightarrow \text{Nat} \]
\[ B[[BD]] = (B[[B]] \times \text{two}) + D[[D]] \]
\[ B[[D]] = D[[D]] \]
\[ D : \text{Binary-digit} \rightarrow \text{Nat} \]
\[ D[[0]] = \text{zero} \]
\[ D[[1]] = \text{one} \]

As an example of evaluating a binary expression to its meaning using a denotational definition, let’s take ‘101’.

\[
B[[101]] = (B[[10]] \times \text{two}) + D[[1]] \\
= ((B[[1]] \times \text{two}) + D[[0]]) \times \text{two}) + D[[1]] \\
= ((D[[1]] \times \text{two}) + D[[0]]) \times \text{two}) + D[[1]] \\
= ((\text{one} \times \text{two}) + \text{zero}) \times \text{two}) + \text{one} \\
= \text{five}
\]

Similar derivations can be performed on strings of binary digits of any length. And using the semantic derivations, we can prove that \( D[[0101]] = D[[101]] \), i.e., even thought the syntactical definitions of the strings are different, they semantically represent the same thing.

3.4 Operational Semantics

Operational semantics for a programming language describes how a valid program is interpreted in a simple abstract machine. A program can be viewed as a sequence of steps where each step takes an abstract machine from one state to another. This sequence then
lead to the meaning of the program.

A state or configuration of the machine is represented by symbols, such as labeled boxes, values in the boxes, and arrows between them. Each step execution starts with an initial state and ends in a final state. The function given by the program determines or maps from one configuration to another. When the program is exhausted or the transition function is undefined for some reason, the process halts producing a final configuration that we take to be the result of the program.

Metacircular interpreters are the earliest forms of using the operational description of the programming languages. These descriptions represent the configurations directly as structures in the language being defined. John McCarthy’s definition of Lisp in Lisp [7] was an early landmark in providing the semantics of a programming language. But these representations failed to give any insight to a target language. Formal semantics demands more clarity in the definition of a language specification.

The Vienna Definition Language (VDL) developed at the Vienna IBM laboratory tried to give the abstract machine of a language in the form of tree representations in which the program being interpreted will be represented with memory as input, output and control trees. A control tree holds the instructions that need to be interpreted. Each configuration or state is represented by a snapshot of the tree. Starting with an initial configuration or state that has all the components of storage properly initialized, input defined as a tree representing the list of input values, output set as an empty tree, and the control tree defines as a single instruction to execute. The transition function given by the instructions of the VDL interpreter performs a step of a computation. One step consists of selecting a leaf node of the control tree and evaluating it according to the
program, producing a new state. This process is continued until all leaf node are used in the final result. This representation of the semantics is complex, and it is hard to prove the correctness of a language.

Peter Ladin proposed an abstract machine called the SECD machine [6,8] for the mathematical evaluation of Lambda expressions. The state in the abstract machine consists of four stacks: a structure for storing the partial result awaiting the subsequent use and represented by S (stack), a collection of bindings of values to the variables represented by E (Environment), a stack for holding the expressions to be evaluated represented by C (control), and a stack for holding the completed states corresponding to evaluation in progress represented by D (Dump). In SECD a state or configuration represented by $cfg(S,E,C,D)$. To evaluate a lambda expression expr, the SECD machine starts with the initial configuration $cfg([],nil,[expr],nil)$ that has empty stacks for S, E, and D. The one item on the control stack is the expression to be evaluated. The SECD machine is defined by a transition function, $\text{transform} : State \rightarrow State$, that maps the current configuration to the next configuration until a final state results, if it ever does. A final state is recognized by its having an empty control stack and an empty dump, indicating that no further computation is possible.

Although abstract machines as in above discussion provide higher-level, implementation independent specifications of program execution, they are not effective in proving program properties, notions of program equivalence and developing an algebra for programs. And, moreover, proofs about program execution are often tedious and cumbersome.

Gordan Plotkin proposed a deductive system that turns the abstract machine into a system of logical inferences called structural operational semantics (SOS) [9]. Since the
semantic descriptions are based on deductive logic, proofs of program properties are derived directly from the definition of language constructs. This formal representation of programs provides properties to programs that are missing from the previous methods of operational semantics. In structural operational semantics definitions are given by inference rules consisting of a conclusion that follows from the set of premises. If there are no premises to the conclusion then the rule become an axiom. The general form of an inference rule has the premises listed above a horizontal line, the conclusion below, and the condition, if present, to the right. A simple derivation for $a \equiv (Init + 5) + (7 + 9)$ in state $\sigma_0$, where Init is a location where $\sigma_0(Init) = 0$ can be seen in the fig-1.

\[
\begin{align*}
\langle Init, \sigma_0 \rangle &\rightarrow 0 & \langle 5, \sigma_0 \rightarrow 5 \rangle &\rightarrow 5 \\
\langle (Init + 5), \sigma_0 \rangle &\rightarrow 5 & \langle (7 + 9), \sigma_0 \rightarrow 16 \rangle &\rightarrow 21 \\
\end{align*}
\]

Plotkin’s structural operational semantics provides an elegant way to provide operational semantics, but it lacks mathematical structure to provide an analysis on programs. Non-determinism and recursion are hard to analyze using mathematical deduction.
CHAPTER 4

Relational Model for Program Semantics

4.1 Relational Model

In an abstract view, a program consists of a sequence of atomic steps, where each step takes the state of the machine from one state to another state that is one step closer to the final state. Collection of all possible individual states and their state transitions can be considered as an abstract machine. A state transition $T$ takes state machine $M$ from $s$ to $s'$ which can be represented as $T : s \rightarrow s'$. If $S$ is the set of all states then $S \times S$ represents a collection of all possible state transitions. A program $P$ is a subset of the set of state transitions $S \times S$ i.e. $P \subseteq S \times S$.

A computer program can be represented as a pair of $(s, p)$ where “s” is the text of the data state and “p” is the program text that needs to be executed. The meaning of the text is represented by $(S, P)$ where $S$ is the state corresponding to “s” and $P$ is the program corresponding to “p”. Each execution of a statement in the program text takes the machine one step closer to the final state. The program $(S, II)$ where $II$ is skip, represents the final step in a program execution where the result of the program execution is stored in the variables of the state $S$.

In the relational model we are going to use binary relations to represent possible states and state transitions, and then we will define a program in terms of a collection of state transitions. We will apply intuitive programming language constraints like error state and strictness of programs to behave like real programs. We use Smyth and Hoare orderings.
to define the orderings on states and programs.

In the following part of this section we will explain the construction of a program in a language of Variables, states and programs and provide an ordering on states and programs in the Hoare and Smyth orderings. In the section-4.2 we provide primitives, operations and some of the properties given by Hoare et al. [1] and In section 4.3 we will give the parallel of Hoare’s properties in relational model and prove the laws. In section 4.4 we are going to present the refinement relation in terms of our relational model. In section 4.5 we define non-determinism in the relational model. In the final section of the chapter, we introduce an operational view of programs in relational mode.

4.1.1 Variables and Values

Variables represented by $V = \{v_1, v_2, ..., v_n, ..\}$ are a countably infinite set of variables. $U = \{u_1, u_2, ... u_n, ..\}$ is the set of values from where the variables get their values. The domain of these values can be the set of natural numbers or any set of mathematical entities.

In the domain of values, no two elements are comparable because neither of the two elements hold more information than the other. To make this domain of values a partially ordered set, we introduce a bottom element $\bot_u$, such that $\bot_u \subseteq x, \forall x \in U$. This set will be represented by $U_\bot$ and will be $U \cup \{\bot_u\}$. Thus, $U_\bot$ is a flat domain. We also define that any function which takes $\bot_u$ as an argument will return $\bot_u$ i.e. $f : U_\bot \rightarrow U_\bot$ and $f(\bot_u) = \bot_u$.

From the formal view of programming $\bot_u$ is the element of error value or any illegal value. The meaning is the same when the variable does not have an assigned value and
when the assigned value is $\bot_u$.

Figure 3 shows the set of values and their ordering. All the values are in the same level, and they are not comparable, and bottom ($\bot_u$) lies below all the other elements in the set.

![Figure 3: Set of variables with bottom.](image)

**Proposition 4.1.1** $U_\bot$ is a complete partial order (CPO).

**Proof:** The value set $U_\bot$ is a flat domain because $\bot_u \sqsubseteq u$, $\forall u \in U_\bot$ and $u_1 \sqsubseteq u_2$ iff $u_1 = u_2$, $\forall u_i \in U$. By proposition 2.3.1 every flat domain is a CPO.

**Proposition 4.1.2** Let $f : U_\bot \rightarrow U_\bot$ be a function. $f$ is monotonic with $\sqsubseteq$ i.e if $x \sqsubseteq y$ then $f(x) \sqsubseteq f(y)$ $\forall x, y \in U_\bot$.

**Proof:** Because $U_\bot$ is flat ordered, $x \sqsubseteq y$ means $x$ can be $\bot_u$ i.e, $(x = \bot_u)$ or $x = y$. We defined $f(\bot_u) = \bot_u$, so it will be $\bot_u \sqsubseteq f(y)$, and $f(x) \sqsubseteq f(x)$ when $x = y$. Thus $f$ is monotonic with respect to $\sqsubseteq$.

### 4.1.2 States

A state in a program execution is the snapshot of all the variables in the program at a given point of time. As the program executes, the values of the variables change,
which means the state changes. We may assume that a state “s” is a function from V to $U_\bot$, i.e. $s : V \rightarrow U_\bot$ with the restriction that only finitely many variables have non $\bot_u$ values. Thus, the set of all states $S = \{ s : V \rightarrow U_\bot : |\{v \in V : s(v) \neq \bot\}| < \infty\}$. Also we assume each state is total.

The bottom element of the states can be defined as $\bot_s \in S$ such that $\bot_s : V \rightarrow U$ and $\bot_s(v) = \bot_u \forall v \in V$. For the set of all states including $\bot_s$, we use the notation $S_\bot$. In our formal view, the bottom state $\bot_s$ represents a nontermination or a blocked state.

**Pointwise ordering:**

**Definition 4.1.1** Pointwise ordering ($\sqsubseteq^p$) on states i.e, $s_1 \sqsubseteq^p s_2$ can be defined as $\forall v \in V, s_1(v) \sqsubseteq^p s_2(v)$.

**Proposition 4.1.3** $\bot_s \sqsubseteq^p s \forall s \in S$.

**Proof:** From the definition of $\bot_s$, the bottom state takes every variable to $\bot_u$. And from the ordering on the value space $\bot_u \subseteq u \forall u \in U$. Using the pointwise ordering on the set of states, it follows that $\forall v \in V \ \bot_s(v) = \bot_u \subseteq s(v)$ so $\bot_s \subseteq s$.

Assuming that we do not include the variable assignments that are assigned to the $\bot_u$, we will derive the Smyth and Hoare orderings on the set of states. Our assumption means that if $s \in S$ such that $s(v_1) = u_1$, and for $v \neq v_1$, $S(v) = \bot_u$, then we will let $s = \{(v_1, u_1)\}$ instead of $\{(v_1, u_1)\} \cup \{(v_1, \bot_u), v \neq v_1\}$.

**Smyth ordering:**

**Definition 4.1.2** Smyth ordering ($\sqsubseteq^\sharp$) on states : $s_1 \sqsubseteq^\sharp s_2$ can be defined as $\forall y \in s_2, \exists x \in s_1$, such that $x \sqsubseteq y$. 
This means if \((v, u) \in s_2\) then there exists \((v, u') \in s_1\) where \(u' \sqsubseteq u\). However, since we are not including \((v^*, u^*)\) in a state if \(u^*\) is \(\bot_u\). It follow that if \((v, u) \in s_2\), then \((v, u) \in s_1\). Thus, \(s_2 \subseteq s_1\).

**Hoare Ordering:**

**Definition 4.1.3** Hoare ordering \((\sqsubseteq^\flat)\) on states: \(s_1 \sqsubseteq^\flat s_2\) can be defined as \(\forall x \in s_1 \exists y \in s_2\), such that \(x \sqsubseteq y\).

Similar to the Smyth ordering the Hoare ordering on sets \(s_1, s_2\) in terms of set inclusion means that \(s_1 \subseteq s_2\).

### 4.1.3 Programs

A program is collection of executable statements, and each such statement is a state transition. Each program statement will change one program state to another state. If \(S_\bot\) is a set of all program states, then \(S_\bot \times S_\bot\) represents the set of state transitions that are possible. Thus, any program \(P\) is subset of state transitions, i.e. \(P \subseteq S_\bot \times S_\bot\).

Any program that starts with a non-termination or error state always goes to the non-termination state. So programs in our programming language are strict i.e \(P(\bot_s) = \bot_s \forall P\). Thus, programs are not completely arbitrary subsets of \(S_\bot \times S_\bot\). However, they are arbitrary subsets of \(S \times S_\bot\) with the possibility including \((\bot_s, \bot_s)\). We are going to use the notation of “\(\text{Prog}\)” to represent these set of possible state transitions i.e.

\[
\text{Prog} = \{S \times S_\bot\} \cup \{(\bot_s, \bot_s)\}.
\]

The relational model allows us to better include the non-determinism into our programming language because one program may have more then one state transition from a single state.

Before we can define an ordering on the programs, we need to define the ordering on the
Definition 4.1.4 (Ordering on Transitions): Our ordering on state transitions, i.e. on elements of $Prog$ is the pointwise ordering. $(s_1, s_2) \sqsubseteq (s_3, s_4)$ if and only if $s_1 = s_3$ and $s_2 \sqsubseteq s_4$.

A state transition is said to be more defined than that of the other, i.e. $(s_1, s_2) \sqsubseteq (s_3, s_4)$ if the pointwise ordering on its elements holds, i.e. $(s_1, s_2) \sqsubseteq (s_3, s_4)$ iff $s_1 = s_3$ and $s_2 \sqsubseteq s_4$.

Both Smyth and Hoare orderings in the relational model are pre-orders, because they satisfy the reflexive and transitive properties but there may not be antisymmetry.

Smyth ordering: Let $p_1$ and $p_2$ be two programs, i.e. $p_1, p_2 \subseteq Prog$, then the Smyth ordering on the programs $p_1 \sqsubseteq_p p_2$ is defined as $\forall t \in p_2, \exists t' \in p_1$ such that $t' \sqsubseteq t$.

For every state transition in program $p_2$ there exists a state transition which is less than or equal to the one in program $p_1$, i.e. for each $(s, s_2) \in p_2$, there is $(s, s_1) \in p_1$ with $s_1 \sqsubseteq s_2$. In formal sense $p_2$ is more defined than that of the program $p_1$.

A bottom element in Smyth ordering is any program that contains the state transitions $S_\bot \times \{s_\bot\}$, where $S_\bot$ is the set of states and $s_\bot$ is the bottom state. The universal relation $Prog (\{S \times S_\bot\} \cup \{\bot_\bot, \botS\})$ would also be a bottom element.

Proposition 4.1.4 $Q$ is a bottom element for the set of all programs if and only if $S_\bot \times \{s_\bot\} \subseteq Q$

Proof: Let $P$ be a program i.e. $P \subseteq Prog$. And $Q$ is a bottom element i.e. $S_\bot \times \{s_\bot\} \subseteq Q$. $Q$ has all the bottom transitions. Hence, $\forall (s, s_1) \in P, \exists (s, s_\bot) \in Q$, and $s_\bot \sqsubseteq s_1, \forall s_1 \in S_\bot$, so $Q$ is a bottom element.
In Smyth ordering the top element ($\top_p^\sharp$) is an empty set i.e. $\top_p^\sharp = \emptyset$. But if we are not going to allow a empty program as a real program, then there can only be maximal elements. In Smyth ordering, all the singleton elements are maximal elements, where singleton element is a program with only one state transition.

**Proposition 4.1.5** If $\top_p^\sharp = \emptyset$, then $P \sqsubseteq \top_p^\sharp$

**Proof:** There are no state transitions in the top program $\top_p^\sharp$, so it holds that $\forall t \in \top_p^\sharp$, $\exists t' \in P$ such that $t' \sqsubseteq t$ where $P \subseteq \text{Prog}$. so empty program is the top in Smyth ordering.

**Proposition 4.1.6** $P \sqcap^\sharp Q = P \cup Q \sqsubseteq P, Q$.

**Proof:** Let $R \subseteq \text{Prog}$ and $R = P \cup Q$. Let $t_1 \in P$ and $t_2 \in Q$ then $t_1, t_2 \in P \cup Q$. From definition of Smyth ordering it follows that $R \sqsubseteq P$ and $R \sqsubseteq Q$ because $\forall t \in P, Q \exists t' \in R$ with $t' \sqsubseteq t$, So $R$ is a lower bound of $P$ and $Q$.

To prove that this is a greatest lower bound, let $X$ be a lower bound of $P$ and $Q$, then $R \sqsubseteq X \sqsubseteq P, Q$. Let $t \in R$, Thus, $t \in P$ or $t \in Q$ because $R = P \cup Q$. However, since $X \sqsubseteq P, Q$, then there is $t' \in X$ where $t' \sqsubseteq t$. Thus, $X \sqsubseteq R$, and so $R \equiv X$.

**Proposition 4.1.7** $P, Q \sqsubseteq P \cap Q$ where $P \cap Q$ is an upper bound of $P$ and $Q$.

**Proof:** Let $R \subseteq \text{Prog}$ and $R = P \cap Q$. Let $t \in R$, then $t \in P$ and $t \in Q$ and $t \sqsubseteq t$. So $P \sqsubseteq R$ and $Q \sqsubseteq R$, i.e., $R$ is an upper bound of $P$ and $Q$.

**Hoare Ordering:** Lets $p_1$ and $p_2$ are two programs, i.e, $p_1, p_2 \subseteq \text{Prog}$, then the Hoare ordering on the programs $p_1 \sqsupseteq p_2$ is defined $\forall t \in p_1$, $\exists t' \in p_2$ such that $t \sqsubseteq t'$. 
For every state transition in program \( p_1 \), there exists a state transition in \( p_2 \) which is greater than or equal to the one in program \( p_1 \). In other words, for each \((s, s_1) \in p_1\) there is \((s, s_2) \in p_2\) with \( s_1 \sqsubseteq s_2 \). In the Hoare ordering a bottom element is an empty set \( \emptyset \). If we make the assumption for program \( P \), that if for \( s \in S_\bot \), there is no \( s' \in S_\bot \) with \((s, s') \in P\), then \( P \) is equivalent to \( P \cup \{(s, s_\bot)\} \). Then any subset of \( S_\bot \times \{s_\bot\} \) is a bottom element in the set of all programs with the Hoare ordering.

**Proposition 4.1.8** A program \( Q \) is a bottom element with respect to the Hoare ordering if and only if \( Q \subseteq S_\bot \times \{s_\bot\} \).

**Proof:** Let \( P \) be a program i.e, \( P \subseteq \text{Prog} \). And \( Q \) is a bottom element i.e, \( Q \subseteq S_\bot \times \{s_\bot\} \). \( Q \) has only bottom transitions, Hence, \( \forall (s, s_\bot) \in Q, \exists (s, s') \in P \), and \( s_\bot \sqsubseteq s' \forall s' \in S_\bot \), so \( Q \) is a bottom element.

**Proposition 4.1.9** \( P \sqsubseteq^b \text{Prog} \).

**Proof:** Let \( P \) be a program i.e, \( P \subseteq \text{Prog} \). Since \( \text{Prog} \) contains all the state transitions possible i.e, \( \text{Prog} = \{S \times S_\bot\} \cup \{(\bot_s, \bot_s)\} \), so for any \((s, s') \in P\), \((s, s')\) also exists in \( \text{Prog} \), so \( P \sqsubseteq^b \text{Prog} \).

**Proposition 4.1.10** \( P \cap Q \subseteq P, Q \). Where \( P \cap Q \) is a lower bound of \( P \) and \( Q \).

**Proof:** Let \( R \subseteq \text{Prog} \) and \( R = P \cap Q \), then, for all \( t \in R \), \( t \) also exists in \( P \) and \( Q \), and \( t \sqsubseteq t \). So \( R \) is a lower bound of \( P \) and \( Q \) in Hoare ordering.

**Proposition 4.1.11** \( P, Q \subseteq P \cup^b Q = P \cup Q \).

**Proof:** Let \( R \subseteq \text{Prog} \) and \( R = P \cup Q \). Then \( \forall t \in P \exists t' \in R \) such that \( t \sqsubseteq t' \), So \( P \subseteq R \).

And \( \forall t \in Q \exists t' \in Q \) such that \( t \sqsubseteq t' \), so \( Q \subseteq R \) i.e. \( R \) is an upper bound of both \( P \) and
To prove that \( R \) is the least upper bound, let \( X \) be an upper bound, such that, \( P, Q \subseteq X \subseteq R \). Let \( t \in R \), then \( t \in P \) or \( t \in Q \). because \( R = P \cup Q \), However, since \( P, Q \subseteq X \), there exists \( t' \in X \), such that \( t \sqsubseteq t' \). Thus \( R \subseteq X \), so \( R \equiv X \).

4.2 Hoare’s Properties of Programs

In this section we are going to introduce primitives and operations of programming languages and their properties as used by Tony Hoare et al. [1,10].

**Hoare’s Primitives and Operations:**

**Abort(\( \bot \)):** Abort is a program where it places no constraint on the behavior of the program in executing or non executing, and terminating or not terminating. Abort is like a broken machine, where its end state is not predictable. Program developers hope to avoid this program.

**Miracle(\( \top \)):** Miracle \( \top \) can be used to serve any purpose; this program is better than any program. Miracle satisfies specifications for every program and no program can satisfy all the specifications of this program. This program can not be implemented in real life.

**Skip(\( I \)):** Skip is program that will not change state of the machine on completion and it always terminates in the starting state.

**Sequential composition:** Sequential composition allows us to combine two programs to form a composite program. If \( P \) and \( Q \) are programs,\( (P; Q) \) is a program that is executed by first executing \( P \). If \( P \) does not terminate, neither does \( (P; Q) \). If and when \( P \) terminates, \( Q \) is started; and then \( (P; Q) \) terminates when \( Q \) does.
**Non-Determinism:** Non-deterministic choice between two programs $P$ and $Q$ is defined as the union of the programs, i.e., $P \cup Q$. This means that the developer of the program deliberately postponed the decision to the later stages of the execution or delegated the decision to the executing machine.

**Recursion:** Recursion can be represented as the continuous execution of a program by itself. Let $X$ be the name of the program that is defined recursively, and $P(X)$ be a program text that defines the program behavior. Then $\mu X :: P(X)$ is a program that behaves like $P(\mu X :: P(X))$, i.e., all occurrences of the program name have been replaced by the whole recursive program. Where $\mu$ is the weakest or least fixed point.

**Refinement Ordering ($P \sqsubseteq Q$):** The refinement ordering will give us the ability to reason about program transformations. Let $P$ and $Q$ be two programs, then the refinement ordering on the programs defined as $P \sqsubseteq Q$ means that $Q$ is at least as good as $P$ in the sense that it will meet every purpose and satisfy every specification of $P$.

**Properties of Programming language:**

In following part of the section, we are going to present some of the programming language properties given by Hoare et al..

**Law 4.2.1** : $(II; P) = P = (P; II)$

Execution of skip ($II$) always terminates and leaves everything unchanged, so execution of skip followed or preceded by any program will not make any difference.

**Law 4.2.2** : $P; (Q; R) = (P; Q); R$

Sequential composition is associative.

**Law 4.2.3** : $(\bot; P) = \bot = (P; \bot)$
Sequentially composing a program with bottom will leave the machine in the bottom state and vice versa.

**Law 4.2.4** : \((\top; P) = \top\)

Sequentially composing miracle(\(\top\)) with a program will leave the machine in the miraculous state.

**Law 4.2.5** : \(P; (Q \cap R) ; S = (P; Q; S) \cap (P; R; S)\)

Sequential composition is distributed through non-determinism.

**Law 4.2.6** : \(\bot \sqsubseteq P \sqsubseteq \top\)

Bottom (\(\bot\)) is the least element and the top(\(\top\)) is the greatest element.

**Law 4.2.7** : \((R \sqsubseteq P \land R \sqsubseteq Q) \equiv R \sqsubseteq (P \sqcap Q)\)

If program \(R\) is less than \(P\) and \(Q\), then program \(R\) is less than the greatest lower bound, i.e., \(R \sqsubseteq (P \sqcup Q)\).

**Law 4.2.8** : If \(P \sqsubseteq Q\) then \((P \sqcup R) \sqsubseteq (Q \sqcup R)\)

Non-deterministic choice is monotonic with respect to refinement ordering.

### 4.3 Properties in Relational Model

In this section we are going to provide the interpretations for Hoare’s primitives and operations in the relational model, and we are going to prove some of the laws given by Hoare.

**Abort(\(\bot\))**: In the relational model point of view, abort is a program that takes all the states to the bottom state (\(\bot_s\)), i.e. \(\bot(s) = \bot_s \ \forall s \in S_\bot\). In the set programs using
Smyth ordering, from the proposition 4.1.4, any program \( Q \) where \( S \times \{s_\perp\} \subseteq Q \) acts as a bottom element. From the proposition 4.1.8, a bottom program in Hoare ordering is the one with subset of elements from \( S \times \{s_\perp\} \). If the programs are total then \( S \times \{s_\perp\} \) is the only bottom element.

**Miracle(\( \top \)):** From Hoare’s ordering it follows that set of all states will become the miracle of the programs i.e. \( \top^p = \text{Prog} \). In case of Smyth ordering the top element (\( \top^s \)) is a program with empty transactions i.e. \( \emptyset \). If we assume that programs are total, as in our model, then no true top program exists in Smyth ordering.

**Skip(\( \text{II} \)):** Skip is program that will not change state of the machine on completion and it always terminates in the starting state, which is equivalent to the identity relation. A program \( \text{II} \) is a identity relation with every state will move to the same state, i.e. \( \text{II} = \{(s, s) | s \in S_\perp\} \). In both Smyth and Hoare Orderings the skip remains the same i.e. \( \text{II}^s = \text{II}^p = \{(s, s) | s \in S_\perp\} \).

**Sequential composition:** Sequential composition is equivalent to composite function in the relations.

Let \( P \subseteq \text{Prog} = \{(x, y) | (x, y) \in P \text{ and } x, y \in S\} \) and \( Q \subseteq \text{Prog} \) defined as $= \{(y, z) | (y, z) \in Q \text{ and } y, z \in S\}$ then \( P;Q \) defined as $\{(x, z) | (\exists z) [(x, y) \in P \text{ and } (y, z) \in Q]\}$.

**Sequential Composition Properties**

Let \( P \) and \( Q \) be programs, i.e. \( P, Q \subseteq \text{Prog} \), then sequentially composting \( P \) and \( Q \) \((P;Q)\) will results in a new set of state transitions \( R \) such that \( R \subseteq \text{Prog} \).
Proposition 4.3.1: The set of programs is closed under the sequential composition(;;).

Proof: Let \( P = \{(s_1, s_2)|s_1, s_2 \in S_\bot \text{ and } (s_1, s_2) \in \text{Prog}\} \)
and \( Q = \{(s_2, s_3)|s_2, s_3 \in S_\bot \text{ and } (s_2, s_3) \in \text{Prog}\} \) then from the definition of sequential composition \( P;Q = \{(s_1, s_3)|\exists s_2|s_1, s_3 \in S_\bot \text{ and } (s_1, s_3) \in S_\bot \times \{s_\bot\}\} \), so \( \forall t \in P;Q, \exists t' \in S_\bot \times S_\bot \) such that \( t = t' \).

From the definition of the skip(II), it follows that sequentially composing a program with a skip results in the same state. It will act as left and right identity.

Law 4.3.1 : \(< II;P > = P = < P;II >\)

Proof: Skip defined as \( II = \{(s, s)|\forall s \in S_\bot\} \) and Let \( P \) be a program define as \( P = \{(s_1, s_2)|s_1, s_2 \in S_\bot \text{ and } (s_1, s_2) \in \text{Prog}\} \) then
\[
R = II;P = \{(s_1, s_2)|\exists s_1|(s_1, s_1) \in II \text{ and } (s_1, s_2) \in P\}; \text{ but all } (s_1, s_2) \in II;P \text{ also exists in } P. \text{ so it follows that } II;P = P.
\]

Similarly we can easily prove the other part of the law, i.e., \( P;II = P \).

Law 4.3.2 : \(< P; (Q;R) > = (P; Q); R >\)

Sequential composition is associative as composition is associative in relations.

Proposition 4.3.2: Let \( S \) be set of states and \( P \) be a program\((P \subseteq \text{Prog})\), then the set of programs is a monoid over sequential composition(;;).

Proof: From the law-4.3.1, the set of programs has an identity element i.e., Skip (II).

And from law-4.3.2, programs are associative with respect to the sequential composition.

So the set of programs is monoid over sequential composition.
**Law 4.3.3** : 

\[(P; \bot) = \bot\]

Abort represented by bottom \(\bot\) acts as right zero over the sequential composition. In Smyth’s ordering of programs, a program \(P\) is bottom if \(S_\bot \times \{s_\bot\} \subseteq P\).

\[P; \bot = \{(x, y)|\exists z (x, z) \in P \text{ and } (z, y) \in \bot\}\]

\[= \{(x, y)|\exists z (x, z) \in P \text{ and } z \in S_\bot \text{and } y = s_\bot\}\]

\[= \{(x, y)| (\exists z \in S_\bot) [x \in \text{dom}(P) \text{ and } y = s_\bot]\}\]

\[= \{(x, y)| [x \in \text{dom}(p) \text{ and } y = s_\bot]\}\]

\[= \{(x, y)| [x \in S_\bot \text{ and } y = s_\bot]\}\]

\[= \bot\]

In Hoare’s ordering, a program is bottom if \(S_\bot \times \{s_\bot\} \supseteq P\). But in our model, programs are total, i.e., any state without a state transition will always goes to the bottom state. So the only possible bottom program is \(S_\bot \times \{s_\bot\}\). We can easily prove the right zero property \(P; \bot = \{s_\bot\}\) in Hoare’s order using similar proof as Smyth order.

**Law 4.3.4** : 

\[(\bot; P) = \bot\]

Bottom \(\bot\) acts as left zero over the sequential composition.

In Hoare’s ordering of programs, a program \(P\) is bottom if \(S_\bot \times \{s_\bot\} \supseteq P\).

\[P; \bot = \{(x, y)|\exists z (x, z) \in \bot \text{ and } (z, y) \in P\}\]

\[= \{(x, y)|\exists z [x \in S_\bot \text{ and } z = s_\bot \text{ and } (z, y) \in P]\}\]

\[= \{(x, y)| (\exists z = s_\bot) [x \in S_\bot \text{ and } (s_\bot, s_\bot) \in P]\} \quad \text{[Strictness of programs]}\]

\[= \{(x, y)| [x \in S_\bot \text{ and } y = s_\bot]\}\]

\[= \bot\]
Similarly we can prove the left zero property of bottom \((S_\perp \times \{s_\perp\} \subseteq P)\) in Smyth order, i.e, \((\bot_s; P) = \bot_s\).

**Proposition 4.3.3** \(P; \top^b = \top^b\)

Let \(P\) be a program i.e \(P \subseteq \text{Prog}\). In Hoare ordering \(\top^b = \text{Prog}\), then

\[
P; \top^b = \{(x, y) | \exists z (x, z) \in P \text{ and } (z, y) \in \top^b\}
\]

\[
= \{(x, y) | \exists z (x, z) \in P \text{ and } z \in S_\perp \text{ and } y \in S_\perp\}
\]

\[
= \{(x, y) | (\exists z \in S_\perp) [x \in \text{dom}(P) \text{ and } y \in S_\perp]\}
\]

\[
= \{(x, y) | [x \in \text{dom}(P) \text{ and } y \in S_\perp]\}
\]

\[
= \top^b
\]

The left sequential composition of a program with a miracle in Hoare order may not result in a miracle because range of a program is subset of states i.e., \(\text{range}(P) \subset S_\perp\).

In Smyth ordering, the miracle is an empty set \(\emptyset\). Composing an empty set with a program results in an empty set(miracle). We avoid empty programs in our model, so there wont be any miraculous program present in the Smyth ordering.

**Definition:** Let \(D\) be a complete partial order(CPO) with \(\bot\), then \(f : D \to E\) is Scott continuous, if whenever \(E\) is a directed subset of \(D\), then \(f(\bigsqcup) = \bigsqcup f(E)\).

**Proposition:** If \(D\) is a complete partial order(CPO) with \(\bot\) and, if \(f : D \to D\) is a Scott continuous, then \(\bigsqcup_{n \in \mathbb{N}} f^n(\bot)\) is the least fixed point of \(F\). Where \(F : \text{program} \to \text{program}\) such that \(F(g) = f \circ g\) and \(f^0 = \bot_\mu\).

**Proof:** Since \(\bot \sqsubseteq d \ \forall d \in D\) and \(f\) is order preserving, then by induction \(\bot = f^0(\bot) \sqsubseteq \)
\[ f^1(\bot) \subseteq f^2(\bot) \subseteq f^3(\bot) \ldots \subseteq f^n(\bot). \]

Thus \( \{ f^n(\bot) \mid n \in \mathbb{N} \} \) is a directed subset of \( D \), Hence,

\[
f \left( \bigcup_{n \in \mathbb{N}} f^n(\bot) \right) = \bigcup_{n \in \mathbb{N}} f^0(\bot) = \bigcup_{n \in \mathbb{N}} f^n(\bot) \cup \{ \bot \} = \bigcup \{ f^n(\bot) \mid n \in \mathbb{N} \} = \bigcup_{n \in \mathbb{N}} f^n(\bot).
\]

Thus, \( \bigcup_{n \in \mathbb{N}} f^n(\bot) \) is a fixed point of \( F \).

Suppose \( d \) is a fixed point of \( F \), i.e. \( F(d) = d \). Since \( \bot \subseteq d \), then \( F(\bot) = F(d) = d \), by induction \( f^n(\bot) \subseteq d \) \( \forall n \in \mathbb{N} \). Hence \( \bigcup_{n \in \mathbb{N}} f^n(\bot) \subseteq d \), Hence \( \bigcup_{n \in \mathbb{N}} f^n(\bot) \) is the least fixed point of \( F \).

### 4.4 Refinement Relation

We want to give the semantics of refinement on programs, i.e., for given programs \( P, Q \) we want to define how one program is better or worst or equal to other. The notation for refinement is \( \sqsubseteq \). So the refinement \( P \sqsubseteq Q \) means that program \( P \) can be transformed into program \( Q \) such that every possible outcome of the program \( Q \) is at least as defined as the program \( P \). This means \( Q \) must be as defined as the program \( P \) and have no more non-determinacy than program \( P \).

The refinement Relation \( (P \sqsubseteq Q) \) means that \( Q \) is at least as good as \( P \) in the sense that it will meet every purpose and satisfy every specification specified by \( P \). This means substitution of \( Q \) for \( P \) in any context can only be an improvement. If \( Q \) is better than \( P \) in all situations, then the non-deterministic choice between \( P \) and \( Q \) is as bad as \( P \).

As we discussed in the previous section, we have two kind of orderings that can be
possible, i.e. Smyth and Hoare orderings on the set of programs.

Refinement using the Smyth ordering is defined as

\[ P \sqsubseteq Q \text{ iff } \forall y \in Q, \exists x \in P \text{ such that } x \sqsubseteq y. \]

So from the Smyth ordering, \( P \sqsubseteq Q \) means for every element (i.e. transition) in \( Q \) there exists an element in \( P \) which is less than or equal. The semantic view of the definition means that \( Q \) is less or at least no more non-deterministic than \( P \). But this can also be viewed as the loss of information because of elimination of non-determinism.

Refinement using the Hoare ordering is defined as

\[ P \sqsubseteq Q \text{ iff } \forall x \in P, \exists y \in Q \text{ such that } x \sqsubseteq y \]

Hoare ordering infers that \( P \sqsubseteq Q \) means for every element in \( P \) there exists an element in the \( Q \) which is greater then or equal. So in semantic view it will keep the non-determinism.

**Law 4.4.1** : \( \bot \sqsubseteq P \sqsubseteq \top \)

From the proposition 4.1.4 and proposition 4.1.5 of the Smyth ordering (\( \bot^\sharp_p \sqsubseteq P \) and \( P \sqsubseteq \top^\sharp_p \)), it follows that \( \bot^\sharp_p \sqsubseteq P \sqsubseteq \top^\sharp_p \).

Similarly in Hoare ordering we proved the properties \( \bot^\flat_p \sqsubseteq P \) and \( P \sqsubseteq \top^\flat_p \) so it follows that \( \bot^\flat_p \sqsubseteq P \sqsubseteq \top^\flat_p \). So in both Smyth and Hoare orderings top is the highest element and bottom is the lowest element in the orderings. i.e. \( \bot \sqsubseteq P \sqsubseteq \top \).

### 4.5 Non-determinism

If \( P \) and \( Q \) are programs, then \( P \sqcap Q \) is a program that is executed by executing either \( P \) or \( Q \). The choice between them is arbitrary. The programmer has deliberately postponed the decision, possibly to a later stage in the development of the program, or
possibly has even delegated the decision to the machine that executes the program. Non-deterministic choice between two programs \( P \) and \( Q \) is \( P \cup Q \), i.e. a program that includes state transitions from both the programs. We use notation \( R_{\text{nod}} \) to represent the programs or relation of non-deterministic choice. In Smyth ordering the non-deterministic choice between programs \( P \) and \( Q \) is the meet (glb) of the two programs i.e. \( P \sqcap^\sharp Q \).

**Proposition 4.5.1** \( PR_{\text{nod}}Q = P \cup Q = P \sqcap^\sharp Q \subseteq^\sharp P, Q \).

**Proof:** From the proposition 4.1.6 of greatest lower bound of the Smyth ordering, it follows that \( glb \) of two programs \( P \) and \( Q \) is \( P \cup Q \) and it is less refined than that of \( P \) and \( Q \).

**Proposition 4.5.2** \( PR_{\text{nod}}Q = P \cup Q = P \sqcup^\flat Q \supseteq^\flat P, Q \).

**Proof:** From the proposition 4.1.11 of least upper bound on the Hoare ordering, it follows that, non-deterministic choice between two programs is an improvement, i.e, \( PR_{\text{nod}}Q \supseteq^\flat P, Q \).

From the above postulates we can deduce that the Smyth ordering views non-determinism as demonic and try improve a program by reducing the non-determinism. Contrastingly, The Hoare ordering views the non-determinism as angelic, and keeps the non-determinism in its refinements.

**Proposition 4.5.3** If \( P \sqsubseteq^\sharp Q \) then \( PR_{\text{nod}}Q = P \).

**Proof:** The non-deterministic choice between programs \( P \) and \( Q \) is \( P \cup Q \). Because \( P \sqsubseteq^\sharp Q \), it means \( P \supseteq Q \), so \( P \cup Q = P \).
Similarly we can prove the following proposition that if $Q$ is a refinement to $P$, then non-deterministic choice in the Hoare ordering is $Q$.

**Proposition 4.5.4** If $P \sqsubseteq^b Q$ then $PR_{nod}Q = Q$.

**Law 4.5.1** : $P; (QR_{nod}R); S = (P; Q; S)R_{nod}(P; R; S)$

**Proof**: sequential composition is distributive through the non-determinism. The non-deterministic choice of two programs $Q$ and $R$ is $Q \cup R$ in both the Smyth and Hoare orderings and sequential composition is distributive through union of two program transaction sets, so the sequential composition is distributive.

**Law 4.5.2** : If $P \sqsubseteq Q$ then $(PR_{nod}R) \sqsubseteq (QR_{nod}R)$

This law proves that non-deterministic choice is distributive over refinement.

**Proof**:

- $P \sqsubseteq^s Q$ then $P \sqcap^s R \sqsubseteq^s Q \sqcap^s R$

  From the Smyth ordering, non-deterministic choice $P \sqcap^s R = P \cup R$ and $Q \sqcap^s R = Q \cup R$. Because $P \sqsubseteq^s Q$ it follows that $\forall t' \in Q \cup R$, there exists a $t \in P \cup R$ such that $t \sqsubseteq^s t'$.

- $P \sqsubseteq^b Q$ then $P \sqcup^b R \sqsubseteq^b Q \sqcup^b R$.

  We can use similar approach as above to prove that, non-determinism is distributive on Sequential composition with respect to the Hoare ordering.

**Law 4.5.3** : $(R \sqsubseteq P \land R \sqsubseteq Q) \equiv R \sqsubseteq (P \sqcap Q)$

**Proof**:
\begin{itemize}
  \item \((R \sqsubseteq^x P \land R \sqsubseteq^x Q) \equiv R \sqsubseteq^x (P \sqcap^x Q)\)

  In the Smyth ordering \((R \sqsubseteq^x P \land R \sqsubseteq^x Q)\) means that every element \(t'\) in \(P\) or \(Q\), there exist an element \(t\) in \(R\) such that \(t \sqsubseteq t'\). So each elements in the meet of the programs \(P\) and \(Q\) \((P \cup Q)\) has an element in \(R\) which is less then or equal to it.

  \item \((R \sqsupset^y P \land R \sqsupset^y Q) \equiv R \sqsupset^y (P \sqcap^y Q)\).

  similarly we can prove this in Hoare ordering where the meet of the programs is \(P \cap Q\).
\end{itemize}
CHAPTER 5

Future Work

5.1 Operational Semantics view in Relational model

To give an operational view of the semantics we need provide a methodology to show how a program is going to execute on a machine. In this context we are referring to the actual program statements. We are going to use the same methodology used by Hoare et al. [1] where single step in operational semantics is defined as refinement using algebraic laws of improvement, but we are going to use relational semantics developed in the previous section and refinement to achieve this goal.

From our view of a machine as collection of states and state transitions, it follows that programs can be divided into simple atomic steps, where each step corresponds to a single state transition. As each statement (atomic step) executes the state of the machine is changed. It will continue to execute this until no more statements in the program or no more transitions are possible.

A machine state in a stored program machine can be represented as \((s, p)\) where \(s\) is the data state of the program and \(p\) is the program text. Meaning will be represented with capital letters, i.e. \((S, P)\) where \(S\) is the program state itself and \(P\) is the state transitions. Instead of viewing programs as sequence of statements or equations, we are going to view them as atomic state transitions. This assumption is intuitive as every statement can be divided into a sequence of atomic statement, and each atomic statement
can be represented as a state transition using the relational model. This view of programs will enable us to get properties of programs and their execution.

**Definition 5.1.1** (Step Relation): \((s, P) \longrightarrow (s', Q) \stackrel{(def)}{=} (s, P) \sqsubseteq (s', Q)\)

Let \(s\) is the initial data state and \(P\) is the program \((P \subseteq \text{Prog})\), then the execution of a transition \(T: s \rightarrow s'\) where \(T \in P\) will result in a new machine state \((s', Q)\) where \(s'\) is the new data state and \(Q\) is the new program. The new machine state will be an improvement to the old machine state, i.e. we can replace the old machine state \((s, P)\) with more refined \((s', Q)\).
CHAPTER 6

Conclusion

In this thesis we showed a relational model for specifying the program semantics and some of its core properties. Relational model enables us to better understand the program specifications and their properties. Relational semantics may seem less abstract and more complex than some other formal semantics, but traditional semantics can not express the idea of a single step in a program and non-determinism as effectively as the relational semantics. Future work should consist of proving the equivalence between different semantic methodologies so that programming language designers can use these methodologies interchanging as their requirement demands.


