Abstract. We present a semantic framework for logic programs based on first-order hereditary Harrop formulas. The denotational and operational semantics are defined in a uniform way by characterizing some basic algebraic operators, which are directly related to the syntactic structure of the language. This allows us to address problems such as the relation between the operational semantics and the denotational semantics, the existence of a denotation of hereditary Harrop programs and their properties of compositionality, correctness and minimality. We define also an abstract interpretation framework based on the abstract interpretation theory. With this approach, the abstract denotational definition, the transition system and goal-independent denotations are systematically derived from the concrete ones, by replacing the concrete semantic operators by their abstract optimal versions. An extension to the higher-order context is considered. The adaptation of the first-order semantic framework to the higher-order language allows us the study of new properties such as abstraction and types.

October 2002

A mi jinetuelo
Acknowledgments

First of all I would like to thank my advisor, Giorgio Levi, for his infinite patience and unconditional assistance in any circumstance. In particular, I wish to thank him for making possible my research in Pisa; a thing that changed my life. A very special thanks to Luciano García, for showing me a different way of thinking the computer science and for giving me his friendship with affection.

I am very grateful to the people whom encouraged me to go ahead in difficult periods. To all of them my deep and sincere appreciation.

Finally, I specially thank my parents, Ivo and my friends for giving me the force and support in every moment.
Contents

1 Preliminaries 1
  1.1 Basic Set Theory ................................. 1
      1.1.1 Sets ................................... 1
      1.1.2 Relations and Functions ................. 2
  1.2 First Order Languages ......................... 4
      1.2.1 Semantics of a First-Order Language ... 6
  1.3 Substitutions ................................ 7
  1.4 Domain Theory ................................ 8
      1.4.1 Complete Lattices and Continuous Functions ... 8
      1.4.2 Complete Partial Orders and Lattices ...... 9
      1.4.3 Continuous and Additive Functions ........ 9
      1.4.4 Function Space ........................... 10
      1.4.5 Fixed-point Theory ....................... 10
  1.5 Abstract Interpretation ....................... 12
      1.5.1 Correctness and precision of Abstract Semantic Functions .. 13

2 First Order Hereditary Harrop Formulas 17
  2.1 The Language .................................. 17
  2.2 Intuitionistic Proof Procedure ................ 18
      2.2.1 Labelled Unification ...................... 19
      2.2.2 The Proof Procedure ..................... 19
  2.3 Semantic Domains .............................. 22
  2.4 Denotational Semantics ....................... 22
      2.4.1 An Example of Bottom-Up Denotation ........ 28
  2.5 Operational Semantics ....................... 32
      2.5.1 A New Transition System ................ 33
  2.6 Semantic Properties ........................... 46
  2.7 Discussion .................................... 51
# 3 An Abstract Interpretation Framework  53
  3.1 The Abstraction Framework  .................................... 53
      3.1.1 The Observables  ........................................ 54
      3.1.2 Abstract Semantics  ..................................... 55
      3.1.3 Perfect Observables  .................................... 58

# 4 Higher-Order Hereditary Harrop Formulas  69
  4.1 The Language  ................................................ 69
      4.1.1 Equivalence between terms  .............................. 71
      4.1.2 Normalization  .......................................... 72
      4.1.3 Unification  ............................................ 72
      4.1.4 The $hohh$-Formulas  .................................. 75
  4.2 The Proof Procedure  .......................................... 75
  4.3 Denotational and Operational Semantics  .................... 78
  4.4 Semantic Properties  ......................................... 84
  4.5 Higher-order Abstraction and Types  ......................... 85
      4.5.1 The Abstraction Framework  ............................. 85
      4.5.2 Monotype Abstraction  ................................ 88

Conclusions  95

Bibliography  97
Introduction

The aim of this work is the definition of a semantics for logic programs based on hereditary Harrop formulas, expressed in terms of intuitionistic proofs, in order to study the various properties of such programs. This work is also aimed at the construction of a semantic framework to systematically derive more abstract semantics, using the formal tools of abstract interpretation, that will allow the study of the relationship between semantics at different levels of abstraction.

Traditionally, the most studied logic programming languages are based on the logic of Horn clauses. Nevertheless, the simplicity of this logic prevents the development of high level languages according to concepts of modern day programming languages. For example, the transparent implementation of features like abstraction and modularity is not possible. With only Horn clauses, we are unable to realize, in a natural fashion, mechanisms of scoping of terms and predicates to make them visible only in specific contexts.

Higher-order programming is a desired feature of modern programming languages. There are several implementations of functional, logic and procedural language paradigms that allow procedures and sentences to be encapsulated into data structures in such a way that they can subsequently be retrieved and used to complete computations. Higher-order programming allows programs to reason on, calculate and execute procedures. In logic programming, early implementations of Prolog developed higher-order mechanisms by means of some pseudo-logical predicates for terms and predicate construction, commonly referred as meta-programming predicates. However, the use of ad hoc mechanisms, in contraposition with the declarative nature of pure logic programs lead to a loss of the expressiveness and the notion of what it really means with respect to the fundamental of a logic system. For this reason, in the last years, the study of richer logics has become an interesting topic of investigation.

Hereditary Harrop formulas can be divided in two types: first-order hereditary Harrop formulas (fohh-formulas for short) and higher-order hereditary Harrop formulas (hohh-formulas). Many operational interpretations of the intuitionistic theory of hereditary Harrop formulas are the fundamentals of programming features like higher-order programming, polymorphic typing, modular programming, abstract data types and \(\lambda\)-abstraction on data structures.

There exist a relation between the hereditary Harrop formulas and other logic
systems like Horn clauses. The hereditary Harrop formulas preserve the features of Horn clauses and extend it by permitting implication and universal quantifiers in goals. In particular, \textit{hohh}-Formulas can be seen as an extension to first-order Horn clauses (\textit{fohc}) along two dimensions. Both of these extensions provide for different notions of abstractions. The extension from first-order to higher order allows for higher-order variables and quantification: this extension provides for higher-order programming and \textit{\lambda}-abstractions in terms. The extension to hereditary Harrop formulas allows for the nesting of implication and universal quantifiers: this extension allows for hypothetical reasoning, abstract data type, etc. The \textit{hohh}-Formulas are very related with the notion of intuitionistic provability in the sense that an interpretation in such language corresponds to the construction of an intuitionistic proof for a formula (goal) given some set of clauses. First-order Horn clauses constitutes the logical foundation of languages like Prolog, while \textit{hohh}-Formulas are at the base of languages like \textit{\lambda}Prolog.

\section{The Semantics}

The proof procedures presented by Nadathur in [43] constitute the basis of the proof mechanism of languages like \textit{\lambda}Prolog and specify an operational model of such kind of languages. There are various attempts of defining semantics for \textit{fohh}-programs. In [39] a semantics is presented, which is based on a generalization of the standard immediate consequences \(T_P\) ground operator [35], to describe a Kripke-like model theory of positive Horn clauses programs permitting implications. There is also a \(T_P\)-like approach using categories [20]. These semantics are based on the notion of success set and therefore are not adequate to reason about important properties of programs as well as about their behavior.

One important point in understanding the program meaning is the equivalence of programs, which is based on our ability to detect when two programs can not be distinguished by observing their behaviors. The inadequacy of the previously mentioned approaches to study the observational equivalence of classical logic programs is well known [16]. In that sense the \(s\)-semantics [18, 22], by extending the Herbrand interpretation, really captures the operational semantics, and is therefore suitable for semantic based logic analysis and to reason in terms of program equivalences. Unfortunately, this approach can not be used in \textit{fohh}-programs, because the introduction of implication in goals invalidates the use of the computed answers obtained from pure atomic goals for defining program equivalences. In fact, the implication in the \textit{fohh} case is operationally treated by adding a new clause to the program. This approach can dramatically change the meaning of programs that are equivalent in terms of the \(s\)-semantics. Hence, to have a goal-independent semantics, it is not enough to reason about the computed answers in the way it is achieved with the \(s\)-semantics. We need therefore a semantics capable of representing how the answers are calculated. Furthermore, it must be compositional with respect to all language
operators. That means that our semantics must be compositional with respect to the union of programs, i.e., the addition of new program modules. In [34], a complete semantics for \textit{hohh}-logic is presented. The authors developed a sound and complete semantics for the intuitionistic fragment of the Church's theory of types and adapt it to the \(\lambda\)-Prolog fragment of this theory by means of the Uniform Algebras. Nevertheless, our approach requires a semantics that can be easily translated in terms of effective algorithms for program analysis and verification.

Our work follows the methodology of the semantics first presented in [6, 7, 8]. However, we have to face the problem that more complex structure of derivations makes difficult to find the properties on which our approach is based, i.e., the properties of compositionality. We propose a more general operational semantics (a top-down denotation) based on derivations, associating to each program all derivations constructed by the proof procedure for \textit{fohh}-programs, from all possible pure atomic goals. Besides a notion of derivation, we also want to have a denotational semantics which allows us to define goal-independent denotations, and therefore to define the semantics of a \textit{fohh}-program as a set of procedures. In addition, the denotational semantics we present is equivalent to the top-down denotation, and it is defined by means of various operators that provide the compositional nature of the semantics. The semantic operators used to define the bottom-up denotation are the counterpart of the syntactic ones. From an intuitive point of view this semantics is minimal with respect to the properties we are trying to preserve. Furthermore, in our case the semantics is the most concrete, provided that the derivations reflect all low-level operational details of the proof procedure. All results obtained for \textit{fohh}-programs were extended as well to \textit{hohh}-programs. We have reused and redefined all theoretic conclusions and applied it to the higher-order context.

Once we have defined a concrete semantics for our logic programming language, we can start to model some interesting properties of computations like computed answer substitutions, partial answers, call patterns and correct answer substitutions. For this purpose, it is necessary to develop a theory that allows us to create model abstractions of denotations based on \textit{fohh}-derivations and \textit{hohh}-derivations, inheriting all properties of the concrete semantics. All this approach is based on the Abstract Interpretation Theory [11].

Abstract Interpretation Theory is successfully used to reason about the relationship among different semantics and to statically analyze programs. With abstract interpretation we can compare different semantics at different levels of abstraction. Also, it is possible to systematically derive abstract semantics which are correct approximations of the concrete semantics.

One of the aims of this work, is also to establish a basis for techniques such as static analysis, verification, debugging and other programming tools. In particular, program verification, which consists in determining whether or not a given program has certain properties, is a suitable field of application for observables modeled by an abstract semantics. Basically, we can express a specification of the program behavior by abstracting the intended semantics of the program with respect to some
particular observables. By analyzing the relationship between the specification and the modeled behavior of the program, it is possible to establish when the behavior of the program is correct with respect to the desired one.

I.2 Plan of the thesis

The thesis is organized as follows. Notations and basic concepts of programming syntax and semantics are introduced in Chapter 1. We also give an introduction to abstract interpretation.

In Chapter 2 we define a semantic framework to reason about compositional properties of $fohh$-program derivations (proofs). We obtain the collecting semantics which gives the minimal amount of information on computations, necessary to rebuild all the processed computations. This semantics will allow us to observe all internal details of derivations. Moreover, we address problems such as the relation between the operational semantics and the denotational semantics, the existence of a denotation for a set of definite clauses and their properties of compositionality with respect to various operators.

In Chapter 3 we present the semantic abstraction framework together with a class of perfect observables which preserves the most desired properties of the concrete semantics, such as OR-compositionality and compositionality of abstract operators. The observables are formalized by using Galois insertions between the domain of derivations and abstract domains. By using the interaction between the properties of the abstraction and the properties of the concrete primitive semantic operators, we can easily inherit, in the abstract case, all those properties of the collecting semantics for which the suitable lemmata on the semantic operators hold.

In Chapter 4 we define a semantic framework for $hohh$-programs as an extension to the semantics of $fohh$-programs. We have reused and redefined all theoretic results and applied it to the higher-order context. A semantic abstraction framework is presented, and specifically, an observable for type inferencing in higher-order derivations is analyzed.

Finally, the last chapter is dedicated to the conclusive remarks.
Chapter 1

Preliminaries

In this chapter we introduce some notations which will be used in the thesis.


1.1 Basic Set Theory

To define the basic notions we will use the standard meta logical notation to denote conjunction, disjunction, quantification and so on (and, or, for each, ...). For statements (assertions) $A$ and $B$ we will use abbreviations like:

- $A, B$ for $(A$ and $B)$, the conjunction of $A$ and $B$,
- $A \Rightarrow B$ for $(A$ implies $B)$, meaning (if $A$ then $B$)
- $A \Leftrightarrow B$ for $(A$ if and only if $B)$, meaning logical equivalence

We will also make statements by forming disjunctions ($A$ or $B$) with the evident meaning, and negations (not $A$), sometimes written $\neg A$, which is true if and only if $A$ is false.

A statement $P(x, y)$, involving variables $x$ and $y$, is called a predicate (property, relation, condition) and it becomes true or false when $x$ and $y$ stands for particular things. We use logical quantifiers $\exists$ ("there exists") and $\forall$ ("for all") to write assertions.

1.1.1 Sets

Intuitively, a set is a collection of objects, which are elements of it. We write $a \in S$ when $a$ is an element of the set $S$. Moreover we write $\{a, b, c, \ldots\}$ for the set of elements $a, b, c, \ldots$.

A set $S'$ is said to be a subset of a set of a set $S$, written $S' \subseteq S$, if and only if every element of $S'$ is an element of $S$, i.e., $S' \subseteq S \Leftrightarrow \forall x \in S'.x \in S$. A set is
1.1.2 Relations and Functions

A set can be determined by a property \( P \). We will write \( S := \{ x \mid P(x) \} \), meaning that the set \( S \) has as elements all those \( x \) for which \( P(x) \) is true. We will not be formal about it, but we will avoid problems like Russell’s paradox and will have a world of sets rich enough to support most mathematics. This will be achieved by assuming that certain given sets exists from the start and by using safe methods for constructing sets.

We write \( \emptyset \) for the null or empty set and \( \mathbb{N} \) for the set of natural numbers \( 0,1,2,\ldots \). The cardinality of a set \( S \) is denoted by \( \text{card}(S) \). A set \( S \) is called \textit{denumerable} if \( \text{card}(S) = \text{card}(\mathbb{N}) \) and \textit{countable} if \( \text{card}(S) \leq \text{card}(\mathbb{N}) \).

Let \( S \) be a set and \( P(x) \) be a property. By \( \{ x \mid P(x) \} \) we denote the set \( \{ x \mid x \in S, P(x) \} \). The powerset of a set \( S \), \( \{ S' \mid S' \subseteq S \} \), is denoted by \( \wp(S) \).

Let \( I \) be a set. By \( \{ x_i \}_{i \in I} \) we denote the set of unique objects \( x_i \), for any \( i \in I \). The elements \( x_i \) are said to be \textit{indexed} by the elements \( i \in I \).

The \textit{union} of two sets is \( S \cup S' := \{ x \mid x \in S \text{ or } x \in S' \} \). Let \( S \) be a set of sets, \( \bigcup S = \{ x \mid \exists S \in S. x \in S \} \). When \( S = \{ S_i \}_{i \in I} \), we write \( \bigcup S \) as \( \bigcup_{i \in I} S_i \). The \textit{intersection} of two sets is \( S \cap S' := \{ x \mid x \in S \text{ and } x \in S' \} \). Let \( S \) be a set of sets, \( \bigcap S = \{ x \mid \forall S \in S. x \in S \} \). When \( S = \{ S_i \}_{i \in I} \), we write \( \bigcap S \) as \( \bigcap_{i \in I} S_i \).

The Cartesian product of \( S \) and \( S' \) is the set \( S \times S' := \{ (x,y) \mid x \in S, y \in S' \} \).

1.1.2 Relations and Functions

A \textit{binary relation} between \( S \) and \( S' \) (\( R : S \times S' \)) is an element of \( \wp(S \times S') \). We write \( x R y \) for \( (x,y) \in R \).

A \textit{partial function} from \( S \) to \( S' \) is a relation \( f : S \times S' \) for which \( \forall x,y,y'. (x,y) \in f, (x,y') \in f \Rightarrow y = y' \). By \( f : S \rightarrow S' \) we denote a partial function of the set \( S \) (the \textit{domain} ) into the set \( S' \) (the \textit{range} ). The set of all partial functions from \( S \) to \( S' \) is denoted by \( [S \rightarrow S'] \). Moreover, we use the notation \( f(x) = y \) if there exists a \( y \) such that \( (x,y) \in f \) and we will say \( f(x) \) is defined, otherwise \( f(x) \) is undefined.

A (total) function \( f \) from \( S \) to \( S' \) is a partial function from \( S \) to \( S' \) such that, for all \( x \in S \), there is some \( y \in S' \) such that \( f(x) = y \). Although total functions are a special kind of partial functions, it is a tradition to understand something described as simply a function to be a total function. So we will always say explicitly when a function is partial. To indicate that a function \( f \) from \( S \) to \( S' \) is total, we write \( f : S \rightarrow S' \). Moreover, the set of all (total) functions from \( S \) to \( S' \) is denoted by \( [S \rightarrow S'] \).

A function \( f : S \rightarrow S' \) is \textit{injective} if and only if for each \( x,y \in S \) if \( f(x) = f(y) \) then \( x = y \). \( f \) is \textit{surjective} if and only if for each \( x' \in S' \) there exists \( x \in S \) such that \( f(x) = x' \).

We denote by \( f = g \) the extensional equality, i.e., for each \( x \in S \), \( f(x) = g(x) \). Furthermore, \( g := f[^v/x] \) denotes the function \( g \) which differs from \( f \) only for the
assignment of \( v \) to \( x \), i.e., \( g(x) = v \) and, for each \( y \neq x \), \( g(y) = f(y) \). The application of any element of a set \( S' \subseteq S \) is denoted by \( f_{|S'} \).

**Lambda Notation**

It is sometimes useful to use the lambda notation to describe functions. It provides a way of referring to functions without having to name them. Suppose \( f : S \to S' \) is a function which, for any element \( x \in S \), gives a value \( f(x) \) which is exactly described by the expression \( E \), probably involving \( x \). Then we can write \( \lambda x \in S. E \) for the function \( f \). Thus, \( (\lambda x \in S.E) := \{(x, E[x]) \mid x \in S\} \) and so \( \lambda x \in S.E \) is just an abbreviation for the set of input-output values determined by the expression \( E[x] \).

We use also the lambda notation to denote partial functions by allowing expressions in lambda-terms that are not always defined. Hence, a lambda expression \( \lambda x \in S.E \) denotes a partial function \( S \to S' \) which, on input \( x \in S \), assumes the value \( E[x] \in S' \), if the expression \( E[x] \) is defined, and otherwise it is undefined.

**Composing Relations and Functions**

We compose relations, and so partial functions and total functions, \( R : S \times S' \) and \( Q : S' \times S'' \) by defining their composition (a relation between \( S \) and \( S'' \)) by \( Q \circ R := \{(x, z) \in S \times S'' \mid y \in S', (x, y) \in R, (y, z) \in Q\} \). \( R^n \) is the relation

\[
\underbrace{R \circ \ldots \circ R}_n
\]

i.e., \( R^1 := R \) and (assuming \( R^n \) is defined) \( R^{n+1} := R \circ R^n \). Each set \( S \) is associated with an identity function \( \text{Id}_S := \{(x, x) \mid x \in S\} \), which is the neutral element of \( \circ \).

Thus we define \( R^0 := \text{Id}_S \). The transitive and reflexive closure \( R^* \) of a relation \( R \) on \( S \) is \( R^* := \bigcup_{i \in \mathbb{N}} R^i \).

The function composition of \( g : S \to S' \), and \( f : S' \to S'' \) is the partial function \( f \circ g : S \to S'' \), where \( (f \circ g)(x) := f(g(x)) \), if \( g(x) \) (first) and \( f(g(x)) \) (then) are defined, and it is otherwise undefined. When it is clear from the context \( \circ \) will be omitted.

A function \( f : S \to S' \) is bijective if it has an inverse \( g : S' \to S \), i.e., if and only if there exists a function \( g \) such that \( f \circ g = \text{Id}_{S'} \). Then the sets \( S \) and \( S' \) are said to be in 1-1 correspondence. Any set in 1-1 correspondence with a subset of the natural numbers \( \mathbb{N} \) is said to be countable. Notice that a function \( f \) is bijective if and only if it is injective and surjective.

**Direct and Inverse Image of a Relation**

We extend relations, and thus partial and total functions, \( R : S \times S' \) to functions on subsets by taking \( R(X) := \{y \in S' \mid \exists x \in X. (x, y) \in R\} \) for \( X \subseteq S \). The set \( R(X) \) is called the direct image of \( X \) under \( R \). We define \( R^{-1}(Y) := \)
\{ x \in S \mid \exists y \in Y. (x, y) \in R \} for Y \subseteq S'. The set \( R^{-1}(Y) \) is called the inverse image of \( Y \) under \( R \). Thus, if \( f: S \to S' \) is a partial function, \( X \subseteq S \) and \( X' \subseteq S' \), we denote by \( f(X) \) the image of \( X \) under \( f \), i.e., \( f(X) := \{ f(x) \mid x \in X \} \) and by \( f^{-1}(X) \) the inverse image of \( X' \) under \( f \), i.e., \( f^{-1}(x) := \{ x \mid f(x) \in X' \} \).

### Equivalence Relations and Congruences

An equivalence relation \( \approx \) on a set \( S \) is a binary relation on \( S \) (\( \approx := S \times S \)) such that, for each \( x, y, z \in S \),

\[
\begin{align*}
xRx & \quad \text{(reflexivity)} \\
xRy & \Rightarrow yRx \quad \text{(symmetricity)} \\
xRy, yRz & \Rightarrow xRz \quad \text{(transitivity)}
\end{align*}
\]

The equivalence class of an element \( x \in S \), with respect to \( \approx \), is the subset \([x]_\approx := \{ y \mid x \approx y \} \). When clear from the context we abbreviate \([x]_\approx \) by \([x]\) and often abuse notation by letting the elements of a set denote their correspondent equivalence classes. The quotient set \( S/\approx \) of \( S \) modulo \( \approx \) is the set of equivalence classes of elements in \( S \) (with respect to \( \approx \)).

An equivalence relation \( \approx \) on \( S \) is a congruence with respect to a partial function \( f: S^n \to S \) if and only if, for each pair of elements \( a_i, b_i \in S \) such that \( a_i \approx b_i \) (if \( f(a_1, \ldots, a_n) \) is defined then also \( f(b_1, \ldots, b_n) \) is defined and)

\[ f(a_1, \ldots, a_n) \approx f(b_1, \ldots, b_n). \]

Then, we can define the partial function \( f_\approx : (S/\approx)^n \to S/\approx \) as

\[ f_\approx ([a_1]_\approx, \ldots, [a_1]_\approx) := [f(a_1, \ldots, a_n)]_\approx, \]

since, given \([a_1]_\approx, \ldots, [a_n]_\approx\), the class \([f(a_1, \ldots, a_n)]_\approx\) is uniquely determined independently of the choice of the representatives \( a_1, \ldots, a_n \).

### 1.2 First Order Languages

A first order language consists of an alphabet and a set of well-formed formulas defined on it. An alphabet contains two kinds of symbols: the logical symbols and the non-logical ones. Are the set we consider are countable.

We will mainly concerned with classical logic proof and semantics so the logical symbols consists on the sets

- logical connectives: \( \neg \) (negation), \( \Rightarrow \) (implication), \( \lor \) (conjunction), \( \land \) (disjunction), \( \Leftrightarrow \) (equivalence),
- propositional constants: \( true \) and \( false \),
1.2. FIRST ORDER LANGUAGES

• quantifiers: \( \exists \) (there exists) and \( \forall \) (for all),
• punctuation symbols: parentheses, comma and full stop.

They are common to all first order languages we consider. The non-logical symbols determine specific first order languages and consists on the sets

• a set \( \Sigma \) of function symbols of fixed arities,
• an infinite set \( V \) of variables,
• a set \( \Pi \) of predicate symbols of fixed arities.

A first order language is denoted by a triple \( \langle \Sigma, \Pi, V \rangle \), which lists the sets of its non-logical symbols. A (predicate or function symbol) with arity \( n \), will be denoted as \( f^{(n)} \). Function symbols of arity 0 are called also constants symbols. We write \( f, g \) for function symbols, \( p, q \) for predicate symbols, \( x, y \) for variables and \( x, y \) for tuples of distinct variables.

The set \( T_{\Sigma,V} \) of the terms of a language \( L = \langle \Sigma, \Pi, V \rangle \) is the least set such that

1. \( V \subseteq T_{\Sigma,V} \),
2. \( f^{(0)} \in \Sigma \) implies \( f^{(0)} \in T_{\Sigma,V} \),
3. \( f^{(n)} \in \Sigma \) and \( t_1, \ldots, t_n \in T_{\Sigma,V} \) implies \( f(t_1, \ldots, t_n) \in T_{\Sigma,V} \).

We write \( t, s \) for terms in \( T_{\Sigma,V} \), \( t, s \) for tuples of terms. A term is ground if it does not contain variables. We will denote the set of ground terms by \( T_\Sigma \).

Finally we can define the set of well formed formulas, or simply, formulas of a language \( L = \langle \Sigma, \Pi, V \rangle \) as follows

1. \( \textit{true} \) and \( \textit{false} \) are well formed formulas,
2. if \( p^{(n)} \in \Pi \) and \( t_1, \ldots, t_n \in T_{\Sigma,V} \) then \( p(t_1, \ldots, t_n) \) is a well formed formula, also called \( \textit{atomic formula} \), \( \textit{atom} \) or also called \( \textit{pure atomic formula} \) when \( t_1, \ldots, t_n \) are distinct variables,
3. if \( F \) and \( G \) are well formed formulas, then \( \neg F \), \( F \Rightarrow G \), \( F \lor G \), \( F \land G \) and \( F \iff G \) are well formed formulas,
4. if \( F \) is well formed formula and \( x \in V \), then \( \exists x F \) and \( \forall x F \) are well formed formulas,
5. no other expression is a well order formula.
1.2.1 Semantics of a First-Order Language

The semantics of a first order language can be defined as follows.

An interpretation $I = \langle D, \Lambda, \Gamma \rangle$ for a first order language $L = \langle \Sigma, \Pi, V \rangle$ consists of

1. a non-empty set $D$, the domain of the interpretation,
2. a function $\Lambda_f : D^n \rightarrow D$, for each function symbol $f^{(n)} \in \Sigma$,
3. a subset $\Gamma_p \subseteq D^n$, for each predicate symbol $p^{(n)} \in \Pi$.

Observe that to each element $f^{(0)} \in \Sigma$, corresponds an element of $D$.

A variable assignment $\sigma : V \rightarrow D$ maps each variable into an element of $D$. It can be lifted homomorphically to a function, still denoted by $\sigma$ and called valuation, which maps terms of $T_{\Sigma, V}$ to elements of $D$.

The semantics (validity) of a formula $F$ under a variable assignment $\sigma$ over $I$, written $I \models_\sigma F$, can be defined inductively on its structure as follows:

- $I \models_\sigma true$ and not $I \models_\sigma false$,
- $I \models_\sigma p(t_1, \ldots, t_n)$ if and only if $(\sigma(t_1), \ldots, \sigma(t_n)) \in \Gamma_p$,
- $I \models_\sigma \neg F$ if and only if not $I \models_\sigma F$,
- $I \models_\sigma F \land G$ in and only if $I \models_\sigma F$ and $I \models_\sigma G$,
- $I \models_\sigma (\exists x)F$ if and only if there exists $d \in D$ such that $I \models_{\sigma[x/d]} F$.

The semantics of the other connectives can be expressed by means of the semantics of $\neg$, $\land$ and $\exists$, by defining

- $I \models_\sigma F \lor G$ if and only if $I \models_\sigma \neg (\neg F \land \neg G)$,
- $I \models_\sigma F \Rightarrow G$ if and only if $I \models_\sigma \neg F \lor G$,
- $I \models_\sigma F \Leftrightarrow G$ if and only if $I \models_\sigma F \Rightarrow G$ and $I \models_\sigma G \Rightarrow F$,
- $I \models_\sigma (\forall x)F$ if and only if $I \models_\sigma \neg (\exists x \neg F)$.

We say that $F$ is true in an interpretation $I$, and write $I \models F$, if and only if, for each variable assignment $\sigma$, $I \models_\sigma F$ is true. Moreover, $I$ is a model of a set of formulas $S$ if and only if each formula of $S$ is true in $I$. A set of formulas $S$ is satisfiable (or consistent) if and only if $S$ has a model. Otherwise it is unsatisfiable (or inconsistent). Given two sets of formulas $S_1$ and $S_2$, $S_2$ is logical consequence of $S_1$ if and only if every model of $S_1$ is a model of $S_2$. 
A particular important will be given to Herbrand interpretations. The Herbrand universe $U_L$ for a language $L$, is the set of terms in $T_\Sigma$ while the Herbrand base $B_L$ of $L$ is the set of all ground terms of $L$. A Herbrand interpretation $H$ is an interpretation $\langle U_L, \Lambda, \Gamma \rangle$, with domain the set of ground terms, for each $f^{(n)} \in \Sigma$,

$$\Lambda_f(t_1, \ldots, t_n) = f(t_1, \ldots, t_n), \text{ for each } t_1, \ldots, t_n \in U_L,$$

and, for each $p^{(n)} \in \Pi$, $\Gamma_p \subseteq U^n_L$. Each Herbrand interpretation $H$ can be uniquely determined by a set $I \subseteq B_L$, such that $p(t_1, \ldots, t_n) \in I$ if and only if $(t_1, \ldots, t_n) \in \Gamma_p$, then, we can identify a Herbrand interpretation with the set of atoms it makes true. Herbrand interpretations (as set of atoms) with the subset ordering are a complete lattice (see next section).

A non ground Herbrand interpretation $H = \langle T^{n}_{\Sigma, V}, \Lambda, \Gamma \rangle$ is an interpretation, where the domain is the set of all terms, for each $f^{(n)} \in \Sigma$,

$$\Lambda_f(t_1, \ldots, t_n) = f(t_1, \ldots, t_n), \text{ for each } t_1, \ldots, t_n \in T^{n}_{\Sigma, V},$$

and for each $p^{(n)} \in \Pi$, $\Gamma_p \subseteq T^{n}_{\Sigma, V}$.

### 1.3 Substitutions

Let $\Sigma$ be a set of function symbols and $V$ be an infinite set of variables. A substitution is a mapping $\theta : V \rightarrow T^{n}_{\Sigma, V}$ which differs from the identity on a finite set of variables. Given a substitution $\theta$, the expression $\text{dom}(\theta)$ denotes the set $\{x \in V \mid \theta(x) \neq x\}$. We will often use the set-theoretic notation

$$\theta = \{[t_1/x_1], \ldots, [t_n/x_n]\}$$

to represent $\theta$. Notice that by definition of substitution, the variables in $x_1, \ldots, x_n$ are distinct and $t_i \neq x_i$ for $i = 1, \ldots, n$. A pair $[t/x]$ is called a binding and $[t/x] \in \theta$ means that $\theta(x) = t$. The empty substitution $\varepsilon$ is defined as $\varepsilon(x) = x$, for each $x \in V$.

A substitution $\theta$ can be extended homomorphically to any syntactic expression $E$. The application of $\theta$ to an expression $E$ is denoted by $E\theta$ and corresponds to the expression obtained by replacing each variable $x$ in $E$ by the term $\theta(x)$. $E\theta$ is called an instance of $E$. Given two terms $t_1, t_2$ we write $t_1 \preceq t_2$ (1 is more general than $t_2$) if $t_2$ is an instance of $t_1$. The relation $\preceq$ is a preorder (called subsumption) and by $\equiv$ we denote the induced equivalence relation (variance). Therefore, two terms $t_1, t_2$ are variants, written $t_1 \equiv t_2$ if and only if $t_1$ is an instance of $t_2$ and vice versa. This notion can be obviously extended to any kind of expression.

The composition $\theta \sigma$ of the substitutions $\theta$ and $\sigma$ is defined as $(\theta \sigma)(x) := (x\theta)\sigma$. By using set-theoretic notation, if $\theta = \{[t_1/x_1], \ldots, [t_n/x_n]\}$ and $\sigma = \{[s_1/y_1], \ldots, [s_n/y_n]\}$, the composition $\theta \sigma$ is defined by removing from the set

$$\{[t_1/x_1], \ldots, [t_n/x_n]\} \setminus \{[s_1/y_1], \ldots, [s_n/y_n]\}.$$
the bindings \([t_i/x_i] \sigma\) for which \(x_i = t_i \sigma\) and the bindings \([s_i/x_i]\) such that \(y_j \in \{x_1, \ldots, x_n\}\). Composition is associative and the empty substitution \(\varepsilon\) is the neutral element. Moreover, for each expression \(E\), \(E(\theta \sigma) = (E \theta) \sigma\).

A renaming is a substitution \(\rho\) for which there exists the inverse \(\rho^{-1}\) such that \(\rho \rho^{-1} = \rho^{-1} \rho = \varepsilon\). It can be easily shown that two terms \(t_1, t_2\) are variants if and only if there exists a renaming \(\rho\) such that \(t_1 = t_2 \rho\).

### 1.4 Domain Theory

We will present here the (abstract ) concepts of the complete lattices, continuous functions and fixed-point theory, which are the standard tools of denotational semantics.

#### 1.4.1 Complete Lattices and Continuous Functions

A binary relation \(\leq\) on \(S\) (\(\leq : S \times S\)) is a partial order if, for each \(x, y \in S\),

\[
\begin{align*}
    x &\leq x \quad \text{(reflexivity)} \\
    x \leq y, y \leq x &\implies x = y \quad \text{(antisymmetry)} \\
    x \leq y, y \leq z &\implies x \leq z \quad \text{(transitivity)}
\end{align*}
\]

A partially ordered set (poset) \((S, \leq)\) is a set \(S\) equipped with a partial order \(\leq\). A set \(S\) is totally ordered if it is partially ordered and, for each \(x, y \in S\), \(x \leq y\) or \(y \leq x\). A chain is a (possibly empty) totally ordered subset of \(S\).

A preorder is a binary relation which is reflexive and transitive. A preorder \(\leq\) on a set \(S\) induces on \(S\) an equivalence relation \(\approx\) defined as follows: for each \(x, y \in S\),

\[
    x \approx y \iff x \leq y, y \leq x
\]

Moreover, \(\leq\) induces on \(S/\approx\) the partial order \(\leq_{\approx}\) such that, for each \(\[x\]_{\approx}, \[y\]_{\approx} \in S/\approx\),

\[
    \[x\]_{\approx} \leq_{\approx} \[y\]_{\approx} \iff x \leq y
\]

A binary relation \(<\) is strict if and only if it is anti-reflexive (i.e., not \(x < x\)) and transitive.

Given a poset \((S, \leq)\) and \(X \subseteq S\), \(y \in S\) is an upper bound for \(X\) if and only if, for each \(x \in X\), \(x \leq y\). Moreover, \(y \in S\) is the least upper bound (called also join) of \(X\), if \(y\) is an upper bound of \(X\) and, for every upper bound \(y'\) of \(X\), \(y \leq y'\). A least upper bound of \(X\) is often denoted by \(\text{lub}_S X\) or by \(\sqcup_S X\). We also write \(\sqcup_S \{d_1, \ldots, d_n\}\) as \(d_1 \sqcup_S \cdots \sqcup_S d_n\). Dually an element \(y \in S\) is a lower bound for \(X\)
1.4.3 Continuous and Additive Functions

Let \((S, \leq)\) if and only if, for each \(x \in X, y \leq x\). Moreover, \(y \in S\) is the greatest lower bound (called also meet) of \(X\), if \(y\) is a lower bound of \(X\) and for every lower bound \(y'\) of \(X, y' \leq y\). A greatest lower bound of \(X\) is often denoted by \(\text{glb}_S X\) or by \(\cap_S X\). We also write \(\cap_S \{d_1, \ldots, d_n\} = d_1 \cap_S \cdots \cap_S d_n\). When it is clear from the context, the subscript \(S\) will be omitted. Moreover, \(\cup \{D_i\}_{i \in I}\) and \(\cap \{D_i\}_{i \in I}\) can be denoted by \(\sqcup_{i \in I} D_i\) and \(\sqcap_{i \in I} D_i\). It is easy to check that if the lub and glb exist, then they are unique.

1.4.2 Complete Partial Orders and Lattices

A direct set is a poset in which any subset of two elements (and hence any finite subset) has an upper bound in the set. A complete partial order (CPO) \(S\) is a poset such that every chain \(D\) has a least upper bound (i.e., there exists \(\sqcup D\)). Notice that any set ordered by the identity relation forms a CPO, of course without a bottom element. Such CPOs are called discrete. We can add a bottom element to any poset \((S, \leq)\) which does not have one (even to a poset which already has one).

The new poset \(S_\perp\) is obtained by adding a new element \(\perp\) to \(S\) and by extending the ordering \(\leq\) as \(\forall x \in S, \perp \leq x\). If \(S\) is a discrete CPO, then \(S_\perp\) is a CPO with bottom element, which is called flat.

A complete lattice is a poset \((S, \leq)\) such that for every subset \(X\) of \(S\) there exists \(\sqcup X\) and \(\sqcap X\). Let \(\top\) denote the top element \(\sqcup S = \sqcup \emptyset\) and \(\bot\) denote the bottom element \(\sqcap S = \sqcap \emptyset\) of \(S\). The elements of a complete lattice are thought of as points of information and the ordering as an approximation relation between them. Thus, \(x \leq y\) means \(x\) approximates \(y\) (or, \(x\) has less or the same information as \(y\) ) and so \(\bot\) is the point of least information. It is easy to check that, for any set \(S, \wp(S)\) under the subset ordering \(\subseteq\) is a complete lattice, where \(\sqcup\) is union, \(\sqcap\) is intersection, the top element is \(S\) and the bottom element is \(\emptyset\). Also \((\wp(S))_\perp\) is a complete lattice.

Given a complete lattice \((L, \leq)\), the set of all partial functions \(F = [S \rightarrow L]\) inherits the complete lattice structure of \(L\). Let simply define \(f \sqsubseteq g := \forall x \in S.f(x) \leq g(x)\), \((f \sqcup g)(x) := f(x) \sqcup g(x)\), \((f \sqcap g)(x) := f(x) \sqcap g(x)\), \(\bot_F := \lambda x \in S. \bot_L\) and \(\top_F := \lambda x \in S. \top_L\). In the following \(f + g\) will be used to denote \(f \sqcup g\).

1.4.3 Continuous and Additive Functions

Let \((L, \leq)\) and \((M, \subseteq)\) to be (complete) lattices. A function \(f : L \rightarrow M\) is monotonic if and only if

\[\forall x, y \in L. x \leq y \Rightarrow f(x) \subseteq f(y).\]

Moreover, \(f\) is continuous if and only if, for each non-empty chain \(D \subseteq L\),

\[f\left(\sqcup_L D\right) = \sqcup_M f(D)\]
Every continuous function is also monotonic, since \( x \leq y \Rightarrow \sqcup M \{ f(x), f(y) \} = f(\sqcup_L \{x, y\}) = f(y) \Rightarrow f(x) \sqsubseteq f(y) \).

Complete partial orders correspond to types of data (data that can be used as input or output to a computation) and computable functions are modeled as continuous functions between them.

A partial function \( f : S \rightarrow S' \) is additive if and only if the previous continuity condition is satisfied for each non-empty set. Hence, every additive function is also continuous. Dually we define co-continuity and co-additivity, by using \( \sqcap \) instead of \( \sqcup \).

It can be proved that the composition of monotonic, continuous or additive functions is, respectively, monotonic, continuous or additive.

A continuous function \( f : D \rightarrow E \) between CPOs \( D \) and \( E \) is said to be an isomorphism if there is a continuous function \( g : E \rightarrow D \) such that \( g \circ f = Id_D \) and \( f \circ g = Id_E \). Thus \( f \) and \( g \) are mutual inverses. This is actually an instance of a general definition which applies to a class of objects and functions between them (CPOs and continuous functions in this case). It follows from the definition that isomorphic CPOs are essentially the same but for a renaming of elements. It can be proved that a function \( f : D \rightarrow E \) is an isomorphism if and only if \( f \) is bijective and, for all \( x, y \in D, x \leq_D y \iff f(x) \leq_E f(y) \).

### 1.4.4 Function Space

Let \( D, E \) be CPOs. It is a very important fact that the set of all continuous functions from \( D \) to \( E \) can be made into a complete partial order. The function space \([D \rightarrow E]\) consists of continuous functions \( f : D \rightarrow E \) ordered pointwise by \( f \sqsubseteq g \iff \forall d \in D. f(d) \sqsubseteq g(d) \). This makes the function space a complete partial order.

Notice that, provided \( E \) has a bottom element \( \bot_E \) such a function space of CPOs has a bottom element, the constantly \( \bot_E \) function \( \bot_{[D \rightarrow E]} := \lambda d \in D. \bot_E \). Least upper bounds of chains of functions are given pointwise, i.e., a chain of functions \( f_0 \sqsubseteq f_1 \sqsubseteq \ldots \sqsubseteq f_n \ldots \) has a \( \text{lub} \sqcup_{[D \rightarrow E]} f_n := \lambda d \in D. \sqcup_E \{f_n(d)\}_{n \in \mathbb{N}} \).

It is not hard to see that the partial functions \( L \rightarrow D \) are in 1-1 correspondence with the (total) functions \( L \rightarrow D_\bot \), and that, in this case, any total function is continuous; the inclusion order between partial functions corresponds to the “pointwise order” \( f \sqsubseteq g \iff \forall \sigma \in L. f(\sigma) \sqsubseteq g(\sigma) \) between functions \( L \rightarrow D_\bot \). Because partial functions from a CPO so does the set of functions \([L \rightarrow D_\bot]\) ordered pointwise.

### 1.4.5 Fixed-point Theory

Given a poset \((S, \leq)\) and a function \( f : S \rightarrow S \), a fixed-point of \( f \) is an element \( x \in S \) such that \( f(x) = x \). A pre-fixpoint of \( f \) is an element \( x \in S \) such that \( f(x) \leq x \) and dually a post-fixpoint of \( f \) is an element \( x \in S \) such that \( x \leq f(x) \). Moreover, we say that \( x \in S \) is the least fixed-point of \( f \) (denote by \( \text{lfp} f \)) if and only if \( x \) is a
fixed-point of \( f \) and for all fixed-point \( y \) of \( f \), \( x \leq y \). Dually, we define the greatest fixed-point (denoted by \( \text{gfp} \ f \ )

The fundamental theorem of Knaster-Tarski states that the set of fixpoints of a monotonic function \( f \) is a complete lattice.

**Theorem 1.4.1 (Knaster-Tarski)** [47] A monotonic function \( f \) on a complete lattice \( (L, \leq) \) has the least fixed-point and the greatest fixed-point. Moreover,

\[
\text{lfp}(f) = \bigcap \{ x \mid f(x) \leq x \} = \bigcap \{ x \mid x = f(x) \}
\]

\[
\text{gfp}(f) = \bigcup \{ x \mid x \leq f(x) \} = \bigcup \{ x \mid x = f(x) \}
\]

The Knaster-Tarski Theorem is important because it applies to any monotone function on a complete lattice. However, most of the time we will concerned with least fixpoints of continuous functions which we will construct by techniques of the previous section, as least upper bounds of chains in a \( \text{CPO} \). Therefore, it is useful to state some more notations and results on fixpoints of continuous functions defined on (complete) lattices.

First of all we have to introduce the notation of ordinal. We assume that an ordinal is a set, where every element of an ordinal is still an ordinal and the class of ordinals is ordered by membership relation (\( \alpha < \beta \) means \( \alpha \in \beta \)). Consequently, every ordinal coincides with the set of all smaller ordinals. The least ordinals are \( 0, 1 := \{0\} \), \( 2 := \{0, \{0\}\} \), etc. Intuitively, the class of ordinals is the transfinite sequence \( 0 < 1 < 2 < \ldots < \omega < \omega + 1 < \ldots < \omega^\omega \), etc. Ordinals will be often denoted by Greek letters. An ordinal \( \gamma \) is a limit ordinal if it is neither 0 nor the successor of an ordinal; so, if \( \beta < \gamma \), then there exists \( \sigma \) such that \( \beta < \sigma < \gamma \). The first limit ordinal, which is equipotent with the set of natural numbers, is denoted (by an abuse of notation) by \( \omega \). Often, in the definitions of \( \text{CPO} \) and of continuity, directed sets are used instead of chains. It is possible to show that if the set \( S \) is denumerable, then the definitions are equivalent.

The ordinal powers of a monotonic function \( T : S \rightarrow S \) on a \( \text{CPO} \ S \) are defined as

\[
T \uparrow \alpha(x) := \begin{cases} x & \text{if } \alpha = 0 \\ T(T \uparrow (\alpha - 1)(x)) & \text{if } \alpha \text{ is a successor ordinal} \\ \sqcup \{ T \uparrow \beta(x) \mid \beta < \alpha \} & \text{if } \alpha \text{ is a limit ordinal} \end{cases}
\]

In the following, we will use the standard notation \( T \uparrow \alpha := T \uparrow \alpha (\bot) \), where \( \bot \) is the least element of \( S \). In particular, \( T \uparrow \omega := \sqcup_{n<\omega} T \uparrow n, T \uparrow n + 1 := T(T \uparrow n) \), for \( n < \omega \), and \( T \uparrow 0 := \bot \), where \( \sqcup \) is the lub operation of \( S \). Sometimes, \( T \uparrow \alpha(x) \) may be denoted simply by \( T^\alpha(x) \).

The next important result is usually attributed to Kleene and gives an explicit construction of the least fixed-point of a continuous function \( f \) on a \( \text{CPO} \ D \).
Theorem 1.4.2 (Fixpoint Theorem) Let \( f : D \to D \) be a continuous function on a CPO \( D \) and \( d \in D \) be a pre-fixpoint of \( f \). Then \( \sqcup \{ f \uparrow n(d) \mid n \leq \omega \} \) is the least fixed-point of \( f \) greater than \( d \). In particular \( f \uparrow \omega \) is the least pre-fixpoint and least fixed-point of \( f \).

Each CPO \( D \) with bottom \( \bot \) is associated with a fixed-point operator \( \text{fix} : [D \to D] \to D \), \( \text{fix} := \sqcup_{n<\omega} (\lambda f. f^n(\bot)) \), i.e., \( \text{fix} \) is the least upper bound of the chain of the functions \( \lambda f. \bot \sqsubseteq \lambda f. f(\bot) \sqsubseteq \lambda f. f(f(\bot)) \sqsubseteq \ldots \), where each of these is continuous and so an element of the CPO \([D \to D]\).

1.5 Abstract Interpretation

In this section we give the basic notations and concepts of approximation theory in semantics as firstly developed in [10] and [11].

Abstract Interpretation is a theory developed to reason about the relation between two different semantics (the concrete and the abstract semantics). The main idea is that of approximating properties from the exact (concrete) semantics into an approximate (abstract) semantics, that explicitly exhibits a structure which is somehow present in the richer concrete structure associated to program execution. The approximation relation between the concrete and the abstract semantics is formalized by a pair of functions, the abstraction \( \alpha \) and the concretization \( \gamma \), which form a Galois connection.

Galois connections can be defined on preordered sets, however we will restrict our attention to complete lattices.

Let \((C, \sqsubseteq)\) (concrete domain) be the domain of the concrete semantics and \((A, \leq)\) (abstract domain) be the domain of the abstract semantics. The partial order relations reflect an approximation relation. Since in approximation theory a partial order specifies the precision degree of any element in a poset, it is obvious to assume that if \( \alpha \) is a mapping associating an abstract object in \((A, \leq)\) for any concrete element in \((C, \sqsubseteq)\) then the following holds: if \( \alpha(x) \leq y \), then \( y \) is also correct, although less precise. The same argument holds if \( x \sqsubseteq \gamma(y) \). Then \( y \) is also a correct approximation of \( x \), although \( x \) provides more accurate information than \( \gamma(y) \). This gives raise to the following formal definition.

Definition 1.5.1 Let \((C, \sqsubseteq)\) and \((A, \leq)\) be two posets (the concrete and the abstract domain). A Galois Connection \( \langle \alpha, \gamma \rangle : (C, \sqsubseteq) \rightleftharpoons (A, \leq) \) is a pair of maps \( \alpha : C \to A \) and \( \gamma : A \to C \) such that

1. \( \alpha \) and \( \gamma \) are monotonic,
2. for each \( x \in C \), \( x \sqsubseteq (\gamma \circ \alpha)(x) \) and
3. for each \( y \in A \), \( (\alpha \circ \gamma)(y) \leq y \).
Moreover, a Galois insertion (of $A$ in $C$) $\langle \alpha, \gamma \rangle : (C, \sqsubseteq) \rightleftharpoons (A, \leq)$ is a Galois connection where $\alpha \circ \gamma = Id_A$.

In the context of abstract interpretation the map $\alpha$ is called the lower (or left) adjoint or abstraction, while $\gamma$ is called upper (or right) adjoint or concretization.

The following basic properties are satisfied by any Galois connection.

1. for each $c \in C$ and $a \in A$, $\alpha(c) \leq a$ if and only if $c \sqsubseteq \gamma(a)$.
2. $(\alpha \circ \gamma \circ \alpha) = \alpha$ and $(\gamma \circ \alpha \circ \gamma) = \gamma$.
3. $\alpha$ is additive and $\gamma$ is co-additive.
4. Upper and lower adjoints uniquely determine each other. Namely
   \[
   \gamma = \lambda y. \bigsqcup_{C} \{ x \in C \mid \alpha(x) \sqsubseteq y \}
   \]
   \[
   \alpha = \lambda x. \bigwedge_{A} \{ y \in A \mid x \leq \gamma(y) \}
   \]
5. $\alpha$ is surjective if and only if $\gamma$ is injective if and only if $\alpha \circ \gamma = Id_A$.

By property 4 a Galois connection can be defined each time $(C, \sqsubseteq)$ is a complete lattice and $\alpha : C \to A$ is additive or $(A, \leq)$ is a complete lattice and $\gamma : A \to C$ is additive. When in a Galois connection $\langle \alpha, \gamma \rangle$, $\gamma$ is not injective, several distinct elements of the abstract domain $(A, \leq)$ are mapped to the same element of $C$. This is usually considered useless [11]; thus a Galois insertion can always be forced by considering a more concise abstract domain $(A/\approx, \leq/\approx)$, such that for each $x, y \in A : x \approx y$ if and only if $\gamma(x) = \gamma(y)$.

Abstract interpretation can be formalized in a hierarchical framework. Given the Galois connections (insertions) $\langle \alpha_1, \gamma_1 \rangle : (C, \sqsubseteq) \rightleftharpoons (A, \leq)$ and $\langle \alpha_2, \gamma_2 \rangle : (A, \leq) \rightleftharpoons (B, \preceq)$, it can be shown that $\langle \alpha, \gamma \rangle : (C, \sqsubseteq) \rightleftharpoons (B, \preceq)$ is a Galois connection (insertion), where $\alpha := \alpha_2 \circ \alpha_1$ and $\gamma := \gamma_1 \circ \gamma_2$. Therefore, abstract interpretations can be designed by successive approximations.

1.5.1 Correctness and precision of Abstract Semantic Functions

Given a concrete semantics on the lattice $(C, \sqsubseteq)$ and a Galois connection $\langle \alpha, \gamma \rangle : (C, \sqsubseteq) \rightleftharpoons (A, \leq)$, between the concrete and the abstract domain, we can define an abstract semantics on $(A, \leq)$. The theory requires the concrete semantics to be the least fixed-point of a semantic monotonic function $F : C \to C$. A monotonic function $\tilde{F} : A \to A$ is a correct (abstract) approximation of $F$ if, for each $c \in C$, $\alpha(F(c)) \leq \tilde{F}(\alpha(c))$. We consider a correct approximation $\tilde{F}$ as an abstract semantic function on $A$. 
If we define the abstract function \( F^\alpha : A \to A \) as \( F^\alpha = \alpha \circ F \circ \gamma \), we obtain the \textit{best (abstract) approximation} of \( F \) on \( A \). Furthermore, it can be shown that \( F^\alpha \) is a correct approximation of \( F \) and, for each other correct approximation \( \bar{F} \), \( F^\alpha(a) \leq \bar{F}(a) \), for each \( a \in A \).

\( F \) in turn is often defined as composition of "primitive" functions. Let \( f : C^n \to C \) be one such functions. As before, a monotonic abstract function \( \bar{f} : A^n \to A \) is a \textit{correct (abstract) approximation} of \( f \) if, for each \( c_1, \ldots, c_n \in C, \alpha(f(c_1, \ldots, c_n)) \leq \bar{f}(\alpha(c_1), \ldots, \alpha(c_n)) \). The \textit{best (abstract) approximation} \( f^\alpha : A^n \to A \) is defined as \( f^\alpha(a_1, \ldots, a_n) = \alpha(f(\gamma(a_1), \ldots, \gamma(a_n))) \), for each \( a_1, \ldots, a_n \in A \). Notice that the composition of correct approximations of primitive functions is still a correct approximation of their composition. On the other side, the composition of the best approximations of primitive functions is not necessarily the best approximation of their composition. It is true if the abstract primitive functions are \textit{precise}.

\textbf{Definition 1.5.2} Let \( \langle \alpha, \gamma \rangle : (C, \sqsubseteq) \rightrightarrows (A, \leq) \) be a Galois connection and \( f : C^n \to C \). A \textit{monotonic abstract function} \( f : A^n \to A \) is a \textit{precise (abstract) approximation} of \( f \) if, for each \( c_1, \ldots, c_n \in C, \alpha(f(c_1, \ldots, c_n)) = \bar{f}(\alpha(c_1), \ldots, \alpha(c_n)) \).

Observe that the composition of precise approximations of concrete functions is a precise approximation of the composition of the concrete function.

In general, there may exist different precise approximations of a concrete function \( f \) and it can be shown that whenever a precise approximation of \( f \) exists, it coincides on \( \alpha(C) \) with the best approximation of \( f \). Therefore, for Galois insertions, a precise approximation, if exists, is always equal to the best approximation.

\textbf{Lemma 1.5.3} Let \( \langle \alpha, \gamma \rangle : (C, \sqsubseteq) \rightrightarrows (A, \leq) \) be a Galois connection, \( f : C^n \to C \) monotonic and \( \bar{f} : A^n \to A \) a precise approximation of \( f \). Then, for each \( c_1, \ldots, c_n \in C, f^\alpha(\alpha(c_1), \ldots, \alpha(c_n)) = \bar{f}(\alpha(c_1), \ldots, \alpha(c_n)) \).

\textbf{Proof.} The proof follows by observing that

\[
\begin{align*}
f^\alpha(\alpha(c_1), \ldots, \alpha(c_n)) & \quad \text{[By definition of } f^\alpha] \\
\alpha(f(\gamma(\alpha(c_1)), \ldots, \gamma(\alpha(c_n)))) & \quad \text{[Since } \bar{f} \text{ is precise]} \\
= \bar{f}(\alpha(\gamma(\alpha(c_1))), \ldots, \alpha(\gamma(\alpha(c_n)))) & \quad \text{[Since, for Galois connections, } (\alpha \circ \gamma \circ \alpha) = \alpha]\end{align*}
\]

The following theorem gives a sufficient and necessary condition for the existence of a precise approximation of the concrete function \( f : C^n \to C \).
Theorem 1.5.4 Let \( \langle \alpha, \gamma \rangle : (C, \sqsubseteq) \Rightarrow (A, \leq) \) be a Galois connection and \( f : C^n \to C \) be a monotonic function. Then there exists a precise abstract approximation of \( f \) if and only if, for each \( c_1, \ldots, c_n \in C \),
\[
\alpha(f(c_1, \ldots, c_n)) = \alpha(f(\gamma(\alpha(c_1)), \ldots, \gamma(\alpha(c_n)))).
\]

Proof. If \( \bar{f} : A^n \to A \) is a precise approximation of \( f : C^n \to C \), then
\[
\alpha(f(c_1, \ldots, c_n)) = \bar{f}(\alpha(c_1), \ldots, \alpha(c_n)).
\]

By previous lemma
\[
f^a(\alpha(c_1), \ldots, \alpha(c_n)) = \alpha(f(\gamma(\alpha(c_1)), \ldots, \gamma(\alpha(c_n))))
\]

For example, note that if \( \sqcup \) is the join operation over \( (C, \sqsubseteq) \) and \( \langle \alpha, \gamma \rangle : (C, \sqsubseteq) \Rightarrow (A, \leq) \) is a Galois connection then
\[
\alpha(c_1 \sqcup c_2) = \alpha(\gamma(\alpha(c_1))) \sqcup \alpha(\gamma(\alpha(c_2))).
\]
Furthermore, since \( \alpha \) is join-additive, we have that \( \sqcup \circ \alpha = \alpha \circ \sqcup \), where \( \sqcup \) is the join on \( (A, \leq) \). Then \( \sqcup \) is a precise approximation of \( \sqsubseteq \).

Finally we have the following results concerning the approximation of the least fixed-point of a concrete semantic function.

Theorem 1.5.5 Let \( \langle \alpha, \gamma \rangle : (C, \sqsubseteq) \Rightarrow (A, \leq) \) be a Galois connection, \( F : C \to C \) be a monotonic function and \( \bar{F} : A \to A \) a correct approximation of \( F \). Then
\[
\alpha(lfp(F)) \leq lfp\left(\bar{F}\right).
\]

Proof. Let \( q = lfp\left(\bar{F}\right) \). Since \( \bar{F} \) is a correct approximation of \( F \), we have that
\[
\alpha(F(\gamma(q))) \leq \bar{F}(\alpha(\gamma(q))).
\]
Since \( \bar{F} \) is monotonic and since \( \alpha(\gamma(a)) \leq a \), for each \( a \in A \), we have that
\[
\alpha(F(\gamma(q))) \leq \bar{F}(q) = q.
\]
By Property 1 of Galois connections follows that \( F(\gamma(q)) \sqsubseteq \gamma(q) \), i.e. \( \gamma(q) \) is a pre-fixpoint of \( F \). Therefore, by Tarski’s theorem we have that \( lfp(F) \sqsubseteq \gamma(q) \) and hence
\[
\alpha(lfp(F)) \leq lfp\left(\bar{F}\right).
\]

If \( \bar{F} : A \to A \) is a precise approximation of \( F : C \to C \), then the abstraction of the concrete fixed-point coincides with the abstract fixed-point.

Theorem 1.5.6 Let \( \langle \alpha, \gamma \rangle : (C, \sqsubseteq) \Rightarrow (A, \leq) \) be a Galois connection, \( F : C \to C \) be a monotonic function and \( \bar{F} : A \to A \) a precise approximation of \( F \). Then
\[
\alpha(lfp(F)) = lfp\left(\bar{F}\right).
\]

Proof. Since \( \bar{F} \) is an approximation of \( F \), we have that
\[
\alpha(lfp(F)) \leq lfp\left(\bar{F}\right).
\]
Let us now show that \( lfp\left(\bar{F}\right) \leq \alpha(lfp(F)) \). Since \( F(lfp(F)) = lfp(F) \), we have that
\[
\alpha(F(lfp(F))) = \alpha(lfp(F)).
\]
On the other side, since \( \bar{F} \) is precise, it
follows that $\tilde{F}(\alpha(lfp(F))) = \alpha(lfp(F))$, i.e. $\alpha(lfp(F))$ is a fixed-point of $\tilde{F}$, therefore, by Tarski’s theorem, $lfp(\tilde{F}) \leq \alpha(lfp(F))$. ■
Chapter 2

First Order Hereditary Harrop Formulas

A language that uses implication and universal quantifiers in goals can be described as an extension to the language based on Horn clauses. In Horn clauses based programs a goal can be an atom, a conjunction or a disjunction of other goals (a set negative Horn clauses). Clauses of programs are positive Horn clauses in which the head is an atom and the body is a goal. In the interpretation of first-order Horn programs disjunction and conjunction specify OR and AND branches in a search and the existential quantifier in goals also permits answers to be extracted from computations. Namely, we can interpret a goal with free variables as a request to solve the existential closure of the formula and to produce instantiations for the introduced quantifiers that lead to successful solutions.

The extended language we want, should also allow implications and universal quantifiers in goals. This additions are incorporated into a language that is based on first-order hereditary Harrop formulas \[41\]. Intuitively, we can interpret implication and universal quantifier as scoping mechanisms with respect to program clauses and names respectively. This is the effect we obtain if the idea of solving a goal with respect to a program is clarified by using the notion of intuitionistic provability.

2.1 The Language

The first order hereditary Harrop formulas (fOhh-formulas) \[28, 38\] are divided in two groups: the \(G\)-formulas (goals) and the \(D\)-formulas (definite clauses). They are defined by the following syntax rules, where \(A\) is an atom:

\[
G \ ::= \ A \mid G \land G \mid G \lor G \mid \exists x.G \mid D \supset G \mid \forall x.G \\
D \ ::= \ A \mid G \supset A \mid D \land D \mid \forall x.D
\]

We assume that the formulas are defined over a first order signature where \(Vars\) denotes the set of variables and \(Consts\) the set of constants. By \(vars\) and \(consts\)
we denote the functions that, respectively, give the set of variables and constants occurring in a formula or in a set of them.

These formulas define a logic programming language in the sense that a $G$-formula can be seen as a query or a goal to be resolved using a program which is a set of definite clauses ($D$-formulas). In this case the process of answering consists in constructing an intuitionistic proof of the existential closure of the query from the given program. In the following we will refer to a $D$-formula as a program clause and to a $G$-formula as a goal. We denote by $\mathbb{P}$ the set of all $fohh$-programs.

To show the results described in [43] we need first to define some notions concerning the logic programming language we are dealing with.

**Definition 2.1.1** The elaboration of a program clause $D$, denoted by $elab(D)$, is a set defined as:

- If $D ::= A$ or $D ::= G \supset A$, then $elab(D) = \{D\}$.
- If $D ::= D_1 \land D_2$ then $elab(D) = elab(D_1) \cup elab(D_2)$.
- If $D ::= \forall x.D'$, then $elab(D) = \{\forall x.D'' \mid D'' \in elab(D')\}$

It is clear that in $elab(D)$ the conjunctions are eliminated, i.e., all the clauses are atoms or implication formulas universally quantified.

**Definition 2.1.2** Let $D$ be a clause. An instance of a formula $\forall x_1...\forall x_n.F \in elab(D)$ is any formula that can be written as $F\theta$, where $\theta$ is a substitution whose domain is $\{x_1,...,x_n\}$. The instances of a $D$-formula are all the instances of the formulas in $elab(D)$.

### 2.2 Intuitionistic Proof Procedure

In this section we describe the proof procedure presented by Nadathur in [43] for determining a proof (if it exists) of a goal formula from a $fohh$-program. The idea of the proof mechanism for a given goal is reduced to a search for simpler proof for the formulas in which the goal is composed. This method can lead to a disjunctive search, a conjunctive search or a search in the augmented set of program clauses, depending on the propositional connectives found in the goal. When a top level symbol in the goal is an existential quantifier, the involved variable is instantiated by a new one. If the connective is an universal quantifier, the variable is instantiated by a new function of the variables created in the preceding steps, i.e., all variables whose existential quantifier scope governs the universally quantified variable. This method is known by dynamic skolemization [21]. In order to obtain an efficient implementation, a variation of this scheme is presented in [43], in which instead of Skolem functions, labelled constants and variables are used.

Before to proceed with the presentation of the non-deterministic proof procedure we need to introduce some notions of the unification mechanisms on which the proof procedure is based.
2.2.1 Labelled Unification

The non-deterministic proof procedure described in [43] makes use of a different notion of unification that restricts the standard unification, creating a sort of scoping between variables and constants.

**Definition 2.2.1** The labelling function is a partial function \( L : \text{Vars} \cup \text{Consts} \rightarrow \mathbb{N} \) associates to each constant or variable a natural number. We call this number the level (label) of the variables and constants. The function \( L \), defined over formulas, gives the labelling function on the variables and constants in such formulas.

**Definition 2.2.2** Let \( \theta = \{ [t_i/x_i] \mid 1 \leq i \leq n \} \) be a substitution and let \( L \) be a labelling function on the variables and constants in \( \theta \). The substitution \( \theta \) is said to be a proper labelled substitution with respect to \( L \), if and only if, \( L(c) \leq L(x_i) \) for every constant \( c \) appearing in \( t_i \). The induced labelling is the labelling \( L' \) of the substitution \( \theta \) which is obtained from \( L \) in the following manner:

\[
L'(x) = \min (\{L(x)\} \cup \{L(x_i) \mid [t_i/x_i] \in \theta \text{ and } x \text{ appears in } t_i\})
\]

**Definition 2.2.3** Let \( T = \{ (t_i, s_i) \mid 1 \leq i \leq n \} \) be a set of pairs of terms and let \( L \) be a labelling function on the variables and constants occurring in \( T \). A labelled unifier (l-unifier) for \( T \) under \( L \) is a substitution \( \theta \) which is proper with respect to \( L \) and for each pair \( (t_i, s_i) \in T \) the equality \( t_i\theta = s_i\theta \) holds. A most general l-unifier (lmgu) for \( T \) under \( L \) is a l-unifier \( \theta \) such that for any other l-unifier \( \lambda \) for \( T \) there is a substitution \( \sigma \) with \( \theta\sigma = \lambda \).

2.2.2 The Proof Procedure

This algorithm, as presented in [43], determines the transition of one computation state to other. The initial state represents the information about the goal and the program. The final state contains the empty goal (if there is a proof). The information about all substitutions needed to calculate the computed answer can be retrieved from all preceding states.

**Definition 2.2.4** A state \( S \) is a tuple \( \langle W, k, L, \theta \rangle \) where

- \( W \) is a set of tuples \( \langle G, P, n \rangle \) called decorated goals where \( G \) is a goal formula, \( P \) is a program and \( n \) is a natural number,
- \( k \) is a clause used to reduce a goal in \( W \) (\( \kappa \) denotes the void clause, i.e., no reduction),
- \( L \) is a labelling function on \( \text{vars}(W) \cup \text{consts}(W) \), and
- \( \theta \) is a substitution (\( \epsilon \) denotes the empty substitution).
Actually, the program clause stored in the state is not used in the proof procedure. We maintain this information only for semantic purposes. In advance we will refer to decorated goals as “goals”. In case of ambiguity we will clarify whether we are talking about a goal formula or a decorated goal.

**Definition 2.2.5 (proof procedure)** The state \( S_1 = \langle W_1, k_1, L_1, \theta_1 \rangle \) can derive in one step to the state \( S_2 = \langle W_2, k_2, L_2, \theta_2 \rangle \) using the following rules:

- **and** If \( \exists w_1 = \langle G_1 \wedge G_2, P, n \rangle \in W_1 \) then \( W_2 = (W_1 - \{w_1\}) \cup \{\langle G_1, P, n \rangle, \langle G_2, P, n \rangle\} \), \( L_2 = L_1 \), \( k_2 = \kappa \) and \( \theta_2 = \epsilon \).

- **or** If \( \exists w_1 = \langle G_1 \vee G_2, P, n \rangle \in W_1 \) then \( W_2 = (W_1 - \{w_1\}) \cup \{\langle G_1, P, n \rangle\} \) for \( i \in \{1, 2\} \), \( L_2 = L_1 \), \( k_2 = \kappa \) and \( \theta_2 = \epsilon \).

- **augment** If \( \exists w_1 = \langle D \supset G, P, n \rangle \in W_1 \) then \( W_2 = (W_1 - \{w_1\}) \cup \{\langle G, P \cup \{D\}, n \rangle\} \), \( L_2 = L_1 \), \( k_2 = \kappa \) and \( \theta_2 = \epsilon \).

- **instance** If \( \exists w_1 = \langle \exists x. G, P, n \rangle \in W_1 \) then \( W_2 = (W_1 - \{w_1\}) \cup \{\langle G[y/x], P, n \rangle\} \) for \( y \notin \text{vars}(W_1) \), \( L_2 = L_1 \cup \{\langle y, n \rangle\} \), \( k_2 = \kappa \) and \( \theta_2 = \epsilon \).

- **generic** If \( \exists w_1 = \langle \forall x. G, P, n \rangle \in W_1 \) then \( W_2 = (W_1 - \{w_1\}) \cup \{\langle G[c/x], P, n + 1 \rangle\} \) for \( c \notin \text{consts}(W_1) \), \( L_2 = L_1 \cup \{\langle c, n + 1 \rangle\} \), \( k_2 = \kappa \) and \( \theta_2 = \epsilon \).

- **identity** If \( \exists w_1 = \langle A, P, n \rangle \in W_1 \) and \( \exists k = \forall x_1..\forall x_m. A' \in \text{elab}(P) \) such that \( \exists \theta = \text{lmgv}(A, A') \) relative to \( L' = L_1 \cup \{\langle y, n \rangle \mid y \in \text{range}(\rho)\} \), where \( \rho \) is a renaming over all variables of \( A' \) with \( \text{range}(\rho) \cap \text{vars}(W_1) = \emptyset \), then \( W_2 = \{w\theta \mid w \in W_1 - \{w_1\}\} \), \( k_2 = k\rho \), \( L_2 \) is the labelling induced by \( \theta \) from \( L' \), and \( \theta_2 = \theta \).

- **backchain** If \( \exists w_1 = \langle A, P, n \rangle \in W_1 \) and \( \exists k = \forall x_1..\forall x_m. (G' \supset A') \in \text{elab}(P) \) such that \( \exists \theta = \text{lmgv}(A, A') \) relative to \( L' = L_1 \cup \{\langle y, n \rangle \mid y \in \text{range}(\rho)\} \), where \( \rho \) is a renaming over all variables of \( A' \) with \( \text{range}(\rho) \cap \text{vars}(W_1) = \emptyset \), then \( W_2 = \{w\theta \mid w \in W_1 - \{w_1\}\} \cup \{\langle G', P, n \rangle \theta\} \), \( k_2 = k\rho \), \( L_2 \) is the labelling induced by \( \theta \) from \( L' \), and \( \theta_2 = \theta \).

The derivation step between states \( S_1 \) and \( S_2 \) is denoted by \( S_1 \rightarrow S_2 \).

**Definition 2.2.6** The possible infinite sequence of states \( S_0, ..., S_n \) is a derivation sequence \( (S_0 \rightarrow^* S_n) \) if \( S_i \rightarrow S_{i+1}, i \geq 0 \).

**Definition 2.2.7** The initial state of a computation \( G \) in \( P \) is defined as \( S_0 = \langle \{\langle G, P, 0 \rangle\}, \kappa, L_0, \epsilon \rangle \), where \( L_0 = \emptyset \). A state is said to be proper if for each \( w = \langle G, P, n \rangle \in W \) is the case that \( L_{|w|} \leq n \).
2.2. INTUITIONISTIC PROOF PROCEDURE

Definition 2.2.8 A derivation \( d = S_0 \rightarrow^* S_n \) is a successful derivation if \( S_0 \) is an initial state and \( S_n = \langle \emptyset, k_n, L_n, \theta_n \rangle \) for some \( L_n, k_n, \) and \( \theta_n \). Moreover, we say that a non successful derivation is a failure if it can not be extended by applying a derivation step to its last state.

Given a derivation \( d = \langle W_0, k_0, L_0, \theta_0 \rangle, ..., \langle W_n, k_n, L_n, \theta_n \rangle \) we denote by \( \text{first}(d) \) and \( \text{last}(d) \) the set of all goals occurring in the first and the last derivation state respectively. By \( \text{clauses}(d) \) we denote the set of all program clauses selected to reduce goals at each state of a derivation \( d \), and by \( \text{prefix}(d) \) we denote the set of all derivations which are prefixes of \( d \).

Definition 2.2.9 Let \( S_0 = \langle \{\langle G, P, 0 \rangle\}, \kappa, L_0, \epsilon \rangle \) be an initial state and \( S_0 \rightarrow^* S_n \) a successful derivation, its associated answer substitution is the restriction of \( \sigma = \theta_1...\theta_n \) to \( \text{vars}(G, P) \). The computed answer is the formula \( G\sigma \).

By completeness theorem (theorem 15, [43]) we know that a computed answer of a proof (successful derivation) exists regardless the goal selected at each stage in constructing the derivation. Therefore, it is possible to obtain many different derivations for a given formula that differ only in the order in which the goals are reduced. Following this argument we can establish an equivalence relation between these derivations, observing the rules used to reduce each goal, but not the order they are applied. For example, given the program \( P = \{p(Y, Y), q(Y)\} \), the following two derivations are equivalent

\[
\langle\{\langle p(X, a) \land q(X), P, 0 \rangle\}, \kappa, \epsilon \rangle \rightarrow \langle\{\langle p(X, a), P, 0 \rangle, \langle q(X), P, 0 \rangle\}, \kappa, \epsilon \rangle
\rightarrow \langle\{\langle q(a), P, 0 \rangle\}, p(Y, Y), \{[Y/X], [a/Y]\} \rangle
\rightarrow \langle\emptyset, q(Y), [a/Y]\rangle
\]

\[
\langle\{\langle p(X, a) \land q(X), P, 0 \rangle\}, \kappa, \epsilon \rangle \rightarrow \langle\{\langle p(X, a), P, 0 \rangle, \langle q(X), P, 0 \rangle\}, \kappa, \epsilon \rangle
\rightarrow \langle\{\langle p(Y, a), P, 0 \rangle\}, q(Y), \{[Y/X]\} \rangle
\rightarrow \langle\emptyset, p(Y, Y), \{[a/Y]\}\rangle
\]

The specification of equivalence classes of derivation will be useful for the definition of deterministic semantic operators. These operators create instances of classes of derivations to construct in a bottom-up way an interpretation of a given goal.

Definition 2.2.10 The computation tree of a goal \( w = \langle G, P, 0 \rangle \) has \( w \) as root node, and each node (goal) has as children all the goals that it generates when is reduced. The computation set of a derivation \( d \) is composed by all trees generated by the goals in \( \text{first}(d) \).

Definition 2.2.11 We say that two derivations \( d_1 = S_1 \rightarrow^* S_n \) and \( d_2 = S_1' \rightarrow^* S_n' \) are equivalent if they have the same computation set, modulo renaming of variables and constants with label greater than 0. In that case we say that \( d_1 \) is a variant of \( d_2 \) (\( d_1 \equiv_c d_2 \)).
Since there are new constants in the derivations (with label greater than 0), which are introduced by the non-deterministic proof procedure for resolving the universal quantification in goals, two equivalent derivations can contain different constants. In that case we say that a derivation is a variant of other, if there exists a renaming of constants and variables, where each one is replaced by a constant or variable of the same level, that makes both derivations syntactically equal.

2.3 Semantic Domains

Now we use the derivation concept introduced in the previous section to formalize the domains of the semantics definitions.

A set of derivations $S$ is said to be well formed if and only if for any $d \in S$ we have $\text{prefix}(d) \subseteq S$. We denote by $WFD$ the complete lattice of well-formed set of derivations ordered by $\subseteq$.

**Definition 2.3.1** A collection is a partial function $C : \text{Goals} \rightarrow \text{WFD}$, such that if $C(G)$ is defined, then it is a well-formed set of derivations all starting from a goal $G$. $C$ is the domain of all collections ordered by $\subseteq$, where $C \subseteq C'$ iff $\forall G. C(G) \subseteq C'(G)$ and if $C(G)$ is defined then $C'(G)$ is defined too. A pure collection is a collection defined only for pure atomic goals and its domain is denoted by $\text{PC}$.

We introduce the relation of equivalence modulo variance $\equiv_{\subseteq}$ defined on collections. Namely, $C \equiv_{\subseteq} C'$ if and only if for any $G$ there exists a variant $G'$ of $G$ such that, if $C(G)$ is defined, then $C'(G')$ is defined, and for any $d \in C(G)$ there exists $d' \in C'(G')$ such that $d \equiv_{\subseteq} d'$.

**Definition 2.3.2** An interpretation $I$ is a pure collection modulo variance. The set of all interpretations is denoted by $\mathcal{I}$. The pair $(\mathcal{I}, \subseteq)$ is a complete lattice with the induced quotient order.

In other words, an interpretation associates derivations only to pure atomic goals.

**Definition 2.3.3** Let $G$ be a goal and $P$ a program. The identity collection of a goal will be a collection that associates to $G$ the initial state $\langle \{ \langle G, P, 0 \rangle \}, \kappa, 0, \epsilon \rangle$ and is denoted as $\text{Id}_G$.

2.4 Denotational Semantics

We define a denotational semantics inductively on the following syntax of fohon-logic programs.

\[
\begin{align*}
\text{QUERY} & ::= \text{GOAL} \text{ in } \text{PROG} \\
\text{GOAL} & ::= G\text{-formula} \\
\text{PROG} & ::= \{D\text{-formula}\} \cup \text{PROG} | \emptyset
\end{align*}
\]
where the syntax of $G$-formulas and $D$-formulas is already defined in the section 2.1.

The semantic functions are

$$
\begin{align*}
Q &: \text{QUERY} \rightarrow \mathcal{C} \\
G &: \text{GOAL} \rightarrow (\Pi \rightarrow \mathcal{P} \rightarrow \mathcal{C}) \\
P &: \text{PROG} \rightarrow (\Pi \rightarrow \Pi) \\
C &: \text{D-formula} \rightarrow (\Pi \rightarrow \mathcal{P} \rightarrow \Pi)
\end{align*}
$$

and are defined by means of some specific operators on which is based the compositionality. The operator $\oplus$ computes the nondeterministic union of two classes of collections. The operator $\emptyset$ computes the interpretation obtained by replacement, adding to the first derivation step (computed by the operator $\Theta$) all possible derivations of the clause goal. The operator $\triangledown$ restricts (instantiates) an interpretation to such derivations that can be “matched” with an atomic goal. The operator $\otimes$ computes the conjunction of two interpretations, leaving only the unifiable derivations of the two goals, while the operator $\oplus$ computes the extension of the union of two classes of interpretations. Finally we introduce the operator $\triangledown$ to calculate an extended interpretation given a new clause.

$$
\begin{align*}
Q[G \text{ in } P] &: = G[G]_{\text{if} \ P[P].P} \\
P[P]_I &: = \sum_{c \in P} C[C]_{I.P} \\
C[p(t)]_{I.P} &: = \Theta(p(t))_P \\
C[D_1 \land D_1]_{I.P} &: = C[D_1]_{I.P} + C[D_2]_{I.P} \\
C[D \lor p(t)]_{I.P} &: = \Theta(D \lor p(t))_{I.P} \triangledown G[G]_{I.P} \\
C[\forall x. D]_{I.P} &: = C[D]_{I.P} \\
G[p(t)]_{I.P} &: = \triangledown(p(t))_{I.P} \\
G[G_1 \land G_2]_{I.P} &: = G[G_1]_{I.P} \otimes G[G_2]_{I.P} \\
G[G_1 \lor G_2]_{I.P} &: = G[G_1]_{I.P} \oplus G[G_2]_{I.P} \\
G[\exists x. G]_{I.P} &: = \exists(G[G]_{I.P})_{x} \\
G[\forall x. G]_{I.P} &: = \forall(G[G]_{I.P})_{x} \\
G[D \lor G]_{I.P} &: = \Delta(G[G]_{\text{isc}[D] \cup Q(D)})_{D \lor G,P}
\end{align*}
$$

Before describing the semantic operators, we need to define some concepts and operations on derivations that will help us to define all compositional operators over collections.

**Definition 2.4.1** Let $d_1 = S_1^1, ..., S_n^1$ and $d_2 = S_1^2, ..., S_m^2$ be derivations, the concatenation of $d_1$ and $d_2$, $(d_1 :: d_2)$ is defined if $S_1^1 = S_2^1$ and $\text{vars}(d_1) \cap \text{vars}(d_2) = \emptyset$, then $d_1 :: d_2 = S_1^1, ..., S_n^1, S_2^2, ..., S_m^2$.

The restriction introduced on the occurrence of variables asserts that all variables introduced in the second derivation must be new with respect to the first one.

**Definition 2.4.2** Let $d = S_1, ..., S_n$ be a derivation and let $W'$ be a set of goals. The insertion of $W'$ in $d$ ($d \leftarrow W'$) is defined if each state $S_i' = (W_i \cup (W' \sigma_i), k_i, L_i \cup L'_i, \theta_i)$
is a proper state, where $S_i = \langle W_i, k_i, L_i, \theta_i \rangle \in d$ and $L_i'$ is the labelling resulting when applying $\sigma_i$ to $W'$, with $\sigma_i = \theta_1...\theta_i$. In this case $d < W' = S_1', ..., S_n'$.

The insertion of a goal in a derivation means that the goal is added to each state, and it remains unsolved at the end of the derivation, but modified by all the substitutions executed in the preceding states.

**Definition 2.4.3** Let $d = S_1, ..., S_n$ be a derivation and $\theta$ an idempotent substitution such that $\text{vars}(\text{first}(d)\theta) \cap \text{vars}(\text{clauses}(d)) = \emptyset$ and let $L'$ be a labelling function for all variables in $\text{range} (\theta)$. Then the application of $\theta$ to $d$, $d\theta = S_1', ..., S_n'$ is defined if for each state $S_i = \langle W_i, k_i, L_i, \theta_i \rangle \in d$, we have that $S_i' = \langle W_i', k_i, L_i', \theta_i' \rangle$ where

- if $k_i \neq \kappa$ then $\exists \theta_i' = \text{lmgu}(A\theta, A')$ relative to $L_i \cup L'$ where $k_i = \forall x_1...\forall x_n.(G' \supset A')$, $\langle A, P, n \rangle \in W_{i-1}$ and $\theta_i = \text{lmgu}(A, A')$. In that case $W_i' = W_{i-1}'\theta_i' - \langle A, P, n \rangle \theta_i' \cup G', P, n \rangle$ and $L_i'$ is the labelling induced by $\theta_i'$ from $L_i \cup L'$.

- if $k_i \neq \kappa$ then $\exists \theta_i' = \text{lmgu}(A\theta, A')$ relative to $L_i \cup L'$ where $k_i = \forall x_1...\forall x_n. A'$, $\langle A, P, n \rangle \in W_{i-1}$ and $\theta_i = \text{lmgu}(A, A')$. In that case $W_i' = W_{i-1}'\theta_i' - \langle A, P, n \rangle \theta_i' \cup G', P, n \rangle$ and $L_i'$ is the labelling induced by $\theta_i'$ from $L_i \cup L'$.

- if $k_i = \kappa$ then $S_i'$ is a proper state, where $W_i' = W_i\theta$, $L_i'$ is the new induced labelling function for all variables in $\text{range} (\theta) \cup (\text{Dom}(L_1) - \text{Dom}(\theta))$ and $\theta_i' = \epsilon$.

Note that the substitution applied to a derivation attempts to reconstruct the derivation, starting from a new goal (more instantiated or just renamed) using the same clauses, until a failure of a lmg or a substitution that yields a non proper state. In any other case the substitution has success and the result is a derivation sequence.

**Definition 2.4.4** Let $d = S_1, ..., S_n$ be a derivation and $G$ be a goal, the instantiation $\delta_G(d)$ of $d$ is defined if there exists a labelled substitution $\theta$, such that $G = G\theta$ and if $d\theta$ is defined, where $\text{first}(d) = \{G'\}$. In that case $\delta_G(d) = d\theta$.

**Definition 2.4.5** Let $d_1 = S_1^1, ..., S_n^1$ and $d_2 = S_1^2, ..., S_m^2$ be derivations. The fusion of $d_1$ and $d_2$ ($d_1 \ast d_2$) is defined if $\text{vars}(d_1) \cap \text{vars}(d_2) \subseteq \text{vars}(\text{first}(d_1) \cup \text{first}(d_2))$ and if the concatenation $d'_1 :: d'_2$ is defined, where $d'_1 = d_1 \ast \text{first}(d_2)$, $d'_2 = \text{last}(d_1), (d_2\sigma_1) \ast \text{last}(d_1)$ and for all $x \in \text{vars}(d_1) \cap \text{vars}(d_2)$ is the case that $\sigma_1(x) = \sigma_2(x)$, where $\sigma_1$ and $\sigma_2$ are the computed substitution of $d_1$ and $d_2$ respectively. In this case $d_1 \ast d_2$ is $d'_1 :: d'_2$.

This operator attempts to combine in an unique derivation two different derivations $d_1$ and $d_2$. The resulting derivation must be equivalent to the derivation starting with the conjunction of the goals in $\text{first}(d_1)$ and $\text{first}(d_2)$. Is easy to
see that this operator is commutative \((d_1 \ast d_2 = d_2 \ast d_1)\) and associative \(((d_1 \ast d_2) \ast d_3 = d_1 \ast (d_2 \ast d_3))\) modulo variance between derivations. This fact which reflects the commutative and associative properties of conjunction operation on goals.

**Definition 2.4.6** We say that a derivation \(d\) is raised with respect to the variable \(x\), if the substitution \(d'[c/x]\) is defined, where \(c\) is a new constant with \(L(c) = 1\) and all levels of decorated goals in \(d'\) are increased by 1 with respect to \(d\). The raising of a derivation is denoted by \(d \uparrow x\).

Analogously we can establish the raising with respect to a program clause \(C\), where the programs in all decorated goals are augmented with the clause \(C\).

**Definition 2.4.7** The compatible extension of a collection \(C_1\) by a collection \(C_2\) is denoted by the operator \(\bowtie \circ \triangleright\) and is defined as
\[
C_1 \bowtie \circ \triangleright C_2 = \lambda G.C_1(G) \cup \{ d_1 :: (S_n \ast d_2') \mid d_1 \in C_1(G), d_2 \in C_2(G'') \text{ and } d_2' = \delta_{G'}(d_2) \text{ with } G'' \in W_n \}
\]
where \(S_n\) is the last state of \(d_1\).

This operator extends the derivation in \(C_1\) with all possible instances of derivations in \(C_2\). In other words, this operator continues an stopped computation (represented by a derivation of \(C_1\)), by “executing” the steps of other computation (represented by a derivation of \(C_2\)). Note that the second derivation only reduces (derives) a pending goal in \(d_1\), by this reason the extended derivation has all the remaining goals pending.

Now we describe and define all semantic operators and auxiliary functions used in our denotational semantic.

- The operator \(\Theta\), given a clause \(G \supset p(t)\) and a program \(P\), creates the interpretation
  \[
  I = \{\langle p(x), \{d_0, d_1\} \rangle\}
  \]
  where
  \[
  d_0 = \langle\{\langle p(x), P, 0\rangle\}, \kappa, L_0, \epsilon \rangle \text{ and } \\
  d_1 = \langle\{\langle p(x), P, 0\rangle\}, \kappa, L_0, \epsilon \rangle \rightarrow \langle\{\langle G, P, 0 \rangle \theta \}, G \supset p(t) \rho, L_1, \theta \rangle.
  \]

Given a clause \(p(t)\), the operator creates the interpretation
\[
I = \{\langle p(x), \{d_0, d_1\} \rangle\}
\]
where
\[
\begin{align*}
  d_0 &= \langle\{\langle p(x), P, 0\rangle\}, \kappa, L_0, \epsilon \rangle \\
  d_1 &= \langle\{\langle p(x), P, 0\rangle\}, \kappa, L_0, \epsilon \rangle \rightarrow \langle\{\emptyset, p(t) \rho, L_1, \theta \} \rangle.
\end{align*}
\]

We also have, \(L_0 = L(P) \cup \{\langle x, 0\rangle\}, L' = L_0 \cup \{\langle y, 0 \rangle \mid y \in \text{range}(\rho)\}\) where \(\rho\) is a renaming over \(x\) and \(L_1\) is the labelling induced by \(\theta = \{x/\cdot \rho\} \) from \(L'\).
• The operator $+$ is the sum of classes of interpretations:

$$C_1 + C_2 := \lambda G. C_1(G) \cup C_2(G)$$

i.e., it assigns to each goal the union of its interpretations in $C_1$ and $C_2$. The sum $C_1 + \ldots + C_k$ is denoted by $\sum_{i=0}^{k} C_i$.

• The operator $\bowtie \succeq$ computes the concatenation of the collection $C_1$ by $C_2$. It is defined as

$$C_1 \bowtie \succeq C_2 := \lambda p(t). C_1((p(t)) \cup \{ \ d_1 :: d_2 \theta \mid d_1 \in C_1(p(t)), \ d_2 \in C_2(G) \}$$

where $\theta$ is the substitution applied in the last state of $d_1$.

By means of this operator the function $C$ constructs the maximum number of derivations that can be obtained for a pure goal.

• Given a program $P$, the operator $\nabla$ makes the instantiation of an interpretation $I$ with an atom $p(t)$. It is defined as

$$\nabla(p(t))_{I,P} := \begin{cases} \lambda p(t). \{ \delta_{p(t)}(d) \mid d \in I(p(x)) \} & \text{if } I(p(x)) \neq \emptyset \\ \lambda p(t). \{ \{ (p(t), P, 0) \}, \kappa, 0, \epsilon \} & \text{if } I(p(x)) = \emptyset \end{cases}$$

This operator associates to $p(t)$ all the instantiated derivations in $I$.

• The product of $C_1$ and $C_2$ is

$$C_1 \otimes C_2 = \lambda (G_1 \land G_2). \{ S_0, (d_1 \# d_2) \mid d_1 \in C_1(G_1), \ d_2 \in C_2(G_2) \}$$

and $S_0 = \langle \{ (G_1 \land G_2, P, n) \}, \kappa, L, \epsilon \rangle$

where $P$ and $n$ are the program and level occurring in the unique goal of $\text{first}(d_1)$ and $\text{first}(d_2)$.

Note that the collection defined by this operator only operates over conjunctions of goals. The operator takes the derivations associated to each component of the conjunction and fuses them (when it is possible) in an unique derivation, adding a head state $(S_0)$ containing the conjunction goal.

• The extension of the union of $C_1$ and $C_2$ is

$$C_1 \oplus C_2 = \lambda (G_1 \lor G_2). \{ S_0, d \mid d \in C_1(G_1) \cup C_2(G_2) \}$$

and $S_0 = \langle \{ (G_1 \lor G_2, P, n) \}, \kappa, L, \epsilon \rangle$

where $P$ and $n$ are the program and level occurring in the unique goal of $\text{first}(d)$.

This operator is very similar to the sum of collections, it differs only in the head state added to the resulting derivations.
2.4. DENOTATIONAL SEMANTICS

- The existential quantifier of variable $x$ over a collection $C$ is defined as
  \[ \exists(C)_x = \lambda(\exists x.G'). \{ \quad S_0, d' \mid d \in C(G'), d' = d[y/x] \text{ and} \]
  \[ S_0 = \{ \langle \exists x.G', P, n \rangle, \kappa, L, \epsilon \} \]
  where $L(y) = n$, $L(x) = n$ for a new variable $y$ and $P$ is the program occurring in $\text{first}(d)$

- The universal quantifier of variable $x$ over a collection $C$ is defined as
  \[ \forall(C)_x = \lambda(\forall x.G'). \{ \quad S_0, d \uparrow x \mid d \in C(G') \text{ and} \]
  \[ S_0 = \{ \langle \forall x.G', P, 0 \rangle, \kappa, L, \epsilon \} \]
  where $P$ is the program occurring in $\text{first}(d)$

- The operator $\Delta$ given an implication goal $G$, a collection $C$ and a program $P$, constructs the collection
  \[ \Delta(C)_{D \supseteq G', P} = \lambda(D \supseteq G'). \{ \quad S_0, d \mid d \in C(G') \text{ and} \]
  \[ S_0 = \{ \langle D \supseteq G', P, n \rangle, \kappa, \epsilon \} \]
  where $n$ is the level occurring in $\text{first}(d)$.

**Definition 2.4.8** The composition of two interpretations $I_1$ and $I_2$ defined on Goals of programs $P_1$ and $P_2$ respectively is

\[ I_1 \uplus I_2 = \text{lfp} \Phi_{I_1 \uplus P_2 + I_2 \uplus P_1} \]

where

\[ \Phi_P(I) = \begin{cases} I' & \text{if } I = \bot \\ I \triangleright \circ \triangleleft I' & \text{otherwise.} \end{cases} \]

Note that the mutual rising of both interpretation to the respectively associated programs is required in order to guarantee the compatible extension between interpretations. Intuitively, this operator defines an OR-composition between program interpretations. It takes the least fixed point of all possible mutual extensions of both interpretations. In advance we will denote the napplication $\Phi_P(\Phi_P(\ldots \Phi_P(\bot)\ldots))$ of $\Phi_P$ as $I' \uparrow^n$ and its least fixed point as $I' \uparrow$. In that way, the previous definition can be reformulated as

\[ I_1 \uplus I_2 = (I_1 \uparrow P_2 + I_2 \uparrow P_1) \uparrow \]

It is easy to show that this operator is commutative by observing that $I_1 \uparrow P_2 + I_2 \uparrow P_1 = I_2 \uparrow P_2 + I_1 \uparrow P_1$ by commutativity of $+$. 
2.4.1 An Example of Bottom-Up Denotation

To show how our semantics works consider a problem commonly referred to as the sterile jar problem. Assume we have the following facts:

1. A jar is sterile if every germ in it is dead,
2. A germ in a heated jar is dead, and
3. Some jar is heated.

We can write the $fohh$-clauses that represent this knowledge as

\[
\forall Y (\forall X (\text{germ}(X) \supset (\text{in}(X, Y) \supset \text{dead}(X))) \supset \text{sterile}(Y)) \\
\forall Y \forall X (\text{heated}(Y) \land (\text{in}(X, Y) \land \text{germ}(X)) \supset \text{dead}(X))
\]

For simplicity we will implicitly define the labelling function by indicating a superscript in variables and constants. Also we will rewrite the program $P$ more simplified:

\[
C_1 := \forall Y (\forall X (\text{germ}(X) \supset (\text{in}(X, Y) \supset \text{dead}(X))) \supset \text{sterile}(Y)) \\
C_2 := \forall Y \forall X (\text{heated}(Y) \land (\text{in}(X, Y) \land \text{germ}(X)) \supset \text{dead}(X)) \\
C_3 := h(a)
\]

In addition, we will eliminate all suffixes of already calculated derivations if they are not useful for further computations.

Let’s calculate the interpretation $I_P = \text{lfp} \mathcal{P}[P]$

\[
P := \{C_1, C_2, C_3\} \\
P_1 := P \cup \{\text{g}(X)\} \\
P_2 := P \cup \{\text{g}(X), \text{i}(X, Y)\}
\]

\[
\mathcal{P}[P]_0 := C[P]_0 + C[P]_0 + C[P]_0
\]

\[
C[P]_0 := \Theta(C_1)_P \Theta(C_2)_P \Theta(C_3)_P
\]

\[
C[P]_1 := \Theta(C_1)_P \Theta(C_2)_P \Theta(C_3)_P
\]

\[
C[P]_2 := \Theta(C_1)_P \Theta(C_2)_P \Theta(C_3)_P
\]

\[
C[P]_3 := \Theta(C_1)_P \Theta(C_2)_P \Theta(C_3)_P
\]

\[
C[P]_4 := \Theta(C_1)_P \Theta(C_2)_P \Theta(C_3)_P
\]

\[
C[P]_5 := \Theta(C_1)_P \Theta(C_2)_P \Theta(C_3)_P
\]

\[
C[P]_6 := \Theta(C_1)_P \Theta(C_2)_P \Theta(C_3)_P
\]
2.4. DENOTATIONAL SEMANTICS

Finally, we have in the first iteration:

\[
\Theta(C_1)_P := \{ s(X), \{
\langle (s(X), P, 0), \kappa, \epsilon \rangle \rightarrow \\
\langle \forall X(g(X) \supset (i(X, Y) \supset d(X))), P, 0 \rangle, C_1, [X/X_1] \}\}\n\]

\[
\Theta(C_2)_P := \{ d(X_0), \{
\langle d(X), P, 0 \rangle, \kappa, \epsilon \rangle \rightarrow \\
\langle \langle h(Y) \land (i(X, Y) \land g(X)), P, 0 \rangle, C_2, [X/X_1] \\rangle \}\}
\]

\[
\Theta(C_3)_P := \{ h(X), \{
\langle h(X), P, 0 \rangle, \kappa, \epsilon \rangle \rightarrow \{ \emptyset, C_3, [a/X] \}\}
\]

\[
\Theta(g(X))_P := \{ g(X), \{ \langle (g(X), P_1, 0), \kappa, \epsilon \rangle \}\}
\]

\[
\Theta(i(X, Y))_P := \{ i(X, Y), \{ \langle (i(X, Y), P_2, 0), \kappa, \epsilon \rangle \}\}
\]

\[
I_1 := C[g(X)]_0, P_1
\]

\[
I_2 := I_1 \uplus C[i(X, Y)]_{I_1, P_2}
\]

\[
:= \{ \{ g(X), \{ \langle (g(X), P_1, 0), \kappa, \epsilon \rangle \}\}, \{ i(X, Y), \{ \langle (i(X, Y), P_2, 0), \kappa, \epsilon \rangle \}\}\}
\]

\[
\mathcal{G}[\forall X(g(X) \supset (i(X, Y) \supset d(X)))]_{0, P}
\]

\[
:= \forall X\mathcal{G}[g(X) \supset (i(X, Y) \supset d(X))]_{0, P}
\]

\[
:= \forall X \Delta (\mathcal{G}[i(X, Y) \supset d(X)]_{I_1, P_1})_{g(X) \supset (i(X, Y) \supset d(X)), P}
\]

\[
:= \forall X \Delta \left( \Delta (\mathcal{G}[d(X)]_{I_2, P_2})_{i(X, Y) \supset d(X), P_1} \right)_{g(X) \supset (i(X, Y) \supset d(X)), P}
\]

\[
:= \forall X \Delta \left( \Delta (\nabla (\mathcal{G}[d(X)])_{I_2, P_2})_{i(X, Y) \supset d(X), P_1} \right)_{g(X) \supset (i(X, Y) \supset d(X)), P}
\]

\[
:= \{ \{ \langle (\forall X(g(X) \supset (i(X, Y) \supset d(X))), P, 0 \rangle, \kappa, \epsilon \rangle \rightarrow \\
\langle (g(c^1) \supset (i(c^1, Y) \supset d(c^1)), P, 1 \rangle, \kappa, \epsilon \rangle \rightarrow \\
\langle (i(c^1, Y) \supset d(c^1), P_1, 1 \rangle, \kappa, \epsilon \rangle \rightarrow \langle (d(c^1), P_2, 1 \rangle, \kappa, \epsilon \rangle \}
\]

\[
\mathcal{G}[h(Y) \land (i(X, Y) \land g(X))]_{0, P}
\]

\[
:= \mathcal{G}[h(Y)]_{0, P} \otimes \mathcal{G}[i(X, Y)]_{0, P} \otimes \mathcal{G}[g(X)]_{0, P}
\]

\[
:= (\nabla (h(Y))_{0, P} \otimes (\nabla (i(X, Y)))_{0, P} \otimes (\nabla (g(X)))_{0, P}
\]

\[
:= \{ \{ (h(Y) \land (i(X, Y) \land g(X)), P, 0 \rangle, \kappa, \epsilon \rangle \rightarrow \\
\{ (h(Y), P, 0 \rangle, (i(X, Y) \land g(X), P, 0 \rangle, \kappa, \epsilon \rangle \rightarrow \\
\{ (i(X, a), P, 0 \rangle, (g(X), P, 0 \rangle, \kappa, \epsilon \rangle \rightarrow \\
\{ (i(X, a), P, 0 \rangle, (g(X), P, 0 \rangle, C_3, [a/Y]) \}
\}
\]

Finally, we have in the first iteration:
\[ I^0 = P[P]_0 := \{ \]
\[ s(X), \{ \]
\[ \langle \langle s(X), P, 0 \rangle, \kappa, \epsilon \rangle \rightarrow \]
\[ \langle \langle \forall X \, (g(X) \supset (i(X, Y) \supset d(X))), P, 0 \rangle, C_1, [X/X_1] \rangle \rightarrow \]
\[ \langle \langle g(c^1) \supset (i(c^1, Y) \supset d(c^1)), P, 1 \rangle, \kappa, \epsilon \rangle \rightarrow \]
\[ \langle \langle i(c^1, Y) \supset d(c^1), P_1, 1 \rangle, \kappa, \epsilon \rangle \rightarrow \langle \langle d(c^1), P_2, 1 \rangle, \kappa, \epsilon \rangle \} \]
\[ d(X), \{ \]
\[ \langle \langle d(X), P, 0 \rangle, \kappa, \epsilon \rangle \rightarrow \]
\[ \langle \langle h(Y) \land (i(X, Y) \land g(X)), P, 0 \rangle, C_2, [X/X_1] \rangle \rightarrow \]
\[ \langle \{ [h(Y) \land g(X)], P, 0 \rangle, \{ i(X, Y), P, 0 \}, \{ g(X), P, 0 \} \}, \kappa, \epsilon \rangle \rightarrow \]
\[ \langle \{ [i(X, a), P, 0 \rangle, \{ g(X), P, 0 \}, C_3, [a/Y] \} \} \]
\[
\]
\[ h(X_0), \{ \]
\[ \langle \langle h(X), P, 0 \rangle, \kappa, \epsilon \rangle \rightarrow \{ \emptyset, C_3, [a/X] \} \} \]
\[
\]
Now we can start the second iteration:

\[ P[P]_1 := C[C_1]_{P_0, P} + C[C_2]_{P_0, P} + C[C_3]_{P_0, P} \]

\[ C[C_1]_{P_0, P} := \Theta(C_1)^0 P \bowtie \mathcal{G}[\forall X \, (g(X) \supset (i(X, Y) \supset d(X))))]_{P_0, P} \]
\[ C[C_2]_{P_0, P} := \Theta(C_2)^0 P \bowtie \mathcal{G}[h(Y) \land (i(X, Y) \land g(X))]_{P_0, P} \]
\[ C[C_3]_{P_0, P} := \Theta(C_3)^0 P \]
\[ C[g(X)]_{P_1, P_1} := \Theta(g(X))_{P_1} \]
\[ C[i(X, Y)]_{P_2, P_2} := \Theta(i(X, Y))_{P_2} \]

\[ I_3 := I_0 \cup C[g(X)]_{P_1, P_1} \]
\[ I_4 := I_3 \cup C[i(X, Y)]_{I_3, P_2} \]
\[ \mathcal{G}[\forall X (g(X) \supset (i(X,Y) \supset d(X))))]_{I_0,P} \]
\[ \mathcal{G}[h(Y) \land (i(X,Y) \land g(X)))]_{P_0,P} := \mathcal{G}[h(Y)]_{P_0,P} \otimes \mathcal{G}[i(X,Y)]_{P_0,P} \otimes \mathcal{G}[g(X)]_{P_0,P} \]
\[ := (h(Y) \cdot I^0) \otimes (i(X,Y) \cdot I^0) \otimes (g(X) \cdot I^0) \]
\[ := \mathcal{G}[h(Y) \land (i(X,Y) \land g(X))]_{\emptyset,P} \]

Finally, we obtain that

\[ I^1 = \mathcal{P}[P]_{P_0} := \{ \]
\[ s(X), \{ \]
\[ \langle (s(X), P, 0), \kappa, \epsilon \rangle \rightarrow \]
\[ \langle (\forall X (g(X) \supset (i(X,Y) \supset d(X))), P, 0), C_1, [X/X_1] \rangle \rightarrow \]
\[ \langle (g(c^1) \supset (i(c^1,Y) \supset d(c^1)), P, 1), \kappa, \epsilon \rangle \rightarrow \]
\[ \langle (i(c^1,Y) \supset d(c^1), P_1, 1), \kappa, \epsilon \rangle \rightarrow \]
\[ \langle (d(c^1), P_2, 1), \kappa, \epsilon \rangle \rightarrow \]
\[ \langle (h(Y^1) \land (i(c^1,Y^1) \land g(c^1)), P_2, 1), C_2, [c^1/X_1^1] \rangle \rightarrow \]
\[ \langle (\{h(Y^1), 0, P\}, \{i(c^1,Y^1) \land g(c^1), P_2, 1\}), \kappa, \epsilon \rangle \rightarrow \]
\[ \langle (\{h(Y^1), P_2, 1\}, \{i(c^1,Y^1), P_2, 1\}, \{g(c^1), P_1\}), \kappa, \epsilon \rangle \rightarrow \]
\[ \langle (\{i(c^1), a, P_2, 1\}, \{g(c^1), P_2, 1\}), C_3, [a/Y^1_1] \rangle \rightarrow \]
\[ \langle (\{i(c^1), a, P_2, 1\}, \{g(c^1), P_2, 1\}), C_3, \epsilon \rangle \rightarrow \]
\[ \langle (\{g(c^1), P_2, 1\}, i(X^1,Y^1), \{[c^1/X^1], [a/Y^1_1]\}) \rightarrow \]
\[ \langle \emptyset, g(X^1), [c^1/X^1] \} \rangle \]
CHAPTER 2. FIRST ORDER HEREDITARY HARROP FORMULAS

\[ d(X), \{ \langle \langle d(X), P, 0 \rangle, \kappa, \epsilon \rangle \rightarrow \langle \langle h(Y) \land (i(X,Y) \land g(X)), P, 0 \rangle, C_2, [X/X_1] \rangle \rightarrow \langle \{ \langle h(Y), P, 0 \rangle, \langle i(X,Y) \land g(X), P, 0 \rangle \}, \kappa, \epsilon \rangle \rightarrow \langle \{ \langle i(X,a), 0, P \rangle, \langle g(X), P, 0 \rangle \}, C_3, [a/Y] \rangle \} \]

\[ h(X), \{ \langle \langle h(X), P, 0 \rangle, \kappa, \epsilon \rangle \rightarrow \langle \emptyset, C_3, [a/X] \rangle \} \]

In the next iteration we can see that \( I^2 \equiv C I^1 \), so \( I^1 \) is a fixed point of the denotational semantic function of our program.

2.5 Operational Semantics

The procedure presented in section 2.2 can be seen as a transition system between configuration states of an operational semantics. This operational semantics describes, in the most concrete way, the resolution principle of the \( fohh \)-programs. We present now an extended semantics based on derivations.

**Definition 2.5.1** Let \( P \) be a \( fohh \)-program. The behavior of a query \( G \) in \( P \) is defined as

\[ B[G in P] = \lambda G. \{ S_0 \rightarrow^* S_n | S_0 \text{ is the initial state } \langle \{ \langle G, P, 0 \rangle \}, \kappa, 0, \epsilon \rangle \} / \equiv_C \]

The behavior modulo variance is a collection that assigns to every goal \( G \) the equivalence classes of derivation sequences which initial state contains only the goal \( G \). The derivation sequences assigned to goals are constructed in a top-down way as described in definition 2.2.5, so we have not only all the successful derivations, but also all failed and infinite derivations, including their prefixes.

**Definition 2.5.2** Let \( P \) be a \( fohh \)-program. Its top-down intuitionistic derivation denotation is

\[ O[P] = \sum_{p(x) \in Goals} B[p(x) in P] \]

Note that \( O[P] \) is defined only for pure atoms, assigning a non empty set of derivations to the atoms that match any head of clause in the program \( P \).

Using the previous denotations we can define the equivalence of two programs \( P_1, P_2 \) as

\[ P_1 \approx P_2 \iff \forall G \in Goals, B[G in P_1] = B[G in P_2]. \]
2.5. OPERATIONAL SEMANTICS

2.5.1 A New Transition System

By employing the low-level operations used in our denotation semantics, we can define a new transition system \( \mathcal{I} \) to obtain the operational semantics. Instead of operating over derivations, this transition system derives a collection into another collection by adding an execution step to the derivations. The transition system is composed by one rule:

\[
\frac{C \in \mathcal{C}, C \neq C \triangleright \circ \triangleleft \text{su}(\Theta(P))_P}{C \mapsto C \triangleright \circ \triangleleft \text{su}(\Theta(P))_P}
\] (2.1)

where

\[
\text{su}(I)_P = \sum_{G \in \text{Goals}} G[G]_I, P
\]

and

\[
\Theta(P) = \sum_{c \in P} \Theta(c)
\]

Note that \( \Theta(P) \) is an interpretation where all derivations contain the steps of \textit{identity} and \textit{backchain}, for any clause \( p(t), G \triangleright p(t) \in P \). The function \( \text{su}(I) \) will attempt the instantiation of all pure derivations in \( I \) with every goal \( G \). Hence, \( \text{su}(\Theta(P))_P \) is a proper collection of the form

\[
\langle \{\langle G_1 \land G_2, P, 0 \rangle \}, \kappa, 0, \epsilon \rangle \rightarrow \langle \{\langle G_1, P, 0 \rangle, \{G_2, P, 0\}\}, \kappa, 0, \epsilon \rangle \cdots
\]

\[
\langle \{\langle G_1 \lor G_2, P, 0 \rangle \}, \kappa, 0, \epsilon \rangle \rightarrow \langle \{\langle G_i, P, 0 \rangle \}, \kappa, 0, \epsilon \rangle, i \in \{1, 2\} \cdots
\]

\[
\langle \{\langle G \triangleright D, P, 0 \rangle \}, \kappa, 0, \epsilon \rangle \rightarrow \langle \{\langle G, P \cup \{D\}, 0 \rangle \}, \kappa, 0, \epsilon \rangle \cdots
\]

\[
\langle \{\exists x . G, P, 0 \rangle \}, \kappa, 0, \epsilon \rangle \rightarrow \langle \{\langle G[y/x], P, 0 \rangle \}, \kappa, 0, \epsilon \rangle \cdots
\]

\[
\langle \{\forall x . G, P, 0 \rangle \}, \kappa, 0, \epsilon \rangle \rightarrow \langle \{\langle G[c/x], P, 0 \rangle \}, \kappa, 0 \cup \{\{c, 1\}, \epsilon \} \rangle \cdots
\]

\[
\langle \{\langle p(x), P, 0 \rangle \}, \kappa, 0, \epsilon \rangle \rightarrow \langle \{\langle G, P, 0 \rangle \theta \}, G \triangleright p(t) \rho, 0, \theta \rangle \cdots
\]

\[
\langle \{\langle p(x), P, 0 \rangle \}, \kappa, 0, \epsilon \rangle \rightarrow \langle \emptyset, p(t) \rho, 0, \theta \rangle
\]

(2.2)

Note that \( \Theta(P) \) will be the interpretation of the form

\[
\langle \{\langle p(x), P, 0 \rangle \}, \kappa, 0, \epsilon \rangle \rightarrow \langle \{\langle G, P, 0 \rangle \theta \}, G \triangleright p(t) \rho, 0, \theta \rangle
\]

\[
\langle \{\langle p(x), P, 0 \rangle \}, \kappa, 0, \epsilon \rangle \rightarrow \langle \emptyset, p(t) \rho, 0, \theta \rangle
\]

(2.3)

The intuitive idea of this transition system is to simulate the derivation process of the intuitionistic proof procedure, by extending a set of derivations (collections), with any legal piece of derivation unfolded by the \text{su} operator. Note that this transition system is based on the low-level operators we have defined for our denotation semantics. This peculiarity, will help us to define in advance an operational semantics that uses the same instruments of the denotational semantics. This fact will allow the proof of the equivalence of both denotations. In advance we will omit the program argument of \( \text{su} \) and will assume it by the context.

Now we will show some lemmata that will help us to proof important properties of our semantics, useful for establishing the relationships between the denotational and operational semantics.
Lemma 2.5.3  Let \( d_1, d_2 \) be derivations with \( d_1 \in C_P = \sum_{G \in Goals} B[G \text{ in } P] \). If the concatenation \( d' = d_1 :: d_2 \) is defined then \( d' \in C_P \) too.

Proof.  We prove this lemma by induction on the length of derivations. So, we should prove that \( d \in C_P \Rightarrow d :: d' \in C_P \), given that the same condition holds for a shorter derivation \( d' = d'' \), \( s \) where \( d'' \) is an arbitrary derivation and \( s \) is a single state.

- **Base case** \( (d' = s_0, s_1) \) In this case we can see that there is a possible transition that given a state \( s_0 \) we can obtain \( s_1 \). Since \( \text{last}(d) = s_0 \), by definition of :: is obvious that from \( d \in C_P \) we can derive a single step \( s_1 \).

- **General case** \( (d' = d'', s) \) This proof is very similar to the first one. We have that \( d :: d'' \in C_P \). On the other hand we know that there is a derivation step starting from \( d'' \) that give us \( d' \). So, we can apply to \( d :: d'' \) this derivation step to obtain \( d :: d' \in C_P \).

Symmetrically we can prove that \( d, d' \in C_P \Rightarrow d :: d' \in C_P \), \( d' = s, d'' \) by extending the first derivation in the other direction.

This lemma asserts that a concatenation belongs to the behavior of a program if the two composing derivations belong too. Analogously we can state the following theorem that establishes the same property for the fusion \((\hat{\circ})\) operator. As an immediate consequence of this operator we can see that it is symmetric, which is not in contradiction with the intuitive idea of AND-compositionality. In fact we can see that this operator does not change the computation trees of the respective derivations. Therefore, the resulting tree is the composition of the two subtrees with a common root node associated to the starting state \( S_0 \).

Lemma 2.5.4  Let \( d_1, d_2 \) be derivations with \( d_1, d_2 \in C_P = \sum_{G \in Goals} B[G \text{ in } P] \). If the concatenation \( d' = d_1 \hat{\circ} d_2 \) is defined then \( S_0, d' \in C_P \) too, where

\[
S_0 = \{\langle \{G_1 \land G_2, P, 0\} \rangle, \kappa, 0, \epsilon \}
\]

with \( \{\langle G_1, P, 0 \rangle \} = \text{first}(d_1) \) and \( \{\langle G_2, P, 0 \rangle \} = \text{first}(d_2) \).

Proof.  We prove this lemma by induction on the length of derivations. So we should prove that \( d, d' \in C_P \Rightarrow d \hat{\circ} d' \in C_P \) given that the condition holds for a shorter derivation \( d' = d'' \), \( s \) where \( d'' \) is an arbitrary derivation and \( s \) is a single state.

- **Base case** \( (d' = s_0) \) In this case the resulting derivation is \( d \triangleleft s_0, s_0 \). It is evident that for this derivation we can start a derivation with state \( \text{first}(d) \cup s_0 \) and follow the steps in \( d \) without never deriving a non proper state.
general case \((d = d', s)\) This case is simpler. In fact, \(d\theta d''\) and \(d\theta d'\) differ only in the last state, where \(\text{last}(d\theta d') = \text{last}(d') \triangleleft \text{last}(d)\). Hence we can derive \(d\theta d''\) using the last step in \(d'\) obtaining a proper derivation.

It is easy to see that this operator is symmetric, so the proof is complete.

The following lemma establishes the same property for the instantiation derivation.

**Lemma 2.5.5** The operation of instantiation \(\delta\) is closed with respect to the derivations in behavior of a program \(C_P = \sum_{G \in \text{Goals}} B[G \in P]\), i.e. if \(d = d_1\theta\) is defined for \(d_1 \in C_P\) then \(d \in C_P\).

**Proof.** We prove this theorem by induction on the length of derivations. So we should proof that if \(d_1 \in C_P \Rightarrow d_1\theta \in C_P\) then \(d'_1 \in C_P \Rightarrow d'_1\theta \in C_P\) and \(d'_1 = d_1, s\).

**base case** Straightforward, given that every starting state is in \(C_P\).

**general case** If \(d_1\theta, d'_1 \in C_P\) is easy to see that \(d'_1\theta\) also is in \(C_P\) provided that we can extend \(d_1\theta\) with the last step in \(d'_1\).

**Lemma 2.5.6** The operation of compatible extension \(\triangleright \circ \triangleleft\) is closed with respect to the collections in behavior of a program \(C_P = \sum_{G \in \text{Goals}} B[G \in P]\), i.e. if \(C = C_1 \triangleright \circ \triangleleft C_2\) is defined for \(C_1, C_2 \in C_P\) then \(C \in C_P\).

**Proof.** By definition of \(\triangleright \circ \triangleleft\) we have that

\[
C = C_1 \triangleright \circ \triangleleft C_2 = \lambda G.C_1(G) \cup \{d_1 :: (S_n \triangleright \circ \triangleleft d'_2) \mid d_1 \in C_1(G), \ d_2 \in C_2(G) \text{ and } d'_2 = \delta_{G'}(d_2) \text{ with } G' \in W_n\}
\]

where \(S_n\) is the last state of \(d_1\). If \(C_1, C_2 \in C_P\) then by theorem 2.5.3 is evident that \(d_1 :: (S_n \triangleright \circ \triangleleft d'_2) \in C_P\) if the operations are defined.

The compatible extension operator has interesting distributivity properties with respect to operation of sum, instantiation, fusion and raising. The following lemma shows these properties. These properties are at the base of the compositionality and equivalence between the bottom-up and the top-down denotation.

**Lemma 2.5.7** Let \(C, C_1\) and \(C_2\) be collections and let \(I\) be an interpretation. Then, the following statements hold

1. \((C_1 + C_2) \triangleright \circ \triangleleft C = (C_1 \triangleright \circ \triangleleft C) + (C_2 \triangleright \circ \triangleleft C)\)

2. \(\delta_p(t)(I) \triangleright \circ \triangleleft C = \delta_p(t)(I \triangleright \circ \triangleleft C)\)

3. \((C_1 \triangleright C_2) \triangleright \circ \triangleleft C = (C_1 \triangleright \circ \triangleleft C) \triangleright (C_2 \triangleright \circ \triangleleft C)\)
4. \((C_1 \uparrow x) \circ \circ \lambda C = (C_1 \circ \circ C) \uparrow x\)

**Proof.** We prove each point separately.

**Point 1** Follows immediately from definition of + and \(\circ \circ \circ\) operators.

\[
(C_1 \circ \circ C) + (C_2 \circ \circ C) =
\]

\[
= \lambda G. C_1(G) \cup \left\{ d_1 : (S_n \circ d_1') \mid d_1 \in C_1(G), d \in C(G') \wedge d_1' = \delta_{C_1'}(d) \right\} \bigcup C_2(G) \cup \left\{ d_2 : (S_n \circ d_2') \mid d_2 \in C_2(G), d \in C(G') \wedge d_2' = \delta_{C_2'}(d) \right\}
\]

\[
= \lambda G. (C_1 + C_2)(G) \cup \{ d_1 : (S_n \circ d_1') \mid d_1 \in (C_1 + C_2)(G), d \in C(G') \wedge d_1' = \delta_{C_1'}(d) \}
\]

where \(S_n\) is the last state of \(d_1 \in (C_1 + C_2)(G)\) and \(S_n^i\) is the last state of \(d_i \in C_i(G)\).

**Point 2**

\[
\delta_{p(t)}(I) \circ \circ C =
\]

\[
= \lambda G. \delta_{p(t)}(I)(G) \cup \{ d_1 : (S_n \circ d_2') \mid d_1 \in \delta_{p(t)}(I)(G), d_2 \in C(G') \wedge d_2' = \delta_{G'}(d_2) \}
\]

(note that \(G'\) is a goal instantiated by \(\delta_{p(t)}\), so we can rewrite)

\[
= \lambda G. \delta_{p(t)}(I)(G) \cup \{ \delta_{p(t)}(d_1) : (S_n \circ d_2') \mid d_1 \in I(G), d_2 \in C(G') \wedge d_2' = \delta_{G'}(d_2) \}
\]

\[
= \lambda G. \delta_{p(t)}(I)(G) \cup \{ \delta_{p(t)}(d_1) : (S_n \circ d_2') \mid d_1 \in I(G), d_2 \in C(G') \wedge d_2' = \delta_{G'}(d_2) \}
\]

\[
= \delta_{p(t)}(I \circ \circ C)
\]

**Point 3** This point is more complex to prove and requires some assumptions. From definition of \((\circ)\) we can see that this operator is symmetric and associative. So the following equalities hold

\[
(d_1 \circ d_2) = d_2 \circ d_1
\]

\[
((d_1 \circ d_2) \circ d_3) = d_1 \circ (d_2 \circ d_3).
\]
By definition of compatible extension operator we can write \((C_1 \triangleright \omega \triangleleft C) \# (C_2 \triangleright \omega \triangleleft C)\) by using a simplified notation as

\[(C_1 + C_1 \# C')(C_2 + C_2 \# C'')\]

where \(C''\) is the set of all possible instantiations of \(C\) given a goal of \(last(C_1)\)

\[C'' = \{ \delta_{G'}(d) | d \in C(G), G' \in last(C_1(G'')) \}\]

analogously

\[C''' = \{ \delta_{G''}(d) | d \in C(G), G'' \in last(C_2(G''')) \}\]

By applying the properties of \(\#\) we can derive

\[
(C_1 + C_1 \# C')(C_2 + C_2 \# C'') = C_1 \# C_2 + \]

\[
(C_1 \# C')(C_2 + (C_2 \# C'')) = C_1 \# C_2 + (C_1 \# C') \# (C_2 \# C'') +
\]

\[
(C_1 \# C') \# C_2 + \]

\[
(C_1 \# C') \# (C_2 \# C'') = C_1 \# C_2 + (C_1 \# C') \# (C_2 \# C'') +
\]

\[
(C_1 \# C') \# C_2 + (C_1 \# C') \# (C_2 \# C'') = C_1 \# C_2 + (C_1 \# C') \# (C_2 \# C'')
\]

By definition of \(C', C''\) we know that

\[
C' + C''' = C'' + (C_1 \# C') =
\]

\[
\{ \delta_{G'}(d) | d \in C(G), G' \in last(C_1(G'')) \} +
\]

\[
\{ \delta_{G_1'}(d) \# \delta_{G_2'}(d) | d \in C(G), G_1', G_2' \in last(C_1(G'')) \}
\]

\[
\{ \delta_{G_1'}(d) \# \delta_{G_2'}(d) | d \in C(G), G_1', G_2' \in last(C_1(G'')) \}
\]

\[
\{ \delta_{G''}(d) | d \in C(G), G'' \in last((C_1 \# C')(G''')) \}
\]

\[
C'''
\]

finally we have that

\[
(C_1 \triangleright \omega \triangleleft C) \# (C_2 \triangleright \omega \triangleleft C) = C_1 \# C_2 + (C_1 \# C_2) \# C'''
\]

\[
= (C_1 \# C_2) \triangleright \omega \triangleleft C
\]
Point 4

\[(C_1 \uparrow x) \triangleright \circ \triangleleft C =\]
\[= \lambda G.C_1(G) \uparrow x \cup \{d_1 : (S^1_n \# d'_1)|d_1 \in C_1(G) \uparrow x, d \in C(G'') \]
\[\quad \text{and } d'_1 = \delta_{G_1}(d) \text{ with } G'_1 \in W'_n\} \]

(note that \(d \uparrow x = d[c/x]\), so we can replace each part of \(d_1 \in C_1(G) \uparrow x\)
by \(d_1 \uparrow x\) with \(d_1 \in C_1(G)\))

\[= \lambda G.C_1(G) \uparrow x \cup \{d_1 : (S^1_n \# d'_1) \uparrow x | d_1 \in C_1(G), \]
\[\quad d \in C(G'') \text{ and } d'_1 = \delta_{G_1}(d) \text{ with } G'_1 \in W'_n\} \]

\[= \lambda G.C_1(G) \uparrow x \cup \{d_1 : (S^1_n \# d'_1) \uparrow x | d_1 \in C_1(G), d \in C(G'') \]
\[\quad \text{and } d'_1 = \delta_{G_1}(d) \text{ with } G'_1 \in W'_n\} \]

\[= (C_1 \triangleright \circ \triangleleft C) \uparrow x\]

The first step for showing the equivalence between the operational and denotational semantics is to show that a goal function evaluated in an interpretation that is a subset of the semantics of the program is always a subset of the behavior of the program.

**Theorem 2.5.8** Let \(G\) be a goal \((G \in \text{Goals})\) and \(I\) a given interpretation such that \(I \sqsubseteq \mathcal{O}[P]\), then \(\mathcal{G}[G]_I \sqsubseteq \mathcal{B}[G \text{ in } P]\)

**Proof.** By definition of \(\triangleright \circ \triangleleft\) we can reformulate the thesis as

\[\forall I \sqsubseteq \mathcal{O}[P], d \in \mathcal{G}[G]_I(G) \Rightarrow d \in \mathcal{B}[G \text{ in } P](G) \quad (2.4)\]

Now we proceed by induction on the structure of \(G\).

\(G := p(t)\): In this case we have \(\mathcal{G}[p(t)] = \nabla(p(t))_I = \lambda p(t).\{\delta_{p(t)}(d)|d \in I(p(x))\}\).

Since \(d \in I(p(x))\) and \(I \sqsubseteq \mathcal{O}[P] \sqsubseteq \Sigma_{G \in \text{Goals}} \mathcal{B}[G \text{ in } P]\), by theorem 2.5.6 we have that \(\delta_{p(t)}(d) \in \mathcal{B}[G \text{ in } P](p(t))\) too.

\(G := G_1 \land G_2\): Suppose that (2.4) holds for \(G_1\) and \(G_2\). By definition of \(\otimes\) we have that \(\mathcal{G}[G_1]_I \otimes \mathcal{G}[G_2]_I =\)

\[\lambda(G_1 \land G_2)\{S_0, (d_1 \# d_2)|d_1 \in \mathcal{G}[G_1]_I, d_2 \in \mathcal{G}[G_1]_I\}\]

where \(S_0 = \langle\{\langle G_1 \land G_2, P, 0\rangle\}, \kappa, 0, \epsilon\rangle\). By definition of \(\#\) we can observe that \(d_1 \# d_2\) is defined, so by theorem 2.5.5 we have that \(S_0, (d_1 \# d_2)\) is also a derivation present in \(\mathcal{B}[G_1 \land G_2 \text{ in } P](G_1 \land G_2)\).

\(G := G_1 \lor G_2\): Immediately, by definition of \(\oplus\) given that (2.4) holds for \(G_1\) and \(G_2\) and applying the or rule to the initial state \(\langle\{G_1 \lor G_2, P, 0\}\rangle, \kappa, 0, \epsilon\).
2.5. OPERATIONAL SEMANTICS

\[ G := \exists x. G' \]: Follows directly from definition of \( \exists \), provided that \( \exists (G[G']_I)_x \) is a renaming in derivations of the variable \( x \) by a new variable \( y \).

\[ G := \forall x. G' \]: Suppose that (2.4) holds for \( G' \). By definition of \( \forall \) we have that

\[ \forall (G[\forall x. G']_I)_x = \lambda(\forall x. G') \{ S_0, (d \uparrow x) | d \in G[G']_I \} \]

where \( S_0 = \{ \{ \forall x. G', P, 0 \}, \kappa, 0, \epsilon \} \). By definition of \( \uparrow \) we have that \( d \uparrow x \) is defined, so by theorem 2.5.5 we have that \( S_0, (d \uparrow x) \) is also a derivation present in \( B[\forall x. G' \text{ in } P](\forall x.G') \).

\[ G := D \supset G' \]: Suppose (2.4) holds for \( G' \) and an augmented program \( P \cup \{ D \} \), i.e. \( d \in G[G']_I(G') = d \in B[G' \text{ in } P \cup \{ D \}](G) \) with \( I' \subseteq O[P \cup \{ D \}] \). By definition of \( G[.]. \), we have that

\[ G[D \supset G']_I = \lambda(D \supset G'). \{ S_0, d | d \in G[G']_I \cup \{ D \} \} \]

where \( S_0 = \{ \{ D \supset G', P, 0 \}, \kappa, 0, \epsilon \} \). Let \( I'' \) be the interpretation \( I \uplus C[D]_I \).

It is sufficient to show that \( I'' \subseteq O[P \cup \{ D \}] \). In that case, by assumption \( d \in G[G']_I(G') \Rightarrow d \in B[G' \text{ in } P \cup \{ D \}](G) \) and by theorem 2.5.10 we could conclude that \( S_0, d \uparrow \{ D \} \) is also included in \( B[D \supset G' \text{ in } P](G) \). Note that analyzing the \textbf{augment} rule, we can see that the only difference between \( B[D \supset G' \text{ in } P](G) \) and \( B[G' \text{ in } P \cup \{ D \}](G) \) is the first state \( S_0 \) in derivations. So, it remains to show that \( I'' \subseteq O[P \cup \{ D \}] \). By definition of \( \uplus \)

\[ I \uplus C[D]_I = (I \uparrow \{ D \} \triangleright \circ \triangleleft C[D]_I \uparrow P) \uparrow \langle C[D]_I \uparrow P \triangleright \circ \triangleleft I \uparrow \{ D \} \rangle \uparrow \]

Therefore, by straightforward induction we can see that it is sufficient to prove that

\[ (I \triangleright \circ \triangleleft C[D]_I) + (C[D]_I \triangleright \circ \triangleleft I) \subseteq O[P \cup \{ D \}] \]

We can derive two alternatives

\[ (I \triangleright \circ \triangleleft \Theta(p(t))) + (\Theta(p(t)) \triangleright \circ \triangleleft I) \subseteq O[P \cup \{ D \}] \]

or

\[ (I \triangleright \circ \triangleleft \Theta(p(t))) \triangleright \triangleleft \triangleright G[G'']_I + (\Theta(p(t)) \triangleright \triangleleft \triangleright G[G'']_I \triangleright \circ \triangleleft I) \subseteq O[P \cup \{ D \}] \]

Note that we intentionally omit the program complementation, in order to simplify the notation. The first one can be easily proved, provided that \( I, \Theta(p(t)) \subseteq O[P \cup \{ D \}] \). In the second case, we know by assumption that \( G[G']_I(G') \subseteq B[G' \text{ in } P \cup \{ D \}](G''), \) hence, \( \Theta(p(t)) \triangleright \triangleright G[G']_I \subseteq O[P \cup \{ D \}] \), which proves that \( I'' \subseteq O[P \cup \{ D \}] \).
Therefore, we can conclude that every derivation included in $\mathcal{G}[G]_I(G)$ is included in $\mathcal{B}[G in P](G)$ too. ■

The former theorem asserts an important relation between the top-down denotation and the bottom-up one. In fact, as the following corollary shows, the query denotational function is a subset of the behavior of a program.

**Corollary 2.5.9** Given a program $P$ the following inclusion holds

$$\mathcal{Q}[G in P] \subseteq \mathcal{B}[G in P]$$

**Proof.** Straightforward from definition of $\mathcal{Q}[.]$ and theorem 2.5.8. ■

Up to now we have proved one half of the equivalence theorem. It remains to show a more complex implication, i.e. $\mathcal{B}[G in P] \subseteq \mathcal{Q}[G in P]$. For this purpose, we will use the new transition system and will take advantage of its denotational nature. In fact it uses the same operators used in the denotational semantics. Consequently, as a first step we need to redefine the behavior of a program in terms of this new transition system.

**Lemma 2.5.10** Let $G$ be a goal ($G \in \text{Goals}$) and $C$ a given collection such that $C \subseteq \mathcal{B}[G in P]$, then $C \triangleright \circ \circ \mathcal{su}(\Theta(P)) \subseteq \mathcal{B}[G in P]$

**Proof.** By definition of $\mathcal{su}$ we can reformulate the thesis as

$$d \in C \triangleright \circ \circ \mathcal{G}[G]_{\Theta(P)}(G) \Rightarrow d \in \mathcal{B}[G in P](G)$$

(2.5)

By theorem 2.5.6 we know that $\triangleright \circ \circ$ is closed, therefore, we need to prove that

$$d \in \mathcal{G}[G]_{\Theta(P)}(G) \Rightarrow d \in \mathcal{B}[G in P](G)$$

(2.6)

to show that (2.5) holds, which is easily proved using the theorem 2.5.8. ■

Now we can state the following theorem that establishes the equivalence between the new transition system and the transition system derived from the proof procedure we have seen in the previous section. This theorem is very important, because it defines an operational semantics of the program in terms of semantics operations over derivations, which are the same operators used in the denotational semantics, and establishes, indeed, the equivalence between the new transition system $\mathcal{I}$ and the proof procedure.

**Theorem 2.5.11**

$$\mathcal{B}[G in P] = \sum \{C | \text{Id}_G \rightarrow^* C\}$$

**Proof.** The proof is made by induction on the length of derivations and on the structure of derivation rules. Define $C^k$ to be the collection with derivations up to length $k$. We will show that $\forall m \exists i$ such that $\mathcal{B}[G in P]^{n+i} \subseteq \sum \{C | \text{Id}_G \rightarrow^{m+1} C\}$ if $\mathcal{B}[G in P]^n \subseteq \sum \{C | \text{Id}_G \rightarrow^m C\}$ for some $n, i > 0$, and (the converse case) $\mathcal{B}[G in P]^{n+i} \supseteq \sum \{C | \text{Id}_G \rightarrow^{m+1} C\}$ given that $\mathcal{B}[G in P]^n \supseteq \sum \{C | \text{Id}_G \rightarrow^m C\}$.
2.5. OPERATIONAL SEMANTICS

\( n, m = 1 \): By definition of \( B[G in P]^1 \) we have a derivation of the form
\[
\{\{G, P, 0\}\}, \kappa, 0, \epsilon
\]
on the other hand we can see that the first state of \( \exists \) is \( Id_G \). So, we have that \( B[G in P]^1 \equiv_c \sum \{C|Id_G \mapsto^0 C\} \).

\( n, m > 0 \): Now we proceed by induction on the derivation rules.

\( \sqsubseteq_c \): Suppose there is a derivation \( d' = d, S_{n+1} \) such that \( d' \in B[G in P]^{n+1} \), i.e., in some derivation \( d \in B[G in P]^n \) a rule was applied.

If the \textbf{and} rule was applied, then
\[
S_{n+1} = \langle W \cup \{(G_1, P, 0)\} \cup \{(G_2, P, 0)\} - \{(G_1 \land G_2, P, 0)\}, \kappa, L, \theta \rangle
\]
From definition of \( su(I) \) and from (2.2) is straightforward that exists \( d_1 \in G[G_1 \land G_2]_{\Theta(P)} \), such that
\[
d_1 = \langle\{(G_1 \land G_2, P, 0)\}, \kappa, L, \theta\rangle \rightarrow \langle\{(G_1, P, 0), (G_2, P, 0)\}, \kappa, L, \theta\rangle
\]
By hypothesis \( d \in \sum \{C|Id_G \mapsto^m C\} \), then we have that \( d \triangleright \circ \triangleleft d_1 = d' \). So, we can conclude that in this case \( B[G in P]^{n+1} \) is always a subset of \( \sum \{C|Id_G \mapsto^{m+1} C\} \). In the same way we obtain that \( d \in \sum \{C|Id_G \mapsto^{m+1} C\} \) for the rules \textbf{or}, \textbf{augment}, \textbf{instance} and \textbf{generic}. The proof for the cases of \textbf{identity} and \textbf{backchain}, follows directly from the definition of \( su(I) \) and from (2.2). In fact we have that
\[
\langle\{(p(x), P, 0)\}, \kappa, L_0, \epsilon\rangle \rightarrow \langle\{(G, P, 0)\}, \kappa, L_0, \epsilon\rangle \rightarrow \langle\{, \epsilon\}, \kappa, L_1, \theta\rangle.
\]
belongs to \( su(\Theta(P)) \) for every clauses \( p(x), G \supset p(t) \) of \( P \). Therefore, we have that \( d' \in \sum \{C|Id_G \mapsto^m C\} \triangleright \circ \triangleleft su(\Theta(P)) \).

\( \sqsupseteq_c \): Now we will prove the converse case. Suppose there is a derivation \( d \in \sum \{C|Id_G \mapsto^n C\} \), then we should prove that for every last state in \( d_1 \in B[G in P]^n \) it is possible to apply a derivation rule that yields to a new derivation which is equivalent to \( d \). Note that \( \sum \{C|Id_G \mapsto^n C\} = \sum \{C|Id_G \mapsto^{n-1} C\} \triangleright \circ \triangleleft su(\Theta(P)) = B[G in P]^n \triangleright \circ \triangleleft su(\Theta(P)) \).

Therefore, by observing (2.2) and the definition of \( \triangleright \circ \triangleleft \), we can see that for each applied rule, we can find a derivation \( d_2 \in su(\Theta(P)) \) such that \( d = d_1 \triangleright \circ \triangleleft d_2 \).

The previous theorem shows that we can construct the behavior of a program by extending the identity interpretation with pieces of derivations obtained from \( su(\Theta(P)) \), until no further changes occurs, just like the intuitionistic procedure does. The following corollary is a consequence of this theorem. We rewrite the transitive closure of \( Id_G \mapsto C \) for any goal \( G \), as a function with a fixed-point denotation, for showing that it is equivalent to the operational semantics of the program.
Corollary 2.5.12 Let $O[P]$ be the operational semantics of a program $P$, then the following equality holds

$$O[P] = \text{lfp } T[P]$$

where $T[P]$ is a function defined as

$$T[P] = \lambda I. \begin{cases} \Theta(P) & \text{if } I = \bot \\ I \triangleright \circ \lhd \text{su}(\Theta(P)) & \text{otherwise} \end{cases}$$

Proof. We have by definition that $O[P] = \sum_{p(x) \in Goals} B[p(x) \in P]$. So, we can derive the equalities

$$O[P] = \text{lfp } T[P] = \sum_{p(x) \in Goals} \sum \{ C | Id_{p(x)} \mapsto^* C \} \quad (2.7)$$

Now it is sufficient to prove that (2.7) holds. The proof can be easily constructed by induction on the number of application of $T$ and $\exists$. Note that

$$\sum \{ C | Id_{p(x)} \mapsto^* C \} = \sum \{ C | Id_{p(x)} \mapsto^* C, p(x) \in Goals \}$$

therefore the behavior of a program can be expressed as the set of collections obtained by using $\exists$ and starting from all possible pure atomic goals. In fact, for $n = 1$, we have $T[P] = \text{su}(\Theta(P))$ which is equivalent to

$$\sum \{ C | Id_{p(x)} \text{su}(\Theta(P)), p(x) \in Goals \}$$

Now suppose that

$$T[P]^n = \sum \{ C | Id_{p(x)} \mapsto^n C, p(x) \in Goals \},$$

for $n > 1$, then, as result we have

$$T[P]^{n+1} = \sum \{ C \triangleright \circ \text{su}(\Theta(P)) | Id_{p(x)} \mapsto^n C, p(x) \in Goals \}$$

$$T[P]^{n+1} = \sum \{ C | Id_{p(x)} \mapsto^{n+1} C, p(x) \in Goals \}$$

The following lemma is a building block for constructing the proof of equivalence between denotations.

Lemma 2.5.13 Let $I$ be an interpretation such that $I \sqsubseteq O[P]$ and let $\Psi^k$ be the expression

$$\sum \mathcal{G}[G]_{\Theta(P)} \triangleright \circ_1 \ldots \triangleright \circ_k \sum \mathcal{G}[G]_{\Theta(P)}$$

then for any goal $G$ and $k > 0$ we have

$$\mathcal{G}[G]_I \triangleright \circ \Psi^k \sqsubseteq \mathcal{G}[G]_{I \triangleright \circ \Psi^k}$$
Proof. By induction on the structure of $G$

$G := p(t)$

$G[p(t)]_I \triangleright \circ \Psi^k = \delta_{p(t)}(I) \triangleright \circ \Psi^k = \delta_{p(t)}(I) \triangleright \circ \Psi^k$ (by theorem 2.5.7)

$G := G_1 \land G_2$

$G[G_1 \land G_2]_I \triangleright \circ \Psi^k = \lambda(G_1 \land G_2). \left\{ S_0, (d_1, d_2) \mid d_1 \in G[G_1]_I(G_1), \quad d_2 \in G[G_2]_I(G_2) \right\} \triangleright \circ \Psi^k = \lambda(G_1 \land G_2). \left\{ S_0, (d_1, d_2) \mid d_1 \in G[G_1]_{I \triangleright \circ \Psi^k}(G_1), \quad d_2 \in G[G_2]_{I \triangleright \circ \Psi^k}(G_2) \right\} = (by \ hypothesis)$

$G := G_1 \lor G_2$

$G[G_1 \lor G_2]_I \triangleright \circ \Psi^k = \lambda(G_1 \lor G_2). \left\{ S_0, d \mid d \in G[G_1]_I(G_1) \right\} \triangleright \circ \Psi^k = \lambda(G_1 \lor G_2). \left\{ S_0, d \mid d \in G[G_1]_{I \triangleright \circ \Psi^k}(G_1) \right\} = \lambda(G_1 \lor G_2). \left\{ S_0, d \mid d \in G[G_2]_{I \triangleright \circ \Psi^k}(G_2) \right\} = (by \ theorem \ 2.5.7)$

$G := \exists x. G'$

$G[\exists x. G']_I \triangleright \circ \Psi^k = \lambda(\exists x. G'). \left\{ S_0, d[y/x] \mid d \in G[G']_I(G') \right\} \triangleright \circ \Psi^k = \lambda(\exists x. G'). \left\{ S_0, d[y/x] \mid d \in G[G']_{I \triangleright \circ \Psi^k}(G') \right\} = \lambda(\exists x. G'). \left\{ S_0, d[y/x] \mid d \in G[G']_{I \triangleright \circ \Psi^k}(G') \right\} = (by \ hypothesis)$

where $S_0 = \{(G_1 \land G_2, P, 0)\}, \kappa, 0, \epsilon$.

where $S_0 = \{(G_1 \lor G_2, P, 0)\}, \kappa, 0, \epsilon$.

where $S_0 = \{(\exists x. G', P, 0)\}, \kappa, 0, \epsilon$.
$G := \forall x. G'$

\[
G[\forall x. G']_I \triangleright \circ \Psi^k =
\lambda (\forall x. G'). \{ S_0, d \uparrow x \mid d \in [G[G']_I]_I (G') \} \triangleright \circ \Psi^k =
\lambda (\forall x. G'). \{ S_0, d \uparrow x \mid d \in [G[G']_I \triangleright \circ \Psi^k] (G') \} =
\text{(by theorem 2.5.7)}
\lambda (\forall x. G'). \{ S_0, d \uparrow x \mid d \in [G[G']_{\Psi^k}] (G') \} =
\text{(by hypothesis)}
\]

$G := D \supset G'$

\[
G[D \supset G']_I \triangleright \circ \Psi^k =
\lambda (D \supset G'). \{ S_0, d \mid d \in [G[G_1]_{\Psi^k[D]}]_I (G_1) \} \triangleright \circ \Psi^k =
\lambda (D \supset G'). \{ S_0, d \mid d \in [G[G_1]_{\Psi^k[D]} \triangleright \circ \Psi^k] (G_1) \} =
\lambda (D \supset G'). \{ S_0, d \mid d \in [G[G_1]_{\Psi^k[D]}]_I (G_1) \} \subseteq
\text{(\triangleright \circ \Psi^k is undefined for these arguments)}
\lambda (D \supset G'). \{ S_0, d \mid d \in [G[G_1]_{\Psi^k[D]}]_I (G_1) \} =
\lambda (\triangleright \circ \Psi^k \text{ by monotonicity of } G[.] )
\]

where $S_0 = \langle \{ \langle \forall x. G', P, 0 \rangle \}, \kappa, 0, \epsilon \rangle$. Note that $G[G_1]_{\Psi^k[D]} \triangleright \circ \Psi^k$ is undefined because the arguments have incompatible program context, in consequence derivations of $G[G_1]_{\Psi^k[D]}$ can't be fused with derivations in $\Psi^k$.

\[ \square \]

We can state now the theorem that asserts the equivalence relation between the bottom-up denotation and the top-down denotation.

**Theorem 2.5.14** Let $P$ be a folh-program, then

\[ O[P] = \text{lfp } P[P], \tag{2.8} \]

moreover

\[ B[G \text{ in } P] = Q[G \text{ in } P]. \tag{2.9} \]

**Proof.** By corollary 2.5.9 we know that $Q[G \text{ in } P] \subseteq B[G \text{ in } P]$, therefore it rest to prove only the inclusion in the other way. By theorem 2.5.11 and corollary 2.5.12 we can rewrite the thesis (2.8) in the following manner

\[ \text{lfp } T[P] = \text{lfp } P[P] \tag{2.10} \]

Now we will prove (2.10) by induction on the number of application of $T$ and $P$. In other words, given that $T^n \subseteq P^m$, we will show that $T^{n+1} = P^{m'}$ from some $m' \geq m$. Note that by definition of $T$ and $P$

\[
P^m = \sum_{G \in P} \Theta(G \supset p(t)) \bowtie G[G]_{\Theta(p)} + \sum_{p(t) \in P} \Theta(p(t))
\]

\[
T^n = T^{n-1} \triangleright \circ \sum_{G \in \text{Goals}} \Theta(G) \Theta(p)
\]
2.5. OPERATIONAL SEMANTICS

n = 1 Since \( G[G] \) is a monotonic function and \( P^1 = \mathbb{G}(G \supset p(t)) \bowtie G[G]_{\Theta(P)} \), we can derive

\[
T^1 = (\sum \Theta(G \supset p(t)) + \sum \Theta(p(t))) \triangleright \triangleleft \sum_{G \in \text{Goals}} G[G]_{\Theta(P)} = \sum \Theta(G \supset p(t)) + \sum \Theta(p(t)) = \sum \Theta(G \supset p(t)) \bowtie G[G]_{\Theta(P)} + \sum \Theta(p(t))
\]

Note that \( \triangleright \triangleleft \) is distributive over \( + \), \( C_1 \triangleright \triangleleft C_2 = C_1 \triangleright \triangleright C_2 \) if \( \text{last}(C_1) = \text{first}(C_2) \), and by property (1) of \( \triangleright \triangleleft \), \( \Theta(P) \triangleright \triangleleft \sum_{G \in \text{Goals}} G[G]_{\Theta(P)} = \Theta(P) \).

n > 1 Is easy to see that \( \sum_{p(t) \in P} \Theta(p(t)) \) is a subset of \( P^n \) and \( T^m \) for \( m, n > 1 \), so, it is sufficient to show

\[
T^n \triangleright \triangleleft \sum_{G \in \text{Goals}} G[G]_{\Theta(P)} \subseteq \sum_{G \supset p(t) \in P} \Theta(G \supset p(t)) \bowtie G[G]_{P^n}
\]

Hence, by hypothesis is sufficient to prove that for every clause of \( P \)

\[
\Theta(G' \supset p(t)) \triangleright \triangleleft \Psi^{m-1} \Sigma \Theta(G' \supset p(t)) \bowtie G[G']_{\Theta(P)} \triangleright \triangleleft \Psi^{m-2}
\]

Analyzing the definition of \( \triangleright \triangleleft \) we can see that \( \triangleright \triangleleft \) is defined only for \( G' \), as consequence

\[
\Theta(G' \supset p(t)) \triangleright \triangleleft \Psi^{m-1} = \Theta(G' \supset p(t)) \triangleright \triangleleft G[G']_{\Theta(P)} \triangleright \triangleleft \Psi^{m-2}
\]

On the other hand, we can also see that \( \Theta(G' \supset p(t)) \triangleright \triangleleft G[G']_{\Theta(P)} \triangleright \triangleleft \Psi^{m-2} \subseteq \)

\[
\Theta(G' \supset p(t)) \bowtie G[G']_{\Theta(P)} \triangleright \triangleleft \sum_{G \in \text{Goals}} G[G]_{\Theta(P)} + \sum \Theta(p(t))\]

At this point, we need to prove that

\[
G[G']_{\Theta(P)} \triangleright \triangleleft \Psi^{m-2} \subseteq G[G']_{\sum_{G \supset p(t)} \bowtie G[G']_{\Theta(P)} + \sum \Theta(p(t))}
\]

which is shown using lemma 2.5.13. Therefore, we have finally that (2.8) holds.

Now to prove (2.9), it is sufficient to show

\[
\text{Id}_G \triangleright \triangleleft \text{fp} \Psi \subseteq G[G]_{\text{fp} \Psi[P]}
\]

\[
\text{Id}_G \triangleright \triangleleft \text{fp} \Psi = G[G]_{\Theta(P)} \triangleright \triangleleft \text{fp} \Psi \subseteq G[G]_{\Theta(P)} \bowtie \text{dfp} \Psi \subseteq \text{corollary 2.5.12)}
\]

\[
G[G]_{\Theta(P)} = G[G]_{\Theta(P)} \bowtie \text{fp} \Psi \subseteq \text{corollary 2.5.12)}
\]
2.6 Semantic Properties

The program denotation $O[P]$ has several interesting properties. These can all be viewed as compositionality properties, and are based on the semantics operators defined in section 2.4. The first result is a theorem which proves the minimality of our semantics. The other result is a theorem that shows the compositionality of the semantic function $B$ with respect to procedure calls and the different syntactic operators of the language.

It can be easily shown that the denotation $O$ is correct and minimal with respect to the equivalence between programs ($\approx$).

**Theorem 2.6.1** Let $P_1$ and $P_2$ be fohh-programs, then $P_1$ and $P_2$ are equivalent if and only if its operational semantics are equivalent, i.e.,

$$P_1 \approx P_2 \iff O[P_1] = O[P_2]$$

**Proof.** We prove both implications separately

$\Rightarrow$ We need to prove $P_1 \approx P_2 \Rightarrow O[P_1] = O[P_2]$, which is immediately derived from definition of $\approx$.

$\Leftarrow$ To prove the implication is this direction we reduce to a contradiction. Suppose $O[P_1] = O[P_2]$ and suppose $\exists G \in \text{Goals}$ such that

$$B[G \text{ in } P_1] \neq B[G \text{ in } P_2]$$

then, we can derive

$$Q[G \text{ in } P_1] \neq Q[G \text{ in } P_2] \quad (\text{by theorem 2.5.14})$$

$$G[G]_{\uparrow P_1} \neq G[G]_{\uparrow P_2} \quad (\text{by definition of } Q)$$

$$G[G]_{\uparrow P_1} \neq G[G]_{\uparrow P_2} \quad (\text{by theorem 2.5.14})$$

which is a contradiction because we have assumed that $O[P_1] = O[P_2]$.

Another interesting property can be easily verified when fixed-point interpretations are involved. In fact,

$$(O[P_1] \uparrow P_2 + O[P_2] \uparrow P_1) \uparrow = (O[P_1] \uparrow P_2 \triangleright \circ \triangleright O[P_2] \uparrow P_1) \uparrow + (O[P_2] \uparrow P_1 \triangleright \circ \triangleright O[P_1] \uparrow P_2) \uparrow$$

Since $O[P_1] \uparrow P_2 \triangleright \circ \triangleright O[P_1] \uparrow P_2 = O[P_1] \uparrow P_2$ and $O[P_2] \uparrow P_1 \triangleright \circ \triangleright O[P_2] \uparrow P_1 = O[P_2] \uparrow P_1$.

Finally we can state the following theorem, which establishes the $OR$-compositionality property between program denotations.
Theorem 2.6.2 Let $P_1$ and $P_2$ be programs. Then
\[
\mathcal{O}[P_1 \cup P_2] = \mathcal{O}[P_1] \uplus \mathcal{O}[P_2]
\] (2.11)

Proof. We use a similar approach to that used in the proof of theorem 2.5.11. By definition of $\uplus$ we can rewrite (2.11) as
\[
\mathcal{O}[P_1 \cup P_2] = (\mathcal{O}[P_1] + \mathcal{O}[P_2]) \uparrow
\] (2.12)

For simplicity in this proof, we omit the program complementation $\uparrow P_i$ and assume that all derivations in $\mathcal{O}[P_i]$ are already complemented with their respective counterpart program. From theorem 2.5.14 we have that
\[
\mathcal{O}[P_1] = \Theta(P_1) \bowtie \circ \Psi_{P_1} \uparrow
\]
\[
\mathcal{O}[P_2] = \Theta(P_2) \bowtie \circ \Psi_{P_2} \uparrow
\]
where
\[
\Psi_P = \sum_{G \in \text{Goals}} G[G]_{\Theta(P)}
\]

Therefore the expression (2.12) can be stated as
\[
\sum \mathcal{B}[p(x) \text{ in } P_1 \cup P_2] = \left[[(\Theta(P_1) \bowtie \circ \Psi_{P_1} \uparrow) \bowtie \circ \Theta(P_2) \bowtie \circ \Psi_{P_2} \uparrow]\right] \uparrow + \left[[(\Theta(P_2) \bowtie \circ \Psi_{P_2} \uparrow) \bowtie \circ \Theta(P_1) \bowtie \circ \Psi_{P_1} \uparrow]\right] \uparrow
\]

Thus, we need to prove the inclusion in both directions, i.e.,
\[
d \in \sum \mathcal{B}[p(x) \text{ in } P_1 \cup P_2] \Rightarrow
\]
\[
d \in \left[[(\Theta(P_1) \bowtie \circ \Psi_{P_1} \uparrow) \bowtie \circ \Theta(P_2) \bowtie \circ \Psi_{P_2} \uparrow]\right] \uparrow + \left[[(\Theta(P_2) \bowtie \circ \Psi_{P_2} \uparrow) \bowtie \circ \Theta(P_1) \bowtie \circ \Psi_{P_1} \uparrow]\right] \uparrow
\]
and
\[
d \in \left[[(\Theta(P_1) \bowtie \circ \Psi_{P_1} \uparrow) \bowtie \circ \Theta(P_2) \bowtie \circ \Psi_{P_2} \uparrow]\right] \uparrow + \left[[(\Theta(P_2) \bowtie \circ \Psi_{P_2} \uparrow) \bowtie \circ \Theta(P_1) \bowtie \circ \Psi_{P_1} \uparrow]\right] \uparrow \Rightarrow
\]
\[
d \in \sum \mathcal{B}[p(x) \text{ in } P_1 \cup P_2]
\]

Without loss of generality, we will consider only derivations starting with $\Theta(P_1)$. The same proof could be repeated for those derivations starting with $\Theta(P_2)$. Now we use an inductive method based on the length of derivations. In other words, we need to prove in both directions that for every $n$, exists $i, m' > m \geq 0$ such that
\[
\sum \mathcal{B}[p(x) \text{ in } P_1 \cup P_2]^{n+i} \subseteq \left[[(\Theta(P_1) \bowtie \circ \Psi_{P_1} \uparrow) \bowtie \circ \Theta(P_2) \bowtie \circ \Psi_{P_2} \uparrow]\right] \uparrow^{m'}
\]

provided that
\[
\sum \mathcal{B}[p(x) \text{ in } P_1 \cup P_2]^{n} \subseteq \left[[(\Theta(P_1) \bowtie \circ \Psi_{P_1} \uparrow) \bowtie \circ \Theta(P_2) \bowtie \circ \Psi_{P_2} \uparrow]\right] \uparrow^{m}
\]
CHAPTER 2. FIRST ORDER HEREDITARY HARROP FORMULAS

Note that the interpretation \( \Theta(P_1) \triangleright \circ \triangleleft \Psi_{P_1} \uparrow \) \( \triangleright \circ \triangleleft (\Theta(P_2) \triangleright \circ \triangleleft \Psi_{P_2} \uparrow) \) can be expressed in terms of the union

\[
\Theta(P_1) + \\
\Theta(P_1) \triangleright \circ \triangleleft \Psi_{P_1} \uparrow + \\
\Theta(P_1) \triangleright \circ \triangleleft \Theta(P_2) + \\
\Theta(P_1) \triangleright \circ \triangleleft \Psi_{P_1} \uparrow \triangleright \circ \triangleleft \Theta(P_2) + \\
\Theta(P_1) \triangleright \circ \triangleleft \Psi_{P_1} \uparrow \triangleright \circ \triangleleft \Theta(P_2) \triangleright \circ \triangleleft \Psi_{P_2} \uparrow
\]

(2.13)

\( n = 1 \): This case is obvious. In fact, the first state of \( B[p(x) \text{ in } P_1 \cup P_2] \) has the form \( \{\langle p(x), P_1 \cup P_2, 0 \rangle \}, \kappa, L_0, \epsilon \), which is also the first state of derivations of \( \Theta(P_1) \).

\( n > 1 \): Now we prove separately the inclusion in both directions.

\( \sqsubseteq \): Suppose \( d \in \sum B[p(x) \text{ in } P_1 \cup P_2]^{n+1} \), then by hypothesis

\[
first(d)^n \in \sum B[p(x) \text{ in } P_1 \cup P_2]^n
\]

The only difference between \( d \) and \( first(d)^n \) is the last state added to \( d \), which is the consequence of the application of a transition rule of the proof procedure. Suppose that it was applied the and rule, then the last state of the will have the form \( S_{n+1} = \{W \cup \{G_1 \cup \{G_2\} - \{G_1 \land G_2\}, \kappa, L, \theta\} \). By hypothesis we know that \( first(d)^n \) is contained by (2.13).

From definition of \( su(I) \) and from (2.2) we obtain that exists \( d_1 \in G[G_1 \land G_2]_{\Theta(P_1)} \) such that

\[
d_1 = \{\langle G_1 \land G_2, P_1, 0 \rangle, \kappa, L, \theta\} \rightarrow \{\langle G_1, P_1, 0 \rangle, \kappa, L, \theta\}
\]

By hypothesis \( d \in \sum \{C|Id_G \mapsto^m C\} \), then we have that \( d \triangleright \circ \triangleleft d_1 = d' \). So, we can conclude that in this case \( B[G \text{ in } P_1 \cup P_2]^{n+1} \) is always a subset of \( \sum \{C|Id_G \mapsto^m C\} \). In the same way we obtain that \( d \in \sum \{C|Id_G \mapsto^{m+1} C\} \) for the rules or, augment, instance and generic.

The proof for the cases of identity and backchain, follows directly from the definition of \( su(I) \) and from (2.2). In fact we have that

\[
\{\langle p(x), P_1, 0 \rangle, \kappa, L_0, \epsilon\} \rightarrow \{\langle G, P_1, 0 \rangle, G \supset p(t), p(t), L_1, \theta\}, \\
\{\langle p(x), P_1, 0 \rangle, \kappa, L_0, \epsilon\} \rightarrow \{\emptyset, p(t), p(t), L_1, \theta\}
\]

belongs to \( su(\Theta(P)) \) for every clauses \( p(x), G \supset p(t) \) of \( P_1 \). Therefore, we have that \( d' \in \sum \{C|Id_G \mapsto^m C\} \triangleright \circ \triangleleft su(\Theta(P_1)) \).
The same arguments can be provided for showing the converse case. Suppose
\[ d \in [(\Theta(P_1) \triangleright \circ \Psi_{P_1}) \triangleright \circ \triangleleft (\Theta(P_2) \triangleright \circ \Psi_{P_2})] \uparrow^{m+1} \]
then by hypothesis we have \( d = d' \triangleright \circ \triangleleft d'' \) where
\[ \sum B[p(x) \text{ in } P_1 \cup P_2]^n \supseteq d' \in [(\Theta(P_1) \triangleright \circ \Psi_{P_1}) \triangleright \circ \triangleleft (\Theta(P_2) \triangleright \circ \Psi_{P_2})] \uparrow^m \]
and \( d'' \) is a derivation of the form (2.13) such that
\[ \sum B[p(x) \text{ in } P_1 \cup P_2]^n \supseteq \sum B[p(x) \text{ in } P_1 \cup P_2]^n \supseteq d'', d' \]
By applying lemma 2.5.7, we obtain
\[ \sum B[p(x) \text{ in } P_1 \cup P_2]^n \supseteq d = d' \triangleright \circ \triangleleft d'' \]
Since \( d' \) and \( d'' \) are finite we can always find a natural number \( i > 0 \) such that
\[ \sum B[p(x) \text{ in } P_1 \cup P_2]^{n+i} \supseteq d = d' \triangleright \circ \triangleleft d'' \]
Finally, we can conclude
\[ \sum B[p(x) \text{ in } P_1 \cup P_2]^{n+i} \supseteq [(\Theta(P_1) \triangleright \circ \Psi_{P_1}) \triangleright \circ \triangleleft (\Theta(P_2) \triangleright \circ \Psi_{P_2})]^{m+1} \]
As an immediate consequence of this theorem we can construct a new method to calculate the semantics of the program.

**Corollary 2.6.3** Let \( P \) be a program, then
\[ \mathcal{O}[P] = \biguplus_{c \in P} C[c] \perp \]

**Proof.** Obvious by straightforward induction using the previous theorem. It is enough to show that for any clause \( c \)
\[ \mathcal{O}[P \cup \{c\}] = \mathcal{O}[P] \sqcup C[c] \perp \]
which can be easily proved by observing that \( \text{lfp } \mathcal{P}[c] = (C[c] \perp) \uparrow \subseteq \mathcal{O}[P] \sqcup C[c] \perp \).

The following theorem shows the compositionality of the semantic function \( B \) with respect to atomic procedure calls and the different syntactic operators of the language.

---

\[ \sum B[p(x) \text{ in } P_1 \cup P_2]^n \supseteq \sum B[p(x) \text{ in } P_1 \cup P_2]^n \supseteq d'', d' \]

By applying lemma 2.5.7, we obtain
\[ \sum B[p(x) \text{ in } P_1 \cup P_2]^n \supseteq d = d' \triangleright \circ \triangleleft d'' \]
Since \( d' \) and \( d'' \) are finite we can always find a natural number \( i > 0 \) such that
\[ \sum B[p(x) \text{ in } P_1 \cup P_2]^{n+i} \supseteq d = d' \triangleright \circ \triangleleft d'' \]
Finally, we can conclude
\[ \sum B[p(x) \text{ in } P_1 \cup P_2]^{n+i} \supseteq [(\Theta(P_1) \triangleright \circ \Psi_{P_1}) \triangleright \circ \triangleleft (\Theta(P_2) \triangleright \circ \Psi_{P_2})]^{m+1} \]
As an immediate consequence of this theorem we can construct a new method to calculate the semantics of the program.
Theorem 2.6.4 Let $A$ be an atom, $D$ be a program clause and $G, G_1, G_2$ be goals. Then

1. $B[p(t) \text{ in } P] = \nabla(p(t))_{O[P],P}$,
2. $B[G_1 \land G_2 \text{ in } P] = B[G_1 \text{ in } P] \otimes B[G_2 \text{ in } P]$,
3. $B[G_1 \lor G_2 \text{ in } P] = B[G_1 \text{ in } P] \oplus B[G_2 \text{ in } P]$,
4. $B[\exists x. G \text{ in } P] = \exists x B[G \text{ in } P]$, 
5. $B[\forall x. G \text{ in } P] = \forall x B[G \text{ in } P]$, 
6. $B[D \supset G \text{ in } P] = \Delta(B[G \text{ in } P \cup \{D\})_{D \supset G,P}$.

Proof. This points can be proved separately by using the theorem 2.5.14

Point 1.

\[
\begin{align*}
B[p(t) \text{ in } P] &= Q[p(t) \text{ in } P] = (\text{by theorem 2.5.14}) \\
G[p(t)]_{O[P],P} &= (\text{by definition of }Q) \\
G[p(t)]_{O[P],P} &= (\text{by theorem 2.5.14}) \\
\n\end{align*}
\]

Point 2.

\[
\begin{align*}
B[G_1 \land G_2 \text{ in } P] &= Q[G_1 \land G_2 \text{ in } P] = (\text{by theorem 2.5.14}) \\
G[G_1 \land G_2]_{O[P],P} &= (\text{by definition of }Q) \\
G[G_1]_{O[P],P} \otimes G[G_2]_{O[P],P} &= (\text{by definition of }G) \\
Q[G_1 \text{ in } P] \otimes Q[G_2 \text{ in } P] &= (\text{by definition of }Q) \\
B[G_1 \text{ in } P] \otimes B[G_2 \text{ in } P] &= (\text{by theorem 2.5.14}) \\
\end{align*}
\]

Point 3.

\[
\begin{align*}
B[G_1 \lor G_2 \text{ in } P] &= Q[G_1 \lor G_2 \text{ in } P] = (\text{by theorem 2.5.14}) \\
G[G_1 \lor G_2]_{O[P],P} &= (\text{by definition of }Q) \\
G[G_1]_{O[P],P} \oplus G[G_2]_{O[P],P} &= (\text{by definition of }G) \\
Q[G_1 \text{ in } P] \oplus Q[G_2 \text{ in } P] &= (\text{by definition of }Q) \\
B[G_1 \text{ in } P] \oplus B[G_2 \text{ in } P] &= (\text{by theorem 2.5.14}) \\
\end{align*}
\]

Point 4.

\[
\begin{align*}
B[\exists x. G \text{ in } P] &= Q[\exists x. G \text{ in } P] = (\text{by theorem 2.5.14}) \\
G[\exists x. G]_{O[P],P} &= (\text{by definition of }Q) \\
\exists x G[G]_{O[P],P} &= (\text{by definition of }G) \\
\exists G[G \text{ in } P] &= (\text{by definition of }Q) \\
\exists B[G \text{ in } P] &= (\text{by theorem 2.5.14}) \\
\end{align*}
\]
2.7 Discussion

The first outcome of this section is that we have now a powerful framework where we can reason about operational properties of fohh-programs. If we are only concerned with the input-output behavior of programs, we should observe only computed answers and finite failures. However there are tasks, such as program analysis and optimization, where we are forced to observe and take into account other features of derivations. In principle, we could be interested in the complete information of derivations. By this reason we have defined two equivalent semantics according to the intuitionistic proof procedure [43], which models all the computation process. We obtain a type of collecting semantics which gives the maximum amount of information and allows us to observe all the internal details of derivations.

The peculiarity of our framework is that we handle the denotational and operational semantics in a uniform way by using a set of primitive operators, directly related to the syntactic structure of the language and with operational features of the proof procedure. This allows us to address problems such as the relation between the top-down and bottom-up semantics, the denotation of fohh-programs and their properties of compositionality, correctness and minimality.

Point 5.

\[ B[\forall x.G \in P] = \]
\[ Q[\forall x.G \in P] = (by\ theorem\ 2.5.14) \]
\[ G[\forall x.G][lfp P] = (by\ definition\ of\ Q) \]
\[ \forall_s G[P] = (by\ definition\ of\ G) \]
\[ \forall_s Q[G \in P] = (by\ definition\ of\ Q) \]
\[ \forall_s B[G \in P] = (by\ theorem\ 2.5.14) \]

Point 6.

\[ B[D \supset G \in P] = \]
\[ Q[D \supset G \in P] = (by\ theorem\ 2.5.14) \]
\[ G[D \supset G][lfp P] = (by\ definition\ of\ Q) \]
\[ \Delta(G[P] \cup \{D\}[D \cup \{D\}]) = (by\ definition\ of\ G) \]
\[ \Delta(Q[G \in P \cup \{D\}]) = (by\ definition\ of\ Q) \]
\[ \Delta(B[G \in P \cup \{D\}]) = (by\ theorem\ 2.5.14) \]

The property expressed in point 1 is sometimes referred in program analysis as the condensing property. It essentially shows that the behavior of an atomic goal can be derived from the goal independent denotation \( O[P] \). The former theorem also asserts that we can always reconstruct a derivation of a generic goal by assembling pieces of derivations obtained from pure atomic goals.
This semantics provides also a powerful tool for dealing with a variety of applications related to the semantics of fohh-programs. In the following chapter we will see how general semantic frameworks, taking into account approximation, can be defined by using Abstract Interpretation, a theory developed to reason about the relation among different semantics, including the approximate semantics useful for static program analysis.
Chapter 3

An Abstract Interpretation Framework

3.1 The Abstraction Framework

Some semantics-based techniques (such as program analysis, debugging and transformation) require semantics which are able to effectively model computational properties. The most relevant difference between semantics is related to the property which is intended to model. Several ad hoc semantics modelling various properties have been defined. These include correct answer substitutions [19, 4], computed answer substitutions [17], partial answers [15], OR-compositional correct answers [24], call patterns [23], proof trees [32, 33]. In addition there are several semantics specifically designed for static program analysis, which can handle various observables such as types and groundness dependencies.

General semantic frameworks taking into account approximation can be defined by using Abstract Interpretation [11, 12], a theory developed to reason about the relation among different semantics, including the approximate semantics useful for static program analysis. This is the approach taken in [6], where an observable is an abstraction according to abstract interpretation theory, and in [25], where abstract interpretation is used to discuss the relations among different semantics.

We push forward the approach in [6], by defining a semantic framework whose ingredients are, as in the case of most abstract interpretation frameworks, a concrete semantics and an observable. Our concrete semantics models fohh-computations and is formalized both denotationally and operationally.

An observable is a Galois insertion between the domain of derivations and an abstract domain describing the properties to be modeled. The abstract denotational definition, transition system and goal-independent denotations are systematically derived from the concrete ones, by replacing the concrete semantic operators by their abstract optimal versions. Therefore, the definition style (denotational semantics and transition system) of the concrete semantics will be inherited by all the abstract
semantics.

The next step is the definition of classes of observables by their semantics properties. An observable belongs to a class if it satisfies a set of conditions relating the concrete semantic operators and the Galois insertion. Once we have shown that an observable belongs to a class, we know how to derive the “best” semantics and which the properties of such a semantics are. The perfect observables defined in this chapter are precise and have all the properties of the concrete semantics.

3.1.1 The Observables

We want to develop a theory that allows us to create model abstractions of denotations based on fohh-derivations inheriting all properties of its semantics. We will model the abstractions using the Abstract Interpretation Theory [11].

Informally, an observable property domain is a set of properties of derivations with an ordering relation which can be viewed as an approximation structure. An observation consists of looking at a fohh-computation (derivation), and then extracting some property (abstraction). Since any computation is equivalent to a collection, an observable is a function from $C$ to a suitable property domain, which preserves the approximation structure. Such a function must be a Galois insertion.

Let $(D, \preceq)$ be a complete lattice in the abstract domain. A function $\alpha : WFD \to D$ is a domain abstraction if there exist $\gamma$ such that $\langle \alpha, \gamma \rangle : (WFD, \sqsubseteq) \cong (D, \preceq)$ is a Galois Insertion. Given an abstract domain $D$, we are interested in the existent relation between goals and elements in $D$. An abstract collection is a function $C : Goals \to D$. By $A$ we denote the domain of abstract collections ($A$-collections), ordered by the extension $\leq$ of $\preceq$. A $A$-collections defined for pure atomic goals is denoted by $PA$. The abstract equivalence modulo variance $\equiv_A$ is defined on $A$-collections as follows: $C \equiv_A C'$ if and only if $\gamma(C) \equiv_C \gamma(C')$.

Definition 3.1.1 Let $A$ be a complete lattice of $A$-collections. A function $\alpha : C \to A$ is an observable if it maps finite elements of $C$ to finite elements of $A$ and there exist a function $\gamma$ such that

1. $\langle \alpha, \gamma \rangle : (C, \sqsubseteq) \cong (A, \leq)$ is a Galois insertion,
2. $\alpha(\mathbb{P}C) = PA$ and $\gamma(\mathbb{P}A) = \mathbb{P}C$,
3. $\forall C, C' \in \mathbb{P}C$ we have $C \equiv_C C' \implies (\alpha \circ \gamma)(C) \equiv_C (\alpha \circ \gamma)(C')$.

Definition 3.1.2 An $A$-interpretation is a pure $A$-collection modulo $\equiv_A$. We denote by $(A, \leq)$ the complete lattice $A$-interpretations with the induced quotient order.

Each observable $\alpha$ induces an observational equivalence $\approx_\alpha$ on programs which is defined in the following way

$$P_1 \approx_\alpha P_2 \iff \forall G \in Goals, \alpha(B[G in P_1]) = \alpha(B[G in P_2]).$$
Therefore, if two programs are equivalent with respect to $\alpha$ we are not able to distinguish them by looking at the abstraction of their concrete behaviors.

**Example 3.1.3** As an example of observable consider the abstraction function $\xi : \mathcal{WFD} \rightarrow \wp(\text{Subst})$ where $\wp(\text{Subst})$ is the powerset of the set of substitutions. The computed answer observable $\xi : \mathcal{C} \rightarrow \mathcal{A}_{ca}$ is defined by

\[
\xi(C) \mathrel{:=} \lambda G. \{d \mid d \in C(G), \text{ last}(d) \text{ is a final state}\}
\]

\[
\xi^\gamma(C) \mathrel{:=} \lambda G. \{d \mid \text{first}(d) = G, \text{ answer}(d) \in C(G), \text{ last}(d) \text{ is a final state}\}
\]

where $\mathcal{A}_{ca} \subseteq [\text{Goals} \rightarrow \wp(\text{Subst})]$. We can also define the abstract enhanced variance relation $\equiv_{\mathcal{A}_{ca}}$ on $\mathcal{A}_{ca}$ as mentioned before. It is easy to show that $C \equiv_{\mathcal{A}_{ca}} C'$ if and only if, for any $p(x)$, there exists $p(y)$ such that $C(p(x))$ is defined when $C'(p(y))$ is defined, and for $\theta \in C(p(x))$ there exists $\theta' \in C'(p(y))$ such that $p(x)\theta \equiv p(y)\theta'$ and vice versa. For two programs $P_1$ and $P_2$ we can see that $P_1 \equiv_{\xi} P_2$ if and only if, for any goal both programs obtain the same computer answer substitutions. Now we prove that $\langle \xi, \xi^\gamma \rangle$ is an observable.

**Proof.** By definition $\xi$ maps finite elements to finite elements, so $\xi$ and $\xi^\gamma$ satisfy points 1 and 2 of definition of observable. Then, we have to prove the last point

\[
\forall C, C' \in \mathcal{P}\mathcal{C}, C \equiv_C C' \implies (\xi \circ \xi^\gamma)(C) \equiv_C (\xi \circ \xi^\gamma)(C')
\] (3.1)

It is easy to see that the condition holds if the collections $C, C'$ are identical up to renaming of variables and constants of higher level. Hence, we need to take into consideration equivalent derivations having a different order of resolution steps.

Suppose toward a contradiction that (3.1) does not holds, i.e., for some pure collections $C, C'$ with $C \equiv_C C'$ we have at least a derivation $d_{\alpha\gamma}$ such that $d_{\alpha\gamma} \in (\xi \circ \xi^\gamma)(C)$ and $d_{\alpha\gamma} \notin (\xi \circ \xi^\gamma)(C')$. By completeness theorem (theorem 15, [43]) we know that a computed answer of a proof (successful derivation) exists regardless the goal selected at each stage in constructing the derivation. Hence, we know that exists $d'$ such that $\xi(d') = \xi(d)$ where $d' \in C'$ and $d_{\alpha\gamma} \in (\xi \circ \xi^\gamma)(d)$, i.e., $d'$ is a variant of $d \in C$. Finally, by definition of $\xi^\gamma$ we can see that $d_{\alpha\gamma} = \xi \circ \xi^\gamma(d) = \xi \circ \xi^\gamma(d')$, which contradicts the assumption. Therefore, $\langle \xi, \xi^\gamma \rangle$ is an observable. □

### 3.1.2 Abstract Semantics

Given an observable $\alpha : \mathcal{C} \rightarrow \mathcal{A}$, we can obtain new abstract semantics. We need to define the optimal abstract counterparts of all operators in the concrete semantics, and we also need to obtain the conditions under which the abstract semantics is optimal.
Let $C, C_1, C_2$ be abstract collections and $I_1, I_2$ be abstract interpretations in some abstract domain $A$. Then the optimal abstract operators are:

$$
\begin{align*}
\tilde{\Theta}(P) & := \alpha(\Theta(P)) \\
C \uparrow P & := \alpha(\gamma(C) \uparrow P) \\
C_1 \uparrow C_2 & := \alpha(\gamma(C_1) + \gamma(C_2)) \\
C_1 \bowtie C_2 & := \alpha(\gamma(C_1) \bowtie \gamma(C_2)) \\
\tilde{\nabla}(A)_{C,P} & := \alpha(\nabla(A)(\gamma), P) \\
C_1 \bowtie C_2 & := \alpha(\gamma(C_1) \bowtie \gamma(C_2)) \\
\tilde{\exists} C & := \alpha(\exists \gamma(C)) \\
\tilde{nabla} C & := \alpha(\nabla \gamma(C)) \\
\tilde{\Delta}(C)_{G,P} & := \alpha(\Delta(\gamma(C))_{G,P}) \\
C_1 \bowtie \bowtie \bowtie C_2 & := \alpha(\gamma(C_1) \bowtie \bowtie \bowtie \gamma(C_2))
\end{align*}
$$

Once we have the optimal abstract operators, we can define the corresponding abstract semantics, obtained from the concrete denotational and operational semantics by replacing the basic semantic operators by their optimal abstract versions.

The abstract denotational semantics is defined by means of the abstract semantic functions:

$$
\begin{align*}
Q_\alpha[G \in P] & := G_\alpha[G][\uparrow \downarrow P_\alpha[P], P] \\
P_\alpha[P] & := \sum_{c \in P} C_\alpha[c], P \\
C_\alpha[p(t)] & := \tilde{\Theta}(p(t)) \\
C_\alpha[G \supset p(t)] & := \tilde{\Theta}(G \supset p(t)) \bowtie \bowtie G_\alpha[G], P \\
C_\alpha[D_1 \land D_2] & := C_\alpha[D_1] \bowtie \bowtie C_\alpha[D_2], P \\
C_\alpha[\forall x.D] & := C_\alpha[D] \bowtie \bowtie \nabla \gamma(C) \\
G_\alpha[A] & := \tilde{nabla}(A), P \\
G_\alpha[G_1 \land G_2] & := G_\alpha[G_1], P \bowtie \bowtie G_\alpha[G_2], P \\
G_\alpha[G_1 \lor G_2] & := G_\alpha[G_1], P \bowtie \bowtie G_\alpha[G_2], P \\
G_\alpha[\exists x.G] & := \exists x \tilde{G}[G], P \\
G_\alpha[\forall x.G] & := \forall x \tilde{G}_\alpha[G], P \\
G_\alpha[D \supset G] & := \tilde{\Delta}(G_\alpha[G], \uparrow \downarrow C_\alpha[D], P \cup (D))_{D \supset G,P}
\end{align*}
$$

where the abstract composition operator is defined as

$$
I_1 \bowtie I_2 = (I_1 \uparrow P_2 \bowtie \bowtie I_2 \uparrow P_1) \uparrow \alpha
$$

where $P_1, P_2$ are the programs in $\text{first}(\gamma(I_1))$ and $\text{first}(\gamma(I_2))$ respectively. As in the concrete semantics, by $I \uparrow \alpha$ we intend the least fixed point of the function $\Xi_I$ where

$$
\Xi_I(l) = \begin{cases} 
    l' & \text{if } l = \bot \\
    I \bowtie \bowtie l' & \text{otherwise}
\end{cases}
$$
3.1. THE ABSTRACTION FRAMEWORK

The abstract operational semantics is based on the abstract behavior. Given a query \( G \) in \( P \) it is defined as
\[
B_\alpha[G \text{ in } P] = \alpha \left( \lambda G. \{ S_0 \rightarrow^\ast S_n \mid S_0 = \langle \{G, P, 0\}, \kappa, 0, e \rangle \} \right) / \equiv_A
\]
the abstract top-down denotation is defined as
\[
\mathcal{O}_\alpha[P] = \sum_{p(x) \in \text{Goals}} B_\alpha[p(x) \text{ in } P].
\]
or by means of the following transition system \( \mathfrak{S}_\alpha \)
\[
\frac{C \in A, \ C \neq C \triangleright \triangleleft \circ su_\alpha(\widetilde{\Theta}(P))}{C \mapsto C \triangleright \triangleleft \circ su_\alpha(\widetilde{\Theta}(P))}
\]
where
\[
su_\alpha(l) = \sum_{G \in \text{Goals}} G_\alpha[G]_l
\]
Thus, the abstract behavior of a program can be rewritten as
\[
B_\alpha[G \text{ in } P] = \sum \{ C | \alpha(Id_G) \mapsto^\ast C \}
\]
Any abstract interpretation is implicitly considered also as an arbitrary abstract collection obtained by choosing an arbitrary representative of the interpretation. It is easy to see that for any abstract interpretations \( l_1, l_2 \) such that \( l_1 \equiv_A l_2 \) we have that \( \nabla(A)_{t_1, P} = \nabla(A)_{t_2, P} \). Analogously we can see that all the semantic operators that we have introduced are independent from the choice of the representative. This is the reason why we defined the operators in terms of their counterparts defined on the abstract domain.

Example 3.1.4 3.1.3 Consider again the computed answer observable. Now we can define the semantics operators for the abstract domain.

\[
\begin{align*}
\widetilde{\Theta} &:= \lambda P. \{ [t/x] \mid p(t) \in P \} \\
C \upharpoonright P &:= C \\
C_1 \sqcap C_2 &:= \lambda G. \{ \theta \mid \theta \in C_1(G) \cup C_2(G) \} \\
C_1 \sqcup C_2 &:= \lambda G \supset p(t). \{ \theta_1 \circ \theta_2 \mid \theta_1 \in C_1(G \supset p(t)), \theta_2 \in C_2(G) \} \\
\nabla(p(t))_{t,P} &:= \lambda G \supset p(t). \{ \theta \circ [t/x] \mid \theta \in l(p(t)) \text{ and } x \in \text{vars}(\theta) \} \\
C_1 \otimes C_2 &:= \lambda G_1 \wedge G_2. \left\{ \theta_1 \circ \theta_2 \mid \theta_1 \in C_1(G_1), \theta_2 \in C_2(G_2) \text{ where } \forall x. x \in \theta_1, x \in \theta_2, \theta_1(x) = \theta_2(x) \right\} \\
\exists_x C &:= \lambda (\exists x. G). \{ \theta \circ [y/x] \mid \theta \in C(G) \} \\
\nabla_x C &:= \lambda (\forall x. G). \{ \theta \circ [x^{n+1}/x^n] \mid \theta \in C(G) \text{ where } n \text{ is the level} \} \\
\n\Delta(C)_{G,P} &:= C \\
C_1 \triangleright \triangleright \triangleleft C_2 &:= \lambda G \left\{ \theta_1 \circ (\theta_2 \circ \theta_3) \mid \theta_1 \in C_1(G), \theta_2 \in C_2(G') \text{ where } \exists G' = G \theta_3 \text{ and vars}(\theta_3) \cap \text{vars}(\theta_1) \neq \emptyset \right\} \text{ and } \forall x. x \in \theta_1, x \in \theta_2, \theta_1(x) = \theta_2(x)
\end{align*}
\]
In the definition of \( \triangleright \circ \triangleleft \) we have that \( \theta_3 \) is a non idempotent substitution on \( G \), which produces a more instantiated goal. In that way we can simulate the instantiation in the substitutions of the second abstract collection.

Note that some operators are not precise. If we work in the abstract domain, we can derive substitutions that are not computed answers of the concrete semantics. In this case we lose significant informations of the computing process. For example, suppose we have the following derivations in simplified notation

\[
\begin{align*}
C_1(p(x)) &= \xi \left( p(x) \overset{g(y)/x}{\rightarrow} q(g(y)) \overset{a/y}{\rightarrow} r(g(a)) \right) = [g(a)/x] \\
C_2(r(z)) &= \xi \left( r(z) \overset{f(z)/z}{\rightarrow} s(f(z)) \right) = [f(z)/z] \\
C_2(t(z)) &= \xi \left( t(z) \overset{h(z)/z}{\rightarrow} s(f(z)) \right) = [h(z)/z]
\end{align*}
\]

by applying the definition of \( \triangleright \circ \triangleleft \) we can obtain the following substitutions

\[
C_1 \triangleright \circ \triangleleft C_2 = [f(g(a))/x] \cup [h(g(a))/x]
\]

where the second one is the result of the fusion of two incompatible derivations.

This semantics is very related to the class of denotational observables defined in [5, 37]. We relax optimality condition of some operators by allowing approximations. Nevertheless, we can replace \( C_\alpha [.] \) by the optimal abstraction \( \tilde{C}[.] \) of \( C[.] \) to make the semantic definition precise.

\[
\tilde{C}[c] := \alpha \circ C[.] \circ \gamma
\]

With this new semantic function it is possible to define a more precise denotational semantics by redefining the program denotation function as

\[
\mathcal{P}_\alpha [P]_I := \sum_{c \in P} \tilde{C}[c]_{I,P}
\]

By following the arguments of the proof provided in [5, 37], we can show that the abstract denotational semantics and the bottom-up denotation are precise.

### 3.1.3 Perfect Observables

The ideal class of observable is the one for which the abstract denotational and the abstract operational semantics coincide. With a precise abstract semantics we can equivalently compute in a top-down and in a bottom-up way, the abstract property by mimicking the concrete computations.

Note that for any observable, the abstract sum (\( + \)) operator is the union of sets of abstract interpretations, as defined in the concrete semantics. In fact, for any Galois insertion we have,

\[
\alpha \left( \sum_{d \in I} d \right) = \alpha \left( \sum_{d \in I} \gamma(\alpha(d)) \right) \quad (3.4)
\]
Therefore we can derive

\[
\begin{align*}
\alpha(\Theta(P)) &= \alpha(\gamma(\alpha(\Theta(P)))) \\
\alpha(C \uparrow P) &= \alpha(\gamma(\alpha(C)) \uparrow P) \\
\alpha(C_1 \triangleright C_2) &= \alpha(\gamma(\alpha(C_1)) \triangleright \gamma(\alpha(C_2))) \\
\alpha(\nabla(A)_{C,P}) &= \alpha(\nabla(A)_{\gamma(\alpha(C)),P}) \\
\alpha(C_1 \otimes C_2) &= \alpha(\gamma(\alpha(C_1)) \otimes \gamma(\alpha(C_2))) \\
\alpha(C_1 \oplus C_2) &= \alpha(\gamma(\alpha(C_1)) \oplus \gamma(\alpha(C_2))) \\
\alpha(\exists C) &= \alpha(\exists \gamma(\alpha(C))) \\
\alpha(\forall C) &= \alpha(\forall \gamma(\alpha(C))) \\
\alpha(\Delta(C)_{G,P}) &= \alpha(\Delta(\gamma(\alpha(C)))_{G,P}) \\
\alpha(C_1 \circ \triangleright C_2) &= \alpha(\gamma(\alpha(C_1)) \circ \triangleright \gamma(\alpha(C_2)))
\end{align*}
\]

We want to show that this class of observable preserves the most important properties we have seen in the concrete semantics. In particular, we will show that any semantics constructed on a perfect observable is compositional and the top-down denotation is equivalent to the bottom-up one. To conclude with this results we need first to proof some necessary lemmata and theorems.

The following theorem establishes the optimality of \( \tilde{\sqcup} \) when the observable is perfect.

**Theorem 3.1.6** Let \( \alpha : C \to \mathbb{A} \) be a perfect observable and \( I_1, I_2 \in \mathbb{I} \). Then

\[
\alpha(I_1) \tilde{\sqcup} \alpha(I_2) = \alpha(I_1 \sqcup I_2)
\]

**Proof.** By definition of \( \sqcup, \tilde{\sqcup}, \uparrow \) and by straightforward induction on \( \Xi \) we can see that it is sufficient to prove that

\[
\alpha(I) \circ \triangleright \left( \alpha(I_1) \uparrow P_2 + \alpha(I_2) \uparrow P_1 \right) = \alpha(I \circ \triangleright (I_1 \uparrow P_2 + I_2 \uparrow P_1))
\]

for any \( I, I_1, I_2 \in \mathbb{I} \). First of all, we can observe that

\[
\begin{align*}
\alpha(I_1 \uparrow P) &= \\
\alpha(\alpha \gamma(I) \uparrow P) &= \text{ (by definition 3.1.5)} \\
\alpha(I) \uparrow P &= \text{ (by (3.2))}
\end{align*}
\]

Therefore we can derive

\[
\begin{align*}
\alpha(I \circ \triangleright (I_1 \uparrow P_2 + I_2 \uparrow P_1)) &= \\
\alpha(\alpha \gamma(I) \circ \triangleright (\alpha \gamma(I_1 \uparrow P_2 + I_2 \uparrow P_1))) &= \text{ (by definition 3.1.5)} \\
\alpha(I) \circ \triangleright \alpha(I_1 \uparrow P_2 + I_2 \uparrow P_1) &= \text{ (by (3.2))} \\
\alpha(I) \circ \triangleright (\alpha(I_1 \uparrow P_2) + \alpha(I_2 \uparrow P_1)) &= \text{ (by (3.4))} \\
\alpha(I) \circ \triangleright \left( \alpha(I_1 \uparrow P_2) + \alpha(I_2 \uparrow P_1) \right) &= \text{ (by (3.2))} \\
\alpha(I) \circ \triangleright \left( \alpha(I_1) \uparrow P_2 + \alpha(I_2) \uparrow P_1 \right) &= \text{ (by (3.5))}
\end{align*}
\]
The same result can be shown with respect to a goal denotation. We can equivalently calculate in the abstract domain using abstract operators or calculate in the concrete domain with concrete operators and interpretation and then apply the abstraction function.

**Theorem 3.1.7** Let \( \alpha : C \to A \) be a perfect observable. Then

\[
\forall I \in \mathbb{I}, G \in \text{Goals. } \alpha(\mathcal{G}[G]_I) = \mathcal{G}_\alpha[G]_{\alpha(I)}
\]

**Proof.** Now we will prove by induction on the structure of \( G \).

\( G := p(t) : \)

\[
\alpha(\mathcal{G}[p(t)]_I) = \alpha(\nabla(p(t))_I) \quad \text{by definition of } \mathcal{G}[.] \\
= \alpha(\nabla(p(t))_{\alpha(I)}) \quad \text{by definition 3.1.5} \\
= \nabla(p(t))_{\alpha(I)} \quad \text{by (3.2)} \\
= \mathcal{G}_\alpha[p(t)]_{\alpha(I)} \quad \text{by definition of } \mathcal{G}_\alpha[.]
\]

\( G := G_1 \land G_2 : \)

\[
\alpha(\mathcal{G}[G_1 \land G_2]_I) = \alpha(\mathcal{G}[G_1]_I \otimes \mathcal{G}[G_2]_I) \quad \text{by definition of } \mathcal{G}[.] \\
= \alpha(\alpha(\mathcal{G}[G_1]_I) \otimes \mathcal{G}[G_2]_I) \quad \text{by definition 3.1.5} \\
= \alpha(G_1)_I \otimes \alpha(G_2)_I \quad \text{by (3.2)} \\
= \mathcal{G}_\alpha[G_1]_{\alpha(I)} \otimes \mathcal{G}_\alpha[G_2]_{\alpha(I)} \quad \text{by assumption} \\
= \mathcal{G}_\alpha[G_1 \land G_2]_{\alpha(I)} \quad \text{by definition of } \mathcal{G}_\alpha[.]
\]

\( G := G_1 \lor G_2 : \)

\[
\alpha(\mathcal{G}[G_1 \lor G_2]_I) = \alpha(\mathcal{G}[G_1]_I \oplus \mathcal{G}[G_2]_I) \quad \text{by definition of } \mathcal{G}[.] \\
= \alpha(\alpha(\mathcal{G}[G_1]_I) \oplus \mathcal{G}[G_2]_I) \quad \text{by definition 3.1.5} \\
= \alpha(G_1)_I \oplus \alpha(G_2)_I \quad \text{by (3.2)} \\
= \mathcal{G}_\alpha[G_1]_{\alpha(I)} \oplus \mathcal{G}_\alpha[G_2]_{\alpha(I)} \quad \text{by assumption} \\
= \mathcal{G}_\alpha[G_1 \lor G_2]_{\alpha(I)} \quad \text{by definition of } \mathcal{G}_\alpha[.]
\]

\( G := \forall x. G' : \)

\[
\alpha(\mathcal{G}[\forall x. G']_I) = \alpha(\forall x \mathcal{G}[G']_I) \quad \text{by definition of } \mathcal{G}[.] \\
= \alpha(\forall x \alpha(\mathcal{G}[G']_I)) \quad \text{by definition 3.1.5} \\
= \forall x \alpha(\mathcal{G}[G']_I) \quad \text{by (3.2)} \\
= \mathcal{G}_\alpha[G']_{\alpha(I)} \quad \text{by assumption} \\
= \mathcal{G}_\alpha[\forall x. G']_{\alpha(I)} \quad \text{by definition of } \mathcal{G}_\alpha[.]
\]
3.1. THE ABSTRACTION FRAMEWORK

\[G := \exists x. G':\]

\[\alpha(G[\exists x. G']) = \alpha(\exists x \alpha G[G']) \quad \text{by definition of } G[.].\]

\[\alpha(\exists x \alpha G[G']) = \exists x \alpha(G[G']) \quad \text{by definition 3.1.5}\]

\[\exists x \alpha G[G'] = \exists x G' \quad \text{by (3.2)}\]

\[\exists x G' = \exists x G' \quad \text{by assumption}\]

\[G[\exists x. G'] = \exists x G' \quad \text{by definition of } G[.]\]

\[G := D \sqcup G':\]

\[\alpha(G[D \sqcup G']) = \alpha(\Delta \alpha(G[G'] \sqcup D \sqcup C[D]_\bot)) \quad \text{by definition of } G[.].\]

\[\alpha(\Delta \alpha(G[G'] \sqcup D \sqcup C[D]_\bot)) = \Delta \alpha(G[G'] \sqcup D \sqcup C[D]_\bot) \quad \text{by definition 3.1.5}\]

\[\Delta \alpha(G[G'] \sqcup D \sqcup C[D]_\bot) = \Delta (G_\alpha[G'] \sqcup D \sqcup C[D]_\bot) \quad \text{by (3.2)}\]

\[\Delta (G_\alpha[G'] \sqcup D \sqcup C[D]_\bot) = \Delta (G_\alpha[G'] \sqcup D \sqcup C[D]_\bot) \quad \text{by assumption}\]

\[\Delta (G_\alpha[G'] \sqcup D \sqcup C[D]_\bot) = \Delta (G_\alpha[G'] \sqcup D \sqcup C[D]_\bot) \quad \text{by theorem 3.1.6}\]

\[\Delta (G_\alpha[G'] \sqcup D \sqcup C[D]_\bot) = \Delta (G_\alpha[G'] \sqcup D \sqcup C[D]_\bot) \quad \text{by assumption and (3.2)}\]

\[\Delta (G_\alpha[G'] \sqcup D \sqcup C[D]_\bot) = G_\alpha[D \sqcup G'] \quad \text{(by definition of } G_\alpha[.].)\]

\[G_\alpha[D \sqcup G'] = G_\alpha[D \sqcup G'] \quad \text{(by definition of } G_\alpha[.].)\]

---

The former theorem has an immediate consequence. In fact, we can extend the result to the denotational semantics of a program.

**Corollary 3.1.8** Let \(\alpha : \mathbb{C} \to \mathbb{A}\) be a perfect observable. Then

\[\forall I \in \mathbb{I}, \alpha(P[I]) = P_\alpha[P]_{\alpha(I)}\]

**Proof.** Straightforward from definition of \(P[.].\) and theorem 3.1.7.

\[\alpha(P[I]) = \alpha(\sum C[c[I]) \quad \text{by definition of } P[.].\]

\[\alpha(\sum C[c[I)) = \alpha(\sum C[c[I)) \quad \text{by definition 3.1.5}\]

\[\sum C[c[I]) = \sum C[c[I)) \quad \text{by (3.2)}\]

\[\sum C[c[I)) = \sum C[c[I)) \quad \text{(from theorem 3.1.7)}\]

\[\sum C[c[I)) = \sum C[c[I)) \quad \text{(by definition of } P_\alpha[.].)\]

---

The following corollary also follows immediately from the above theorem. Since the abstract operational semantics is defined by means of a transition system based on the sequential unfolder function, just like in the concrete operational semantics, the following corollary will help us to prove the equivalence between the abstract operational semantics and the top-down denotation.
Corollary 3.1.9 Let \( \alpha : \mathcal{C} \rightarrow \mathcal{A} \) be a perfect observable. Then
\[
\alpha(su(\Theta(P))) = su_\alpha(\tilde{\Theta}(P))
\]

Proof. By definition of \( su(\cdot) \) we obtain
\[
\alpha(su(\Theta(P))) = \alpha(\sum_{G \in Goals} G[[C]]_{\Theta(P)}) = \alpha(\omega(\sum_{G \in Goals} G[[C]]_{\Theta(P)})) \text{ by definition 3.1.5}
\]
\[
= \sum_{G \in Goals} \alpha(G[[C]]_{\Theta(P)}) \text{ by definition 3.2}
\]
\[
= \sum_{G \in Goals} \alpha(G[[C]]_{\alpha(\Theta(P))}) \text{ by theorem 3.1.7}
\]
\[
= su_\alpha(\tilde{\Theta}(P)) \text{ by definition of } su_\alpha(\cdot)
\]

Now we can show that the abstract transition system \( \mathfrak{S}_\alpha \) is precise for perfect observables.

Theorem 3.1.10 Let \( \alpha : \mathcal{C} \rightarrow \mathcal{A} \) be a perfect observable. Then
\[
\forall C, C' \in \mathcal{C}. C \mathbin{\rightarrow^*} C' \Rightarrow \alpha(C) \mathbin{\overset{\alpha}{\rightarrow}^*} \alpha(C')
\]
moreover
\[
\forall C' \in \mathcal{A}. \alpha(C) \mathbin{\overset{\alpha}{\rightarrow}^*} C' \Rightarrow \exists C', C = \alpha(C'), C \mathbin{\rightarrow^*} C'
\]

Proof. We prove the first part of the theorem by induction on the length of \( \mathfrak{S} \)-derivations.

\( n = 1 \): For the base case we need to prove
\[
C \mathbin{\rightarrow} C' \Rightarrow \alpha(C) \mathbin{\overset{\alpha}{\rightarrow}} \alpha(C')
\]

By definition of \( \mathfrak{S} \) we have that
\[
C \mathbin{\rightarrow} C' \Rightarrow C' = C \circ \circ \leftrightarrow su(\Theta(P))
\]

Thus,
\[
\alpha(C') = \alpha(C \circ \circ \leftrightarrow su(\Theta(P))) = \alpha(\alpha(\gamma(C) \circ \circ \leftrightarrow \alpha(\gamma(su(\Theta(P)))))) \text{ by definition 3.1.5}
\]
\[
= \alpha(C) \circ \circ \leftrightarrow \alpha(su(\Theta(P))) \text{ by (3.2)}
\]
\[
= \alpha(C) \circ \circ \leftrightarrow \alpha(su_\alpha(\tilde{\Theta}(P))) \text{ by corollary 3.1.9}
\]

Therefore, \( C \mathbin{\rightarrow} C' \Rightarrow \alpha(C) \mathbin{\overset{\alpha}{\rightarrow}} \alpha(C') \).
THE ABSTRACTION FRAMEWORK

3.1. Now we need to prove

\( C \mapsto^n C' \mapsto C'' \Rightarrow \alpha(C) \mapsto^{\alpha} n \alpha(C') \mapsto^{\alpha} \alpha(C'') \)

given that

\( C \mapsto^n C' \Rightarrow \alpha(C) \mapsto^{\alpha} n \alpha(C') \)

which can be immediately proved as shown in the base case. In fact, is sufficient
to show that

\( C' \mapsto C'' \Rightarrow \alpha(C') \mapsto^{\alpha} \alpha(C'') \)

The second part of the theorem is analogous and hence it is omitted. ■

In correspondence with the above theorem we can state now the following theo-
rem that establishes the precision of the abstract semantics for a perfect observable,
i.e., is not important what domain we use to calculate the behavior or the semantics
of a program, we will obtain always the same result.

**Theorem 3.1.11** Let \( \alpha \) be a perfect observable. Then the abstract operational se-

- \( \alpha(B[G in P]) = B_\alpha[G in P] \)
- \( \alpha(O[P]) = O_\alpha[P] \)

**Proof.** The points will be proved separately

**Point 1:**

\[
\begin{align*}
\alpha(B[G in P]) &= \alpha(\sum \{ C | Id \mapsto^{*} C \}) \\
&= \alpha(\sum \{ \alpha\gamma(C) | Id \mapsto^{*} C \}) \\
&= \alpha(\sum \{ \gamma(C) | \alpha(Id_G) \mapsto^{*} C \}) \\
&= B_\alpha[G in P] \\
\end{align*}
\]

(by definition of \( B[.]. \))

(by definition 3.1.5)

(by theorem 3.1.10)

(by (3.2) and def. of \( B_\alpha[.]. \))

**Point 2:** Follows immediately from previous point and from definition of abstract
equivalence \( \equiv_\Lambda \).

■

**Theorem 3.1.12** Let \( \alpha \) be a perfect observable. Then the abstract denotational
semantics and the bottom-up denotation are precise and equivalent:

1. \( \alpha(\text{lfp } P[P]) = \text{lfp } P_\alpha[P] \)
2. \( \alpha(\text{Q}[G in P]) = Q_\alpha[G in P] \)
3. \( \text{lfp } P_\alpha[P] = O_\alpha[P] \)
4. $Q_\alpha[G \ in \ P] = B_\alpha[G \ in \ P]$

**Proof.** We prove the points separately

**Point 1:** First of all we need to prove that $P_\alpha[\_]$ is continuous and have a fixed point. Let $I_{\text{ch}}$ be the chain $I_1 \subseteq \ldots \subseteq I_n \subseteq A$. Since $\sum$ is a lub operation on $A$ it is sufficient to prove that

$$\sum_{l \in I_{\text{ch}}} P_\alpha[P]_l = P_\alpha[P]_{I_{\text{ch}}}$$

$$\alpha \left( \sum_{l \in I_{\text{ch}}} P_\alpha[P]_l \right) = \left( \text{by definition of } \sum \right)$$

$$\alpha \left( \sum_{l \in I_{\text{ch}}} \gamma(P_\alpha[P]_l) \right) = \left( \text{by definition of } \sum \right)$$

$$\alpha \left( \sum_{l \in I_{\text{ch}}} \gamma(P_\alpha[P]_l) \right) = \left( \text{by (3.4)} \right)$$

$$\alpha \left( \sum_{l \in I_{\text{ch}}} \gamma(P_\alpha[P]_l) \right) = \left( \text{by corollary 3.1.8 and def. of } \sum \right)$$

**Point 2:** By definition of $Q[\_], Q_\alpha[\_]$ and by theorem 3.1.7 we have that

$$\alpha(G[G]_{tp\ P[P]}) =$$

$$G[G]_{\alpha(tp\ P[P])} =$$

$$G[G]_{tp\ P_\alpha[P]} =$$

$$Q_\alpha[G \ in \ P]$$

**Point 3:** Follows immediately from point 1 and theorem 3.1.11.

**Point 4:** Follows immediately from point 2 and theorem 3.1.11.

At this point we can show that all the properties of the concrete semantics also hold for the abstract top-down denotation for any perfect observable.

**Theorem 3.1.13** Let $\alpha$ be a perfect observable, $A$ be an atom, $D$ be a program clause and $G, G_1, G_2$ be goals. Then

1. $B_\alpha[p(t) \ in \ P] = \wedge (p(t))_{\alpha[P], P}$
2. $B_\alpha[G_1 \land G_2 \ in \ P] = B_\alpha[G_1 \ in \ P] \odot B_\alpha[G_2 \ in \ P]$
3. $B_\alpha[G_1 \lor G_2 \ in \ P] = B_\alpha[G_1 \ in \ P] \oplus B_\alpha[G_2 \ in \ P]$
4. $B_\alpha[\exists x.G \ in \ P] = \exists_x B_\alpha[G \ in \ P]$
5. $B_\alpha[\forall x. G \text{ in } P] = \bar{\forall}_x B_\alpha[G \text{ in } P]$

6. $B_\alpha[D \supset G \text{ in } P] = \Delta (B_\alpha[G \text{ in } P \cup \{D\}])_{D\supset G,P}$.

Proof. This points can be proved separately by using the theorem 3.1.12.

Point 1.

$B_\alpha[p(t) \text{ in } P] = \quad (\text{by theorem 3.1.12})$

$Q_\alpha[p(t) \text{ in } P] = \quad (\text{by definition of } Q_\alpha)$

$G_\alpha[p(t)]_{\text{ifp } P_\alpha[P], P} = \quad (\text{by definition of } G_\alpha)$

$Q_\alpha[p(t)]_P \circ G_\alpha[p(t)]_P = \quad (\text{by theorem 3.1.12})$

$G_\alpha[p(t)]_P = \quad (\text{by definition of } G_\alpha)$

Point 2.

$B_\alpha[G_1 \land G_2 \text{ in } P] = \quad (\text{by theorem 3.1.12})$

$Q_\alpha[G_1 \land G_2 \text{ in } P] = \quad (\text{by definition of } Q_\alpha)$

$G_\alpha[G_1 \land G_2]_{\text{ifp } P_\alpha[P], P} = \quad (\text{by definition of } G_\alpha)$

$Q_\alpha[G_1 \in P] \circ Q_\alpha[G_2 \in P] = \quad (\text{by definition of } Q_\alpha)$

$B_\alpha[G_1 \in P] \circ B_\alpha[G_2 \in P] = \quad (\text{by theorem 3.1.12})$

Point 3.

$B_\alpha[G_1 \lor G_2 \text{ in } P] = \quad (\text{by theorem 3.1.12})$

$Q_\alpha[G_1 \lor G_2 \text{ in } P] = \quad (\text{by definition of } Q_\alpha)$

$G_\alpha[G_1 \lor G_2]_{\text{ifp } P_\alpha[P], P} = \quad (\text{by definition of } G_\alpha)$

$Q_\alpha[G_1 \in P] \circ Q_\alpha[G_2 \in P] = \quad (\text{by definition of } Q_\alpha)$

$B_\alpha[G_1 \in P] \circ B_\alpha[G_2 \in P] = \quad (\text{by theorem 3.1.12})$

Point 4.

$B_\alpha[\exists x. G \text{ in } P] = \quad (\text{by theorem 3.1.12})$

$Q_\alpha[\exists x. G \text{ in } P] = \quad (\text{by definition of } Q_\alpha)$

$G_\alpha[\exists x. G]_{\text{ifp } P_\alpha[P], P} = \quad (\text{by definition of } G_\alpha)$

$Q_\alpha[G \in P] = \quad (\text{by definition of } Q_\alpha)$

$B_\alpha[G \in P] = \quad (\text{by theorem 3.1.12})$

Point 5.

$B_\alpha[\forall x. G \text{ in } P] = \quad (\text{by theorem 3.1.12})$

$Q_\alpha[\forall x. G \text{ in } P] = \quad (\text{by definition of } Q_\alpha)$

$G_\alpha[\forall x. G]_{\text{ifp } P_\alpha[P], P} = \quad (\text{by definition of } G_\alpha)$

$Q_\alpha[G \in P] = \quad (\text{by definition of } Q_\alpha)$

$B_\alpha[G \in P] = \quad (\text{by theorem 3.1.12})$
Point 6.

\[ B_\alpha[D \supset G \text{ in } P] = \]
\[ Q_\alpha[D \supset G \text{ in } P] = \quad (\text{by theorem 3.1.12}) \]
\[ G_\alpha[D \supset G]_\text{llf } \alpha[P],P = \quad (\text{by definition of } Q_\alpha) \]
\[ \Delta(G_\alpha[G]_\text{llf } \alpha[P],P)_{D \supset G,P} = \quad (\text{by definition of } G_\alpha) \]
\[ \Delta(Q_\alpha[G \text{ in } P \cup \{D\}])_{D \supset G,P} = \quad (\text{by corollary 3.1.12}) \]
\[ \Delta(B_\alpha[G \text{ in } P \cup \{D\}])_{D \supset G,P} = \quad (\text{by theorem 3.1.12}) \]

As a final remark for the class of perfect observables we show the following theorem which establishes the observational equivalence of programs in terms of their abstract semantics.

**Theorem 3.1.14** Let \( \alpha : C \rightarrow A \) be a perfect observable and \( P, P' \) be programs, then

\[ P \approx_\alpha P' \iff O_\alpha[P] = O_\alpha[P'] \]

**Proof.** By (3.4) and theorem 3.1.11, we know that

\[ P \approx_\alpha P' \iff \forall G \in \text{Goals}. \alpha(B[G \text{ in } P]) = \alpha(B[G \text{ in } P']) \]
\[ \iff \forall G \in \text{Goals}. B_\alpha[G \text{ in } P] = B_\alpha[G \text{ in } P'] \]

This proof is very similar to the one presented in the theorem 2.6.1. The minimality can be trivially deduced by definition of \( O_\alpha[.] \). The converse case is proved by contradiction assuming that exists a goal \( G \) such that \( B_\alpha[G \text{ in } P] \neq B_\alpha[G \text{ in } P'] \).

**The \textit{depth}(k) Observable Example**

In contrast with the perfect observables, the \textit{depth}(k) observable is not precise. With this abstraction we lose the necessary information to rebuild the concrete semantics. We give up precision to achieve effectiveness in the construction of the abstract semantics. This type observable is useful to model some of the properties for static program analysis. With this example we show how to approximate an infinite set of computed answers by means of a \textit{depth}(k) cut [47], i.e. by cutting terms which have a depth greater than \( k \). Terms are simplified by replacing each subterm at depth \( k \) with a new fresh variable.

Let \( t \mid_k \) be the reduction of term \( t \) by cutting at depth \( k \). The operator \( \mid_k \) can be extended to substitutions to obtain abstract substitutions as

\[ \theta \mid_k = \{ [t/x] \mid_k | [t/x] \in \theta \} \]
3.1. THE ABSTRACTION FRAMEWORK

We assume that for any binding in \( \theta \) any cut is performed by using distinct variables. We denote by \( \text{Subst} \downarrow_k \) the set of all substitutions at depth \( k \).

Let \( A_{\downarrow_k} \subseteq [\text{Goals} \rightarrow \wp(\text{Subst} \downarrow_k)] \). As in the example of computed answer we can obtain the depth \( k \) observable \( \kappa \) by further abstraction of \( \xi \). Namely \( \mu_k : A_{ca} \rightarrow A_{\downarrow_k} \) is defined as

\[
\mu_k(C) := \lambda G. \{ \theta \downarrow_k \mid \theta \in C(G) \}
\]

Therefore, the depth \( k \) computed answer observable \( \kappa \) is

\[
\kappa(C) := \mu_k(\xi(C)) \\
\kappa^\gamma(C) := \lambda G. \{ d \mid \text{first}(d) = G, \text{answer}(d) \downarrow_k \in C(G), \text{last}(d) \text{ is a final state} \}
\]

**Proof.** It is easy to show that \( \kappa \) is an observable. As showed in the proof for the computed answer observable, \( \kappa \) also maps finite elements to finite elements, so \( \kappa \) and \( \kappa^\gamma \) satisfy points 1 and 2 of definition of observable. Then, we have to prove the last point

\[
\forall C, C' \in \mathbb{PC}, C \equiv_C C' \implies (\kappa \circ \kappa^\gamma)(C) \equiv_C (\kappa \circ \kappa^\gamma)(C') \quad (3.6)
\]

It is easy to see that the condition holds if the collections \( C, C' \) are identical up to renaming of variables and constants of higher level. Hence, we need to take into consideration equivalent derivations having a different order of resolution steps.

Suppose toward a contradiction that (3.6) does not holds, i.e., for some pure collections \( C, C' \) with \( C \equiv_C C' \) we have at least a derivation \( d_{\alpha\gamma} \) such that \( d_{\alpha\gamma} \in (\kappa \circ \kappa^\gamma)(C) \) and \( d_{\alpha\gamma} \notin (\kappa \circ \kappa^\gamma)(C') \). By completeness theorem (theorem 15, [43]) we know that a computed answer of a proof (successful derivation) exists regardless the goal selected at each stage in constructing the derivation. Hence, we know that exists \( d' \) such that \( \kappa(d') = \kappa(d) \) where \( d' \in C' \) and \( d_{\alpha\gamma} \in (\kappa \circ \kappa^\gamma)(d) \), i.e., \( d' \) is a variant of \( d \in C \). Finally, by definition of \( \kappa^\gamma \) we can see that \( d_{\alpha\gamma} = \kappa \circ \kappa^\gamma(d) = \kappa \circ \kappa^\gamma(d') \), which contradicts the assumption. \( \blacksquare \)

Now we can construct the abstract semantics by defining the following abstract operators.
It is clear that the above operators are not optimal. We intentionally relaxed the optimality condition to guarantee computation effectiveness on this abstraction by using only simple abstract operations. However, in this case we ensure a minimal correctness for the semantics as showed for the computed answer observable.

\[ \mathcal{O}_\alpha[P] \equiv \alpha(\mathcal{O}[P]) \]

In consequence we will obtain an approximated abstract semantics.
Chapter 4

Higher-Order Hereditary Harrop Formulas

Higher-order programming is a desired feature of modern programming languages. There are several implementations of functional, logic and procedural language paradigms, that allow procedures and sentences to be encapsulated into data structures in such a way that they can subsequently be retrieved and used to complete computations. So, operationally, the ability of programs to reason on, calculate and execute procedures is commonly referred as higher-order programming. In logic programming early implementations of Prolog developed higher-order mechanisms by means of some pseudo-logical predicates for terms and predicate construction and search. However, the use of ad hoc mechanisms, in contraposition with the declarative nature of pure logic programs leads to a loss of the expressiveness and the notion of what it really means with respect to the fundamentals of a logic system.

The λProlog language is a successful realization of the higher-order logic programming language paradigm. This language is based on the theory of the hereditary Harrop formulas. The hohh-formulas provides abstraction and scoping capabilities at the level of predicate logic to match those present at the level of terms. Since hohh-formulas are an extension to fohh-formulas, now it is possible to have occurrences of implication and universal quantifiers in goals, which leads to a mechanism for controlling the availability of names and clauses that appear in a program. As a difference with the first-order version, we have that the hohh-formulas are typed. Typing is necessary to ensure consistency of the core logic.

4.1 The Language

The typed λ-calculus is the language underlying the formal system commonly used for hohh-formulas. This language is basically composed by types and terms. The types categorize the terms based on a functional hierarchy. The types that are employed in hohh-formulas are often referred as simple types. The simple typed
\(\lambda\)-terms are constructions made of typed versions of \(\lambda\)-abstraction and application.

The simple types are built by sorts and type constructors. Let \(\Sigma\) and \(\Pi\) be the set of sorts and type constructors respectively, such that \(\exists o, s \in \Sigma\) where \(o\) is the type of truth values and \(s\) is the type of individuals. The constructors \(c \in \Pi\) map a positive number (the arity). Now we can define the type expressions that are formed from a set of base types. If \(\sigma, \sigma_1, \ldots, \sigma_n\) and \(\tau\) are types then so are

1. \(c (\sigma_1, \ldots, \sigma_n)\), for every \(c \in \Pi\) of arity \(n\)
2. \((\sigma \rightarrow \tau)\)

The type (2) is the functional type. The non-functional types are referred as atomic types. Alternatively, we can use a simpler notation for functional types called Church's notation, where \((\tau \sigma)\) denotes \((\sigma \rightarrow \tau)\). \(\sigma\) defines the domain of a functional type and \(\tau\) defines its range. By default it is assumed that \(\rightarrow\) associates to the right. Therefore, more complex types like \((\sigma_1 \rightarrow (\ldots \rightarrow (\sigma_n \rightarrow \tau)\ldots))\) will be equivalent to \((\sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \tau)\) or \((\tau \sigma_1 \ldots \sigma_n)\).

The terms are obtained by the operation of abstraction and application from a set of constants and variables. Each variable or constant is associated with a type. The term is defined as follows.

1. A variable or constant of type \(\sigma\) is a term of type \(\sigma\)
2. If \(t_1\) is a term of type \(\sigma \rightarrow \tau\) and \(t_2\) is a term of type \(\sigma\), then \((t_1 t_2)\) is a term of type \(\tau\).
3. If \(x\) is a variable of type \(\sigma\) and \(t\) is a term of type \(\tau\) then \((\lambda x.t)\) is a term of type \(\sigma \rightarrow \tau\).

The variable associated to the \(\lambda\)-constructor is called bound variable and its scope is limited to the function related to the abstraction. An unbound variable is said to be a free variable. To simplify the notation and depending on the context we will also use the syntax \(t : \sigma\) to specify that \(t\) is of type \(\sigma\).

Atomic terms whose top level symbol is not a free variable are called rigid atoms. Those whose top level symbol is a free variable are called flexible atoms.

This language allows us to describe functions of simple types. By means of abstraction we can specify the definition of a function, while the application provides a mean for representing the evaluation of such functions. However, to represent the language of \(hohh\)-formulas, the presence of logical connectives is needed for describing the relation between functions. For this reason are introduced new constants:

1. logical constants: \(\top : o\) and \(\lor, \land, \supset : o \rightarrow o \rightarrow o\).
2. non logical constants: \(\forall, \exists : (\sigma \rightarrow o) \rightarrow o\), for any type \(\sigma\).
4.1. THE LANGUAGE

The type \( o \) is interpreted as the type of propositions, \( \top \) corresponds to the tautology proposition, and the prepositional connectives \( \lor, \land, \supset, \forall \) and \( \exists \) corresponds to the usual logic connectives seen in the \( fohh \)-formulas.

Finally we need to do some simplifications of notation to give our language a more clear and logical seemly look. In particular, the expressions \( \forall (\lambda x. t) \) and \( \exists (\lambda y. t) \) will be written respectively as \( \forall x. t \) and \( \exists y. t \). The expression \( \lor t_1 t_2 \) will be written as \( t_1 \lor t_2 \). In the same way will be written the other binary relations \( \land \) and \( \supset \).

Besides this conventions we will sometimes write \( \forall x_1, \ldots, \forall x_n. t \) as \( \forall x_1, \ldots, x_n. t \) and \( \exists x_1, \ldots, \exists x_n. t \) as \( \exists x_1, \ldots, x_n. t \).

4.1.1 Equivalence between terms

Prior to starting with implementation details like unification of \( fohh \)-formulas, we need to introduce the concept of equality (equivalence) between \( \lambda \)-terms. This concept is based in some rules of rewriting usually called \( \lambda \)-conversion. To define the equivalence we need to define the operation of substitution of free variables in \( \lambda \)-terms.

Definition 4.1.1 The \( \lambda \)-substitution \([t/x] \) is defined as

1. \( x[t/x] = t \) and \( t_1[t_2/x] = t_1 \), where \( t_1 \) is a constant or a variable different from \( x \).
2. \( \lambda y. t_1[t_2/x] = \lambda y. (t_1[t_2/x]) \) and \( \lambda x. t_1[t_2/x] = \lambda x. t_1 \), provided that \( x, y \neq t_1 \).
3. \( (t_1 t_2)[t_3/x] = t_1[t_3/x] t_2[t_3/x] \).

In a different manner from the first order case, the substitution in \( \lambda \)-terms preserves the abstractions which can be instantiated only by means of the application of \( \lambda \)-conversion rules. These rules are required to avoid the bounding of free variables when substitutions are elaborated. The rules will guarantee that the operation of substitution is logically correct. We will refer to such rules as \( \alpha \), \( \beta \) and \( \eta \) conversions.

1. \( \alpha \)-conversion: \( \lambda x. t = \lambda y. t[y/x] \), provided \( y \) is bound in \( t \) and substitutable for \( x \).
2. \( \beta \)-conversion: \( (\lambda x. t_1)t_2 = t_1[t_2/x] \), provided \( t_1 \) is free for \( x \) in \( t_2 \).
3. \( \eta \)-conversion: \( \lambda x. (t x) = t \), provided \( x \) is bound in \( t \).

The rules of conversion can be applied in both directions. When the transformation simplifies the term we say that the conversion is a reduction (\( \alpha \), \( \beta \) and \( \eta \) reductions). The converse operation is called expansion.

It is easy to see that the above rules defines a class of equivalence for \( \lambda \)-terms.

Definition 4.1.2 (\( \lambda \)-conversion) We say that \( t_1 \lambda \)-converts to \( t_2 \) \( (t_1 \equiv_\lambda t_2) \) if there is a sequence of \( \alpha \), \( \beta \), \( \eta \) conversions that transform \( t_1 \) in \( t_2 \) or vice versa.
4.1.2 Normalization

In order to identify the equality between terms, it is very useful to restrict the analysis to the representative elements of the domain by writing the terms in a canonical form.

Definition 4.1.3 A term is said to be in normal $\lambda$-normal form if it does not contain subterms of the form:

1. $(\lambda x.t_1)t_2$.
2. $\lambda x.(t x)$ where $x$ not occurs free in $t$.

As shown in [1],[2, 29] every simply typed $\lambda$-term has a unique $\lambda$-normal form up to $\alpha$-conversions. The normal form can be calculated by substituting subterms of the form $(\lambda x.t_1)t_2$ by $t_1[t_2/x]$ and subterm of the form $\lambda x.(t : x)$ by $t$ when $x$ is not free in $t$. Therefore when a term is $\lambda$-normalized it has the form

$$\lambda x_1, \ldots, x_n(t s_1 \ldots s_m)$$

where $t$ is a variable or a constant of type $\sigma_1 \rightarrow \cdots \rightarrow \sigma_m \rightarrow \tau$ with $\tau$ being an atomic type and each $s_i$ is in the same form. So, extending this idea to the entire language, we have that $\top, t_1 \land t_2, t_1 \lor t_2, t_1 \supset t_2, \forall x.t$ and $\exists x.t$ are in normal form if $t, t_1, t_2$ are formulas in normal form.

4.1.3 Unification

The interpreter for a higher order language must deal with the process of unification. The main difference with respect to the first-order logic is that now we have to consider the unification of $\lambda$-terms which equality is ruled by the $\lambda$-conversion.

The unification in higher order logic can be stated as a generalization of the first order case. Let $T = \{ \langle t_i, s_i \rangle \mid 1 \leq i \leq n \}$ be a set of pairs of $\lambda$-terms. A unifier for the set is a substitution $\theta$ such that $t_i \theta = s_i \theta$ for $1 \leq i \leq n$. This problem has been studied and it is well known that for an arbitrary $T$ it is undecidable [26, 30, 36]. Also, it has been shown that most general unifiers do not always exist for an unifiable $T$.

There is, nevertheless, a method that can be used to find unifiers whenever they exist. A systematic search can be made for unifiers of a given disagreement set. This procedure is described in [31] and consists in two iterative algorithms $\text{SIMP}$ and $\text{MATCH}$ that are repeatedly applied until the unifier is calculated. The first one takes the set of term pairs $(t, s)$ and transforms it into a set with no pairs where $t$ is rigid, or a distinguished symbol $\perp$ that denotes ”no unification” is obtained. The objective of $\text{MATCH}$ is to give a set of substitutions that could be part of some unifier.
The basis for the \textit{SIMP} function is provided by the intuitive idea that two rigid terms of the same type which, are normalized, have an unifier if and only if they have the same functional constructor (head of the term) and each subterm of one term unifies with the corresponding subterm in the other term. This idea is formalized and proved in [31].

Given any term \( t \) and any substitution \( \theta \), it is clear that \( t \theta = t' \theta \), where \( t' \) is the normal form of \( t \). Therefore, the question of unifying two terms can be reduced to unifying their \( \lambda \)-normal forms. If \( t_1 \) and \( t_2 \) are terms of the same type, their \( \lambda \)-normal forms must have binders of the same length. So, we can arrange those binders with a series of \( \alpha \)-transformations to make both terms identical. If they are rigid terms, we can determine whether they have an unifier or we can reduce it to the problem of finding unifiers for the arguments of \( t_1 \) and \( t_2 \). In concrete, this is the mechanism implemented by \textit{SIMP}.

\textbf{Definition 4.1.4} The function \textit{SIMP} on disagreement set \( T \) is defined as follows:

1. If \( T = \emptyset \) then \( \text{SIMP}(T) = \emptyset \).
2. If \( T = \langle t_1, t_2 \rangle \) then
   
   (a) If \( t_1 \) is flexible then \( \text{SIMP}(T) = T \).
   
   (b) If \( t_2 \) is flexible then \( \text{SIMP}(T) = \{ \langle t_2, t_1 \rangle \} \).
   
   (c) If \( t_1, t_2 \) are rigid terms such that \( t_1' = \lambda x. (c \ s_1 \ldots s_n) \) and \( t_2' = \lambda x. (c \ r_1 \ldots r_n) \) where \( t_1', t_2' \) are the \( \lambda \)-normal form of \( t_1, t_2 \) then

   \[
   \text{SIMP}(T) = \text{SIMP}(\{ \langle s_i, r_i \rangle \mid 1 \leq i \leq n \}) .
   \]

   (d) Otherwise \( \text{SIMP}(T) = F \)

3. If \( T = \{ \langle t_i, s_i \rangle \mid 1 < i \leq n \} \)

   (a) If \( \exists i. \text{SIMP}(\langle t_i, s_i \rangle) = F \) then \( \text{SIMP}(T) = F \)

   (b) Otherwise \( \text{SIMP}(T) = \bigcup_i \text{SIMP}(\langle t_i, s_i \rangle) \)

It is clear that \textit{SIMP} transforms a given disagreement set into either the no-unification marker \( F \) or a disagreement set consisting of flexible-flexible or flexible-rigid pairs. In [31] is proved that \textit{SIMP} finishes in a finite number of steps.

If the result of \textit{SIMP} is either empty or has only flexible-flexible pairs, at least one unifier can be easily provided for the set [40]. On the other hand if the set has at least one flexible-rigid pair, then we must consider a substitution for the head of the flexible term. This is the purpose of the \textit{MATCH} function. The general idea of \textit{MATCH} resides in the application of two processes: the imitation and the projection. Given a flexible-rigid pair

\[
\langle \lambda x_1 \ldots x_n(f \ t_1 \ldots t_p), \lambda y_1 \ldots y_m(c \ s_1 \ldots s_q) \rangle
\]
where \( f \) is a variable and \( c \) is a constant, the ‘imitation’ attempts to make a substitution of the form

\[
[\lambda w_1 \ldots w_p(c \ (h_1 \ w_1 \ldots w_p) \ldots (h_q \ w_1 \ldots w_p))/f],
\]

where \( h_1 \ldots h_q \) are new fresh variables.

The ‘projection’ tries the substitution (projection) of the form

\[
[\lambda w_1 \ldots w_p(w_i \ (h_1 \ w_1 \ldots w_p) \ldots (h_j \ w_1 \ldots w_p))/f],
\]

where \( 1 \leq i \leq p \), \( h_1 \ldots h_j \) are new variables and \( \tau_1 \rightarrow \ldots \rightarrow \tau_j \rightarrow \tau \) is the type of \( w_i \). There are, therefore, a set of substitutions, each of which may be investigated separately as a component of a complete unifier.

**Definition 4.1.5** Let \( V \) a set of variables and let \( t'_1 = \lambda x.(f \ s_1 \ldots s_n) \) and \( t'_2 = \lambda x.(c \ r_1 \ldots r_m) \) be the \( \lambda \)-normal form of a flexible term \( t_1 \) and a rigid term \( t_2 \) respectively, and the type of \( f \) is \( \sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \tau \) where \( \tau \) is atomic. The function MATCH is defined as follows:

1. \( MATCH(t_1, t_2, V) = IMIT(t_1, t_2, V) \cup (\bigcup_{i \leq n} PROJ_i(t_1, t_2, V)) \).
2. If \( c \) is a bound variable then
   \[
   IMIT(t_1, t_2, V) = \emptyset,
   \]
   otherwise
   \[
   IMIT(t_1, t_2, V) = \{\langle f, \lambda w_1 \ldots w_n(c \ (h_1 \ w_1 \ldots w_n) \ldots (h_m \ w_1 \ldots w_n))\rangle\}
   \]
   where \( h_i \) are variables of appropriate types not contained in \( V \cup \{w_1 : \sigma_1 \ldots w_n : \sigma_n\} \).
3. If \( \exists \sigma_i \) such that its type is not of the form \( \tau_1 \rightarrow \ldots \rightarrow \tau_j \rightarrow \tau \) then
   \[
   PROJ_i(t_1, t_2, V) = \emptyset,
   \]
   otherwise
   \[
   PROJ_i(t_1, t_2, V) = \{\langle f, \lambda w_1 \ldots w_n(w_i \ (h_1 \ w_1 \ldots w_n) \ldots (h_j \ w_1 \ldots w_n))\rangle\}
   \]
   where \( h_i \) are variables of appropriate types not contained in \( V \cup \{w_1 : \sigma_1 \ldots w_n : \sigma_n\} \).

In [31] is presented a proof of the correctness of the MATCH function, i.e., it is showed that MATCH finds a substitution which is part of the unifier of the disagreement set.

The unification procedure may now be described based on the iterative application of both functions. As we will see in the following section, the SIMP and MATCH functions will be amalgamated into the process of derivation which is the base of the interpreter for hohh-programs.
4.1.4 The hohh-Formulas

The program clauses and goal formulas in the higher-order case are close similar to the first-order versions. The principal difference is the presence of types and $\lambda$-terms to ensure the consistency of the underlying logic and the special term $\top$ that represents the tautologous proposition.

In general the $\lambda$-terms can contain arbitrary quantifiers and connectives. However, to construct a programming language implementable by means of goal directed provability, we need to restrict the set of $\lambda$-normal terms in our language. The first restriction is the elimination of implication within the terms. The second restriction is the elimination of all negative atomic formulas. Hence, the only logical constants a term can contain are $\top$, $\lor$, $\land$, $\forall$ and $\exists$. The last restriction is the limitation of atomic formulas to be rigid terms.

**Definition 4.1.6 (hohh-formulas)** Let $A$ be an atomic formula and $A_r$ a rigid atomic formula then hohh-formulas are defined as

$$
G ::= \top \mid A \mid G \land G \mid G \lor G \mid \exists x. G \mid D \supset G \mid \forall x. G
$$

$$
D ::= A_r \mid G \supset A_r \mid D \land D \mid \forall x. D
$$

where the $G$-formulas are the goals and the $D$-formulas are the definite clauses.

As in the fohh case a program $P$ is a finite collection of definite clauses. The extended program or elaboration of a program is defined as $elab(P) = \{elab(D)|D \in P\}$ where

- If $D ::= A$ or $D ::= G \supset A$, then $elab(D) = \{D\}$.
- If $D ::= D_1 \land D_2$ then $elab(D) = elab(D_1) \cup elab(D_2)$.
- If $D ::= \forall x. D'$, then $elab(D) = \{\forall x. D'' | D'' \in elab(D')\}$

In advance, we will implicitly refer to a hohh-program as its elaboration. The reason for the introduction of the elaboration is only a technical one: to simplify the definitions and proofs of the interpreter.

4.2 The Proof Procedure

As in the first-order context, the idea of programming can be thought of as asking if a proof exists for a goal formula given a hohh-program. Despite the introduction of $\lambda$-terms and the differences in the language, almost all aspects discussed for the first-order case are relevant for the higher-order too. An important aspect to focus is the presence of quantifications over first-order variables in the new context. So, we need to modify the labelled unification mechanism and the proving algorithm as well, to consider the extension of the labelling to apply to constants and variable of
function types. Another problem to analyze is the possible presence of flexible atomic formulas in the set of goals. As shown in [41] the resolution of such goal formulas can be delayed till no other goal remains to be solved. Then, the remaining goals are solved by substituting universal relations for the predicate variable at the head of the flexible atom.

The presence of \( \lambda \)-terms requires a slightly different implementation of higher-order unification problem, in order to deal with the presence of goals of the form \( \forall x. G \). For this reason we use the \textit{labelled unification} [43] for \( \lambda \)-terms which adds an additional constraint to the \textit{MATCH} algorithm [31, 40]. In fact, the ‘imitation’ substitution will be possible if the label of the constant head of the term is less than the label of the head variable. If this condition is satisfied, the substitution may take effect, but the labels associated to the new created function variables must be identical to the label of the original variable function. In that way, later instantiations of such variables do not violate the constraint on substitution terms.

\textbf{Definition 4.2.1} Let \( \theta = \{ [t_i/x_i] | 1 \leq i \leq n \} \) be a \( \lambda \)-substitution and let \( L \) be labelling function on the variables and constants in \( \theta \). The \( \lambda \)-substitution \( \theta \) is said to be a proper labelled \( \lambda \)-substitution with respect to \( L \), if and only if, \( L(c) \leq L(x_i) \) for every constant \( c \) appearing in \( t_i \). The induced labelling is the labelling \( L' \) of the substitution \( \theta \) which is obtained from \( L \) in the following manner:

\[
L'(x) = \min \left( \{ L(x) \} \cup \{ L(x_i) | [t_i/x_i] \in \theta \text{ and } x \text{ appears in } t_i \} \right)
\]

Note that we also consider the constants and variables in head of \( \lambda \)-terms.

\textbf{Definition 4.2.2} Let \( D = \{ (t_i, s_i) | 1 \leq i \leq n \} \) be a set of pairs of \( \lambda \)-terms (disagreement set) and let \( L \) be a labelling function defined over the variables and constants of \( D \). A labelled \( \lambda \)-unifier (\( \lambda \)-unifier) for \( D \) under \( L \) is a substitution \( \theta \) which is proper with respect to \( L \) and for each \((t_i, s_i) \in D\) the equality \( t_i \theta \equiv_\lambda s_i \theta \) holds.

\textbf{Definition 4.2.3} A \( \lambda \)-state \( S \) is a tuple \( \langle W, D, k, L, \theta \rangle \) where

- \( W \) is a set of tuples \( \langle G, P, n \rangle \) called decorated goals where \( G \) is a \( \lambda \)-term of type \( o \), \( P \) is a program and \( n \) is a natural number,
- \( D \) is a disagreement set,
- \( k \) is a clause used to reduce a goal (\( \kappa \) denotes the void clause, i.e., no reduction),
- \( L \) is a labelling function on \( \text{vars}(W) \cup \text{consts}(W) \), and
- \( \theta \) is a \( \lambda \)-substitution (\( \epsilon \) denotes the empty substitution).

In advance we will refer to decorated goals as “goals”. In case of ambiguity we will clarify whether we are talking about a goal formula or a decorated goal.
Definition 4.2.4 (higher-order proof procedure) The state

\[ S_1 = (W_1, D_1, k_1, L_1, \theta_1) \]

derives in one step to the state \( S_2 = (W_2, D_2, k_2, L_2, \theta_2) \) using the following rules:

**top** If \( \exists w_1 = (\top, P, n) \in W_1 \) then \( W_2 = (W_1 - \{ w_1 \}), D_2 = D_1, L_2 = L_1, k_2 = \kappa \) and \( \theta_2 = \epsilon \).

**and** If \( \exists w_1 = (G_1 \land G_2, P, n) \in W_1 \) then \( W_2 = (W_1 - \{ w_1 \}) \cup \{ (G_1, P, n), (G_2, P, n) \} \), \( D_2 = D_1, L_2 = L_1, k_2 = \kappa \) and \( \theta_2 = \epsilon \).

**or** If \( \exists w_1 = (G_1 \lor G_2, P, n) \in W_1 \) then \( W_2 = (W_1 - \{ w_1 \}) \cup \{ (G_1, P, n) \} \) for \( i \in \{ 1, 2 \} \), \( D_2 = D_1, L_2 = L_1, k_2 = \kappa \) and \( \theta_2 = \epsilon \).

**augment** If \( \exists w_1 = (D \supset G, P, n) \in W_1 \) then \( W_2 = (W_1 - \{ w_1 \}) \cup \{ (G, P \cup \{ D \}, n) \} \), \( D_2 = D_1, L_2 = L_1, k_2 = \kappa \) and \( \theta_2 = \epsilon \).

**instance** If \( \exists w_1 = (\exists x, G, P, n) \in W_1 \) then \( W_2 = (W_1 - \{ w_1 \}) \cup \{ (G[y/x], P, n) \} \) for \( y \notin \text{vars}(W_1) \), \( D_2 = D_1, L_2 = L_1 \cup \{ (y, n) \}, k_2 = \kappa \) and \( \theta_2 = \epsilon \).

**generic** If \( \exists w_1 = (\forall x, G, P, n) \in W_1 \) then \( W_2 = (W_1 - \{ w_1 \}) \cup \{ (G[c/x], P, n + 1) \} \) for \( c \notin \text{consts}(W_1) \), \( D_2 = D_1, L_2 = L_1 \cup \{ (c, n + 1) \}, k_2 = \kappa \) and \( \theta_2 = \epsilon \).

**unification** If \( D_1 \) is not solved and for some flexible-rigid pair \( \langle t, s \rangle \in D_1 \), either \( \text{MATCH}(t, s, \text{vars}(L_1)) = \epsilon \) and \( D_2 = \emptyset \), or there is a \( \varphi \in \text{MATCH}(t, s, \text{vars}(L_1)) \) such that \( \theta_2 = \varphi \), \( W_2 = W_1 \theta_2 \), \( D_2 = \text{SIMP}(D_1 \theta_2) \), \( k_2 = \kappa \) and \( L_2 \) is the induced labelling of \( \varphi \).

**identity** If \( \exists w_1 = (A_r, P, n) \in W_1 \) and \( \exists k = \forall x_1 \ldots \forall x_m. A' \in \text{elab}(P) \) such that \( L_{|x_1 \ldots x_m} = \emptyset \). Then \( L_2 = L_1 \cup \{ (x_i, n) \mid 1 \leq i \leq m \}, W_2 = \{ w \mid w \in W_1 - \{ w_1 \} \} \), \( P, n, \theta_2 = \epsilon \), \( k_2 = k \) and \( D_2 = \text{SIMP}(D_1 \cup \{ (A_r, A') \}) \).

**backchain** If \( \exists w_1 = (A_r, P, n) \in W_1 \) and \( \exists k = \forall x_1 \ldots \forall x_m. (G' \supset A') \in \text{elab}(P) \) such that \( L_{|x_1 \ldots x_m} = \emptyset \). Then \( L_2 = L_1 \cup \{ (x_i, n) \mid 1 \leq i \leq m \}, W_2 = \{ w \mid w \in W_1 - \{ w_1 \} \} \cup \{ (G', P, n) \}, \theta_2 = \epsilon \), \( k_2 = k \) and \( D_2 = \text{SIMP}(D_1 \cup \{ (A_r, A') \}) \).

The derivation step between states \( S_1 \) and \( S_2 \) is denoted by \( S_1 \rightarrow S_2 \).

Definition 4.2.5 The possible infinite sequence of states \( S_0, \ldots, S_n \) is a derivation sequence \( (S_0 \rightarrow^* S_n) \) if \( S_i \rightarrow S_{i+1}, i \geq 0 \). The initial state of a computation \( G \) in \( P \) is defined as

\[ S_0 = \langle \{ (G, P, 0) \}, \emptyset, \kappa, L_0, \epsilon \rangle \]

where \( L_0 = \emptyset \). A state is said to be proper if for each \( w = (G, P, n) \in W \) is the case that \( L_{|w} \leq n \). The sequence is a successful derivation if \( S_0 \) is an initial state and \( S_n = \langle \emptyset, \emptyset, k_n, L_n, \theta_n \rangle \) for some \( L_n, k_n, \) and \( \theta_n \). Moreover, we say that a non-successful derivation is a finite failure if it can not be extended by applying a derivation step to its last state.
Given a derivation \( d = \langle W_0, D_0, k_0, L_0, \theta_0 \rangle, \ldots, \langle W_n, D_n, k_n, L_n, \theta_n \rangle \) we denote by \( \text{first}(d) \) and \( \text{last}(d) \) the set of all goals occurring in the first and the last derivation state respectively. By \( \text{clauses}(d) \) we denote the set of all program clauses selected to reduce goals at each state of a derivation \( d \), and by \( \text{prefix}(d) \) we denote the set of all derivations which are prefixes of \( d \).

No rigorous proofs of soundness and completeness appears to have been constructed for this particular procedure. In [40] and [43] an outline of a proof is presented. This outline is based on the proof of soundness and completeness developed for the higher-order Horn clauses. The authors maintain that this proof could amalgamate the arguments in [40] and [43], relative to the fragment of higher-order Horn clauses and \( \text{fohh} \)-formulas, showing that all possible substitutions are considered respect the necessary constrains. In [40] is provided a proof of soundness and completeness for a proof procedure for logic programs based on higher-order Horn clauses, i.e., an algorithm without a labeling function for unification and program augmentation. Actually, this derivation procedure (with modifications) is at the base of our higher-order proof procedure. We do not focus in this aspect of the semantics because it is out of the scope of this work. We accept as a good proof all theoretical and experimental results obtained in this area.

Once we have defined the new proof procedure and the related domain of derivations, we can translate to the higher-order most of the concepts and operators defined in earlier chapters. Since our semantics is based on lowlevel operators and taking into account implementation details, we will able to successfully redefine our semantics, only by redefining such operators in accordance with the new context.

### 4.3 Denotational and Operational Semantics

Now we use the derivation concept introduced in the previous section to formalize the domains of the semantics definitions. In advance, we will use for the higher-order the same notation we have used in the first-order context for defining and denoting the semantic domains. So, we have that our semantics is defined over collections and interpretations that are partial functions from goals to well-formed sets of derivations modulo renaming of variables and new constants.

**Definition 4.3.1** We say that two derivations \( d_1 = S_1 \rightarrow^* S_n \) and \( d_2 = S'_1 \rightarrow^* S'_n \) are equivalent if they are identical modulo renaming of variables and constants with label greater than 0. In that case we say that \( d_1 \) is a variant of \( d_2 \) (\( d_1 \equiv_c d_2 \)).

A set of derivations \( S \) is said to be well formed if and only if for any \( d \in S \) we have \( \text{prefix}(d) \subseteq S \). We denote by \( \text{WFD} \) the complete lattice of well-formed set of derivations ordered by \( \subseteq \).

**Definition 4.3.2** A collection is a partial function \( C : \text{Goals} \rightarrow \text{WFD} \), such that if \( C(G) \) is defined, then it is a well-formed set of derivations all starting from a goal \( G \).
\( \mathbb{C} \) is the domain of all collections ordered by \( \sqsubseteq \), where \( C \sqsubseteq C' \) iff \( \forall G. C(G) \subseteq C'(G) \) and if \( C(G) \) is defined then \( C(G) \) is defined too. A pure collection is a collection defined only for pure atomic goals and its domain is denoted by \( \mathbb{P}\mathbb{C} \).

We introduce the relation of equivalence modulo variance \( \equiv_{\mathbb{C}} \) defined on collections. Namely, \( C \equiv_{\mathbb{C}} C' \) if and only if for any \( G \) there exists a variant \( G' \) of \( G \) such that, if \( C(G) \) is defined, then \( C'(G') \) is defined, and for any \( d \in C(G) \) there exists \( d' \in C'(G') \) such that \( d \equiv_{\mathbb{C}} d' \).

**Definition 4.3.3** An interpretation \( I \) is a pure collection modulo variance. The set of all interpretations is denoted by \( \mathbb{I} \). The pair \( (\mathbb{I}, \sqsubseteq) \) is a complete lattice with the induced quotient order.

In other words, an interpretation associates derivations only to pure atomic goals.

**Definition 4.3.4** Let \( G \) be a goal and \( P \) a program. The identity collection of a goal will be a collection that associates to \( G \) the initial state \( \langle \{\{G, P, 0\}\}, \kappa, 0, \epsilon \rangle \) and is denoted as \( \text{Id}_G \).

To describe the semantics, we need to define some concepts and operations on derivations that are in the base of all operators over collections. We do not repeat here such definitions because basically they remain unchanged with respect to the first-order case.

**Definition 4.3.5** Let \( d = S_1, ..., S_n \) be a derivation, let \( W' \) be a set of goals and let \( D' \) be a disagreement set. The insertion of \( \langle W', D' \rangle \) in \( d \) \( (d \triangleleft \langle W', D' \rangle) \) is defined if each state \( S'_i = \langle W_i \cup (W'\sigma_i), D_i \cup (D'\sigma_i), k_i, L_i \cup L'_i, \theta_i \rangle \) is a proper state, where \( S_i = \langle W_i, D_i, k_i, L_i, \theta_i \rangle \in d \) and \( L'_i \) is the labelling resulting when applying \( \sigma_i \) to \( W' \) and \( D' \), with \( \sigma_i = \theta_1...\theta_i \). In this case \( d \triangleleft \langle W', D' \rangle = S'_1, ..., S'_n \).

The insertion of a pair \( \langle W', D' \rangle \) in a derivation means that the goal and disagreement set are added to each state, and it remains unsolved at the end of the derivation, but it is modified by all the substitutions executed in the preceding states.

The introduction of disagreement sets and in particular the introduction of a new transition (the unification), dramatically changes the concept of equivalence between derivations. There exists now a strong dependence between states of unification that could be derived from the resolution of many goals. In this case the order in which the goals are selected influences the final computation. By this reason the equivalence classes of derivation are restricted only to renaming of variables and constants of the same level. The operator of fusion must take into account this peculiarity of higher-order derivations. Now is not sufficient to merge the derivation in an arbitrary “legal” order, but all possible combinations should be tested, to produce a set of derivations that simulates an exhaustive search.
Definition 4.3.6 Let $d_1 = S_1^1, ..., S_n^1$ and $d_2 = S_1^2, ..., S_n^2$ be derivations. The fusion of $d_1$ and $d_2$ ($d_1 \ast d_2$) is defined if \( \text{vars}(d_1) \cap \text{vars}(d_2) \subseteq \text{first}(d_1) \cup \text{first}(d_2) \) and if the set of derivations $D = \{d \mid d = s_0, ..., s_{i+1}\}$ is not empty, where $s_0 = s_0^1 < s_0^2$ and $s_i = s_i^1 < s_i^2$ with $k_i > k_i-1$ and $j \in [1, 2]$. In addition, for all defined derivations we have $\forall x. x \in \text{vars}(d_1) \cap \text{vars}(d_2)$ is the case that $\sigma_1(x) = \sigma_2(x)$, where $\sigma_1$ and $\sigma_2$ are the computed substitution of $d_1$ and $d_2$ respectively. In this case the fusion is $D$.

The resulting derivations must be equivalent to the derivation starting with the conjunction of the goals in $\text{first}(d_1)$ and $\text{first}(d_2)$. Let see an example. Suppose we have the derivations

$$d_1 = a_1 \rightarrow b_1 \rightarrow c_1$$
$$d_2 = a_2 \rightarrow b_2 \rightarrow c_2$$

then the fusion $d_1 \ast d_2$ will attempt the derivations

$$a_1 \triangleleft a_2 \rightarrow b_1 \triangleleft a_2 \rightarrow c_1 \triangleleft a_2 \rightarrow b_2 \triangleleft a_1 \rightarrow c_1 \triangleleft a_2 \rightarrow b_2 \rightarrow c_2$$
$$a_1 \triangleleft a_2 \rightarrow b_2 \triangleleft a_1 \rightarrow c_2 \triangleleft a_1 \rightarrow b_1 \rightarrow c_2$$
$$a_1 \triangleleft a_2 \rightarrow b_1 \triangleleft a_2 \rightarrow b_1 \triangleleft b_2 \rightarrow c_1 \triangleleft b_2 \rightarrow c_2 \triangleleft c_1$$
$$a_1 \triangleleft a_2 \rightarrow b_1 \triangleleft a_2 \rightarrow b_2 \triangleleft b_1 \rightarrow c_1 \triangleleft b_1 \rightarrow c_2$$

Notice that in the first state by definition of $\triangleleft$ we have that $a_1 \triangleleft a_2 = a_2 \triangleleft a_1$.

In order to deal with the new proof procedure we need to modify the operation of substitution in derivations.

Definition 4.3.7 Let $d = S_1, ..., S_n$ be a derivation and $\theta$ an idempotent substitution such that $\text{vars}(\text{first}(d)\theta) \cap \text{vars}(\text{clauses}(d)) = \emptyset$ and let $L'$ be a labelling function for all variables in $\text{range}(\theta)$. Then the application of $\theta$ to $d$, $d\theta = S_1', ..., S_n'$ is defined if for each state $S_i = \langle W_i, D_i, k_i, L_i, \theta_i \rangle \in d$, we have that $S_i' = \langle W_i', D_i', k_i', L_i', \theta_i' \rangle$, where one of the following condition holds

- if $k_i = \kappa$ and $\theta_i \neq \epsilon$ then $D_{i-1} = \text{SIMP}(A \theta, A')$ for some $S_k$ where $k_{i-j} = \forall x_1...\forall x_n. A'$ and $\langle A, P, n \rangle = W_{i-j-1} - W_j$. $D_{i-1}$ is a not solved and for some flexible-rigid pair $\langle t, s \rangle \in D_{i-1}$ either $\text{MATCH}(t, s, \text{vars}(L_{i-1})) = \epsilon$ and $D'_i = F$, or there is a $\varphi \in \text{MATCH}(t, s, \text{vars}(L_{i-1}))$ such that $\theta'_i = \varphi$, $W_i = W_{i-1}\theta'_i$, $D'_i = \text{SIMP}(D_{i-1}\theta'_i)$, $k'_i = \kappa$ and $L'_i$ is the induced labelling of $\varphi$.

- if $k_i \neq \kappa$ then $D'_i = \text{SIMP}(A \theta, A')$ relative to $L_i \cup L'$ where $k_i = \forall x_1...\forall x_n. (G' \supset A')$ and $\langle A, P, n \rangle = W_{i-1} - W_i$. In that case $W'_i = \{W_{i-1}\theta'_i - (A, P, n) \theta \} \cup \langle G', P, n \rangle \theta'_i$ and $L'_i$ is the new induced labelling function for all variables in $\text{range}(\theta) \cup \text{Dom}(L_1) - \text{Dom}(\theta)$ and $\theta'_i = \epsilon$. 
• if \( k_i \neq \kappa \) then \( D'_i = SIMP(A\theta, A') \) relative to \( L_i \cup L' \) where \( k_i = \forall x_1 \ldots \forall x_n. A' \) and \( \langle A, P, n \rangle = W_{i-1} - W_i \). In that case \( W'_i = \{W'_{i-1}\theta'_i - \langle A, P, n \rangle \theta \} \cup \langle G', P, n \rangle \theta_i' \) and \( L'_i \) is the new induced labelling function for all variables in range(\( \theta \)) \( \cup (\text{Dom}(L_1) - \text{Dom}(\theta)) \) and \( \theta'_i = \epsilon \).

• if \( k_i = \kappa \) then \( S'_i \) is a proper state, where \( W'_i = W_i\theta, D'_i = D_i, L'_i \) is the new induced labelling function for all variables in range(\( \theta \)) \( \cup (\text{Dom}(L_1) - \text{Dom}(\theta)) \) and \( \theta'_i = \epsilon \).

Note that the substitution applied to a derivation attempts to reconstruct the derivation, starting from a new goal (more instantiated or just renamed) using the same clauses, until the achievement of a failure or a substitution that yields a non proper state. In any other case the substitution has success and the result is a derivation sequence.

In this new context we need to adapt the derivation step operator to take into account the delayed unification in the higher order proof procedure. In that sense the operator \( \Theta \) will be redefined as follows.

**Definition 4.3.8** Given a clause \( G \supset p(t) \) and a program \( P \), the operator \( \Theta \) creates the interpretation

\[
I = \{ \langle p(x), \{ [S_0], [S_0, S_1], [S_0, S_1, S_2] \} \rangle \}
\]

where

\[
S_0 = \langle \{ \langle p(x), P, 0 \rangle \}, \emptyset, \kappa, L_0, \epsilon \rangle
\]
\[
S_1 = \langle \{ \langle G, P, 0 \rangle \}, SIMP(p(x), p(t)\rho), G \supset p(t)\rho, L_1, \epsilon \rangle
\]
\[
S_2 = \langle \{ \langle G, P, 0 \rangle \}, SIMP(SIMP(p(x), p(t)\rho)\theta, \kappa, L_1, \theta) \rangle
\]

and for a clause \( p(t) \),

\[
S_0 = \langle \{ \langle p(x), P, 0 \rangle \}, \emptyset, \kappa, L_0, \epsilon \rangle
\]
\[
S_1 = \langle \emptyset, SIMP(p(x), p(t)\rho), p(t)\rho, L_1, \epsilon \rangle
\]
\[
S_2 = \langle \emptyset, SIMP(SIMP(p(x), p(t)\rho)\theta, \kappa, L_1, \theta) \rangle
\]

We also have, \( L_0 = \mathcal{L}(P) \cup \{ \langle x, 0 \rangle \} \), \( L' = L_0 \cup \{ \langle y, 0 \rangle \mid y \in \text{range}(\rho) \} \) where \( \rho \) is a renaming over \( x \) and \( L_1 \) is the labelling induced by

\[
\theta = MATCH(SIMP(p(x), p(t)\rho), \text{vars}(L_1))
\]

from \( L' \).

In this definition of the \( \Theta \) operator we introduce a difference with respect to the first order version. Now we deal with derivations in which the backchain step could be separated from unification because the presence of flexible atomic goals forces the delayed resolution of such goals. By this reason, \( \Theta \) produces an unification step in addition to the backchain step. The fixed-point operator of our semantics will use
this operator to extend the interpretations by adding pieces of derivations starting with backchain states or unification states to produce derivations in which delayed unifications will be present.

Once we have defined the higher-order version of the semantic operators we can describe our denotational semantics for the \textit{hohh}-formulas.

\[
\begin{align*}
Q[G \text{ in } P] & := G[\text{lfp } P], P \\
P[P] & := \sum_{c \in P} C[c], P \\
C[p(t)] & := \Theta (p(t)) \\
C[G \supset p(t)] & := \Theta (G \supset p(t)) \Join G[I], P \\
C[D_1 \land D_2] & := C[D_1], P + C[D_2], P \\
C[\forall x. D] & := C[D], P \\
C[p(t)] & := \nabla (p(t)), P \\
G[G \supset p(t)] & := \Theta (G \supset p(t)) \Join G[I], P \\
G[D \supset G] & := \Delta (G[I], P \supset C[D], P \cup \{D\}) \supset D \supset G, P
\end{align*}
\]

Notice that the denotational interpretation of a program now is defined as \(\Phi(\text{lfp } P)\) because \(P\) produces partial derivations that are not complete derivations of programs.

And analogously we can define the top-down denotation based on the resolution principle for \textit{hohh}-programs as

\[
\begin{align*}
\mathcal{O}[P] & = \sum_{p(x) \in \text{Goals}} \mathcal{B}[p(x) \text{ in } P] \\
\end{align*}
\]

where \(\mathcal{B}[p(x) \text{ in } P]\) is the behavior of \(p(x)\) in a higher-order program and

\[
\mathcal{B}[G \text{ in } P] = \lambda G. \{S_0 \rightarrow^* S_n \mid S_0 \text{ is the initial state } \langle\{G, P, 0\}, 0, \kappa, 0, \epsilon\rangle/\equiv\}
\]

By using the lowlevel semantic operations, we can define a new transition system \(\mathcal{S}\). Rather than operate over derivations, this transition system derives a collection into another collection by adding an execution step to the derivations. The transition system is composed by one rule:

\[
\begin{align*}
C \in \mathcal{C}, C \neq C \supset \circ \Join \text{su}(\Theta(P)) \\
\end{align*}
\]

where

\[
\text{su}(I) = \sum_{G \in \text{Goals}} G[I]
\]
Notice that in this case Θ(\( P \)) is an interpretation where all derivations contain the steps of identity, backchain and unification, for any clause \( p(t), G \supset p(t) \in P \). The function \( su(I) \) will attempt the instantiation of all pure derivations in \( I \) with every goal \( G \). Hence, \( su(\Theta(P)) \) now will be a proper collection of the form

\[
\begin{align*}
\{\{G_1 \wedge G_2, P, 0\}, 0, \kappa, 0, \epsilon\} & \rightarrow \{\{G_1, P, 0\}, \{G_2, P, 0\}, 0, \kappa, 0, \epsilon\} \cdots \\
\{\{G_1 \vee G_2, P, 0\}, 0, \kappa, 0, \epsilon\} & \rightarrow \{\{G_i, P, 0\}, 0, \kappa, 0, \epsilon\}, i \in \{1, 2\} \cdots \\
\{\{G \supset D, P, 0\}, 0, \kappa, 0, \epsilon\} & \rightarrow \{\{G, P \cup \{D\}, 0\}, 0, \kappa, 0, \epsilon\} \cdots \\
\{\{\exists x.G, P, 0\}, 0, \kappa, 0, \epsilon\} & \rightarrow \{\{G[y/x], P, 0\}, 0, \kappa, 0, \epsilon\} \cdots \\
\{\{\forall x.G, P, 0\}, 0, \kappa, 0, \epsilon\} & \rightarrow \{\{G[c/x], P, 0\}, 0, \kappa, 0 \cup \{c, 1\}, \epsilon\} \cdots \\
\{\{p(x), P, 0\}, 0, \kappa, 0, \epsilon\} & \rightarrow \{\{G, P, 0\}, \text{SIMP}(p(x), p(t)\rho), G \supset p(t)\rho, 0, \epsilon\} \cdots \\
\{\{\text{SIMP}(p(x), p(t)\rho), 0, \kappa, 0, \epsilon\} \rightarrow \{\{G, P, 0\}, \text{SIMP}(p(x), p(t)\rho)\theta, \kappa, 0, \theta\} \cdots \\
\{\{p(x), P, 0\}, 0, \kappa, 0, \epsilon\} & \rightarrow \{0, \text{SIMP}(p(x), p(t)\rho)\theta, \kappa, 0, \theta\} \cdots
\end{align*}
\]

Note that \( \Theta(P) \) will be the interpretation of the form

\[
\begin{align*}
\{\{p(x), P, 0\}, 0, \kappa, 0, \epsilon\} & \rightarrow \{\{G, P, 0\}, \text{SIMP}(p(x), p(t)\rho), G \supset p(t)\rho, 0, \epsilon\} \\
\{\{p(x), P, 0\}, 0, \kappa, 0, \epsilon\} & \rightarrow \{\{G, P, 0\}, \text{SIMP}(p(x), p(t)\rho)\theta, \kappa, 0, \theta\} \\
\{\{p(x), P, 0\}, 0, \kappa, 0, \epsilon\} & \rightarrow \{0, \text{SIMP}(p(x), p(t)\rho)\theta, \kappa, 0, \theta\}
\end{align*}
\]

The most important results we have obtained for the first-order version can be also adapted to the higher-order version. All of the proofs developed in the previous chapters can be repeated without substantial variations. In principle, this semantics is a generalization of the first-order case. Namely, in a program with no higher-order variables this semantics produces, as result of intermediate operations of fusion, many different derivations that are equivalent in terms of fohh-semantics.

**Theorem 4.3.9** The behavior of a goal in a program can be defined as the transitive closure of \( \rightsquigarrow \).

\[
\mathcal{B}[G \text{ in } P] = \sum \{C \mid Id_G \rightsquigarrow^* C\}
\]

**Theorem 4.3.10** Let \( \mathcal{O}[P] \) be the operational semantics of a program \( P \), then the following equality holds

\[
\mathcal{O}[P] = \text{Ifp } \mathcal{T}[P]
\]

where \( \mathcal{T}[P] \) is a function defined as

\[
\mathcal{T}[P] = \lambda I. \begin{cases} 
\Theta(P) & \text{if } I = \bot \\
I \circ \circ su(\Theta(P)) & \text{otherwise}
\end{cases}
\]

Finally, We can state the theorem that asserts the equivalence relation between the bottom-up denotation and the top-down denotation.
Theorem 4.3.11 Let $P$ be a hohh-program, then

$$O[P] = \text{lfp } P[P],$$

moreover

$$B[G \text{ in } P] = Q[G \text{ in } P].$$

4.4 Semantic Properties

The program denotation $O[P]$ has several interesting properties. These can all be viewed as compositionality properties, and are based on the semantics operators defined in section 2.4. The first result is a theorem that shows the compositionality of the semantic function $B$ with respect to procedure calls and the different syntactic operators of the language.

It can be easily shown that the denotation $O$ is correct and minimal with respect to the equivalence between programs ($\approx$).

Theorem 4.4.1 Let $P_1$ and $P_2$ be fohh-programs, then $P_1$ and $P_2$ are equivalent if and only if its operational semantics are equivalent, i.e.,

$$P_1 \approx P_2 \iff O[P_1] = O[P_2]$$

Another interesting property can be easily verified when fixed-point interpretations are involved. In fact,

$$(O[P_1] \uparrow P_2 + O[P_2] \uparrow P_1) \uparrow = (O[P_1] \uparrow P_2 \triangleright \circ \triangleleft O[P_2] \uparrow P_1) \uparrow + (O[P_2] \uparrow P_1 \triangleright \circ \triangleleft O[P_1] \uparrow P_2) \uparrow$$

Since $O[P_1] \uparrow P_2 \triangleright \circ \triangleleft O[P_1] \uparrow P_2 = O[P_1] \uparrow P_2$ and $O[P_2] \uparrow P_1 \triangleright \circ \triangleleft O[P_2] \uparrow P_1 = O[P_2] \uparrow P_1$.

Finally we can state the theorem which establishes the OR-composition between program denotations

Theorem 4.4.2 Let $P_1$ and $P_2$ be programs. Then

$$O[P_1 \cup P_2] = O[P_1] \uplus O[P_2]$$

As an immediate consequence of this theorem we can construct a new method to calculate the semantics of the program

$$O[P] = \biguplus_{c \in P} C[c]_{\perp}$$

Theorem 4.4.3 Let $A$ be an atom, $D$ be a program clause and $G, G_1, G_2$ be goals. Then
4.5 Higher-order Abstraction and Types

As in the first-order version of our semantics now we can define an abstract interpretation framework that will allow us to model properties of programs, and in particular, we will be able to model important properties of computations. With the introduction of higher-order terms and connectives, we have a more complex computation environment. As a consequence, we will interested to model inherent properties of higher-order programs.

Types (simple types) are an important feature introduced by the \textit{hohh}-formulas with respect to the \textit{fohh}-formulas. This language allows us to describe functions of simple types. By means of $\lambda$-abstraction we can specify the definition of a function, since application provides a mean for representing the evaluation of such functions. In this section we will focus in the definition of an abstract semantics for type inferencing, based on the \textit{hohh}-semantics by means of the abstract interpretation.

4.5.1 The Abstraction Framework

The most important results obtained for \textit{fohh}-program can be reproduced for the \textit{higher}-order version of our semantics.

Given an observable $\alpha : \mathbb{C} \rightarrow \mathbb{A}$, we can obtain new abstract semantics. We need to define the optimal abstract counterparts of all operators in the concrete semantics, and we also need to obtain the conditions under which the abstract semantics is optimal.

Let $\mathbb{C}, \mathbb{C}_1, \mathbb{C}_2$ be abstract collections and $I_1, I_2$ be abstract interpretations in some

1. $\mathcal{B}[p(t) \text{ in } P] = \nabla (p(t))_{\mathcal{C}[P], P}$,

2. $\mathcal{B}[G_1 \land G_2 \text{ in } P] = \mathcal{B}[G_1 \text{ in } P] \otimes \mathcal{B}[G_2 \text{ in } P]$,

3. $\mathcal{B}[G_1 \lor G_2 \text{ in } P] = \mathcal{B}[G_1 \text{ in } P] \oplus \mathcal{B}[G_2 \text{ in } P]$,

4. $\mathcal{B}[\exists x.G \text{ in } P] = \exists x \mathcal{B}[G \text{ in } P]$,

5. $\mathcal{B}[\forall x.G \text{ in } P] = \forall x \mathcal{B}[G \text{ in } P]$,

6. $\mathcal{B}[D \supset G \text{ in } P] = \Delta (\mathcal{B}[G \text{ in } P \cup \{D\}])_{D \supset G, P}$.
abstract domain \( A \). Then the optimal abstract operators are:

\[
\begin{align*}
\widetilde{\Theta} \ (P) & := \alpha(\Theta(P)) \\
C \uparrow P & := \alpha(\gamma(C) \uparrow P) \\
C_1 + C_2 & := \alpha(\gamma(C_1) + \gamma(C_2)) \\
C_1 \bowtie C_2 & := \alpha(\gamma(C_1) \bowtie \gamma(C_2)) \\
\vec{\nabla} \ (A)_{C,P} & := \alpha(\nabla(A)_{\gamma(C),P}) \\
C_1 \otimes C_2 & := \alpha(\gamma(C_1) \otimes \gamma(C_2)) \\
C_1 \oplus C_2 & := \alpha(\gamma(C_1) \oplus \gamma(C_2)) \\
\vec{\Xi} C & := \alpha(\Xi \gamma(C)) \\
\vec{\forall} C & := \alpha(\forall \gamma(C)) \\
\vec{\Delta} (C)_{G,P} & := \alpha(\Delta(\gamma(C))_{G,P}) \\
C_1 \triangleright \circ \triangleleft C_2 & := \alpha(\gamma(C_1) \triangleright \circ \triangleleft \gamma(C_2))
\end{align*}
\]

(4.1)

Once we have the optimal abstract operators, we can define the corresponding abstract semantics, obtained from the concrete denotational and operational semantics.

The abstract denotational semantics is defined by means of the abstract semantic functions:

\[
\begin{align*}
\mathcal{Q}_\alpha[G \text{ in } P] & := \mathcal{G}_\alpha[G]_{\text{if } P, \alpha[P], P} \\
\mathcal{P}_\alpha[P] & := \sum_{c \in P} C_\alpha[c]_{1, P} \\
C_\alpha[p(t)]_{P} & := \Theta(p(t))_{P} \\
C_\alpha[G \triangleright p(t)]_{P} & := \Theta(G \triangleright p(t))_{P} \widetilde{\bowtie} \mathcal{G}_\alpha[G]_{1, P} \\
\mathcal{G}_\alpha[A]_{P} & := \vec{\nabla} (A)_{P} \\
\mathcal{G}_\alpha[G_1 \land G_2]_{P} & := \mathcal{G}_\alpha[G_1]_{1, P} \widetilde{\bowtie} \mathcal{G}_\alpha[G_2]_{1, P} \\
\mathcal{G}_\alpha[G_1 \lor G_2]_{P} & := \mathcal{G}_\alpha[G_1]_{1, P} \oplus \mathcal{G}_\alpha[G_2]_{1, P} \\
\mathcal{G}_\alpha[\exists x.G]_{P} & := \vec{\Xi} x \mathcal{G}[G]_{1, P} \\
\mathcal{G}_\alpha[\forall x.G]_{P} & := \vec{\forall} x \mathcal{G}[G]_{1, P} \\
\mathcal{G}_\alpha[D \triangleright G]_{P} & := \vec{\Delta} (\mathcal{G}_\alpha[G]_{1 \uplus C_\alpha[D]_{1, P \cup (D)}}, D \triangleright G, P)
\end{align*}
\]

(4.2)

where the abstract composition operator is defined as

\[
l_1 \upharpoonright l_2 = (l_1 \uparrow P_2 \upharpoonright l_2 \uparrow P_1) \uparrow \alpha
\]

where \( P_1, P_2 \) are the programs in \( \text{first}(\gamma(l_1)) \) and \( \text{first}(\gamma(l_2)) \) respectively. As in the concrete semantics, by \( \uparrow \alpha \) we intend the least fixed-point of the function \( \Xi_l \) where

\[
\Xi_l(l) = \begin{cases} 
\uparrow' & \text{if } l = \bot \\
\uparrow_{\mathcal{D} \triangleright \circ \triangleleft} l & \text{otherwise.}
\end{cases}
\]

The abstract operational semantics is based on the abstract behavior. Given a query \( G \) in \( P \) it is defined as

\[
\mathcal{B}_\alpha[G \text{ in } P] = \alpha(\lambda G. \{S_0 \rightarrow^* S_n \mid S_0 = \{\langle G, P, 0 \rangle\}, \kappa, 0, \epsilon\}) / \Xi_A
\]

the abstract top-down denotation is defined as

\[ \mathcal{O}_\alpha[P] = \sum_{p(x) \in \text{Goals}} B_\alpha[p(x) \text{ in } P]. \]

or by means of the following transition system \( \mathfrak{T}_\alpha \)

\[ C \in A, \ C \neq C \dashv \spadesuit \downarrow \downarrow \alpha(I) (\Theta(P)) \Rightarrow C \Downarrow C \dashv \spadesuit \downarrow \downarrow \alpha(I) (\Theta(P)) \quad (4.3) \]

where

\[ \alpha(I) = \sum_{G \in \text{Goals}} G_\alpha[G] \]

Thus, the abstract behavior of a program can be rewritten as

\[ B_\alpha[G \text{ in } P] = \sum \{ C | \alpha(Id_G) \vdash C \} \]

Also we can construct perfect observables which will preserve the most important properties of the concrete semantics. If \( \alpha \) is a perfect observable then

\[
\begin{align*}
\alpha(\Theta(P)) &= \alpha(\gamma(\alpha(\Theta(P)))) \\
\alpha(C \uparrow P) &= \alpha(\gamma(\alpha(C)) \uparrow P) \\
\alpha(C_1 \triangleright C_2) &= \alpha(\gamma(\alpha(C_1)) \triangleright \gamma(\alpha(C_2))) \\
\alpha(\nabla(A))_{C,P} &= \alpha(\nabla(A))_{\gamma(\alpha(C))},P) \\
\alpha(C_1 \otimes C_2) &= \alpha(\gamma(\alpha(C_1)) \otimes \gamma(\alpha(C_2))) \\
\alpha(C_1 \oplus C_2) &= \alpha(\gamma(\alpha(C_1)) \oplus \gamma(\alpha(C_2))) \\
\alpha(\exists C) &= \alpha(\exists \gamma(\alpha(C))) \\
\alpha(\forall C) &= \alpha(\forall \gamma(\alpha(C))) \\
\alpha(\Delta(C))_{C,P} &= \alpha(\Delta(\gamma(\alpha(C)))_{C,P}) \\
\alpha(C_1 \triangleright \circ \triangleleft C_2) &= \alpha(\gamma(\alpha(C_1)) \triangleright \circ \triangleleft \gamma(\alpha(C_2)))
\end{align*}
\]

For any perfect observable \( \alpha : C \to A \) we can say that the abstract composition is equivalent to the abstraction of the composed concrete semantics

\[ \alpha(I_1) \uplus \alpha(I_2) = \alpha(I_1 \uplus I_2) \]

where \( I_1, I_2 \in \mathcal{I} \).

As a consequence we can derive the precision property of perfect observables

\[ \forall I \in \mathcal{I} \alpha(P[I]) = \mathcal{P}_\alpha[P]_{\alpha(I)} \]

**Theorem 4.5.1** Let \( \alpha \) be a perfect observable. Then the abstract operational semantics and the top-down denotation are precise:
• \( \alpha(B[G \in P]) = B_\alpha[G \in P] \)
• \( \alpha(O[P]) = O_\alpha[P] \)

**Theorem 4.5.2** Let \( \alpha \) be a perfect observable. Then the abstract denotational semantics and the bottom-up denotation are precise:

1. \( \alpha(lfp P[P]) = lfp P_\alpha[P] \)
2. \( \alpha(Q[G \in P]) = Q_\alpha[G \in P] \)

All the properties of the concrete semantics also hold for the abstract top-down denotation for any perfect observable.

**Theorem 4.5.3** Let \( \alpha \) be a perfect observable, \( A \) be an atom, \( D \) be a program clause and \( G, G_1, G_2 \) be goals. Then

1. \( B_\alpha[p(t) \in P] = \tilde{\nabla}(p(t))_{O_\alpha[P],P} \)
2. \( B_\alpha[G_1 \land G_2 \in P] = B_\alpha[G_1 \in P] \tilde{\otimes} B_\alpha[G_2 \in P] \)
3. \( B_\alpha[G_1 \lor G_2 \in P] = B_\alpha[G_1 \in P] \tilde{\oplus} B_\alpha[G_2 \in P] \)
4. \( B_\alpha[\exists x.G \in P] = \tilde{\exists}_x B_\alpha[G \in P] \)
5. \( B_\alpha[\forall x.G \in P] = \tilde{\forall}_x B_\alpha[G \in P] \)
6. \( B_\alpha[D \supset G \in P] = \tilde{\Delta}(B_\alpha[G \in P \cup \{D\}])_{D \supset G,P} \).

**Theorem 4.5.4** Let \( \alpha : C \to A \) be a perfect observable and \( P, P' \) be programs, then

\[ P \approx_\alpha P' \iff O_\alpha[P] = O_\alpha[P'] \]

### 4.5.2 Monotype Abstraction

Formally, we can define a higher-order type expression \( \tau \) as

\[ \tau := x_\tau \mid \tau_1 \to \tau_2 \mid c \mid f \tau_1 \ldots \tau_n \]

where \( \tau_1 \) and \( \tau_2 \) are type expressions, \( c \) and \( f \) are a type constructors (atomic types) of the signature \( \Sigma \) with arity 0 and \( n \) respectively, and \( x_\tau \) is a type variable. We denote by \( \mathfrak{T} \) the set of all types constructed on \( \Sigma \). Note that the set of types can be represented by the set of first-order terms constructed on \( \Sigma \cup \{ \to \} \). It is possible to define a partial order \( \langle \mathfrak{T}, \preceq \rangle \) on types by using the relation of instantiation. Namely, we can say that \( \tau_1 \preceq \tau_2 \) if \( t_1 \) is more instantiated than \( t_2 \), where \( t_1 \) and \( t_2 \) are the corresponding terms of \( \tau_1 \) and \( \tau_2 \) respectively. In other words, if there exists
4.5. HIGHER-ORDER ABSTRACTION AND TYPES

a substitution \( \theta \) such that \( t_2\theta = t_1 \). For simplicity, in advance we will indistinctly refer to types as first order terms and vice versa.

A monotype system is a type system defined over a single type primitive, for example, the \( \text{int} \) type. Hence, a monotype expression \( \tau \) is defined as

\[
\tau := x_{\tau} \mid \tau_1 \rightarrow \tau_2 \mid \text{int}
\]

We can assign type of variables and terms with respect to an environment by using the following rules:

1. A constant has the type specified for it.
2. A variable has the type assigned to it by the environment.
3. The application \( (t_1 t_2) \) has a type \( \sigma \) where \( \tau \rightarrow \sigma \) and \( \tau \) are the types assigned by the environment to \( t_1 \) and \( t_2 \) respectively.
4. The abstraction \( \lambda x.t \) has a type \( \tau \rightarrow \sigma \) relative to an environment \( \Gamma \) if \( t \) has type \( \sigma \) in an environment that differs from \( \Gamma \) only in the assignment of type \( \tau \) to the variable \( x \).

We can write now a \texttt{fohh}-program that infers types for closed \( \lambda \)-terms. First of all, it is necessary to write the clauses to associate types to constants in the programs. Then, we need to represent the environment that is necessary to maintain the types assigned to bound variables. So, the environment is represented by means of a set of clauses where implication goals are used to enlarge the environment at the point of abstractions. Suppose, we have only the constants \( 0 : \text{int} \) and \( s : \text{int} \rightarrow \text{int} \), then the following program will infer the type of closed \( \lambda \)-terms.

\[
\begin{align*}
type(0, \text{int}) \\
type(s, \text{int} \rightarrow \text{int}) \\
type(app(e_1, e_2), \tau_2) \subset type(e_1, \tau_1 \rightarrow \tau_2) \land type(e_2, \tau_1) \\
type(abst(e, x), \tau_1 \rightarrow \tau_2) \subset (\forall y.\, type(y, \tau_1) \supset type(e, \tau_2) \land x = y)
\end{align*}
\]

We assume the existence of the built in predicate for unification \( = (x, x) \), which is used in the program in the infix form. Note that we use the \textit{meta}-terms \( \text{abst} \) and \( \text{app} \) to distinguish the syntax of \( \lambda \)-terms expressions. The constructor \( \text{app} \) takes as first argument the expression on which the second argument will be applied, while \( \text{abst} \) receives as second argument the variable on which the abstraction (in the first argument) is constructed. For example, if we want to infer the type of the expressions \( s(s(0)) \) and \( (\lambda f. (\lambda x. f x))(s) \), we need to solve the goal

\[
? - type(app(app(s, 0), 0), \tau_1) \land type(app(abst(app(f, x), x), f), s, \tau_2)
\]

In this case the \texttt{fohh}-interpreter will give the answer

\[
\begin{align*}
\tau_1 &= \text{int} \\
\tau_2 &= \text{int} \rightarrow \text{int}
\end{align*}
\]
We are interested to model the types of the computed answers of a program. To achieve this objective we can start from the answer substitution semantics of a hohh-program. By further refinement on substitutions, we can infer for each variable, the type of the associated term.

The above type inference mechanism can be rewritten formally as set of rules. A constrained type is a pair \( \langle \tau, \theta \rangle \) where \( \tau \) is a monotype and \( \theta \) are its related constraints, which is represented as an idempotent substitution over type variables such that \( \tau \theta = \tau \). An environment \( \Gamma \) maps identifiers to types. Substitutions are always restricted to the set of type variables occurring in \( \Gamma \). The rules for type inferencing are defined as follows

\[
\begin{align*}
\frac{c : \tau \in \Sigma}{\Gamma \vdash c \mapsto \langle \tau, \epsilon \rangle} \\
\frac{c : f \tau_1 \ldots \tau_n \in \Sigma}{\Gamma \vdash c \mapsto \langle f \tau_1 \ldots \tau_n, \epsilon \rangle} \\
\frac{\Gamma \vdash x \mapsto \langle x_\tau, \epsilon \rangle}{\Gamma[x/\tau] \vdash t \mapsto \langle \tau, \theta \rangle} \\
\frac{\Gamma \vdash \lambda x.t \mapsto \langle x_\tau \rightarrow \tau, \theta \rangle}{\Gamma \vdash t_1 t_2 \mapsto \langle \tau_1 \theta, \theta_1 \circ \theta_2 \circ \theta = \text{mgu}(\tau_1 \theta, \tau_2 \theta_1) \rangle}
\end{align*}
\]

(4.4)

Our abstract domain consist of a set of type bindings, i.e., a set of variables and the inferred type assigned by a substitution. We denote by \( \text{Types} \) the set of all types bindings \( t = \langle x, \langle \tau, \phi \rangle \rangle \). Now we need to define a function which maps substitutions to type bindings. This function can be defined by means of the rules (4.4).

\[
\mathcal{T} = \lambda \theta. \{ \langle x, \langle \tau, \phi \rangle \rangle \mid t \mapsto \langle \tau, \phi \rangle \text{ and } [t/x] \in \theta \}
\]

Let \( \mathbb{A} \subseteq [\text{Goals} \rightarrow \varphi(\text{Types})] \). As in the computed answer example we can obtain the observable \( \eta \) by further abstraction of \( \xi \). Abstract collections map goals to abstract types (set of type bindings), by assigning to each goal the most general types associated to the variables in the computation. Namely, \( \mu : \mathbb{A}_{\text{ca}} \rightarrow \mathbb{A} \) is defined as

\[
\mu(C) := \lambda G. \text{lub}_\tau (\{ \mathcal{T}(\theta) \mid \theta \in C(G) \})
\]

where \( \text{lub}_\tau \) is the least upper bound defined on set of types

\[
\text{lub}_\tau(T) = \{ \text{lub}(t_1, \ldots, t_n) \mid t_1 = \langle x, \langle \tau_1, \phi_1 \rangle \rangle \in T, \ldots, t_n = \langle x, \langle \tau_n, \phi_n \rangle \rangle \} \in T
\]

Therefore, the computed answer type observable \( \eta \) is

\[
\begin{align*}
\eta(C) & : = \mu(\xi(C)) \\
\eta^\gamma(C) & : = \lambda G. \{ d \mid \text{first}(d) = G, \ \mu(\text{answer}(d)) \in C(G), \ \text{last}(d) \text{ is a final state} \}
\end{align*}
\]
It is easy to show that \( \eta \) is an observable. As showed in the proof for the computed answer observable, \( \eta \) also maps finite elements to finite elements, so \( \eta \) and \( \eta^\gamma \) satisfy points 1 and 2 of definition of observable. Then, we have to prove the last point

\[
\forall C, C' \in \mathbb{P} \mathbb{C}, \ C \equiv C' \implies (\eta \circ \eta^\gamma)(C) \equiv (\eta \circ \eta^\gamma)(C') \tag{4.5}
\]

It is easy to see that the condition holds if the collections \( C, C' \) are identical up to renaming of variables and constants of higher level. Hence, we need to take into consideration equivalent derivations having a different order of resolution steps.

Suppose toward a contradiction that 4.5 does not holds, i.e., for some pure collections \( C, C' \) with \( C \equiv C' \) we have at least a derivation \( d_{\alpha\gamma} \) such that \( d_{\alpha\gamma} \in (\eta \circ \eta^\gamma)(C) \) and \( d_{\alpha\gamma} \notin (\eta \circ \eta^\gamma)(C') \). By completeness theorem (theorem 15, [43]) we know that a computed answer of a proof (successful derivation) exists regardless the goal selected at each stage in constructing the derivation. Hence, we know that exists \( d' \) such that \( \eta(d') = \eta(d) \) where \( d' \in C' \) and \( d_{\alpha\gamma} \in (\eta \circ \eta^\gamma)(d) \), i.e., \( d' \) is a variant of \( d \in C \). Finally, by definition of \( \eta^\gamma \) we can see that \( d_{\alpha\gamma} = \eta \circ \eta^\gamma(d) = \eta \circ \eta^\gamma(d') \), which contradicts the assumption.

Now we can construct the abstract semantics by defining the following abstract operators.

\[
\begin{align*}
\tilde{\Theta} & := \lambda c. \langle p(x), T([t/x]) \rangle \text{ where } c = p(t) \text{ or } c = G \supset p(t) \\
C \upharpoonright P & := C \\
C_1 \bowtie C_2 & := \lambda G. lub(t_1, t_2) \text{ where } t_1 = C_1(G), t_2 = C_2(G) \\
C_1 \uplus C_2 & := \lambda G \supset p(t). unif y(t_1, t_2) \text{ where } t_1 = C_1(G \supset p(t)), t_2 = C_2(p(t)) \\
\bigtriangledown (p(t))_{l,P} & := \lambda p(t). \left\{ \begin{array}{ll}
\lambda \theta. \langle t, \theta = mgu(l(p(x)), T([t/x])) \rangle & \text{if } l \neq \perp \\
\text{otherwise}
\end{array} \right.
\end{align*}
\]

\[
\begin{align*}
C_1 \bowtie C_2 & := \lambda G_1 \land G_2. unif y(t_1, t_2) \text{ where } t_1 = C_1(G_1), t_2 = C_2(G_2) \\
C_1 \uplus C_2 & := \lambda G_1 \lor G_2. lub(t_1, t_2) \text{ where } t_1 = C_1(G_1), t_2 = C_2(G_2) \\
\bigtriangledown_x C & := C \\
\bigtriangleup_x C & := C \\
\bigtriangleup^G (C)_{G,P} & := C \\
C_1 \bowtie \bowtie \bowtie C_2 & := \lambda G. unif y(t_1, t_2) \text{ where } t_1 = C_1(G), t_2 = \bigtriangledown_{G'' \in \text{Goals}} C_2(G'')
\end{align*}
\]

where \( unif y \) is a function which calculates the \( mgu \) for the types over the same variable

\[
unif y(t_1, t_2) = \left\{ \langle x, \tau_1 \theta \circ \phi_1 \circ \phi_2 \rangle \mid t_1 = \langle x, \tau_1, \phi_1 \rangle, t_2 = \langle x, \tau_2, \phi_2 \rangle \text{ and } \theta = mgu(\tau_1 \phi_1, \tau_2 \phi_2) \right\} \\
\cup \{ t_1 \mid t_1 = \langle x, \tau_1, \phi_1 \rangle, t_2 = \langle y, \tau_2, \phi_2 \rangle, x \neq y \}
\]

It can be easily provided a proof for correctness and precision of this abstract semantics. The proof follows the arguments of the proof of correctness for the compute
answer observable in chapter 3. Let see in an example how the compatible extension of types works. Suppose we have this simple (monotyped) program, where the type of \( s \) is \( \text{int} \rightarrow \text{int} \) and the type of \( 0 \) is \( \text{int} \). Notice that here we are not considering the predicate type.

\[
c_1 : \ p(r) \subset q(\lambda x. s \ x, r) \\
c_2 : \ q(y, r) \subset [t(y, r) \subset t(z, z 0)]
\]

The first thing we need to do is to calculate the abstract interpretation \( \text{ifp} \ P_\alpha [P] \) by using the abstract semantic functions (4.2).

\[
I_0 = P_\alpha [P]_\bot = C_\alpha [c_1]_\bot + C_\alpha [c_2]_\bot \\
= \tilde{\Theta} (q(\lambda x. s \ x, r) \supset p(r)) \hat{\otimes} G_\alpha [q(\lambda x. s \ x, r)]_\bot + \\
\tilde{\Theta} ([t(y, r) \subset t(z, z 0)] \supset q(y, r)) \hat{\otimes} G_\alpha [t(y, r) \subset t(z, z 0)]_\bot \\
= \langle p(r), (r, \langle r, \epsilon \rangle) \rangle \hat{\otimes} \nabla (q(\lambda x. s \ x, r)]_\bot + \\
\langle q(y, r), (r, \langle r, \epsilon \rangle), (y, \langle y, \epsilon \rangle) \rangle \hat{\otimes} \Delta (G_\alpha [t(y, r)]_\bot \ominus c_\alpha [t(z, z 0)]_\bot) \\
= \langle p(r), (r, \langle r, \epsilon \rangle) \rangle \hat{\otimes} \langle q(y, r), (r, \langle r, \epsilon \rangle), (y, \langle \langle \text{int} \rightarrow \text{int} \rangle, \epsilon \rangle) \rangle + \\
\langle q(y, r), (r, \langle r, \epsilon \rangle), (y, \langle y, \epsilon \rangle) \rangle \hat{\otimes} \Delta (G_\alpha [t(y, r)]_\bot \ominus c_\alpha [t(z, z 0)]_\bot) \\
(\text{now we calculate the new environment}) \\
I_0' = \bot \hat{\otimes} C_\alpha [t(z, z 0)]_\bot = C_\alpha [t(z, z 0)]_\bot = \tilde{\Theta} (t(z, z 0)) \\
= \langle t(z, z 1), \langle z, \langle \text{int} \rightarrow \text{int} \rangle, \epsilon \rangle, \langle z, \langle z, \epsilon \rangle \rangle \rangle \\
(\text{now we calculate the implication goal in the augmented program}) \\
I_0 = \langle p(r), (r, \langle r, \epsilon \rangle), (y, \langle \langle \text{int} \rightarrow \text{int} \rangle, \epsilon \rangle) \rangle \cup \\
\langle q(y, r), (r, \langle r, \epsilon \rangle), (y, \langle \langle \text{int} \rightarrow \text{int} \rangle, \epsilon \rangle) \rangle \cup \\
(\text{now we calculate the second iteration } I_1) \\
I_1 = P_\alpha [P]_{I_1} \\
= \tilde{\Theta} (q(\lambda x. s \ x, r) \supset p(r)) \hat{\otimes} G_\alpha [q(\lambda x. s \ x, r)]_{I_0} + \\
\tilde{\Theta} ([t(y, r) \subset t(z, z 0)] \supset q(y, r)) \hat{\otimes} G_\alpha [t(y, r) \subset t(z, z 0)]_{I_0} \\
= \langle p(r), (r, \langle r, \epsilon \rangle) \rangle \hat{\otimes} \langle q(y, r), (r, \langle \text{int} \rangle, \epsilon), (y, \langle \langle \text{int} \rightarrow \text{int} \rangle, \epsilon \rangle) \rangle + \\
\langle q(y, r), (r, \langle r, \epsilon \rangle), (y, \langle y, \epsilon \rangle) \rangle \hat{\otimes} \Delta (G_\alpha [t(y, r)]_{I_0} \ominus c_\alpha [t(z, z 0)]_{I_0}) \\
(\text{now we calculate the new environment from } I_0) \\
I_1' = I_0 \hat{\otimes} C_\alpha [t(z, z 0)]_{I_0} = I_0 \hat{\otimes} \tilde{\Theta} (t(z, z 0)) \\
= (I_0 \hat{\otimes} \tilde{\Theta} (t(y, r), \langle y, \langle \text{int} \rightarrow \text{int} \rangle, \epsilon \rangle, \langle r, \langle z, \epsilon \rangle \rangle) \rangle \hat{\otimes} \tilde{\Theta} (t(z, z 0)) \\
= \langle t(z, z 1), \langle z, \langle \text{int} \rightarrow \text{int} \rangle, \epsilon \rangle, \langle z, \langle z, \epsilon \rangle \rangle \rangle \cup I_0 \\
C_0 = G_\alpha [t(y, r)]_{I_0} = \nabla (t(y, r))_{I_0} \\
= \langle t(y, r), \langle y, \langle \text{int} \rightarrow \text{int} \rangle, \epsilon \rangle, \langle r, \langle z, \epsilon \rangle \rangle \rangle \cup I_0 \\
I_1 = \langle p(r), (r, \langle \text{int} \rangle, \epsilon), \rangle \cup (y, \langle \langle \text{int} \rightarrow \text{int} \rangle, \epsilon \rangle) \rangle \cup \\
\langle q(y, r), (r, \langle z, \epsilon \rangle), \rangle \cup (y, \langle \langle \text{int} \rightarrow \text{int} \rangle, \epsilon \rangle) \rangle \cup \\
(\text{In the next iteration we obtain that } I_2 = I_1 \text{ up to rename of type variables. So, } I_1} \)
is a fixed-point. Now we can calculate the type for a given goal. For example,

\[ \mathcal{G}_a[p(s(0))]_{I_0} = \tilde{\nu} (p(s(0)))_{I_0} \]

\[ = \langle r, \langle \text{int}, \epsilon \rangle \rangle \circ \text{mg}_u(\langle \text{int}, \epsilon \rangle, \mathcal{T}([s(0)/r])) \]

\[ = \langle r, \langle \text{int}, \epsilon \rangle \rangle \]

\[ \mathcal{G}_a[q(\lambda x. s(x)), r]_{I_0} = \tilde{\nu} (q(\lambda x. s(x)), r)_{I_0} \]

\[ = \langle r, \langle \text{int}, \epsilon \rangle \rangle \circ \text{mg}_u(\langle y, \langle \text{int} \rightarrow z_r, \epsilon \rangle \rangle, \mathcal{T}([\lambda x. s(x)/y])) \]

\[ = \langle r, \langle \text{int}, \epsilon \rangle \rangle \]

Let see another interesting example. This program is a variant (of \textit{hohh}) of an ML program presented in [9, 27], which is characterized to be non typable by Damas-Milner’s polymorphic type system[13, 42].

\[ c_1 := gfn\ f\ g\ 0\ x\ g(x) \]

\[ c_2 := gfn\ f\ (\lambda x.(\lambda h.g(h\ x)))\ n\ x\ r \supset gfn\ f\ g\ s(n)\ x\ (r\ f) \]

This program calculates \( g(f^n(x)) \). For example the goal \( gfn\ f\ g\ s(s(0))\ a\ r \) will derive

\[ c_1 : gfn\ f_1\ (\lambda x.(\lambda h.g_1(h\ x)))\ n_1\ x_1\ r_1 \supset gfn\ f_1\ g_1\ s(n_1)\ x_1\ (r_1\ f_1) \]

\[ f_1 = f, g_1 = g, x_1 = a, n_1 = s(s(0)), r_1\ f_1 = r \]

\[ c_1 : gfn\ f_2\ (\lambda x.(\lambda h.g_2(h\ x)))\ n_2\ x_2\ r_2 \supset gfn\ f_2\ g_2\ s(n_2)\ x_2\ (r_2\ f_2) \]

\[ f_2 = f_1, g_2 = \lambda x.(\lambda h.g_1(h\ x)), x_2 = x_1, n_2 = s(0), r_2\ f_2 = r_1 \]

\[ c_1 : gfn\ f_3\ (\lambda x.(\lambda h.g_3(h\ x)))\ n_3\ x_3\ r_3 \supset gfn\ f_3\ g_3\ s(n_3)\ x_3\ (r_3\ f_3) \]

\[ f_3 = f_2, g_3 = \lambda x.(\lambda h.g_2(h\ x)), x_3 = x_2, n_3 = 0, r_3\ f_3 = r_2 \]

\[ c_2 : gfn\ f_4\ g_4\ 0\ x_4\ g_4(x) \]

\[ f_4 = f_3, g_4 = \lambda x.(\lambda h.g_3(h\ x)), x_4 = x_3, g_4\ x_4 = r_3 \]

\[ x_4 = x_3 = x_2 = x_1 = a \]

\[ f_4 = f_3 = f_2 = f_1 = f \]

\[ r = r_1\ f_1 = (r_2\ f_2)\ f_1 = ((r_3\ f_3)\ f_2)\ f_1 = \]

\[ ((g_4\ x_4)\ f_3)\ f_2)\ f_1 = \]

\[ ((g_4\ a)\ f)\ f \]

\[ g_4 = \lambda x.(\lambda h.g_3(h\ x)) \]

\[ \lambda x.(\lambda h.(\lambda x.(\lambda h.g_2(h\ x)))\ (h\ x)) = \lambda x.(\lambda h.(\lambda x.(\lambda h.g_1(h\ x)))\ (h\ x))) \]

\[ = \lambda x.(\lambda h.(\lambda h.g_1(h\ x)))) \]

\[ g_4\ a = \lambda h.(\lambda h.g(h\ (h\ (h\ a)))) \]

\[ [r = g(f(f(f(a)))] \]

This is an optimal simplified version of the program

\[ c_1 := gfn\ f\ g\ 0\ x\ g(x) \]

\[ c_2 := gfn\ f\ (\lambda x.g(f\ x))\ n\ x\ r \supset gfn\ f\ g\ s(n)\ x\ r \]
which produces the following derivation

\[ \begin{align*}
  c_1 & : \text{gfn } f_1 (\lambda x. g_1(f_1 x)) \ n_1 \ x_1 \ r_1 \supset \text{gfn } f_1 \ g_1(s(n_1)) \ x_1 \ r_1 \\
  f_1 & = f, g_1 = g, x_1 = a, n_1 = s(s(0)), r_1 = r \\
  c_1 & : \text{gfn } f_2 (\lambda x. g_2(f_2 x)) \ n_2 \ x_2 \ r_2 \supset \text{gfn } f_2 \ g_2(s(n_2)) \ x_2 \ r_2 \\
  f_2 & = f_1, g_2 = \lambda x. g_1(f_1 x), x_2 = x_1, n_2 = s(0), r_2 = r_1 \\
  c_1 & : \text{gfn } f_3 (\lambda x. g_3(f_3 x)) \ n_3 \ x_3 \ r_3 \supset \text{gfn } f_3 \ g_3(s(n_3)) \ x_3 \ r_3 \\
  f_3 & = f_2, g_3 = \lambda x. g_2(f_2 x), x_3 = x_2, n_3 = 0, r_3 = r_2 \\
  c_2 & : \text{gfn } f_4 \ g_4 \ x_4 \ g_4(x_4) \\
  f_4 & = f_3, g_4 = \lambda x. g_3(f_3 x), x_4 = x_3, g_4(x_4) = r_3 \\
  r_3 & = r_2 = r_1 = r \\
  x_4 & = x_3 \ x_2 = x_1 = a \\
  f_4 & = f_3 = f_2 = f_1 = f \\
  g_4 & = \lambda x. g_3(f_3 x) = \lambda x. (\lambda x. g_2(f_2 x))(f_3 x) = \\
  & = \lambda x. (\lambda x. (\lambda x. g_1(f_1 x))(f_2 x))(f_3 x) = \\
  & = \lambda x. g(f(f(x)))(f x) = \\
  |r = g(f(f(f(a))))|
\end{align*} \]

Now we calculate the abstract interpretation \( \text{lfp } \mathcal{P}_\alpha[P] \)

\[ \begin{align*}
  I_0 & = \mathcal{P}_\alpha[P] \ | = \mathcal{C}_\alpha[c_1] \ | + \mathcal{C}_\alpha[c_2] \ | \\
  \mathcal{C}_\alpha[c_1] & = \widetilde{\Theta}(\text{gfn } f \ g \ 0 \ x \ g(x)) \\
  & = \langle f, f_r \rangle, \langle g, g_r \rangle, \langle x_3, \text{int} \rangle, \langle x, x_r \rangle, \langle x_5, g_r[g_r' = x_r \rightarrow g_r] \rangle \\
  \mathcal{C}_\alpha[c_2] & = \widetilde{\Theta}(\text{gfn } f \ g \ s(n) \ x \ (r \ f)) \widetilde{\Theta}(\text{gfn } f \ (\lambda x. (\lambda h. g(h x))) \ n \ x \ r) \\
  & = \langle f, f_r \rangle, \langle g, g_r \rangle, \langle x_3, \text{int}[n = \text{int}] \rangle, \langle x, x_r \rangle, \langle x_5, r_r[r_r' = f_r \rightarrow r_r] \rangle \widetilde{\Theta} \\
  & = \langle f, f_r \rangle, \langle x_2, x_r' \rightarrow h_r \rightarrow g_r \rangle, (n, n_r), \langle x, x_r \rangle, \langle r, r_r' \rangle \\
  & = \langle f, f_r \rangle, \langle x_2, x_r' \rightarrow h_r \rightarrow g_r \rangle, \langle x_3, \text{int}[n = \text{int}] \rangle, \langle x, x_r \rangle, \langle x_5, r_r \rangle \\
  I_0 & = \langle f, f_r \rangle, \langle g, x_r \rightarrow g_r \rangle, \langle x_3, \text{int} \rangle, \langle x, x_r \rangle, \langle x_5, g_r \rangle \\
\end{align*} \]

In the next iteration we obtain the same interpretation, so we can conclude that the type of \( \text{gfn } f \ g \ s(s(s(0))) \ a \ r \) is

\[ \begin{align*}
  \mathcal{G}_\alpha[\text{gfn } f \ g \ s(s(s(0))) \ a \ r]_{I_0} & = \widetilde{\nabla}(\text{gfn } f \ g \ s(s(s(0))) \ a \ r)_{I_0} \\
  & = I_0 \circ \text{mgu}(I_0, \mathcal{T}([f/x_1 \ g/x_2 \ s(s(s(0)))/x_3 \ a/x_4 \ r/x_5])) \\
  & = \langle f, f_r \rangle, \langle g, a_r \rightarrow g_r \rangle, \langle x_3, \text{int} \rangle, \langle x, a_r \rangle, \langle r, g_r \rangle
\end{align*} \]
Conclusions

In this thesis we have defined a semantic framework for logic programs based on hereditary Harrop formulas. We have showed how this semantic framework can be used for program interpretations which characterize compositionally the observable behavior of logic programs. Our work follows the methodology of the semantics first presented in [6, 7, 8]. Instead of reconstructing a new specific semantics for a given observable property, we have defined a general method for deriving semantics, parametrically with respect to a given notion of observable. The peculiarity of this semantic framework is that we handle a denotational and an operational semantics in a uniform way by specifying some basic algebraic operators which are directly related to the syntactic structure of the language. This allows us to address problems such as the relation between the operational semantics and the denotational semantics, the existence of a denotation of hereditary Harrop programs and their properties of compositionality, correctness and minimality.

In chapter 2 we have defined a more general operational semantics (a top-down denotation) based on derivations, associating to each program all derivations constructed by the proof procedure for fohh-programs, from all possible pure atomic goals. This denotational semantics allows us to define goal-independent denotations, and therefore the semantics of a fohh-program as a set of procedures. In addition, it is proved that the denotational semantics is equivalent to the top-down denotation, and it is defined by means of various operators that provide the compositional nature of the semantics. The semantic operators used to define the bottom-up denotation are the counterpart of the syntactic ones. From an intuitive point of view this semantics is minimal with respect to the properties we are trying to preserve. Furthermore, in our case the semantics is the most concrete, provided that the derivations reflect all low-level operational details of the proof procedure. In particular the principal features we can remark for the collecting semantics are

- the homogeneity in the definition of the top-down and bottom-up denotations, since they are based on the same semantic operators;

- the equivalence of both denotations of the semantics;

- the compositionality of the semantics with respect to syntactic operators and union of programs (OR-compositionality).
These characteristics allowed us, in chapter 3, to define an abstract interpretation framework based on the Abstract Interpretation Theory [11]. With this approach, the abstract denotational definition, transition system and goal-independent denotations are systematically derived from the concrete ones, by replacing the concrete semantic operators by their abstract optimal versions. Therefore, the definition style (denotational semantics and transition system) of the concrete semantics is inherited by all the abstract semantics. Moreover, we define a class of observable (perfect observable) which preserves the most important properties of the concrete semantics. In particular, it preserves the equivalence between abstract denotations and the compositionality. This framework can be successfully used to reason about the relationships between different semantics and to statically analyze programs. Now we can compare different semantics at different levels of abstraction, and also, it is possible to systematically derive abstract semantics which are correct approximations of the concrete semantics.

All the operators, definitions and theorems developed for the fohh-programs, were constructed having in mind a second purpose. In fact, all results obtained for fohh-programs were extended as well to hohh-programs. In chapter 4 we have reused and redefined all definitions and theorems and applied it to the higher-order context. The adaptation of the semantic framework to the higher-order language allows us the study of new properties such as abstraction and types. The operational semantics defined in this chapter is at the base of the λ-Prolog language implementations, and reflects many operational details of the proof procedure for programs based on hohh-Formulas [41] [43]. Thus, the collecting semantics for the higher-order, permits to derive new abstract semantics for analyzing higher-order features like λ-abstractions and type inferencing.

One of the aims of this work, is to establish a basis for the construction of programming tools for λProlog. As future work we point towards the individuation of new classes of observables which will help us to categorize the abstract semantics according to its properties of compositionality, correctness, etc. Traditionally known semantics for logic programs, such as answer substitutions[19, 4], computed answer substitutions [17], partial answers[15], OR-compositional correct answers [24], call patterns [23], proof trees [32, 33], depth(κ) and ground analysis semantics, can be elaborated for λProlog. These semantics could be successfully applied to techniques such as static analysis, verification, diagnosis, debugging and other programming tools. In particular, program verification, which consists in determining whether or not a given program has certain properties, is a suitable field of application for observables modeled by an abstract semantics. Basically, we can express a specification of the program behavior by abstracting the intended semantics of the program with respect to some particular observables. By analyzing the relation between the specification and the modeled behavior of the program, it is possible to establish when the behavior of the program is correct with respect to the desired one.
Bibliography


