Università degli Studi di Pisa

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis

Sequent Calculi and Indexed Categories as a Foundation for Logic Programming

Gianluca Amato

Abstract. We present a semantic framework for sequent calculi which allows us to reason about properties of proofs. Using this framework we try to reconcile the proof theoretic approach to logic programming, essentially based on the concept of uniform proof, with the older model theoretic approach. Using abstract interpretation to model abstractions of proofs, we are able to port well known results for pure logic programming, in the field of abstract debugging and analysis, to richer languages such as hereditary Harrop formulas.

While this approach is based on proof theory, there are cases of languages, like CLP, where models play a fundamental role. Therefore, we also propose a categorical semantic framework which is able to cope with both the syntax, the operational semantics and the model theory of a broad range of logic languages. A program is interpreted in an indexed category such that the base category contains all the possible states which can occur during the execution of the program (such as global constraints or type informations), while each fiber encodes the corresponding relevant logical properties. We define appropriate notions of categorical resolution and models, and we prove the related correctness and completeness properties.

Corso Italia 40, 56125 Pisa, Italy. TEL: +39-050-887248. FAX: +39-050-887226.
E-MAIL: amato@di.unipi.it. HOME PAGE: http://www.di.unipi.it/~amato

Contents

Introduction

One of the greatest benefits of logic programming, as presented in [47] and [5], is that it is based upon the notion of *executable specification*. The text of a logic program is endowed with both an operational (algorithmic) interpretation and an independent mathematical meaning which agree each other in several ways.

An operational interpretation is needed if we really want to write programs which can be executed, while a clear mathematical meaning simplifies the work of the programmer, which can focus himself towards "what to do" instead of "how to do". The problem is that operational expressiveness (intended as the capability of directing the flow of execution of a program) tends to obscure declarative meaning. Research in logic programming strives to find a good balance between these opposite needs.

Horn logic programming was one of the first attempts in this area and surely the most famous. However it has a lot of deficiencies when it comes to real programming tasks. On the one side, its forms of control flow are too primitives: there are simple problems (as computing the reflexive closure of a relation) that cannot be coded in the obvious way since the programs so obtained do not terminate. On the other side, the logical expressive power is too weak, both for programming in the small and in the large: it completely lacks any notion of module, program composition, typing. Moreover, if we want to work with some data structure, we need to manually code the behavior and constraints of such a structure in the Horn logic, often obscuring the intended meaning of the code.

For these reasons, various extensions have been proposed to the original framework of Horn logic programming. Several interwoven threads can be recognized in this research:

- extensions that introduce constructs useful for structuring logic programs, both in the small, expanding the underlying logic [36, 52, 58] or in the large, defining notions of modules and program composition [14, 12];

- extensions to logic programming introducing computation in several kind of domain. This is often obtained expanding and redefining the scope of the original framework to fall in the realm of CLP (constraint logic programming) [40];

- extensions that introduce constructs to control the flow of execution. These can be done in a completely operational fashion, like the “cut” operator in PROLOG [66] that totally disregards the declarative nature of the language, or trying to embed the control component in the logic itself, resorting to high-order logic [58], linear logic [38], logics with negation [17, 28];

- extensions that provide a typing system for logic programming [60] and a way to define abstract data type in a declarative setting, without manually coding objects in Horn logic [46].

The effect of all these activities has been to expand the boundaries of the field and the notion itself of declarative content of a program. There is a need of criteria for good language design and models w.r.t. which compare the new features. The original framework of Horn logic programming is now too far away to be taken as a reference point.

If we restrict ourselves to only consider those extensions of logic programs which retain a full declarative meaning, two main approaches have been pursued. One is based on proof theory, the other on model theory.

I.1 The Proof Theoretic Approach

One of the main directions for extending the logic programming paradigm is moving to a broader fragment of logic than Horn clauses. In principle, one could choose the entire first order logic as a programming language [13]. However, if we want to restrict in some way the nondeterminism in the proof procedure, which in turn is essential to the efficiency of the execution, we need to be more careful.

Uniform proofs [54] have widely been accepted as one of the main tools for approaching the problem and to distinguish between logic without a clear computational flavor and logic programming languages. However, that of uniform proofs being a concept heavily based on proof theory, researches conducted along this line have always been quite far from the traditional approach based on fixpoint semantics. In turn, this latter tradition has brought up several important results concerning the effective utilization of Horn clauses as a real programming language. Among the others, problems such as compositionality of semantics [18], modularity [12, 15], static analysis [27], debugging [19], verification [45] have been tackled in this setting. Adapting these results to the new logic languages developed via the proof theoretic approach, such as λProlog [57] or LinLog [4], would probably require at least two things:

- provide a fixpoint semantics for these new languages;

- generalize a great number of concepts whose definition is too much tied to the case of Horn clauses.

The first part of the thesis proposes a semantic framework which can be useful in such an effort. The main idea is to recognize proofs in the sequent calculi as the general counterpart of SLD resolutions for positive logic programs. Thus, the three well-known semantics (operational, declarative and fixpoint) for Horn clause logic can be reformulated within this general setting and directly applied to all the logic languages based on sequent calculi.

Moreover, these semantics are generalized to be parametric with respect to a *pre-interpretation*, which is essentially a choice of semantic domains and intended meanings for the inference rules. When a pre-interpretation is given, we have fixed a particular property of the proofs we want to focus our attention on (correct answers, resultants, groundness). Hence, classical abstractions such as correct answers or resultants, used in the semantic studies of logic programs, and abstractions for static analysis like groundness, can be retrieved in terms of properties of proofs. Expressed in such a way, rather than referring to a computational procedure like SLD resolution, they are more easily extendable to other logic languages.

It turns out that the most convenient way of defining pre-interpretations is through abstract interpretation theory [24]. In this way, we provide a semantic framework for the new proof-theoretic based logic languages to which most of the studies we have for positive logic programs can be easily adapted.

In particular, we show how *correct answers* can be introduced for the full first order logic, as the recording of all the occurrences of quantifier introduction rules in a proof. In the case of Horn clauses, if we only consider the introductions of existential quantifiers, this turns out to be equivalent to the standard definition. We also introduce a corresponding generalization for *correct resultants*. Then, we consider a common abstraction for the semantics of logic programs, namely *groundness*, and we examine its extension to the case of full first order logic. A prototypical abstract interpreter for this observable has been developed, and we show some of the results we have obtained.

I.2 The Model Theoretic Approach

If we want to extend the pure logic programming paradigm but we do not want to use a more powerful logic than Horn clauses, we can choose a model-theoretic approach. Instead of interpreting Horn clauses within a free syntactical model, such as the Herbrand universe, we can decide that some terms and predicates have a fixed meaning in some properly defined model. For example, this is the approach pursued in CLP [40] or in some extensions of logic programs with data types [46].

Moreover, we need to observe that new logic languages are narrowing the distance between logic and functional programming. Also, the semantic methods for Horn logic programming are becoming similar in spirit to those for functional programming under the stimulus of techniques such as abstract interpretation [18]. This suggests to look for a new foundation for logic programming in the area of categorical logic.

It is obvious that category theory can handle the model theoretic approach far easier than set theory, since the distinction itself between syntax and semantics is blurred. Moreover, categorical methods have been used extensively to give clean approaches to side effects and states, non-determinism and type disciplines. Therefore, there are good reasons to think they will be effective also in providing a clean semantics for control flow in logic programming.

The second part of the thesis wants to be a step forward toward the definition of a new categorical foundation for logic programming. In some sense, we try to build a complete categorical framework by merging the idea of considering indexed categories as a model of logic program transitions [22] with an indexed version of the T_P operator [68] which appears in [30]. The immediate consequence operator seems to be an fundamental cornerstone of logic programming, since it appears, in one form or another, across several semantic treatments of logic programs [9]. Moreover, as we have discussed before, most of the studies in the semantics of logic programming are heavily based on the existence of some fixpoint construction. For these reasons, it seems to us that further investigations of a categorical framework which includes a form of bottom-up semantics is advisable.

The main point in our approach is that a language is defined by an indexed category. The base category represents the class of all the states and state transitions to which program execution can lead. Here the concept of state is quite broad: it can be the value of a global store, a constraint or just the current set of free variables. For each object in the base category, we have a fiber which represents the goals and the transitions between goals which are allowed in the corresponding state. It is essentially the standard interpretation of first order logic in an indexed category [43], but we generally require far less structure than it is customary, since we work on very small fragments of logic.

Once we have defined this indexed category, which we call *syntactic doctrine*, a clause is essentially a new arrow we want to freely adjoin in the fibers of the doctrine and a program is a collection of clauses. A *model* for a program is a indexed functor from the syntactic doctrine $\mathcal{P}$ to another indexed category $\mathcal{Q}$, together with an assignment of arrows in the fibers of $\mathcal{Q}$ to the clauses of the program. If we freely adjoin the clauses to $\mathcal{P}$, we obtain a *free model*.

An operational semantics is given by defining a notion of derivation in an indexed category. In this *categorical derivation*, both clauses and arrows in the syntactic doctrine are used for rewriting goals, so that several constraints can be enforced at the operational level. It is shown that from categorical derivations it is possible to build the free model of a program.

Then, a fixpoint bottom-up semantics is introduced. Given an indexed functor from the syntactic doctrine $\mathcal{P}$ to a semantic doctrine $\mathcal{Q}$ and a program P, it is possible to build a chain of indexed functors such that the colimit of the diagram is a model of P. However, this construction only works when $\mathcal{Q}$ has particular properties: among the others, the existence of left adjoints to reindexing functors.

Other conditions should be considered if we want to prove stronger relationships between the model we obtain and the operational semantics.

In all this setting, fibers in the indexed categories can be endowed with some kind of categorical structure (finite products, monoidal structure or others). In this way, it is possible to model the inner structure of goals and the selection rules.

I.3 Plan of the Thesis

In short, the thesis is organized as follows. Notations and basic concepts of set theory, logic and abstract interpretation are introduced in Chapter 1. We also give some results on the application of abstract interpretation to model theoretic properties, when models are characterized as pre-fixpoints of monotone functions.

In Chapter 2 we define a semantic framework for sequent calculi which is similar in spirit to the treatment of Horn clauses in logic programming. We have three different semantic styles (operational, declarative, fixpoint) which agree on the set of all the proofs for a given calculus. Following the guideline of abstract interpretation, it is possible to define abstractions of the concrete semantics.

In Chapter 3, working within the semantic framework of Chapter 2, we propose a couple of extensions to the concepts of correct answers and correct resultants which can be applied to the full first order logic. We motivate our choice with several examples and we show how to use correct answers to reconstruct an abstraction which is widely used in the static analysis of logic programs, namely groundness. As an example of application, we present a prototypical top-down static interpreter for properties of groundness which works for the full first order logic.

In Chapter 4 we fix the relevant notations for category theory which will be used in the second part of the thesis. The presentation is far from being complete, but particular care is devoted to indexed categories and monoidal structures, since these will play a fundamental role in next chapters.

In Chapter 5 we start by briefly sketching some of the works in the categorical treatment of logic programs which are related to the thesis. Then we introduce the categorical framework we are going to use in this and the next chapter. The presentation we give here is quite simple and it is not able to cope with the most essential structure of logic languages which is the logical “and”.

In Chapter 6 we extend the framework so that it is able to treat conjunctions in goals. This is realized by introducing the use of premonoidal structures on the fibers of the syntactic and semantic doctrines. We also show some examples of logic languages which can be treated inside the framework.

Finally, the last chapter is devoted to some conclusive remarks. Some of these are already presented at the end of the respective chapters, others are new, and involve the thesis in its generality.

Part I

Abstract Interpretation for Sequent Calculi

Chapter 1

Preliminaries

Abstract

In this chapter, we introduce most of the notations and results which will be used through the rest of thesis. Since we need to cover several different branches of computer science, our presentation will be quite shallow. Particular care, however, is devoted to the application of abstract interpretation to model theoretic properties, when models are characterized as pre-fixpoints of monotone functions.

Some more specific notions will be introduced in the chapters where they are needed. For more details, we remand to the bibliography.

1.1 Basic Set Theory

We do not want to be involved in foundational aspects of set theory. We just want to fix some basic terminologies. We use the standard notations for membership $\in$, the empty set $\emptyset$, inclusion $\subseteq$, proper inclusion $\subset$, set-theoretic difference $\setminus$, intersection $\cap$, union $\cup$ and disjoint union $\uplus$. If A and B are sets we use $A \subseteq_f B$ to denote that A is a finite subset of B, $\wp(A)$ for the *powerset* of A and $\wp_f(A)$ for the set of all the finite subsets of A.

With $A \times B$ we denote the *cartesian product* of A and B, whose elements are *ordered pairs* $\langle a, b\rangle$ with $a \in A$ and $b \in B$. We also define cartesian products of more than two sets and we extend the notations accordingly.

The set of *natural numbers*, denoted by $\mathbb{N}$, is assumed to contain the number zero. If $n \in \mathbb{N}$ and $n \geq 2$, we write A^n for the product of n copies of A.

We write $\mathsf{card}(A)$ for the cardinality (finite or infinite) of A. A *singleton* is a set with only one element. An *unordered pair* is a set with two elements. We use the notation $\wr a_1, a_2 \int$ for an unordered pair made of the objects a_1 and a_2, where we are implicitly assuming $a_1 \neq a_2$.

1.1.1 Relations

A *binary relation* R between A and B (denoted by $R : A \times B$) is a subset of the cartesian product $A \times B$. We write $a\ R\ b$ for $\langle a, b\rangle \in R$.

We compose two relations $R : A \times B$ and $S : B \times C$ to obtain their *composition* $S \circ R : A \times C$ defined as

$$S \circ R = \{\langle a, c\rangle \mid \text{ there exists } b \in B \text{ such that } a\ R\ b \text{ and } b\ S\ c\} \ . \tag{1.1.1}$$

The composition of relations is associative. If $R : A \times A$, we define R^n as $R \circ \cdots \circ R$ with n consecutive compositions. If we want to be more formal, it is $R^0 = id_A$ and $R^{n+1} = R^n \circ R$. We also denote by R^ω the relation

$$R^\omega = \bigcup_{n \in \mathbb{N}} R^n \ . \tag{1.1.2}$$

When $R : S \times S$ represents transitions of some abstract machine over the set of states S, we call (S, R) a *transition system*. In this case, we use an arrow such as $\longmapsto$ on the place of R.

Given $R : A \times B$, $A' \subseteq A$ and $B' \subseteq B$, we define the *restriction* of R to $A' \times B'$, denoted by $R\mid_{A' \times B'}: A' \times B'$ as

$$R\mid_{A' \times B'} = R \cap (A' \times B') \ . \tag{1.1.3}$$

A binary relation $\approx: A \times A$ is an *equivalence relation* when, for each $x, y, z \in A$, the following properties hold:

$$x \approx x \qquad \text{(reflexivity)} \ , \tag{1.1.4}$$

$$x \approx y \text{ entails } y \approx x \qquad \text{(symmetry)} \ , \tag{1.1.5}$$

$$x \approx y \text{ and } y \approx z \text{ entails } x \approx z \qquad \text{(transitivity)} \ . \tag{1.1.6}$$

The *equivalence class* of an element $x \in A$ w.r.t. $\approx$ is the set $[x]_\approx = \{y \in A \mid y \approx x\}$. When it is clear from the context, we abbreviate $[x]_\approx$ by $[x]$ and we often abuse notation by letting the elements of a set denote their correspondent equivalence class. The *quotient set* $A/_\approx$ of A modulo $\approx$ is the set of all the equivalence classes of the elements of A, i.e.

$$A/_\approx = \{[x] \mid x \in A\} \ . \tag{1.1.7}$$

1.1.2 Functions

A *partial function*(or *partial map*) from A to B is a relation $f : A \times B$ such that for each $a \in A$ there is at most one $b \in B$ such that $a\ f\ b$. By $f : A \rightharpoonup B$ we denote a partial map from the *domain* A to the *codomain* B. We use the notation $f(a)$ to denote the only b such that $a\ f\ b$, assuming it exists. In this case we say that f is defined (or converge) in a and we write $f(a)\downarrow$. Otherwise, we say that f is undefined (or diverge) in a, and we write $f(a)\uparrow$.

Given two partial function, f and g from A to B, we write $f(a_1) \simeq g(a_2)$ when either $f(a_1)\uparrow$ and $g(a_2)\uparrow$ or $f(a_1) = y = g(a_2)$. We denote by $f = g$ the extensional equality, i.e. for each $a \in A$ it is $f(a) \simeq g(a)$. Furthermore $g = f[a/b]$ denotes a function g such that $g(x) \simeq f(x)$ for each $x \neq a$ and $g(a) = b$.

If $f : A \rightharpoonup B$ and $X \subseteq A$, we call the *image* of X trough f, and we denote it by $f(X)$, the set $\{b \in B \mid \exists x \in X.f(x) = b\}$.

A *total function* (or a *total map*) $f : A \to B$ is a partial function $f : A \rightharpoonup B$ such that $f(a)\downarrow$ for each $a \in A$. When we use the terms "function" or "map" without further specifications, we always mean a total one. The set of partial functions from A to B will be denoted by $[A \rightharpoonup B]$. For the set of total functions, we use the symbol $[A \to B]$ or B^A.

A function $f : A \to B$ is *injective* when from $a \neq b$ follows $f(a) \neq f(b)$. It is *surjective* or *onto* when $f(A) = B$. For each set A there is a total function id_A such that $id_A(a) = a$ for each $a \in A$. It is the *identity function* on A. A *fixpoint* for a function $f : A \to A$ is a an element $x \in A$ such that $f(x) = x$.

Functions are composed exactly like relations. If $f : A \to B$ and $A' \subseteq A$, we use $f\mid_{A'}$ as a short form for $f\mid_{A' \times B}$.

1.1.3 Sequences and Multisets

A *finite sequence* over the set A is a function $\sigma : M \to A$ where M is a possibly empty initial segment of $\mathbb{N}$. The length of σ, denoted by $\mathsf{length}(\sigma)$ is the cardinality of M. If M is empty, then σ is an *empty sequence*, and it is denoted by λ or by a single dot like in "$\cdot$". Otherwise, we denote σ by writing one after the other the elements in its image, from $\sigma(0)$ to $\sigma(\mathsf{length}(\sigma) - 1)$. We often write σ_i instead of $\sigma(i)$. If σ and σ' are two sequences, $\sigma \cdot \sigma'$ is their concatenation. Sometimes, we use a vector, such as $\vec{x}$, to denote a sequence.

With A^* we denote the set of all the finite sequences over A. In the context of formal languages, A is called *alphabet* and A^* is the set of *words* over that alphabet. A *language* over A is a subset of A^*.

A *multiset* is essentially a set in which we keep track of the number of occurrences of every element. Formally, given a set S, a (finite) multiset over S is a function $F : S \to \mathbb{N}$. With multisets we use the standard set theoretical symbols. In particular, we write

$$x \in F \iff F(x) > 0 \ , \tag{1.1.8}$$

and, given F_1, F_2 multisets over S, we define the following new multisets over S:

$$F_1 \cap F_2 = \lambda x \in S. \min(F_1(x), F_2(x)) \ , \tag{1.1.9}$$

$$F_1 \cup F_2 = \lambda x \in S.F_1(x) + F_2(x) \ , \tag{1.1.10}$$

$$F_1 \setminus F_2 = \lambda x \in S. \max(0, F_1(x) - F_2(x)) \ , \tag{1.1.11}$$

where $\min(x, y)$ and $\max(x, y)$ denote respectively the least and greatest among the integer numbers x and y. Given a sequence σ over A, we write $\llbracket \sigma \rrbracket$ for the multiset over A given by the function

$$\llbracket \sigma \rrbracket = \lambda x \in A. \,\mathsf{card}\left(\{n \in \mathbb{N} \mid \sigma(n) = x\}\right) \ . \tag{1.1.12}$$

1.2 Ordered Structures

A binary relation $\leq$ on A (i.e. $\leq \colon A \times A$) is a *preorder* when it is reflexive (1.1.4) and transitive (1.1.6). Moreover, it is a *partial order* when, for each $x, y, z \in A$, the following property holds:

$$x \leq y \text{ and } y \leq x \text{ entails } x = y \qquad \text{(antisymmetry)} \ . \tag{1.2.1}$$

A pair $(A, \leq_A)$ made of a set A and a preorder (partial order) $\leq_A$ is called preordered set (poset).

For each preorder $\leq_A$, it is possible to define an equivalence relation $\approx$ on A such that $a \approx a'$ iff $a \leq_A a'$ and $a' \leq_A a$. Here and in the rest of the thesis we write "iff" as a short from for "if and only if".

On the quotient set $A/_{\approx}$, $\leq_A$ induces a relation $\leq_{\approx}$ such that

$$[x] \leq_{\approx} [y] \text{ iff } x \leq_A y \ , \tag{1.2.2}$$

which happens to be a partial order.

A *chain* in a preordered set A is a $B \subseteq A$ such that B is totally ordered, i.e. for each $x, y \in B$ with $x \neq y$, either $x \leq_A y$ or $y \leq_A x$.

Given a preordered set $(A, \leq_A)$ and $B \subseteq A$ not empty, we write $\min_A(B)$ (*minimum*) to denote, if it exists, the least element of B, namely the only $x \in B$ such that $x \leq_A y$ for each $y \in B$. We say that $x \in A$ is an *upper bound* of B if $y \leq_A x$ for each $y \in B$. If it exists, we denote by $\bigsqcup_A B$ the *least upper bound* of B, i.e.

$$\bigsqcup\nolimits_A B = \min_A \{x \in A \mid x \text{ is an upper bound of } B\} \ . \tag{1.2.3}$$

In the same way, it is possible to define the dual notions of *maximum* ($\max_A$) , *lower bound* and *greatest lower bound* ($\bigsqcap_A$) . We use lub and glb as short forms for least upper bound and greatest lower bound.

If $(A, \leq_A)$ has a lub or a glb for each singleton, then it is a poset. If it has a lub and a glb for every finite set, it is called a *lattice*. It is a *complete lattice* if the lub's and glb's exist for each subset of A. Note that if $(A, \leq_A)$ has lub's for each $B \subseteq A$, it also has glb's for each B, since it is

$$\bigsqcap\nolimits_A B = \bigsqcup\nolimits_A \{x \in A \mid \ x \text{ is a lower bound of } B\} \ . \tag{1.2.4}$$

The dual property trivially holds. If a poset has all finite meets (joins), it is called *meet-semilattice* (*join-semilattice*) . Finally, a poset is a *cposet* when every chain has a least upper bound.

If $(A, \leq_A)$ and $(B, \leq_B)$ are meet-semilattices, we say B is a meet-subsemilattice of A when $B \subseteq A$ and the finite meet operation in B is the restriction of the meet operation in A; in formulas

$$\forall x, y \in B.\ x \sqcap_B y = x \sqcap_A y\ \ . \tag{1.2.5}$$

In the same way, we define a join-subsemilattice. Moreover, B is a sublattice of A when it is both a join-subsemilattice and a meet-subsemilattice. Now, assume $(A, \leq_A)$ and $(B, \leq_B)$ are complete lattices. Then, B is a complete meet-subsemilattice when $B \subseteq A$ and

$$\forall X \subseteq B.\ \bigsqcap\nolimits_B X = \bigsqcap\nolimits_A X\ \ . \tag{1.2.6}$$

The definitions of complete join-subsemilattice and complete sublattices are trivial. Further details on lattices and other ordered structures can be found in [11, 61].

1.2.1 Functions between Ordered Structures

Give two preordered sets $(A, \leq_A)$ and $(B, \leq_B)$, a function $f : A \to B$ is called *monotonic* when, for each $x, y \in A$ with $x \leq_A y$, it is $f(a) \leq_B f(b)$. Assuming the preorders are clear from the context, we write $f : A \overset{m}{\to} B$ for a monotonic function from A to B and we denote by $[A \overset{m}{\to} B]$ the set of monotonic function from A to B.

A function $f : A \to B$ is said to be *continuous* when it preserves least upper bounds of chains. This mean that, if $C \subseteq A$ is a chain and $\bigsqcup_A C$ exists, then $\bigsqcup_B f(C)$ exists and it is

$$\bigsqcup\nolimits_B f(C) = f\left(\bigsqcup\nolimits_A C\right)\ \ . \tag{1.2.7}$$

We write $f : A \overset{c}{\to} B$ to mean that f is continuous and $[A \overset{c}{\to} B]$ for the set of continuous functions from A to B. In the same way, we call *additive* a function $f : A \overset{a}{\to} B$ which preserves all the least upper bounds. Additivity implies continuity which, in turns, implies monotonicity. It is possible to define also the dual notions of co-continuity ($\overset{c^\star}{\to}$) and co-additivity ($\overset{a^\star}{\to}$), as functions preserving meets of chains and arbitrary sets. All these properties of functions are preserved by composition.

Given an $f : A \to A$ between preordered sets, a *pre-fixpoint* for f is an element $x \in A$ such that $f(x) \leq_A x$. Dually, a *post-fixpoint* is an x such that $x \leq_A f(x)$. A fundamental theorem of ordered structures is the following

Theorem 1.2.1 (Knaster-Tarski [67]) *A function $f : A \overset{m}{\to} A$ on a complete lattice has a set $S \subseteq A$ of fixpoints which is a lattice according to the restriction of*

$\leq_A$ to S. In particular, there is a least fixpoint $\mathsf{lfp}(f)$ *and a greatest fixpoint* $\mathsf{gfp}(f)$ *defined as*

$$\mathsf{lfp}(f) = \bigsqcap_A \{x \mid f(x) \leq_A x\} \ , \tag{1.2.8}$$

$$\mathsf{gfp}(f) = \bigsqcup_A \{x \mid x \leq_A f(x)\} \ . \tag{1.2.9}$$

The Knaster-Tarski theorem is important because it applies to any monotone function on a complete lattice. However, most of the time we will be concerned with least fixpoint of continuous functions. The result below is usually attributed to Kleene.

Theorem 1.2.2 *Let $f : A \overset{c}{\to} A$ defined on a cposet A and let $x \in A$ be a post-fixpoint of f. Then,*

$$f^\omega(x) = \bigsqcup_A \{f^i(x) \mid i \in \mathbb{N}\} \tag{1.2.10}$$

is the least fixpoint of f greater than x.

As a corollary, if A is a cposet and $f : A \overset{c}{\to} A$, then f has a least fixpoint.

1.3 Abstract Interpretation

Abstract interpretation [24] is a theory developed to reason about the abstraction relation between two different semantics (the *concrete* and the *abstract* one). The idea of approximating program properties by evaluating a program on a simpler domain of descriptions of concrete program states goes back to the early 70's. Program states and their abstract descriptions can be related by a pairs of functions, an *abstraction* α and a *concretization* γ, which form a *Galois connection*.

Let the poset $(C, \leq_C)$ be the domain of the concrete semantics (*concrete domain*), while the poset $(A, \leq_A)$ is the domain of the abstract semantics (*abstract domain*). The partial order relations reflect an approximation between values, where $x \leq y$ means that x is more precise then y.

Definition 1.3.1 (Galois connection) *Let $(C, \leq_C)$ and $(A, \leq_A)$ be posets (the concrete and the abstract domain). A* Galois connection *$\langle \alpha, \gamma \rangle : (C, \leq_C) \rightleftharpoons (A, \leq_A)$ is a pair of monotonic maps $\alpha : C \overset{m}{\to} A$ and $\gamma : A \overset{m}{\to} C$ such that*

$$\alpha(a) \leq_C c \iff a \leq_A \gamma(c) \tag{1.3.1}$$

for each $a \in A$ and $c \in C$. When α is onto, then $\langle \alpha, \gamma \rangle$ is called a Galois insertion*.*

A Galois connection is also called *adjunction*, since it is essentially a pair of adjoint functors if partial orders are viewed as categories. Given a Galois connection

$\langle \alpha, \gamma \rangle : (C, \leq_C) \rightleftharpoons (A, \leq_A)$, we can define an equivalence relation $\approx : A \times A$ on the abstract domain, such that

$$a_1 \approx a_2 \iff \gamma(a_1) = \gamma(a_2) \ . \tag{1.3.2}$$

This induces a Galois insertion $\langle \alpha_{\approx}, \gamma_{\approx} \rangle : (C, \leq_C) \rightleftharpoons (A/_{\approx}$ given by

$$\alpha_{\approx}(c) = [\alpha(c)] \ , \tag{1.3.3}$$

$$\gamma_{\approx}([a]) = \gamma(a) \ . \tag{1.3.4}$$

Every Galois connection enjoys the following properties:

1. $\alpha(\gamma(a)) \leq_A a$ for each $a \in A$ and $c \leq_C \gamma(\alpha(c))$ for each $c \in C$;

2. γ is injective iff α is onto iff $\alpha \circ \gamma = id_A$;

3. α is additive and γ is co-additive;

4. the abstraction and concretization maps uniquely determine each other, since

$$\gamma(a) = \bigsqcup\nolimits_C \{c \in C \mid \alpha(c) \leq_A a\}, \quad \alpha(c) = \bigsqcap\nolimits_A \{a \in A \mid c \leq_C \gamma(a)\} \ . \tag{1.3.5}$$

Conversely, if $\alpha : C \overset{a}{\rightarrow} A$ and C is a complete lattice, we can use equation (1.3.5) to obtain a $\gamma : A \overset{a^\star}{\rightarrow} C$ such that $\langle \alpha, \gamma \rangle$ is a Galois connection. The same holds if we start from a complete lattice A and a function $\gamma : A \overset{c^\star}{\rightarrow} C$.

Now, assume there is an operator (i.e. a function) $\mathsf{op} : C \overset{m}{\rightarrow} C$ defined in the concrete domain. An operator $\widetilde{\mathsf{op}} : A \overset{m}{\rightarrow} A$ on the abstract domain is said to be *correct* w.r.t. op if

$$\alpha(\mathsf{op}(c)) \leq_A \widetilde{\mathsf{op}}(\alpha(c)) \ , \tag{1.3.6}$$

for each $c \in C$. Among all the correct operators w.r.t. op, there is an *optimal* one, i.e. $\alpha \circ \mathsf{op} \circ \gamma$. Here optimality means that, if there is another correct operator $\widetilde{\mathsf{op}}$, then

$$\alpha(\mathsf{op}(\gamma(a))) \leq_A \widetilde{\mathsf{op}}(a) \ , \tag{1.3.7}$$

for each $a \in A$. In the following, we denote $\alpha \circ \mathsf{op} \circ \gamma$ by op_α.

The abstract operator $\widetilde{\mathsf{op}}$ is said to be *complete* w.r.t. op if (1.3.6) still holds when we replace $\leq_A$ with equality. Obviously, completeness implies correctness. In general, given an operator op and a Galois connection, we are not sure that there is a complete abstract operator $\widetilde{\mathsf{op}}$. When this happens, then $\widetilde{\mathsf{op}}$ is always the optimal abstract operator and we say that the Galois connection is complete w.r.t. op.

Correctness and completeness are preserved by composition. The same does not happen for optimality. Moreover if C is a cposet and $\widetilde{\mathsf{op}}$ is correct w.r.t. op then $\mathsf{lfp}(\widetilde{\mathsf{op}})$ does exist and $\alpha(\mathsf{lfp}(\mathsf{op})) \leq_A \mathsf{lfp}(\widetilde{\mathsf{op}})$. If $\widetilde{\mathsf{op}}$ is complete, then $\leq_A$ can be replaced by an equality.

1.3.1 Abstract Interpretation in Declarative Settings

Abstract interpretation techniques have been widely employed to provide abstractions of denotational and operational semantics. We will give here some accounts on how to use them in declarative settings.

When we work with declarative semantics, we generally have a poset $(\mathcal{I}, \leq)$ of *interpretations* and a $M \subseteq \mathcal{I}$ of *models*. We assume that M is characterized as the set of pre-fixpoints of some monotone operator $T : \mathcal{I} \to \mathcal{I}$.

Given a Galois connection $\langle \alpha, \gamma \rangle : \mathcal{I} \rightleftharpoons \mathcal{A}$, we call $\mathcal{A}$ the poset of *abstract interpretations*. If we want to talk of *abstract models*, the natural way to do it is saying that A is an abstract model when $\gamma(A)$ is a model. However, the T operator has a corresponding optimal abstract operator T_α defined as $T_\alpha = \alpha \circ T \circ \gamma$. The question arises whether there is some connection between abstract models and pre-fixpoints of T_α.

Theorem 1.3.2 *A is an abstract model iff it is a pre-fixpoint of T_α.*

Proof. If A is an abstract model, by definition $T(\gamma(A)) \leq \gamma(A)$. By the fundamental property of Galois connections, $T(\gamma(A)) \leq \gamma(A)$ iff $\alpha(T(\gamma(A))) \leq A$. Since $T_\alpha(A) = \alpha(T(\gamma(A)))$, this proves the theorem. ■

An interesting property of models defined as pre-fixpoints is that they are closed under meet. Therefore, if $\mathcal{I}$ is a complete lattice, there is a least model given by $\mathsf{lfp}(T)$. Moreover, if α is onto, $\mathcal{A}$ is a complete lattice, too. As a result, there is a least abstract model. We just prove the following:

Theorem 1.3.3 *If $\mathcal{I}$ is a (complete) meet-semilattice and α is onto, then $\mathcal{A}$ is a (complete) meet-semilattice, too.*

Proof. The theorem easily follows from the fact that, given $X \subseteq \mathcal{A}$, we have

$$\bigsqcap\nolimits_{\mathcal{A}} X = \alpha\left(\bigsqcap\nolimits_{\mathcal{I}} \gamma(X)\right) \tag{1.3.8}$$

when the right hand side is defined. ■

Now, if A is an abstract model, $\gamma(A)$ is a model by definition. It would be useful if the converse were true, i.e. if given a model I, $\alpha(I)$ were an abstract model. This is false in general. However, the following holds

Theorem 1.3.4 *If $\langle \gamma, \alpha \rangle : \mathcal{I} \rightleftharpoons \mathcal{A}$ is complete w.r.t. T and a I is a pre-fixpoint of T, then $\alpha(I)$ is a pre-fixpoint of T_α.*

Proof. The proof is quite easy. If I is a pre-fixpoint of T, then $T_\alpha(\alpha(I)) = \alpha(T(I)) \leq \alpha(I)$, thus $\alpha(I)$ is a pre-fixpoint of T_α. ■

The completeness of T also gives interesting properties relating least models in the concrete and abstract settings. In particular, we prove the following theorems.

Theorem 1.3.5 *Assume α is complete w.r.t. T and I is the least model. Then $\alpha(I)$ is the least abstract model.*

Proof. If A is an abstract model, then $\gamma(A)$ is a model, hence $I \leq \gamma(A)$. Since we are in a Galois connection, $\alpha(I) \leq A$, which proves the theorem. ■

Theorem 1.3.6 *Assume α is complete w.r.t. T and m_I is the least model greater than $I \in \mathcal{I}$. Then, $\alpha(m_I)$ is the least abstract model greater then $\alpha(I)$.*

Proof. Assume A is an abstract model greater than $\alpha(I)$. We need to prove that $A \geq \alpha(m_I)$. By adjoint considerations $A \geq \alpha(I)$ iff $\gamma(A) \geq I$. Hence, $\gamma(A)$ is a model greater than I. By definition of m_I, it turns out that $\gamma(A) \geq m_I$, hence $A \geq \alpha(m_I)$. ■

1.4 Terms and Substitutions

A *signature* Σ is a map from natural numbers to sets. We write "$f/n \in \Sigma$" or "$f \in \Sigma$ of arity n" as an alternative to "$f \in \Sigma(n)$. In this case, f is a *symbol* while n is the *arity* of f. A symbol of arity 0 is a *constant*. Actually, saying "the arity" is a bit of a mistake, since it is possible for the same symbol to have different arities.

Given a signature Σ and a set V of *variables*, disjoint from Σ, we denote by $T_\Sigma(V)$ the set of *terms* over Σ and V, which are define inductively as

- if $v \in V$, then $v \in T_\Sigma(V)$;
- if $t_1, \ldots, t_n \in T_\Sigma(V)$ and $f/n \in \Sigma$, then $f(t_1, \ldots, t_n) \in T_\Sigma(V)$.

Note that in $f(t_1, \ldots, t_n)$, f is not a function. This is just a conventional notation for an ordered tree with a root labeled by f and subtrees $t_1, \ldots, t_n$ (see [21] for other informations on tree). Terms in $T_\Sigma(\emptyset)$ are called *ground*. In the following, we always assume that variables and symbols are disjoint. Given a term t, we denote by $\mathsf{vars}(t)$ the set of variables which occur in t.

Given a set of variables V, a *substitution* (over V) is a partial function $\theta : V \rightharpoonup T_\Sigma(V)$ such that $\{v \in V \mid \theta(v)\downarrow\}$ is finite. We denote by Subst_V the set of all the substitution over V. If t is a term and θ is a substitution, we denote by $t\theta$ the new term we obtain by replacing every occurrence of a variable v, such that $\theta(v)$ is defined, with $\theta(v)$. When, for each $i \in \{1, \ldots, n\}$, v_i is a variable and t_i a term, then $[v_1/t_1, \ldots, v_n/t_n]$ is the substitution θ such that $\theta(v_i) = t_i$ for each i and $\theta(v)\uparrow$ if $v \notin \{v_1, \ldots, v_n\}$.

Given a substitution θ, we define the *domain* of θ,

$$\mathsf{dom}(\theta) = \{v \mid \theta(v)\downarrow\} \ , \tag{1.4.1}$$

and the *range* of θ,

$$\mathsf{range}(\theta) = \bigcup_{v \in \mathsf{dom}(\theta)} \mathsf{vars}(\theta(v)) \ . \tag{1.4.2}$$

Substitutions θ and η can be composed to obtain a new substitution $\theta\eta$ such that

$$\theta\eta(v) = \begin{cases} (v\theta)\eta & \text{if } v \in \mathsf{dom}(\theta), \\ \eta(v) & \text{if } v \in \mathsf{dom}(\eta) \setminus \mathsf{dom}(\theta), \\ \uparrow & \text{otherwise.} \end{cases} \tag{1.4.3}$$

The composition of substitutions is associative. Moreover, for each term t, we have $(t\theta)\eta = t(\theta\eta)$. A substitution θ is *idempotent* when $\theta\theta = \theta$. This is the case if and only if $\mathsf{dom}(\theta) \cap \mathsf{range}(\theta) = \emptyset$.

If V and W are two sets of variables, we denote by Subst_W^V the subset of $\mathsf{Subst}_{V \cup W}$ such that, if $\theta \in \mathsf{Subst}_W^V$, then $\mathsf{dom}(\theta) \subseteq W$ and $\mathsf{range}(\theta) \subseteq V$. If $V \cap W = \emptyset$, all the substitutions in Subst_W^V are idempotent.

1.5 First Order Logic

A *first order signature* is a pair $\langle \Sigma, \Pi \rangle$ of signatures. Elements of Σ are called *function symbols* while elements of Π are called *predicate symbols*. Given a first order signature $\langle \Sigma, \Pi \rangle$ and a set V of variables, we define a *term* as an element of $T_\Sigma(V)$ and an *atomic formula* as an object $p(t_1, \ldots, t_n)$ where $p/n \in \Pi$ and $t_1, \ldots, t_n$ are terms.

Now, assume $W \subseteq V$ is an infinite set. Elements of W are called *bound variables.* The set of *first order formulas*, denoted by $\mathsf{FOF}(\Sigma, \Pi, V, W)$, is defined inductively as:

- an atomic formula is a formula;
- if ϕ and ψ are formulas, then $(\neg\phi)$, $(\psi \wedge \phi)$, $(\psi \vee \phi)$ and $(\psi \supset \phi)$ are formulas;
- if ϕ is a formula and $\xi \in W$, than $(\forall\xi.\phi)$ and $(\exists\xi.\phi)$ are formulas.

We just write FOF when Σ, Π, V and W are clear from the context.

To ease the notation, we often omit the parentheses. To avoid ambiguities, we specify that the $\neg$ operator has the highest priority, followed by the *universal quantifier* $\forall$ and the *existential quantifier* $\exists$, the binary connectives $\wedge$ and $\vee$, and the binary connective $\supset$ at last. We can express the definition of formula in a concise notation as

$$F ::= A \mid F \vee F \mid F \wedge F \mid F \supset F \mid \neg F \mid \forall\xi.F \mid \exists\xi.F \ , \tag{1.5.1}$$

where F is a formula, A an atomic formula and ξ a bound variable. We will use this notation in the rest of this section. In a formula such as $\forall\xi.\phi$ or $\exists\xi.\phi$, the formula ϕ is called the *scope* of the quantifier.

If $\vec{\xi} = \xi_1, \dots, \xi_n$ is a sequence of bound variables, we write $\exists\vec{\xi}.\phi$ or $\exists\xi_1, \dots, \xi_n.\phi$ as a short form for $\exists\xi_1. \dots \exists\xi_n.\phi$. The same holds for the universal quantifier.

Given a $\phi \in \mathsf{FOF}$, we denote by $\mathsf{FV}(\phi)$ the set of *free variables* in ϕ. It is defined inductively on the structure of formulas, as follows:

$$\begin{aligned}
&\mathsf{FV}(p(t_1,\dots,t_n)) = \bigcup_{i=1}^{n} \mathsf{vars}(t_i) \ \ , \\
&\mathsf{FV}(\neg\phi) = \mathsf{FV}(\phi) \ \ , \\
&\mathsf{FV}(\phi \wedge \psi) = \mathsf{FV}(\phi \vee \psi) = \mathsf{FV}(\phi \supset \psi) = \mathsf{FV}(\phi) \cup \mathsf{FV}(\psi) \ \ , \\
&\mathsf{FV}(\exists\xi.\phi) = \mathsf{FV}(\forall\xi.\psi) = \mathsf{FV}(\psi) \setminus \{\xi\} \ \ .
\end{aligned} \tag{1.5.2}$$

Moreover, with $\mathsf{BV}(\phi)$ we denote the set of *bound variables* which occur in ϕ. It is again defined by induction on the structure of formulas:

$$\begin{aligned}
&\mathsf{BV}(p(t_1,\dots,t_n)) = \emptyset \ \ , \\
&\mathsf{BV}(\neg\phi) = \mathsf{BV}(\phi) \ \ , \\
&\mathsf{BV}(\phi \wedge \psi) = \mathsf{BV}(\phi \vee \psi) = \mathsf{BV}(\phi \supset \psi) = \mathsf{BV}(\phi) \cup \mathsf{BV}(\psi) \ \ , \\
&\mathsf{BV}(\exists\xi.\phi) = \mathsf{BV}(\forall\xi.\psi) = \mathsf{BV}(\psi) \cup \{\xi\} \ \ .
\end{aligned} \tag{1.5.3}$$

For example, given $\phi = \forall\xi_1.(p(\xi_1, x) \wedge r(\xi_2))$, it is $\mathsf{FV}(\phi) = \{x, \xi_2\}$ and $\mathsf{BV}(\phi) = \{\xi_1\}$.

We also write $\vec{\forall}\phi$ or $\vec{\exists}\phi$ to denote the *universal closure* or the *existential closure* of ϕ. If $\mathsf{FV}(\pi) \cap W = \{\xi_1, \dots, \xi_n\}$, then $\vec{\forall}\phi = \forall\xi_1, \dots, \xi_n.\phi$ and $\vec{\exists}\phi = \exists\xi_1, \dots, \xi_n.\phi$. Actually, this definition is ambiguous, since we can quantify on the variables of ϕ in different orders. We will be careful to use the notation when this is not relevant.

If ϕ and ψ are two first order formulas, an *occurrence* of ϕ in ψ is a position in ψ where ϕ occurs. If we want to be formal, we can call *position* an element of $\{0,1\}^\star$. Every position p implicitly denotes a partial function $p : \mathsf{FOF} \rightharpoonup \mathsf{FOF}$ which is defined as follows:

$$p(\phi) = \begin{cases} \phi & \text{if } p = \lambda, \\ p'(\phi_1) & \text{if } p = 0 \cdot p' \text{ and } \phi \text{ is one of } \neg\phi_1,\ \phi_1 \wedge \phi_2, \\ & \phi_1 \vee \phi_2,\ \phi_1 \supset \phi_2,\ \forall\xi.\phi_1,\ \exists\xi.\phi_1, \\ p'(\phi_2) & \text{if } p = 1 \cdot p' \text{ and } \phi \text{ is one of } \phi_1 \wedge \phi_2, \\ & \phi_1 \vee \phi_2,\ \phi_1 \supset \phi_2, \\ \uparrow & \text{otherwise.} \end{cases} \tag{1.5.4}$$

An *occurrence* of a formula ϕ in ψ is a position p such that $p(\psi) = \phi$. A *subformula* of ψ is a position p such that $p(\psi)\downarrow$. In the following, we will denote a subformula p of ψ with the formula $p(\psi)$. Although this is an abuse of notation, since it is possible to have different positions p_1 and p_2 such that $p_1(\psi) = p_2(\psi)$, it simplifies the notation. For example, in $\psi = p(a) \wedge p(a)$, there are three subformulas: λ, 0 and 1. It is the case that $\lambda(\psi) = \psi$ and $0(\psi) = 1(\psi) = p(a)$.

Subformulas can be partitioned into *positive* and *negative* ones, according to the following definition:

- ϕ is a positive subformula of ϕ,
- if ϕ is a positive subformula of ψ, then, for every formula ϕ', ϕ is a positive subformula of $\psi \vee \psi'$, $\psi' \vee \psi$, $\psi \wedge \psi'$, $\psi' \wedge \psi$, $\psi' \supset \psi$, $\forall\xi.\psi$, $\exists\xi.\psi$ and a negative subformula of $\neg\psi$ and $\psi \supset \psi'$;
- dually when ϕ is a negative subformula of ψ.

A *binding* in a formula ψ is a subformula ϕ of ψ such that $\phi = \forall\xi.\phi'$ or $\phi = \exists\xi.\phi'$. In this case, we say that ξ is bound by ϕ. A binding is *essentially universal* if it is a positive universal quantifier or a negative existential quantifier. Otherwise, it is said to be *essentially existential*.

A formula ϕ is *well formed* when, for each $\xi \in W$, there is exactly one binding ψ in ϕ which bounds ξ and all the occurrences of ξ in ϕ fall in the scope of ψ. This implies that there are no bound variables which occur out of the scope of any quantifier, i.e $\mathsf{FV}(\phi) \cap W = \emptyset$.

1.5.1 Sequent Calculus

The sequent calculus LK was introduced by Gentzen in [34] as an extension of natural deduction. While in Hilbert-style systems the fundamental components of proofs were formulas, here the same role is played by *sequents*.

A sequent is an object written as

$$\Gamma \twoheadrightarrow \Delta \ , \tag{1.5.5}$$

where Γ and Δ are finite sequences of well formed first order formulas. Intuitively, if $\Gamma = \gamma_1, \ldots, \gamma_n$ and $\Delta = \delta_1, \ldots, \delta_m$, the sequent corresponds to the formula

$$\bigwedge_{i=1}^{n} \gamma_i \supset \bigvee_{j=1}^{m} \delta_j \ . \tag{1.5.6}$$

The left hand side and right hand side of sequent are called *antecedent* and *consequent* respectively. An empty antecedent has the intuitive meaning of true, while an empty consequent means false. The functions FV and BV can be naturally extended to sequents.

A proof in LK is a rooted tree in which nodes are sequents. The root of the tree is called the *endsequent* while the leaves are called *initial sequents* or *axioms*. Each sequent in an LK-proof must be introduced by one of several *inference rules*. An inference rule is denoted by a figure like

$$\frac{S_1 \cdots S_n}{S} \tag{1.5.7}$$

indicating that the sequent S may be inferred from sequents $S_1, \ldots, S_n$. Here, S is the *lower sequent* while each of the S_i's is an *upper sequent*.

The rules for LK are commonly divided in three groups, the *structural rules*, the *propositional rules* and the *quantifier rules*. In Figure 1.1 we show the schemas to obtain all the inference rules for LK. From every schema we obtain an inference rule by instantiating Γ and Δ with finite sequences of well formed formulas, A and B with well formed formulas, t with terms, ξ and v with variables. For the $\forall R$ and the $\exists L$ inference rules, a side condition must be satisfied, too. The variable v which occurs in this condition is called *eigenvariable*. The side condition for the $\forall L$ and $\exists R$ inference rules is required so that $A[\xi/t]$ is well formed. Note that, thanks to our distinction between free and bound variables, when we apply substitution to formulas we do not need to examine all the formula to check for the scope of the quantifiers.

Apart from the standard results of correctness and completeness w.r.t. the semantics of first order logic, the fundamental result of LK is the following:

Theorem 1.5.1 (Haupsatz [34]) *If π is an LK-proof of the sequent S, then S has an LK-proof without cuts.*

Note that the use well formed formulas is not strictly necessary, but properties of sequent calculi can be stated more easily when this happens. For example, the Haupsatz, as expressed in Theorem 1.5.1, does not hold if we consider general formulas. Actually, as pointed out by [29], there is no cut-free proof of the sequent $p(x, y) \twoheadrightarrow \exists x.\exists y.p(y, x)$.

From LK, other calculi can be obtained very easily. The simplest example is the LJ calculus for intuitionistic logic, which is just like LK with the additional restriction that there is at most one formula on the right hand side of the sequents. If we further restrict LJ by forbidding the use of the "*weakeningR*" schema, we obtain a calculus for minimal logic.

1.5.2 Fragments of Intuitionistic Logic

Particular fragments of intuitionistic logic have been extensively studied in the field of logic programming for their computational properties. We start by examining *Horn clauses*. Consider the sublanguages of first order formulas defined by the following rules

$$\begin{aligned} D &::= A \mid G \supset A \mid D \wedge D \mid \forall\xi.D \ , \\ G &::= A \mid G \wedge G \mid G \vee G \mid \exists\xi.G \ , \end{aligned}$$

where A is an atomic formula and ξ a bound variable. Elements of the language generated by D are called *(Horn) clauses*, while elements of G are *goals*. The sequent calculus for Horn clauses is the restriction of LJ to sequents of the form $D_1, \ldots, D_n \twoheadrightarrow G$ where G is a goal and the D_i's are clauses. In particular, a sequence of clauses is called *(definite) logic program*.

Structural rules

$$\dfrac{\Gamma_1, A, B, \Gamma_2 \twoheadrightarrow \Delta}{\Gamma_1, B, A, \Gamma_2 \twoheadrightarrow \Delta}\ \mathit{exchangeL} \qquad \dfrac{\Gamma \twoheadrightarrow \Delta_1, A, B, \Delta_2}{\Gamma \twoheadrightarrow \Delta_1, B, A, \Delta_2}\ \mathit{exchangeR}$$

$$\dfrac{A, A, \Gamma \twoheadrightarrow \Delta}{A, \Gamma \twoheadrightarrow \Delta}\ \mathit{contractionL} \qquad \dfrac{\Gamma \twoheadrightarrow \Delta, A, A}{\Gamma \twoheadrightarrow \Delta, A}\ \mathit{contractionR}$$

$$\dfrac{\Gamma \twoheadrightarrow \Delta}{A, \Gamma \twoheadrightarrow \Delta}\ \mathit{weakeningL} \qquad \dfrac{\Gamma \twoheadrightarrow \Delta}{\Gamma \twoheadrightarrow \Delta, A}\ \mathit{weakeningR}$$

$$\dfrac{\Gamma \twoheadrightarrow \Delta, A \quad \Gamma, A \twoheadrightarrow \Delta}{\Gamma \twoheadrightarrow \Delta}\ \mathit{cut}$$

Propositional rules

$$\dfrac{}{A \twoheadrightarrow A}\ \mathit{id} \text{ where } A \text{ is atomic}$$

$$\dfrac{\Gamma \twoheadrightarrow \Delta, A}{\neg A, \Gamma \twoheadrightarrow \Delta}\ \neg L \qquad \dfrac{A, \Gamma \twoheadrightarrow \Delta}{\Gamma \twoheadrightarrow \Delta, \neg A}\ \neg R$$

$$\dfrac{\Gamma \twoheadrightarrow \Delta, A}{\Gamma \twoheadrightarrow \Delta, A \vee B}\ \vee R_1 \quad \dfrac{\Gamma \twoheadrightarrow \Delta, A}{\Gamma \twoheadrightarrow \Delta, B \vee A}\ \vee R_2 \quad \dfrac{A, \Gamma \twoheadrightarrow \Delta \qquad B, \Gamma \twoheadrightarrow \Delta}{A \vee B, \Gamma \twoheadrightarrow \Delta}\ \vee L$$

$$\dfrac{A, \Gamma \twoheadrightarrow \Delta}{A \wedge B, \Gamma \twoheadrightarrow \Delta}\ \wedge L_1 \quad \dfrac{A, \Gamma \twoheadrightarrow \Delta}{B \wedge A, \Gamma \twoheadrightarrow \Delta}\ \wedge L_2 \quad \dfrac{\Gamma \twoheadrightarrow \Delta, A \quad \Gamma \twoheadrightarrow \Delta, B}{\Gamma \twoheadrightarrow \Delta, A \wedge B}\ \wedge R$$

$$\dfrac{\Gamma \twoheadrightarrow \Delta, A \quad B, \Gamma \twoheadrightarrow \Delta}{A \supset B, \Gamma \twoheadrightarrow \Delta}\ \supset L \qquad \dfrac{A, \Gamma \twoheadrightarrow \Delta, B}{\Gamma \twoheadrightarrow \Delta, A \supset B}\ \supset R$$

Quantifier rules

$$\dfrac{A[\xi/t], \Gamma \twoheadrightarrow \Delta}{\forall \xi.A, \Gamma \twoheadrightarrow \Delta}\ \forall L \qquad \dfrac{\Gamma \twoheadrightarrow \Delta, A[\xi/t]}{\Gamma \twoheadrightarrow \Delta, \exists \xi.A}\ \exists R$$

provided that $\mathsf{vars}(t) \cap W = \emptyset$

$$\dfrac{A[\xi/v], \Gamma \twoheadrightarrow \Delta}{\exists \xi.A, \Gamma \twoheadrightarrow \Delta}\ \exists L \qquad \dfrac{\Gamma \twoheadrightarrow \Delta, A[\xi/v]}{\Gamma \twoheadrightarrow \Delta, \forall \xi.A}\ \forall R$$

provided that $v \in V \setminus W$ does not appear in Γ, Δ and A.

Figure 1.1: Schemas of rules of inference for the LK calculus

Note that, given a `PROLOG` clause `G :- B`, the corresponding clause in this calculus is the universally quantified formula $\vec{\forall}.(B \supset G)$. In the same way, a query `G` for a definite program becomes $\vec{\exists}.G$. Actually, consider the program

```
q(A) :- p(A).
p(X).
```

and the query `q(Y)` which has a valid SLD refutation. The sequent

$$\forall\xi_1.(p(\xi_1) \supset q(\xi_1)), \forall\xi_2.p(\xi_2) \twoheadrightarrow \exists\xi_3.q(\xi_3) \tag{1.5.8}$$

has an obvious proof, but

$$p(x_1) \supset q(x_1), p(x_2) \twoheadrightarrow q(x_3) \tag{1.5.9}$$

is not provable since free variables are never instantiated in LK.

Hereditary Harrop formulas [51] form another sublanguage of first order logic which extends Horn clauses. They are defined by the following rules

$$\begin{aligned} &D ::= A \mid G \supset A \mid D \wedge D \mid \forall\xi.D \;\; , \\ &G ::= A \mid G \wedge G \mid G \vee G \mid D \supset G \mid \exists\xi.G \mid \forall\xi.G \;\; . \end{aligned}$$

The restriction of LK to these formulas is defined in the same way as the restriction for Horn clauses.

Chapter 2

The General Semantic Framework

Abstract

In this chapter, we propose a semantic treatment for sequent calculi similar in spirit to the treatment of Horn clauses in logic programming. We have three different semantic styles (operational, declarative, fixpoint) which agree on the set of all the proofs for a given calculus. Following the guideline of abstract interpretation, it is possible to define abstractions of the concrete semantics. The properties of the abstract semantics will depend on completeness properties of the abstract semantic operators. Part of this chapter has been published in [3].

2.1 Introduction

In the field of logic programming, two main semantic traditions can be easily recognized. One is the proof theoretic one, grown on the idea of isolating fragments of well known logic systems which have good computational properties. Uniform proofs [54] and focusing proofs [4] are two of the main tools used to distinguish between logic without a clear computational meaning and logic programming languages. λProlog [57] and LinLog [4] are examples of languages developed along this line of research.

The other, older tradition, is heavily based on model theory. Logic languages are endowed with a notion of model for a program, and have three different semantics, declarative, operational and fixpoint, which agree on a particular minimal model [47]. Languages such as PROLOG and CLP [40], in all their variants, have their origin in this context.

Moreover, this latter tradition has brought up several important results concerning the effective utilization of Horn clauses as a real programming language. Among the others, problems such as compositionality of semantics [18], modularity [12, 15], static analysis [27], debugging [19], have been tackled in this setting. Adapting these results to the new logic languages developed via the proof theoretic approach would probably require at least two things:

- provide a fixpoint semantics for these new languages;
- generalize a number of concepts whose definition is too much tied to the case of Horn clauses.

In this chapter we propose a semantic framework which can be useful in such an effort. The main idea is to recognize proofs in the sequent calculi as the general counterpart of SLD resolutions for positive logic programs. Thus, the three well-known semantics (operational, declarative and fixpoint) for Horn clause logic can be reformulated within this general setting and directly applied to all the logic languages based on sequent calculi.

These semantics are generalized to be parametric w.r.t. a *pre-interpretation*, which is essentially a choice of semantic domains and intended meanings for the inference rules. When a pre-interpretation is given, we have fixed a particular property of the proofs we want to focus our attention on. Hence, classical abstractions such as correct answers or resultants, used in the semantic studies of logic programs, and abstractions for static analysis like groundness, can be retrieved in terms of properties of proofs. Expressed in such a way, rather than referring to a computational procedure like SLD resolution, they are more easily extendable to other logic languages.

It turns out that the most convenient way of defining pre-interpretations is through abstract interpretation theory [24]. In this way, we provide a semantic framework for the new proof-theoretic based logic languages to which most of the studies we have for positive logic programs can be easily adapted.

After some preliminaries, we introduce the three semantic styles for sequent calculi, with respect to a particular concrete pre-interpretation. It is shown that the three semantics coincide on the set of proofs for a given calculus. Later, using the theory of abstract interpretation, the concept of observable is introduced, as an abstraction of a set of proofs. This gives corresponding notions of abstract semantics for sequent calculi. In general, the properties of the abstract semantics will depend on completeness properties of the abstract optimal semantic operators.

2.2 Proofs and Proof Skeletons

2.2.1 Basic Definitions

We give here a presentation for sequent calculi which is a wide generalization of most of the calculi which have been developed so far from the introduction of Gentzen's LK calculus [34]. Actually, our formalization is so general that a name like tree calculi would probably be more appropriate. However, since we only use the framework in the context of proofs for logic systems, we will stick to the name of sequent calculi.

Definition 2.2.1 (Proof Skeletons) *Given a set Seq of* sequents*, the set* $\mathsf{Sk}(Seq)$ *of* proof skeletons *over Seq is defined inductively as follows:*

- *every* $S \in Seq$ *is a proof skeleton;*
- *if* $S \in Seq$*,* $n \in \mathbb{N}$ *and* T_i *is a proof skeleton for each* $i \leq n$*, then*

$$\frac{T_1 \cdots T_n}{S} \tag{2.2.1}$$

is a proof skeleton, which we also denote by $\mathsf{tree}(S, T_1, \ldots, T_n)$*.*

We write Sk *in the place of* $\mathsf{Sk}(Seq)$ *when Seq is clear from the context.*

We can also define a partial order relation on $\mathsf{Sk}(Seq)$, which is again defined inductively as

- $S \leq S$ for each $S \in Seq$;
- $S \leq \mathsf{tree}(S, T_1, \ldots, T_m)$ for each $S \in Seq$, $m \in N$ and $\{T_i\}_{i \leq m} \subseteq \mathsf{Sk}$;
- $\mathsf{tree}(S, T_1, \ldots, T_n) \leq \mathsf{tree}(S, T_1', \ldots, T_n')$ if $T_i \leq T_i'$ for each $1 \leq i \leq n$.

If π and π' are proof skeletons with $\pi \leq \pi'$, we say that π is a *prefix* of π. Note that, since proof skeletons are finite objects, the set of prefixes for a given proof skeleton π is finite.

We define two auxiliary functions $\mathsf{hyp} : \mathsf{Sk}(Seq) \to Seq^\star$ and $\mathsf{root} : \mathsf{Sk}(Seq) \to Seq$ such that

- $\mathsf{hyp}(S) = S \quad ,$
- $\mathsf{hyp}(\mathsf{tree}(S, T_1, \ldots, T_n)) = \mathsf{hyp}(T_1) \cdots \mathsf{hyp}(T_n) \quad ,$

and

- $\mathsf{root}(S) = S \quad ,$
- $\mathsf{root}(\mathsf{tree}(S, T_1, \ldots, T_n)) = S \quad .$

Given a proof skeleton π, $\mathsf{hyp}(\pi)$ is the sequence of *hypotheses* of π while $\mathsf{root}(\pi)$ is the *root* of π. When we want to state that π is a proof skeleton with $\mathsf{hyp}(\pi) = S_1, \ldots, S_n$ and $\mathsf{root}(\pi) = S$, we write

$$\pi : S_1, \ldots, S_n \vdash S \quad . \tag{2.2.2}$$

We also define the *height* of a proof skeleton π introducing the function $\mathsf{height} : \mathsf{Sk}(Seq) \to \mathbb{N}$ such that

- $\mathsf{height}(S) = 0 \quad ,$

- $\mathsf{height}(\mathsf{tree}(S, T_1, \ldots, T_n)) = \max\{\mathsf{height}(T_1), \ldots, \mathsf{height}(T_n)\} + 1 \quad ,$

with the obvious assumption that $\max(\emptyset) = 0$.

Note that S, which we also denote by ϵ_S, and $\mathsf{tree}(S)$ are two different proof skeletons. Actually, it is $\mathsf{height}(S) = 0$ and $\mathsf{hyp}(S) = S$ but $\mathsf{height}(\mathsf{tree}(S)) = 1$ and $\mathsf{hyp}(\mathsf{tree}(S)) = \lambda$.

Example 2.2.2

If *Seq* is the language of sequents for a first order language, as defined in (1.5.5), with predicate symbols p and r, the following is a proof skeleton

$$\dfrac{\dfrac{p(x) \twoheadrightarrow r(x) \qquad \cdot \twoheadrightarrow r(y)}{\cdot \twoheadrightarrow r(x) \wedge r(y)} \qquad \cdot \twoheadrightarrow \exists\xi_3.p(\xi_3)}{\forall\xi_1.p(\xi_1) \twoheadrightarrow \forall\xi_2.r(\xi_2)} \tag{2.2.3}$$

Note that it is not a proof in any of the standard logical systems. The following

$$\pi : \quad \dfrac{}{p(x) \twoheadrightarrow p(x)} \qquad\qquad \pi' : \quad q(x) \twoheadrightarrow p(x) \tag{2.2.4}$$

are respectively a proof skeleton $\pi : \cdot \vdash p(x) \twoheadrightarrow p(x)$ and the empty proof skeleton $\pi' : q(x) \twoheadrightarrow p(x) \vdash q(x) \twoheadrightarrow p(x) = \epsilon_{q(x) \twoheadrightarrow p(x)}$.

Now, we fix a set $\mathcal{R}$ of proof skeletons of height one. We call *inference rules* the elements of $\mathcal{R}$. A proof skeleton π, which is obtained by pasting together the empty proof skeletons and the inference rules, is called *proof*. A proof with no hypothesis is said to be *final*. A sequent S is *provable* if there is a final proof rooted at S. Finally, we call *sequent calculus* a pair $\langle Seq, \mathcal{R} \rangle$. Sequent calculi will often be denoted by the symbol $\mathcal{C}$ and its variants, while $\mathsf{Prf}(\mathcal{C})$ will be used for the set of proofs in $\mathcal{C}$. We use $\mathcal{L}$ instead of $\mathcal{C}$ when we want to stress the fact that the calculus is the proof system of a well known logic.

Example 2.2.3

In the hypotheses of Example 2.2.2, assume $\mathcal{R}$ is the set of inference rules for first order logic from Figure 1.1. Then π and π' in the previous example are proofs. In particular, π is a final proof. Another proof, a bit more involved, is the following

$$\dfrac{\dfrac{\Gamma \twoheadrightarrow \forall\xi_1.p(\xi_1)}{\Gamma \twoheadrightarrow p(x)}}{\Gamma \twoheadrightarrow \exists\xi_2.p(\xi_2)} \tag{2.2.5}$$

where Γ is a sequence of formulas.

In the following, most of the examples will be based on the LK calculus and its variants. We will denote by $\mathcal{L}_c^{\Sigma,\Pi}$ the calculus obtained by instantiating the framework with the standard language of sequents for first order logics (1.5.5) according to the first order signature $\langle \Sigma, \Pi \rangle$ and with the inference rules obtained as instances of the schemas in Figure 1.1. Here the c in $\mathcal{L}_c$ stands for *classical* while Σ and Π are often omitted when uninteresting or obvious from the context. We also denote by $\mathcal{L}_i$, $\mathcal{L}_m$, $\mathcal{L}_{hc}$ and $\mathcal{L}_{hhf}$ the obvious restrictions of $\mathcal{L}_c$ to intuitionistic logic, minimal logic, Horn clauses and hereditary Harrop formulas.

When we want to refer to a particular inference rule, obtained as an instance of one of the schemas in Figure 1.1, we use the name of the schema, indexed by the actual values for $A, B, \Gamma, \Gamma_1, \Gamma_2, \Delta, \Delta_1, \Delta_2, \xi, v, t$ in this order. We omit one of the values when the corresponding placeholder does not appear in the schema. For example, $id_{p(a)}$ is the rule

$$\frac{}{p(a) \twoheadrightarrow p(a)} \tag{2.2.6}$$

while $\exists R_{p(\xi),\lambda,\lambda,\xi,a}$ is

$$\frac{\cdot \twoheadrightarrow p(a)}{\cdot \twoheadrightarrow \exists \xi . p(\xi)} \tag{2.2.7}$$

Remember that (2.2.6) and (2.2.7) are inference rules, while those in Figure 1.1 are only schemas used to describe, in a finite form, the set of all the inference rules.

2.2.2 Semantic Operators

Given a sequent S, we denote by Sk_S the set of all the proof skeletons rooted at S. For each $\pi \in \mathsf{Sk}_S$,

$$\pi : S_1, \ldots, S_n \vdash S \ , \tag{2.2.8}$$

we have a corresponding semantic operator $\pi : \mathsf{Sk}_{S_1} \times \cdots \times \mathsf{Sk}_{S_n} \to \mathsf{Sk}_S$ which works by gluing proof skeletons of the "input" sequents $S_1, \ldots, S_n$ into π, to obtain a new proof skeleton of the "output" sequent S. Formally, if $\pi = S$ and $\pi' \in \mathsf{Sk}_S$, then

$$\pi(\pi') = \pi' \ . \tag{2.2.9}$$

Otherwise, if $\pi = \mathsf{tree}(S, \pi_1, \ldots, \pi_n)$, $\mathsf{hyp}(\pi_i) = S_{i,1}, \ldots, S_{i,r_i}$ and $m_i = \sum_{k=1}^{i} r_k$ for each $i \leq n$ and $m = m_n$, then

$$\pi(\pi'_1, \ldots, \pi'_m) = \mathsf{tree}(S, \pi_1(\pi'_1, \ldots, \pi'_{m_1}), \ldots, \pi_n(\pi'_{m_{n-1}+1}, \ldots, \pi'_{m_n})) \ . \tag{2.2.10}$$

Note that π is injective. We can prove the following:

Theorem 2.2.4 *Given a proof skeleton $\pi : S_1, \ldots, S_n \vdash S$ and π', then π is a prefix of π' iff there are $\pi_1, \ldots, \pi_n$ such that $\pi' = \pi(\pi_1, \ldots, \pi_n)$.*

Proof. Assume there are $\pi_1, \ldots, \pi_n$ such that $\pi' = \pi(\pi_1, \ldots, \pi_n)$. Then, by induction on π, if $\pi = S$ for $S \in \mathit{Seq}$, then $n = 1$ and $\pi' = \pi_1$. Since $\mathsf{root}(\pi') = S$, then $\pi \leq \pi'$ trivially. If $\pi = \mathsf{tree}(S, \pi_1, \ldots, \pi_m)$, then

$$\pi' = \mathsf{tree}(S, \pi_1(\pi'_{1,1}, \ldots, \pi'_{1,k_1}), \ldots, \pi_m(\pi'_{m,1}, \ldots, \pi'_{m,k_m})) \ . \tag{2.2.11}$$

By inductive hypothesis,

$$\pi_i \leq \pi_i(\pi_{i,1}, \ldots, \pi'_{i,k_i}) \ , \tag{2.2.12}$$

whence $\pi \leq \pi'$.

With respect to the opposite implication, assume $\pi \leq \pi'$. We proceed by induction on the structure of π. If $\pi = S \in \mathit{Seq}$, then $\pi' = \pi(\pi')$. Otherwise, if $\pi = \mathsf{tree}(S, \pi_1, \ldots, \pi_n)$, then $\pi' = \mathsf{tree}(S, \pi'_1, \ldots, \pi'_n)$ with $\pi_i \leq \pi'_i$ for each $1 \leq i \leq n$. By inductive hypothesis, $\pi'_i = \pi_i(\pi_{i,1}, \ldots, \pi_{i,m_i})$ for each i. Then, it easy to check that

$$\pi' = \pi(\pi_{1,1}, \ldots, \pi_{1,m_1}, \ldots, \pi_{n,1}, \ldots, \pi_{n,m_n}) \tag{2.2.13}$$

and this proves the theorem. ■

Example 2.2.5

Consider the proof π in $\mathcal{L}_{hc}$ given by

$$\frac{\forall\xi.p(\xi) \twoheadrightarrow p(a) \qquad \forall\xi.p(\xi) \twoheadrightarrow r(b)}{\forall\xi.p(\xi) \twoheadrightarrow p(a) \land r(b)} \tag{2.2.14}$$

and the proof' π'

$$\frac{\overline{p(a) \twoheadrightarrow p(a)}}{\forall\xi.p(\xi) \twoheadrightarrow p(a)} \tag{2.2.15}$$

and $\pi'' = \epsilon_{\forall\xi.p(\xi) \twoheadrightarrow r(b)}$. Then, the proof $\pi(\pi', \pi'')$ is

$$\frac{\dfrac{\overline{p(a) \twoheadrightarrow p(a)}}{\forall\xi.p(\xi) \twoheadrightarrow p(a)} \qquad \forall\xi.p(\xi) \twoheadrightarrow r(b)}{\forall\xi.p(\xi) \twoheadrightarrow p(a) \land r(b)} \tag{2.2.16}$$

In particular, note that gluing with empty proof has no effects.

If $\pi : S_1, \ldots, S_n \vdash S \in \mathsf{Sk}$ and $X_i \subseteq \mathsf{Sk}$ for each i, we introduce a collecting variant of the semantic operator π, defined as

$$\pi(X_1, \ldots, X_n) = \{\pi(\pi_1, \ldots, \pi_n) \mid \text{ for each } i,\ \pi_i \in X_i \cap \mathsf{Sk}_{S_i}\} \ . \tag{2.2.17}$$

All these operators are both additive and co-additive. We will write $\bar{\pi}(X)$ as a short form for $\pi(X, \ldots, X)$ with n identical copies of X as the input arguments. Note that $\bar{\pi}$ is not additive; however, the following holds.

Theorem 2.2.6 *Given a proof schema $\pi : S_1, \ldots, S_n \vdash S$, the semantic operator $\bar{\pi}$ is continuous and co-additive.*

Proof. Let $\{X_j\}_{j \in J}$ be a chain of subsets of Sk. Consider a proof schema $\pi' \in \bar{\pi}(\bigcup_{j \in J} X_j)$. It is

$$\pi' = \pi(\pi_1, \ldots, \pi_n) \tag{2.2.18}$$

where $\pi_i \in (\cup_{j \in J} X_j) \cap \mathsf{Sk}_{S_i} = \cup_{j \in J}(X_j \cap \mathsf{Sk}_{S_i})$ for each i. Since $\{X_j\}_{j \in J}$ is a chain, there exists an $n \in J$ such that $\pi_i \in X_n \cap \mathsf{Sk}_{S_i}$ for each i. As a result, $\pi \in \bar{\pi}(X_n) \subseteq \bigcup_{j \in J} \bar{\pi}(X_j)$.

Now, take a collection $\{X_j\}_{j \in J}$ of interpretations and $\pi' \in \bigcap_{j \in J} \bar{\pi}(X_j)$. Since $\pi(\pi_1, \ldots \pi_n)$ is different from $\pi(\pi'_1, \ldots, \pi'_n)$ if there is at least an index i such that $\pi_i \neq \pi'_i$, then $\pi' = \pi(\pi_1, \ldots, \pi_n)$ with $\pi_i \in X_j \cap \mathsf{Sk}_{S_i}$ for all j and i. Hence, $\pi' \in \bar{\pi}(\cap_{j \in J} X_j)$. ■

Working with a semantic operator for each proof skeleton can be uncomfortable, especially when reasoning in terms of abstractions. We can actually resort to a unique *gluing operator*. Given X_1 and X_2 subsets of Sk, we denote by $X_1 \rhd X_2$ the set

$$X_1 \rhd X_2 = \bigcup_{\pi \in X_1} \bar{\pi}(X_2) \ . \tag{2.2.19}$$

In other words, $X_1 \rhd X_2$ is the result of gluing together each proof skeleton in X_1 above all the "compatible" proof skeletons in X_2. It turns out that $\rhd$ is (roughly) the counterpart for sequent calculi of the $\bowtie$ operator for SLD derivations defined in [20]. It enjoys the following properties:

Theorem 2.2.7 *The glue operator $\rhd$ is*

- *additive and co-continuous w.r.t. its first argument,*
- *continuous and co-additive w.r.t. its second argument.*

Proof. The proof that $\rhd$ is left-additive easily follows by its definition. Moreover, it is co-continuous w.r.t. its first argument. We just need to prove that

$$\left(\bigcap_{j \in J} X_j \right) \rhd Y \supseteq \bigcap_{j \in J} (X_j \rhd Y) \ , \tag{2.2.20}$$

for each chain $\{X_j\}_{j \in J} \subseteq \wp(\mathsf{Sk})$. Given a proof skeleton π, let us consider the set X of prefixes of π, which we know to be finite. Now, assume $\pi \in \cap_{j \in J}(X_j \rhd Y)$ but $\pi \notin (\cap_{j \in J} X_j) \rhd Y$. By Theorem 2.2.4, this means that $(\cap_{j \in J} X_j) \cap X = \emptyset$. In other words, for each $\pi' \in X$, there is a $j_{\pi'}$ such that $\pi' \notin X_{j_{\pi'}}$. However, since X is finite, there is $i \in J$ such that $X_i = \cap_{\pi' \in X} X_{j_{\pi'}}$. It happens that $X_i \cap X = \emptyset$, and then $\pi \notin X_i \rhd Y$, which is an absurd. Therefore, it must be $\pi \in (\cap_{j \in J} X_j) \rhd Y$. With respect to the second argument, the continuity and coadditivity of $\rhd$ directly follows from the fact that, given $\pi \in \mathsf{Sk}$, $\bar{\pi}$ is continuous and co-additive (see Theorem 2.2.6). ■

2.3 Semantics for Sequent Calculi

Given a sequent calculus $\langle \mathit{Seq}, \mathcal{R} \rangle$, we can introduce three different styles of semantics, similar in spirit to the operational, declarative and fixpoint semantics of classic logic programming. We follow the idea underlying [20] of having a common set of semantic operators for both the top-down (operational) and the bottom-up (fixpoint) styles.

2.3.1 Declarative Semantics

The fundamental idea is that a sequent calculus can be viewed as a signature for Σ-algebras, where sequents correspond to sorts and inference rules to term symbols. A Σ-algebra gives a choice of a semantic domain for each sequent and of a semantic function for each inference rule. Roughly, a model for a sequent calculus in a given Σ-algebra should assign, to each sequent, an element of its corresponding semantic domain, in such a way that this assignment is well-behaved w.r.t. the inference rules.

To be more precise, we call *pre-interpretation* the choice of a nonempty ordered set $\mathcal{I}(S)$ for each sequent S and of a monotonic function $\mathcal{I}(r)$ for each inference rule r, where if $\mathsf{hyp}(r) = S_1, \ldots, S_n$ and $\mathsf{root}(r) = S$, then

$$\mathcal{I}(r) : \mathcal{I}(S_1) \times \cdots \times \mathcal{I}(S_n) \to \mathcal{I}(S) \ . \tag{2.3.1}$$

Therefore, the concept of *pre-interpretation* is the same of ordered Σ-algebras as defined in [55]. We call *pre-interpreted (sequent) calculus* a triple $\langle \mathit{Seq}, \mathcal{R}, \mathcal{I} \rangle$.

Given a pre-interpretation $\mathcal{I}$, an *interpretation* is a choice of an element $[\![S]\!] \in \mathcal{I}(S)$ for each sequent S. Interpretations are ordered pointwise. An interpretation is called a *model* when, for each inference rule

$$r : S_1, \ldots, S_n \vdash S \ , \tag{2.3.2}$$

the following relation holds

$$\mathcal{I}(r)([\![S_1]\!], \ldots, [\![S_n]\!]) \leq [\![S]\!] \ . \tag{2.3.3}$$

Note that models are closed under arbitrary meets.

Theorem 2.3.1 *If $[\![\_]\!]_j$ for $j \in J$ are models and $[\![\_]\!] = \bigsqcap_{j \in J} [\![\_]\!]_j$ does exists, then $[\![\_]\!]$ is a model, too.*

Proof. Given an inference rule $r : S_1, \ldots, S_n \vdash S$, we have

$$\mathcal{I}(r)([\![S_1]\!], \ldots, [\![S_n]\!]) \leq \mathcal{I}(r)([\![S_1]\!]_j, \ldots, [\![S_n]\!]_j) \leq [\![S]\!]_j$$

for each $j \in J$, whence

$$\mathcal{I}(r)([\![S_1]\!], \ldots, [\![S_n]\!]) \leq \bigsqcap_{j \in J} [\![S]\!]_j$$

which proves that $[\![\_]\!]$ is a model. ■

The notion of pre-interpretation gives us a great flexibility. By a careful choice of $\mathcal{I}$, it is possible to define a lot of different semantics for the same calculus.

Example 2.3.2

Given a calculus $\mathcal{C}$, consider the pre-interpretation $\mathcal{I}_s$ defined as follows.

- $\mathcal{I}_s(S)$ is the set $\{\mathsf{false}, \mathsf{true}\}$ with $\mathsf{false} \leq \mathsf{true}$;
- if $r \in \mathcal{R}$ is the inference rule $r : S_1, \ldots, S_n \vdash S$, then $\mathcal{I}_s(r)$ is the logical conjunction of the n input values. If r has no hypothesis, then $\mathcal{I}_s(r) = \mathsf{true}$.

There exists a least model $[\![\_]\!]_s$ for $\mathcal{I}_s(r)$, given by

$$[\![S]\!]_s = \begin{cases} \mathsf{true} & \text{if } S \text{ is provable,} \\ \mathsf{false} & \text{otherwise.} \end{cases} \tag{2.3.4}$$

We call $\mathcal{I}_s$ the pre-interpretation of *success sets*.

When we work in the field of logic programming, the idea is that a program P corresponds to a sequence of formulas. Given a goal G and a model $[\![\_]\!]$, the corresponding semantics of G in the program P is given by $[\![P \twoheadrightarrow G]\!]$.

Example 2.3.3

In the calculus $\mathcal{L}_{hc}$, consider the pre-interpretation $\mathcal{I}_s$ of success sets. If P is a definite logic program and $[\![\_]\!]$ is an interpretation, the set

$$I_P = \{A \mid [\![P \twoheadrightarrow A]\!] = \mathsf{true} \text{ and } A \text{ is a ground atomic goal}\} \tag{2.3.5}$$

is a Herbrand interpretation. Moreover, if $[\![\_]\!]$ is a model, I_P is an Herbrand model.

2.3.2 Fixpoint Semantics

The definition of the declarative semantics is non-constructive. We now present a bottom-up construction of the least model, when it does exist, using an operator similar to the immediate consequence operator T_P of ordinary logic programming. The idea is that, given a pre-interpreted calculus $\langle Seq, \mathcal{R}, \mathcal{I}\rangle$ where $\mathcal{I}(S)$ is a complete lattice for each sequent S, we can build a model by using a fixpoint construction. The new operator $T_{\mathcal{R}}$ takes interpretations to interpretations, according to the following definition

$$T_{\mathcal{R}}([\![\_]\!])(S) = [\![S]\!] \sqcup \bigsqcup_{r:S_1,\ldots,S_n \vdash S \in \mathcal{R}} \mathcal{I}(r)([\![S_1]\!], \ldots, [\![S_n]\!]) \ . \tag{2.3.6}$$

We can prove that all the common results which hold for the T_P operator, apply to $T_{\mathcal{R}}$ as well. In particular,

Theorem 2.3.4 *An interpretation* $\llbracket \_ \rrbracket$ *is a model iff it is a pre-fixpoint of* $T_{\mathcal{R}}$.

Proof. Given a sequent S,

$$\begin{aligned}
& \llbracket S \rrbracket \geq T_{\mathcal{R}}(\llbracket \_ \rrbracket)(S) \\
\iff & \llbracket S \rrbracket \geq \llbracket S \rrbracket \sqcup \bigsqcup_{r:S_1,\ldots,S_n \vdash S \in \mathcal{R}} \mathcal{I}(r)(\llbracket S_1 \rrbracket, \ldots, \llbracket S_n \rrbracket) \\
\iff & \llbracket S \rrbracket \geq \mathcal{I}(r)(\llbracket S_1 \rrbracket, \ldots, \llbracket S_n \rrbracket) \\
& \quad \text{for each } r : S_1, \ldots, S_n \vdash S \in \mathcal{R}
\end{aligned} \tag{2.3.7}$$

Since this holds for every S, the result follows trivially. ■

Theorem 2.3.5 *If* $\mathcal{I}(r)$ *is continuous (additive) for each inference rule* r*, then* $T_{\mathcal{R}}$ *is continuous (additive), too.*

Proof. Trivial, since $T_{\mathcal{R}}$ is the obtained by composition of continuous (additive) operators. ■

Thus, assuming continuity of the pre-interpretation of inference rules, we have that $T_{\mathcal{R}} \uparrow \omega$, i.e. the least fixpoint of $T_{\mathcal{R}}$, is the least model of the calculus. In general, $T_{\mathcal{R}}^{\omega}(\llbracket \_ \rrbracket)$ will be the least model greater than $\llbracket \_ \rrbracket$.

2.3.3 Models and proofs

There is another standard way of building a model starting from proofs. Assume $\langle Seq, \mathcal{R}, \mathcal{I} \rangle$ is a pre-interpreted calculus, $\llbracket \_ \rrbracket$ is an interpretation and π a proof of the sequent S. We can inductively define a new element $\pi(\llbracket \_ \rrbracket) \in \mathcal{I}(S)$ as follows:

$$\begin{aligned}
\epsilon_S(\llbracket \_ \rrbracket) &= \llbracket S \rrbracket \quad , \\
r(\pi_1, \ldots, \pi_n)(\llbracket \_ \rrbracket) &= \mathcal{I}(r)(\pi_1(\llbracket \_ \rrbracket), \ldots, \pi_n(\llbracket \_ \rrbracket)) \quad .
\end{aligned} \tag{2.3.8}$$

If Prf is the set of proofs for the calculus we can define a model $\mathsf{Prf}_{\llbracket \_ \rrbracket}$ as

$$\mathsf{Prf}_{\llbracket \_ \rrbracket}(S) = \bigsqcup \{ \pi(\llbracket \_ \rrbracket) \mid \pi \in \mathsf{Prf}_S \} \tag{2.3.9}$$

We have the following

Theorem 2.3.6 *If* $\mathcal{I}(r)$ *is additive for each* $r \in R$*, then* $\mathsf{Prf}_{\llbracket \_ \rrbracket}$ *is the least model greater than* $\llbracket \_ \rrbracket$.

Proof. First of all, we prove that $\mathsf{Prf}_{\llbracket \_ \rrbracket}$ is a model. Given an inference rule $r : S_1, \ldots, S_n \vdash S$, we have

$$\mathsf{Prf}_{\llbracket \_ \rrbracket}(S) \geq \bigsqcup \{ \pi(\llbracket \_ \rrbracket) \mid \pi = r(\pi_1, \ldots, \pi_n) \in \mathsf{Prf}_S \}$$

$$
\begin{aligned}
&= \bigsqcup\{\mathcal{I}(r)(\pi_1(\llbracket\_\rrbracket), \ldots, \pi_1(\llbracket\_\rrbracket)) \mid r(\pi_1, \ldots, \pi_n) \in \mathsf{Prf}_S\} \\
&\geq \mathcal{I}(r)\left(\bigsqcup\{\pi_1(\llbracket\_\rrbracket) \mid \pi_1 \in \mathsf{Prf}_{S_1}\}, \ldots, \bigsqcup\{\pi_n(\llbracket\_\rrbracket) \mid \pi_n \in \mathsf{Prf}_{S_n}\}\right) \\
&= \mathcal{I}(r)(\mathsf{Prf}_{\llbracket\_\rrbracket}(S_1), \ldots, \mathsf{Prf}_{\llbracket\_\rrbracket}(S_n)) \ .
\end{aligned}
$$

Now, assume $\llbracket\_\rrbracket'$ is another model with $\llbracket\_\rrbracket' \geq \llbracket\_\rrbracket$. Given a proof π of the sequent S, we prove that $\llbracket S\rrbracket' \geq \pi(\llbracket\_\rrbracket)$, which leads to the result that $\llbracket\_\rrbracket' \geq \mathsf{Prf}_{\llbracket\_\rrbracket}$. The proof proceeds by induction on the structure of π. For the base case, if $\pi = \epsilon_S$, then $\pi(\llbracket\_\rrbracket) = \llbracket S\rrbracket$ and the desired property directly descends from the hypotheses. If $\pi = r(\pi_1, \ldots, \pi_n)$, since $\llbracket\_\rrbracket'$ is a model, we have

$$
\llbracket S\rrbracket' \geq \mathcal{I}(r)(\llbracket S_1\rrbracket', \ldots, \llbracket S_n\rrbracket') \ , \tag{2.3.10}
$$

where $r : S_1, \ldots, S_n \vdash S$. By inductive hypotheses, it is $\llbracket S_i\rrbracket' \geq \pi_i(\llbracket\_\rrbracket)$ for each $i \in \{1, \ldots, n\}$. Hence

$$
\begin{aligned}
\llbracket S\rrbracket' &\geq \mathcal{I}(r)(\llbracket S_1\rrbracket', \ldots, \llbracket S_n\rrbracket') \\
&\geq \mathcal{I}(r)(\pi_1(\llbracket\_\rrbracket , \ldots, \pi_n(\llbracket\_\rrbracket))) \\
&= \pi(\llbracket\_\rrbracket) \ ,
\end{aligned}
$$

and this proves the theorem. ■

2.4 The Syntactical Pre-Interpretation

A major drawback of this approach is that the process of defining a pre-interpretation is quite arbitrary, especially for what concerns the inference rules. In the following, we try to overcome this problem by just sticking to a specific concrete pre-interpretation and deriving all the others by abstraction functions, according to the theory of abstract interpretation.

Definition 2.4.1 (Syntactical pre-interpretation) *Given a calculus $\mathcal{C}$, the* syntactical *pre-interpretation $\mathcal{I}_{\mathcal{C}}$ is given by*

- *$\mathcal{I}_{\mathcal{C}}(S) = \langle \wp(\mathsf{Sk}_S), \subseteq\rangle$ for each sequent S;*
- *$\mathcal{I}_{\mathcal{C}}(r)$ is the semantic function corresponding to $r \in \mathcal{R}$, as in* (2.2.17).

Interpretations for $\mathcal{I}_{\mathcal{C}}$ are called *syntactical interpretations*. In the following, these will be denoted by subsets of Sk. The convention does not rise any ambiguities, since if $S_1 \neq S_2$, then $\mathsf{Sk}_{S_1} \cap \mathsf{Sk}_{S_2} = \emptyset$. A syntactical model, therefore, is a set of proof skeletons closed under application of inference rules. We denote by Int the set of all the syntactical interpretations, which is a complete lattice under subset ordering. In the remaining of this section, when we talk of interpretations or models we always refer to the syntactical ones, unless otherwise stated.

It is possible to concisely express the condition of a syntactical interpretation I being a model using the glue operator. We have the following

Theorem 2.4.2 *An interpretation I is a model iff*

$$\mathcal{R} \rhd I \subseteq I \ . \tag{2.4.1}$$

Proof. First of all, we prove that if $\mathcal{R} \rhd I \subseteq I$, then I is a model. Given an inference rule $r : S_1, \ldots, S_n \vdash S$, we have

$$\begin{aligned}
\mathcal{I}_{\mathcal{C}}(r)(I(S_1), \ldots, I(S_n)) &= r(I \cap \mathsf{Sk}_{S_1}, \ldots, I \cap \mathsf{Sk}_{S_n}) \\
&= \bar{r}(I) && [\,\text{by } (2.2.17)\,] \\
&\subseteq (\mathcal{R} \rhd I) \cap \mathsf{Sk}_S \\
&\subseteq I \cap \mathsf{Sk}_S \\
&= I(S) \ .
\end{aligned}$$

On the contrary, if $\mathcal{I}_{\mathcal{C}}(r)(I(S_1), \ldots, I(S_n)) \subseteq I(S)$ for each $r \in \mathcal{R}$, then

$$\begin{aligned}
\mathcal{R} \rhd I &= \bigcup_{r:S_1,\ldots,S_n \vdash S \in \mathcal{R}} \bar{r}(I) \\
&= \bigcup_{r:S_1,\ldots,S_n \vdash S \in \mathcal{R}} r(I \cap \mathsf{Sk}_{S_1}, \ldots, I \cap \mathsf{Sk}_{S_n}) \\
&\subseteq \bigcup_{r:S_1,\ldots,S_n \vdash S \in \mathcal{R}} I \cap \mathsf{Sk}_S \\
&\subseteq I \ ,
\end{aligned}$$

and this proves the theorem. ■

By Theorem 2.3.1 and since $\mathsf{Sk}(Seq)$ is a model for every calculus, it follows that syntactical models form a complete sub-meet-semilattice of Int. However, it is not a sublattice, since the join operator and the bottom element differ. In particular,

Definition 2.4.3 (Declarative semantics) *Given a calculus $\mathcal{C}$, the least element of the lattice of syntactical models is called* declarative semantics *of $\mathcal{C}$ and will be denoted by $\mathcal{D}(\mathcal{C})$.*

By applying Theorem 2.3.6 to $\mathcal{I}_{\mathcal{C}}$ starting with the empty interpretation, it turns out that $\mathcal{D}(\mathcal{C})$ is the set of final proofs of $\mathcal{C}$. Hence, the declarative semantics precisely captures all the terminating computations. For a valid treatment of compositionality, we also need information about partial computations [12]. If ϵ is the set of all the empty proof skeletons, we call *complete declarative semantics* of $\mathcal{C}$, and we denote it by $\mathcal{D}_c(\mathcal{C})$, the least model greater then ϵ. As before, it is easy to check that $\mathcal{D}_c(\mathcal{C})$ is the set of all the proofs of $\mathcal{C}$.

2.4.1 Top-Down and Bottom-Up Semantics

The $T_{\mathcal{R}}$ operator w.r.t. the syntactical pre-interpretation $\mathcal{I}_{\mathcal{C}}$ will be denoted by $T_{\mathcal{C}}$. Using the isomorphism between syntactical interpretations and subsets of Sk, the definition can be simplified as

$$T_{\mathcal{C}}(I) = I \cup (\mathcal{R} \rhd I) \ . \tag{2.4.2}$$

An interpretation I is a model iff it is a pre-fixpoint of $T_{\mathcal{C}}$ (by Theorem 2.3.4). Moreover $T_{\mathcal{C}}$ is continuous (by Theorem 2.2.7), hence $T_{\mathcal{C}} \uparrow \omega$ is its least fixpoint. We call $T_{\mathcal{C}} \uparrow \omega$ the *fixpoint semantics* of $\mathcal{C}$. It trivially follows that the fixpoint and declarative semantics do coincide. Analogously to the complete declarative semantics, we can define a *complete fixpoint semantics* as $T_{\mathcal{C}}^{\omega}(\epsilon)$. As in the previous case, $T_{\mathcal{C}}^{\omega}(\epsilon) = \mathcal{D}_c(\mathcal{C})$. Note that inference rules are essentially treated like Horn clauses for a predicate $\texttt{is\_a\_proof}/1$. For example, an inference rule like

$$\frac{\Gamma \twoheadrightarrow \phi \quad \Gamma \twoheadrightarrow \psi}{\Gamma \twoheadrightarrow \phi \wedge \psi} \tag{2.4.3}$$

corresponds to the Horn clause

$$\begin{aligned} &\texttt{is\_a\_proof}(\texttt{der}(\Gamma \twoheadrightarrow \phi \wedge \psi, [P_1, P_2])) :- \\ &\qquad P_1 = \texttt{der}(\Gamma \twoheadrightarrow \phi, \_), P_2 = \texttt{der}(\Gamma \twoheadrightarrow \psi, \_), \\ &\qquad \texttt{is\_a\_proof}(P_1),\ \texttt{is\_a\_proof}(P_2) \end{aligned} \tag{2.4.4}$$

where $\texttt{der}(\mathit{Sequent}, \mathit{List\_of\_Proof\_Skeletons})$ is a coding for proof skeletons. In general, we have an infinite set of ground Horn clauses, since every instance of (2.4.3) counts as a different inference rule and variables in the calculus $\mathcal{C}$ are coded as ground objects at the Horn clause level.

The fixpoint construction is essentially a bottom-up process. Real interpreters, on the contrary, follow a top-down approach, since it is generally more efficient. We consider here a transition system $(\mathsf{Sk}, \overset{\mathcal{R}}{\longmapsto})$ which emulates such a behavior. Assume $\pi : S_1, \ldots, S_n \vdash S$ is a proof schema and $r : S'_1, \ldots, S'_m \vdash S_i$ is an inference rule. We can define a new proof schema $\pi' = \pi(\epsilon_{S_1}, \ldots, r, \ldots, \epsilon_{S_n})$ just replacing S_i in the hypotheses of π with the inference rule r. We write $\pi \overset{\mathcal{R}}{\longmapsto} \pi'$ when the above conditions are satisfied. In general, it is possible to replace more than one hypothesis, hence we have the following transition rule

$$\pi \overset{\mathcal{R}}{\longmapsto} \pi(r_1, \ldots, r_n) \text{ when } \begin{cases} \pi : S_1, \ldots, S_n \vdash S, \\ r_i \in \mathcal{R} \cup \epsilon \text{ and } \mathsf{root}(r_i) = S_i \text{ for each } 1 \leq i \leq n. \end{cases} \tag{2.4.5}$$

We call *complete operational semantics* of $\mathcal{C}$ the interpretation

$$\mathcal{O}_c(\mathcal{C}) = \{\pi \in \mathsf{Sk} \mid \exists S.\ \epsilon_S \overset{\mathcal{R}}{\longmapsto}^* \pi\} \ . \tag{2.4.6}$$

It is possible to give a collecting variant of the operational semantics construction, via a fixpoint operator $U_{\mathcal{C}}$ on interpretations which uses the gluing semantic operator:

$$U_{\mathcal{C}}(I) = I \rhd (\mathcal{R} \cup \epsilon) \ . \tag{2.4.7}$$

The idea is that $U_{\mathcal{C}}(I)$ contains all the proof schemas derived by I with a step of the transition system, i.e.

$$U_{\mathcal{C}}(I) = \{\pi' \in \mathsf{Sk} \mid \exists \pi \in I .\ \pi \xmapsto{\mathcal{R}} \pi'\} \ . \tag{2.4.8}$$

Actually, the following property can be proved:

Lemma 2.4.4 *$U^n_{\mathcal{C}}(X)$ is the set of all the proof schemas derivable from X in n steps, i.e.*

$$U^n_{\mathcal{C}}(X) = \{\pi \in \mathsf{Sk} \mid \exists \pi' \in X .\ \pi' \xmapsto{\mathcal{R}}{}^n \pi\} \ . \tag{2.4.9}$$

Proof. The proof proceeds by induction on n. For $n = 0$, the property (2.4.9) trivially holds. Now, assume (3.1.1) holds for n and let us prove it for $n+1$. Given π derivable in $n+1$ steps from X, there exists π' derivable in n steps such that $\pi \xmapsto{\mathcal{R}} \pi'$. By inductive hypothesis, $\pi' \in U^n_{\mathcal{C}}(X)$ and $\pi \in \pi'(\mathcal{R} \cup \epsilon) \subseteq U^{n+1}_{\mathcal{C}}(X)$. On the other side, if $\pi \in U^{n+1}_{\mathcal{C}}(X)$, then $\pi \in \pi'(\mathcal{R} \cup \epsilon)$ with $\pi' \in U^n_{\mathcal{C}}(X)$. Since $\pi'(\mathcal{R} \cup \epsilon) = \{\pi \mid \pi' \xmapsto{\mathcal{R}} \pi\}$, by inductive hypotheses, π is derivable from π' in $n+1$ steps. ■

Since $U_{\mathcal{C}}$ is continuous, being the composition of continuous operators, it is possible to derive the least fixpoint greater than ϵ as $U^{\omega}_{\mathcal{C}}(\epsilon)$. It turns out that

$$\mathcal{D}_c(\mathcal{C}) = U^{\omega}_{\mathcal{C}}(\epsilon) = \mathcal{O}_c(\mathcal{C}) \ . \tag{2.4.10}$$

Theorem 2.4.5 *Given a sequent calculus $\mathcal{C}$, the identities in* (2.4.10) *hold.*

Proof. The equality between $U^{\omega}_{\mathcal{C}}(\epsilon)$ and $\mathcal{O}_c(\mathcal{C})$ is a consequence of Lemma 2.4.4. ■

From the implementation viewpoint, the great advantage of the top-down operational semantic w.r.t. the bottom-up fixpoint one is that we do not need to compute the entire semantics if we are only interested in part of it. An interpreter for a logic language typically works with a program P and a goal G, trying to obtain the proofs of the sequent $P \twoheadrightarrow G$. The semantics of every other sequent in the logic is computed only if it is needed for computing the semantics of $P \twoheadrightarrow G$.

We call *query* whatever sequent in the calculus $\mathcal{C}$. We define the *operational behavior* of $\mathcal{C}$ as a function $\mathcal{B}(\mathcal{C}) : \mathit{Seq} \to \mathsf{Int}$ such that

$$\mathcal{B}(\mathcal{C})_S = \{\pi \in \mathsf{Sk} \mid \epsilon_S \xmapsto{\mathcal{R}}{}^* \pi\} \ . \tag{2.4.11}$$

In other words, $\mathcal{B}(\mathcal{C})_S$ is the set of proofs for the sequent S in the calculus $\mathcal{C}$. The fixpoint operator $U_{\mathcal{C}}$ can be used to compute $\mathcal{B}(\mathcal{C})$ since it is $\mathcal{B}(\mathcal{C})_S = U_{\mathcal{C}}^{\omega}(\{\epsilon_S\})$, as shown in Lemma 2.4.4.

There is an immediate result of compositionality for $\mathcal{B}$. For each sequent S, consider the set $R = \{r_i\}_{i \in I}$ of all the inference rules rooted at S, such that $r_i : S_{i,1}, \ldots, S_{i,m_i} \vdash S$. We have

$$\mathcal{B}(\mathcal{C})_S = \bigcup_{i \in I} r_i \left(\mathcal{B}(\mathcal{C})_{S_{i,1}}, \ldots, \mathcal{B}(\mathcal{C})_{S_{i,m_i}}\right) \quad . \tag{2.4.12}$$

Unfortunately, this result is not what we desire in most of the cases, as shown by the following example.

Example 2.4.6

When we work in $\mathcal{L}_{hc}$, the above compositionality result gives us the following property:

$$\mathcal{B}(\mathcal{L}_{hc})_{P \twoheadrightarrow G_1 \wedge G_2} = \mathcal{B}(\mathcal{L}_{hc})_{P \twoheadrightarrow G_1} \cap \mathcal{B}(\mathcal{L}_{hc})_{P \twoheadrightarrow G_2} \quad . \tag{2.4.13}$$

However, the classical result of and-compositionality for definite logic programs (w.r.t. correct answers or other observables) says that the semantics of $\mathtt{G_1} \wedge \mathtt{G_2}$ can be derived from the semantics of $\mathtt{G_1}$ and $\mathtt{G_2}$. Since goals in definite programs become existentially quantified in our setting, we would like a relationship between $P \twoheadrightarrow \vec{\exists}.G_1 \wedge G_2$, $P \twoheadrightarrow \vec{\exists}.G_1$ and $P \twoheadrightarrow \vec{\exists}.G_2$. Unfortunately, this cannot be derived directly from (2.4.12).

Note that $U_{\mathcal{C}}$ works with proofs with hypotheses. For this reason, it is not possible to retrieve only terminated computations using this fixpoint operator. This is not a flaw in the definition of the operator, but an intrinsic limit of all the kinds of top-down semantic refinements.

2.5 Abstraction Framework

The previous semantics are by far too detailed for most of the needs. However, it is now possible to use the techniques of abstract interpretation [24] to develop a range of abstract semantics for sequent calculi. In the following, we use the term *observable* to denote a Galois connection between Int and a generic abstract domain D.

Fixed an observable for a calculus $\mathcal{C}$, an abstract interpretation is an element of the abstract domain D. Given an interpretation I, we have the corresponding abstract counterpart $\alpha(I)$. Hence, it is possible to define abstract denotational, operational and fixpoint semantics as the abstractions of the corresponding concrete semantics. The question is whether it is possible to derive such abstract semantics working entirely in the abstract domain.

Example 2.5.1

Given a calculus $\mathcal{C}$, take as abstract domain D_s the powerset of all the sequents with the standard subset ordering, and as abstraction function the following

$$\alpha_s(I) = \{S \in \mathit{Seq} \mid \exists \pi \in I.\ \mathsf{root}(\pi) = S \text{ and } \mathsf{hyp}(S) = \emptyset\} \ . \tag{2.5.1}$$

The right adjoint of α_s is the function

$$\gamma_s(A) = \{\pi \in \mathsf{Sk} \mid \mathsf{hyp}(S) \neq \emptyset \text{ or } \mathsf{root}(\pi) \in A\} \ . \tag{2.5.2}$$

We call $\langle \alpha_s, \gamma_s \rangle$ the observable of *success sets*, since it abstracts a set of proofs in the set of the sequents they prove.

2.5.1 Abstract Semantic Operators

The only two operators we use in the specification of the concrete semantics are union and gluing. Once we define an abstraction, we have an abstract operator $\cup_\alpha$ correct w.r.t. $\cup$, defined as

$$\bigcup\nolimits_\alpha \{A_j \mid j \in J\} = \alpha\left(\bigcup \{\gamma(A_j) \mid j \in J\}\right) \ . \tag{2.5.3}$$

In general, $\cup_\alpha$ is the least upper bound of those elements in D which are the image of α. Moreover, it is a complete operator, i.e.

$$\bigcup\nolimits_\alpha \{\alpha(I_j) \mid j \in J\} = \alpha\left(\bigcup \{I_j \mid j \in J\}\right) \tag{2.5.4}$$

for each collection $\{I_j\}_{j \in J}$ of interpretations.

We could define an abstract operator $\rhd_\alpha$ optimal w.r.t. $\rhd$ as done for $\cup_\alpha$ in (2.5.3). However, $\rhd$ is never used in all its generality. Hence we prefer to consider the optimal abstract counterparts of the two fixpoint operators $T_\mathcal{C}$ and $U_\mathcal{C}$, respectively $T_{\mathcal{C},\alpha}$ and $U_{\mathcal{C},\alpha}$.

When either $T_{\mathcal{C},\alpha}$ or $U_{\mathcal{C},\alpha}$ is complete, we say that the observable is respectively *denotational* or *operational*, following the terminology introduced in [2]. If, for each inference rule $r \in \mathcal{R}$, there is an abstract operator $\tilde{r}$ correct w.r.t. the semantic operator $\bar{r}$, a correct abstract operator for $T_\mathcal{C}$ can be defined as

$$\tilde{T}_\mathcal{C}(A) = A \cup_\alpha \bigcup_{r \in \mathcal{R}}{}_\alpha\, \tilde{r}(A) \ . \tag{2.5.5}$$

Moreover, if all the $\tilde{r}$'s are optimal or complete, the same holds for (2.5.5).

Example 2.5.2

With respect to the observable α_s, consider an inference rule $r : S_1, \ldots, S_n \vdash S$. The corresponding optimal abstract operator $\bar{r}_{\alpha_s}$ is given by

$$\bar{r}_{\alpha_s}(X) = \begin{cases} \{S\} & \text{if } \{S_1, \ldots, S_n\} \subseteq X \\ \emptyset & \text{otherwise} \end{cases} \tag{2.5.6}$$

and it can be proved to be complete. Then, it turns out that the observable of success sets is denotational. An observable which is both operational and denotational is that of *plain resultants*, defined as

$$\alpha_r(I) = \{\langle S, (S_1, \ldots, S_n)\rangle \mid \exists \pi \in I.\ \pi : S_1, \ldots, S_n \vdash S\} \ . \tag{2.5.7}$$

with the obvious ordering by subsets. Note that what is generally called resultant in logic programming is the reduced product [25] of α_r and the observable of computed answers, which will be introduced later.

2.5.2 From Observables to Pre-Interpretations

By means of the observables we want to recover the great generality given by the use of pre-interpretations, but in a more controlled way, in order to simplify the definition and comparison of different semantics.

Given a calculus $\mathcal{C}$ and an observable $\langle \alpha, \gamma \rangle : \mathsf{Int} \rightleftharpoons D$, we have a corresponding pre-interpretation $\mathcal{I}_\alpha$ given by

- $\mathcal{I}_\alpha(S) = \langle \{x \in D \mid x \leq \alpha(\mathsf{Sk}_S)\}, \leq \rangle$, where $\leq$ is the ordering for D;
- $\mathcal{I}_\alpha(r) = \alpha \circ r \circ \gamma$.

The idea is that, with the use of pre-interpretations, we break an abstract interpretation in pieces, each one relative to a single sequent. If A is an abstract interpretation, a corresponding interpretation $[\![\_]\!]_A$ w.r.t. $\mathcal{I}_\alpha$ is

$$[\![S]\!]_A = A \cap_\alpha \alpha(\mathsf{Sk}_S) \ , \tag{2.5.8}$$

for each sequent S, where $\cap_\alpha$ is the optimal abstract operator which is correct w.r.t. $\cap$. On the other side, given $[\![\_]\!]$, we have the abstract interpretation

$$A_{[\![\_]\!]} = \bigcup_{S \in Seq}{}_{\alpha} [\![S]\!] \ . \tag{2.5.9}$$

However, in general, (2.5.8) and (2.5.9) do not form a bijection. Actually, an interpretation w.r.t. $\mathcal{I}_\alpha$ always keeps separate the semantics for different sequents, while the same does not happen for abstract interpretations.

Example 2.5.3

Consider the observable $\langle \alpha, \gamma \rangle : \mathsf{Int} \rightleftharpoons D$ where $D = \{\mathsf{true}, \mathsf{false}\}$, $\mathsf{false} \leq \mathsf{true}$ and

$$\alpha(I) = \begin{cases} \mathsf{true} & \text{if } \exists \pi \in I.\ \mathsf{hyp}(\pi) = \emptyset \\ \mathsf{false} & \text{otherwise} \end{cases} \tag{2.5.10}$$

The corresponding pre-interpretation $\mathcal{I}_\alpha$ is the same as $\mathcal{I}_s$ defined in Example 2.3.3. Given the interpretation $[\![\_]\!]$ such that $[\![\bar{S}]\!] = \mathsf{true}$ for a given sequent $\bar{S}$ and $[\![S]\!] =$

false for each $S \neq \bar{S}$, the composition of (2.5.8) and (2.5.9) is the interpretation $[\![\_]\!]'$ such that

$$[\![S]\!]' = \left(\bigcup\nolimits_{\alpha}\{[\![S']\!] \mid S' \text{ is a sequent}\}\right) \cap_{\alpha} \mathsf{true} = \mathsf{true} \tag{2.5.11}$$

for each sequent S.

Under particular hypotheses, an observable has a corresponding "well-behaved" pre-interpretation.

Definition 2.5.4 *Given an observable $\langle\alpha, \gamma\rangle$, we say that it* separates sequents *when*

- $\gamma(\alpha(\mathsf{Sk}_S)) = \mathsf{Sk}_S$ *for each sequent S;*
- $\gamma(\alpha(\bigcup_{S \in Seq} X_S)) = \bigcup_{S \in Seq} \gamma(\alpha(X_S))$ *if $X_S \subseteq \mathsf{Sk}_S$ for each sequent S.*

When we limit ourselves to observable which separate sequents, the following holds:

Theorem 2.5.5 *If the observable $\langle\alpha, \gamma\rangle$ separates sequents, then equations* (2.5.8) *and* (2.5.9) *form a bijection between the abstract interpretations which are in the image of α and the interpretations $[\![\_]\!]_\alpha$ such that $[\![S]\!]_\alpha$ is in the image of α for each sequent S.*

Proof. First of all, if $\langle\alpha, \gamma\rangle$ separates sequents and $[\![\_]\!]_\alpha$ is an interpretation w.r.t. $\mathcal{I}_\alpha$, then

$$\gamma([\![S]\!]_\alpha) \subseteq \gamma(\alpha(\mathsf{Sk}_S)) = \mathsf{Sk}_S \quad . \tag{2.5.12}$$

Now, let us take an interpretation $[\![\_]\!]_\alpha$ for the pre-interpretation $\mathcal{I}_\alpha$. We want to prove that, for each sequent S,

$$\begin{aligned}
\left(\bigcup_{S' \in Seq}\nolimits_{\alpha} [\![S']\!]_\alpha\right) \cap_\alpha \alpha(\mathsf{Sk}_S) &= \alpha\left(\gamma\left(\bigcup_{S'}\nolimits_{\alpha} [\![S']\!]_\alpha\right) \cap \gamma(\alpha(\mathsf{Sk}_S))\right) \\
&\qquad [\text{since } \gamma(\alpha(\mathsf{Sk}_S)) = \mathsf{Sk}_S] \\
&= \alpha\left(\gamma\left(\alpha\left(\bigcup_{S'} \gamma([\![S']\!]_\alpha)\right) \cap \mathsf{Sk}_S\right)\right) \\
&\qquad [\text{since } \alpha \text{ separates sequents}] \\
&= \alpha\left(\bigcup_{S'} \gamma([\![S']\!]_\alpha) \cap \mathsf{Sk}_S\right) \\
&\qquad [\text{since } \gamma([\![S']\!]_\alpha) \subseteq \mathsf{Sk}_{S'}] \\
&= \alpha(\gamma([\![S]\!]_\alpha))
\end{aligned}$$

as we were looking for. We also need to prove that, given an abstract interpretation A, it is

$$\bigcup_{S\in Seq}{}_{\alpha} (A \cap_\alpha \alpha(\mathsf{Sk}_S)) = \alpha(\gamma(A)) \ . \tag{2.5.13}$$

The proof is the following

$$\begin{aligned}
\bigcup_{S\in Seq}{}_{\alpha} (A \cap_\alpha \alpha(\mathsf{Sk}_S)) &= \bigcup_{S}{}_{\alpha} (\alpha(\gamma(A) \cap \gamma(\alpha(\mathsf{Sk}_S)))) \\
&\qquad [\text{since } \textstyle\bigcup_\alpha \text{ is complete}] \\
&= \alpha\left(\bigcup_S (\gamma(A) \cap \mathsf{Sk}_S)\right) \\
&= \alpha\left(\gamma(A) \cap \bigcup_S \mathsf{Sk}_S\right) \\
&= \alpha(\gamma(A)).
\end{aligned}$$

■

In particular, we have the following

Corollary 2.5.6 *If $\langle\alpha,\gamma\rangle$ is a Galois insertions and separates sequents, then* (2.5.8) *and* (2.5.9) *form a bijection.*

From this point of view, observables are even more general than pre-interpretations. On the other side, they only cover a subset of all the pre-interpretations, those whose abstraction function has a right adjoint.

Example 2.5.7

It is easy to prove that α_s separates sequents. The corresponding pre-interpretation $\mathcal{I}_{\alpha_s}$ is isomorphic to $\mathcal{I}_s$. Note that, thanks to abstract interpretation theory, we automatically obtain an optimal candidate for the abstract semantic functions from the choice of the abstract domain.

2.5.3 Abstract Semantics

We say that an abstract interpretation A is an *abstract model* when the corresponding interpretation $[\![\_]\!]_A$ for $\mathcal{I}_\alpha$ given by (2.5.8) is a model. In formulas, this means that, for each inference rule $r : S_1, \ldots, S_n \vdash S$,

$$\alpha\left(r\left(\gamma(A \cap_\alpha \alpha(\mathsf{Sk}_{S_1})), \ldots, \gamma(A \cap_\alpha \alpha(\mathsf{Sk}_{S_n}))\right)\right) \leq A \cap_\alpha \alpha(\mathsf{Sk}_S) \ . \tag{2.5.14}$$

In turn, this is equivalent to say that $\gamma(A)$ is a syntactic model.

Theorem 2.5.8 *A is an abstract model according to equation* (2.5.14) *iff $\gamma(A)$ is a model.*

Proof. First of all, given two abstract interpretation A_1 and A_2, it is $\gamma(A_1 \cap_\alpha A_2) = \gamma(A_1) \cap \gamma(A_2)$. Then, since γ preserves the greatest lower bounds, we have for each abstract interpretation A and inference rule $r : S_1, \dots, S_n \vdash S$,

$$\begin{aligned}
&\text{Equation (2.5.14) holds}\\
\iff& \alpha(r(\gamma(A) \cap \gamma(\alpha(\mathsf{Sk}_{S_1})), \dots, \gamma(A) \cap \gamma(\alpha(\mathsf{Sk}_{S_n})))) \leq A \cap_\alpha \alpha(\mathsf{Sk}_S)\\
&\qquad\text{[by equation (2.2.17)]}\\
\iff& \alpha(r(\gamma(A) \cap \mathsf{Sk}_{S_1}, \dots, \gamma(A) \cap \mathsf{Sk}_{S_n})) \leq A \cap_\alpha \alpha(\mathsf{Sk}_S)\\
&\qquad\text{[by adjoint properties and equation (2.2.17)]}\\
\iff& r(\gamma(A) \cap \mathsf{Sk}_{S_1}, \dots, \gamma(A) \cap \mathsf{Sk}_{S_n}) \subseteq \gamma(A) \cap \mathsf{Sk}_S\\
\iff& \{r\} \rhd \gamma(A) \subseteq \gamma(A)
\end{aligned}$$

Since this holds for every r, by left-additivity of $\rhd$ we have the required result. ■

We would like to define the *abstract declarative semantics* $\mathcal{D}_\alpha(\mathcal{C})$ as the least abstract model for $\mathcal{C}$. However, since our abstract domain is a poset, we are not guaranteed that such an element exists. Nevertheless, when we work with a denotational observable, we have (Theorem 1.3.6):

- $\mathcal{D}_\alpha(\mathcal{C}) = \alpha(\mathcal{D}(\mathcal{C}))$, where $\mathcal{D}_\alpha(\mathcal{C})$ is the least abstract model;
- $\mathcal{D}_{c,\alpha}(\mathcal{C}) = \alpha(\mathcal{D}_c(\mathcal{C}))$, where $\mathcal{D}_{c,\alpha}(\mathcal{C})$ is the least abstract model greater than $\alpha(\epsilon)$.

Other conditions, such as surjectivity of α, imply the existence of $\mathcal{D}_\alpha(\mathcal{C})$ (Theorem 1.3.3), whether or not α is denotational. However, in this case, we cannot be sure of the stated correspondence with $\alpha(\mathcal{D}(\mathcal{C}))$.

As in the concrete case, we want to recover $\mathcal{D}_\alpha(\mathcal{C})$ as the least fixpoint of a continuous operator. If the observable is denotational, the optimal abstract operator w.r.t. $T_\mathcal{C}$, namely

$$T_{\mathcal{C},\alpha}(A) = A \cup_\alpha \alpha(\mathcal{R} \rhd \gamma(A)) \ , \tag{2.5.15}$$

is complete. Then, by well known results of abstract interpretation theory [25],

$$T_{\mathcal{C},\alpha} \uparrow \omega = \alpha(T_\mathcal{C} \uparrow \omega) = \mathcal{D}_\alpha(\mathcal{C}) \ , \tag{2.5.16}$$

$$T^\omega_{\mathcal{C},\alpha}(\alpha(\epsilon)) = \alpha(T^\omega_\mathcal{C}(\epsilon)) = \mathcal{D}_{c,\alpha}(\mathcal{C}) \ , \tag{2.5.17}$$

which are the required equalities.

Finally, let us come to the abstract operational semantics. In general, since we do not have an abstraction on the level of the single derivation, we can only abstract the collecting operational semantics given by $U_{\mathcal{C}}$. If $\rhd$ is operational, the optimal abstract counterpart of $U_{\mathcal{C}}$, namely

$$U_{\mathcal{C},\alpha}(A) = \alpha(\gamma(A) \rhd (\mathcal{R} \cup \epsilon)) \ , \tag{2.5.18}$$

is complete. Then,

$$U^{\omega}_{\mathcal{C},\alpha}(\alpha(\epsilon)) = \alpha(U^{\omega}_{\mathcal{C}}(\epsilon)) = \alpha(D_c(\mathcal{C})) \ , \tag{2.5.19}$$

$$U^{\omega}_{\mathcal{C},\alpha}(\alpha(\{\epsilon_Q\})) = \alpha(U^{\omega}_{\mathcal{C}}(\{\epsilon_Q\})) = \alpha(\mathcal{B}(\mathcal{C})_Q) \ . \tag{2.5.20}$$

Therefore, we have a top-down collecting construction of the abstract declarative semantics and of the operational behavior of $\mathcal{C}$.

Generally, if we replace the first equality with a “greater than” disequality in the equations (2.5.16), (2.5.17), (2.5.19) and (2.5.20), they become true for every observable α. In this case, the semantics computed in the abstract domain are correct w.r.t. the real abstract semantics.

2.6 Conclusions

In this chapter we presented the general framework we propose to use for the semantic analysis of those logic languages which are based on proof theory, especially in the form of sequent calculi. Typical examples of these languages are λProlog [57], FORUM [53], Lolli [38], LinLog [4]. We have shown some basic examples of properties which it is possible to extract from a proof, such as plain resultants or ground answers for Horn clauses.

In the following chapter, we apply the framework to the case of first order logic, introducing several common observables, such as correct answers, resultants and groundness, from the point of view of proof theory. Expressed in such a way, rather than referring to a computational procedure like SLD resolution, it is possible to grasp their logical meaning out of the traditional context of Horne clauses.

Chapter 3

The Case of First Order Logic

Abstract

Working within the semantic framework we have just developed, we propose a couple of extensions to the concepts of correct answers and correct resultants which can be applied to the full first order logic. We motivate our choice with several examples and we show how to use correct answers to reconstruct an abstraction which is widely used in the static analysis of logic programs, namely groundness. As an example of application, we present a prototypical top-down static interpreter for properties of groundness which works for the full first order logic. Part of this chapter has been presented in [1].

In the previous chapter, we have defined a semantic framework which can be easily instantiated to different sequent calculi. Once we define a calculus as an instance of the general framework, we automatically obtain a couple of fixpoint semantics and a standard methodology for defining abstractions of proofs. The framework is not useful by itself, but as the foundation for the proof-theoretic semantic treatment of different logic systems.

It seems natural to start by applying the framework to first order classical or intuitionistic logic, for several reasons. First of all, classical logic is the main example of logic system which has ever been studied. Moreover, two of the main logic languages, namely Horn clauses and hereditary Harrop formulas, are subsets of intuitionistic logic. As previously stated, one of the goal of this thesis is to ease the transition of all the research in static analysis and related areas which has been conducted for pure logic programs to other more powerful logic languages. Therefore, hereditary Harrop formulas and first order classical logic seems a natural candidate to start with.

However, nothing in the framework precludes its application to other logic systems, such as linear logic or temporal logic. Several variants of temporal logic, in particular, are used in model checking, so that our framework could lead to an alternative approach to *abstract model checking* [26].

3.1 Correct Answers and Resultants

The concepts of *correct answer* and *computed answer* are the cornerstones of the theory and practice of logic programming. If we want to extend the results we have for Horn clauses to other logic languages, we need to find an analogous of these concepts in the new settings. Actually, since we are not discussing any computational mechanism, we only focus our attention to correct answers. In this chapter, we will stick to the standard convection of using latin letters $x, y, z, \ldots$ to denote both free and bound variables.

3.1.1 Correct Answers as Proof-Theoretical Properties

Given a sequent $S = \Gamma \twoheadrightarrow \exists\vec{x}.\phi$ in $\mathcal{L}_{hc}$, a *correct answer* for the goal $\exists\vec{x}.\phi$ in the program Γ is traditionally defined as a substitution θ for $\vec{x}$ such that $\Gamma \twoheadrightarrow \phi\theta$ is provable. In the following, we refer to θ as a correct answer for the sequent S. According to this definition, the concept of correct answer seems strictly related to model theory. It is essentially an assignment for the variables in $\vec{x}$ such that ϕ in valid in every term model of Γ.

However, if π is an intuitionistic proof for S, a correct answer for $\exists\vec{x}.\phi$ can be extracted from π by examining the instances in π of the $\exists R$ schema.

Example 3.1.1

If S is the sequent $p(0), \forall x.p(x) \twoheadrightarrow \exists y.p(y)$, then y/t is a correct answer for each term t. If π is the proof

$$\dfrac{\dfrac{}{\forall x.p(x), p(0) \twoheadrightarrow p(0)}\,id}{\forall x.p(x), p(0) \twoheadrightarrow \exists y.p(y)}\,\exists R \tag{3.1.1}$$

the $\exists R$ rule gives origin to the correct answer $y/0$.

When we use hereditary Harrop formulas, we can keep the same definition of correct answers we have for Horn clauses. However, the amount of information we obtain in this way is rather limited. For example, the sequent $\forall x.p(x,x) \twoheadrightarrow \forall y.\exists z.p(y,z)$ only has a trivial empty correct answer, since the right hand side of the sequent is not an existentially quantified formula. On the contrary, let us give a look to a proof of the same sequent:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p(a,a) \twoheadrightarrow p(a,a)}\,id}{p(a,a) \twoheadrightarrow \exists z.p(a,z)}\,\exists R}{\forall x.p(x,x) \twoheadrightarrow \exists z.p(a,z)}\,\forall R}{\forall x.p(x,x) \twoheadrightarrow \forall y.\exists z.p(y,z)}\,\forall L \tag{3.1.2}$$

If we keep track of the occurrences of both the $\exists R$ and $\forall R$ inference rules, we obtain a substitution $\{y/a, z/a\}$. This makes explicit that for each y we have a z such that $p(y, z)$ is true, and that y and z do coincide. Moreover, if we apply the substitution $\{y/a, z/a\}$ to the right hand side, discarding all the quantifiers, we obtain the sequent $\forall x.p(x, x) \twoheadrightarrow p(a, a)$ which is trivially provable.

If we further extend the language to handle the full first order logic, we have to deal with sequents like $\exists x.p(s(x)) \twoheadrightarrow \exists y.p(y)$. Here, again, the standard "model-theoretic" definition of correct answers gives us no interesting information, since there are no correct answers according to that definition. Actually, the sequent $\exists x.p(s(x)) \twoheadrightarrow p(t)$ is not provable for any term t. Let us consider the following proof:

$$\dfrac{\dfrac{\dfrac{}{p(s(a)) \twoheadrightarrow p(s(a))}\,id}{p(s(a)) \twoheadrightarrow \exists y.p(y)}\,\exists R}{\exists x.p(s(x)) \twoheadrightarrow \exists y.p(y)}\,\exists L \tag{3.1.3}$$

If we keep track of all the instances of a quantifier introduction rule, we obtain a substitution $\{x/a, y/s(a)\}$. Here, the role of a is that of a witness. The existential quantifier on the left hand side *produces* a new object a such that $p(s(a))$ holds. The binding $\{y/s(a)\}$ makes clear that the object y such that $p(y)$ holds is $s(a)$, where a is the same variable produced by the other existential quantifier. Again, if we apply the substitution discarding the quantifiers, we obtain the sequent $p(s(a)) \twoheadrightarrow p(s(a))$ which is provable.

In the general case, we cannot bind a variable with a single term. For example, consider the sequent $p(a) \vee p(b) \twoheadrightarrow \exists x.p(x)$, and the proof

$$\dfrac{\dfrac{\dfrac{}{p(a) \twoheadrightarrow p(a)}\,id}{p(a) \twoheadrightarrow \exists x.p(x)}\,\exists R \qquad \dfrac{\dfrac{}{p(b) \twoheadrightarrow p(b)}\,id}{p(b) \twoheadrightarrow \exists x.p(x)}\,\exists R}{p(a) \vee p(b) \twoheadrightarrow \exists x.p(x)}\,\vee L \tag{3.1.4}$$

We have two different instances of the $\exists R$ schema, each with a different term which is bound to the variable x. Therefore, we are led to consider bindings of the kind $\{x/\{a, b\}\}$.

3.1.2 Formalization

We now try to make precise the above informal discussion. In the following, we assume fixed a first order signature $\langle \Sigma, \Pi \rangle$ and a couple of denumerable sets V and W for the free and bound variables respectively. This induces a sequent calculus $\mathcal{L}_c^{\Sigma,\Pi}$ in which we assume to work.

Definition 3.1.2 (Candidate answer) *A* candidate answer *is a function*

$$\theta : W \to \wp_f(T_\Sigma(V \setminus W))$$

such that $\{v \mid \theta(v) \neq \emptyset\}$ *is finite. We denote by* Ans *the set of candidate answers. Note that it only depends from the chosen first order signature.*

Therefore, a candidate answer is a sort of idempotent substitution with multiple bindings for each variable. We denote by $\{x_1/S_1, \ldots, x_n/S_n\}$ the candidate answer θ such that $\theta(x_i) = S_i$ for each $i \in \{1, \ldots, n\}$ and $\theta(x)\uparrow$ if $v \notin \{x_1, \ldots, x_n\}$. When S is a singleton $\{t\}$, we write x/t instead of $x/\{t\}$.

For each proof skeleton π, we have a corresponding candidate answer θ_π or $\mathsf{answer}(\pi)$, such that, for each $x \in W$,

$$\theta_\pi(x) = \begin{cases} \emptyset & \text{if } \pi = id, \\ \{a\} & \text{if } \pi = \exists L_{A,\Gamma,\Delta,x,v}(\pi') \text{ or } \pi = \forall R_{A,\Gamma,\Delta,x,v}(\pi'), \\ \{t\} & \text{if } \pi = \exists R_{A,\Gamma,\Delta,x,t}(\pi') \text{ or } \pi = \forall L_{A,\Gamma,\Delta,x,t}(\pi'), \\ \bigcup_{j=1\ldots n} \theta_{\pi_j}(x) & \text{otherwise, where } \pi = r(\pi_1, \ldots, \pi_n) \text{ for } r \in \mathcal{R}. \end{cases} \tag{3.1.5}$$

If π is a proof, then θ_π is called the *partial correct answer* for π. If π is a final proof of the sequent S, then θ is a *correct answer* for the sequent S. The set of correct answers for the sequent S will be denoted by $\mathsf{CAns}_c(S)$, which is defined as

$$\mathsf{CAns}_c(S) = \{\theta_\pi \mid \pi \in \mathcal{D}(\mathcal{L}_c) \cap \mathsf{Sk}_S\} \ . \tag{3.1.6}$$

We write $\mathsf{CAns}_i(S)$ when we prefer to work with the intuitionistic sequent calculus induced by the chosen signature.

Example 3.1.3

Let us consider the sequent $S = \forall x.(p(x) \supset p(s(x))), p(0) \twoheadrightarrow \exists y.p(y)$. In classical logic, θ is a correct answer for S if and only if

- $\theta(x) \in \wp_f(T_\Sigma(V \setminus W))$,
- $\theta(y) \in \wp_f(T_\Sigma(V \setminus W))$ and there exists $s^i(0) \in \theta(y)$ such that $s^j(0) \in \theta(x)$ for every $j \in \{0, \ldots, i-1\}$.

Here, $s^i(0)$ is the term $s(s(\cdots s(0) \cdots))$, where s is repeated i times. In intuitionistic logic, the form of the correct answers is simpler. In particular, θ is a correct answer for S if and only if

- $\theta(x) \in \wp_f(T_\Sigma(V \setminus W))$,
- $\theta(y) = \{s^i(0)\}$ for some $i \in \mathbb{N}$ such that $s^j(0) \in \theta(x)$ for every $j \in \{0, \ldots, i-1\}$.

The difference is due to the fact we cannot apply the contraction rule on the consequent.

Note that, quite often, a correct answer for the sequent S is meaningful only if there are no two different bindings for the same variable. In the following, we will call *pure* every sequent which satisfies this condition.

Our definition of correct answer essentially collects all the occurrences of introduction rules for quantifiers in a proof. A problem is that most of the answers we obtain are trivial. For example, given a sequent $\forall x.\phi \twoheadrightarrow \Delta$ and a correct answer θ, then $\theta[x/L]$ is a correct answer, too, for each $L = \theta(x) \cup T$ where T is a set of terms such that

- $\mathsf{vars}(t) \cap W = \emptyset$ for each $t \in T$,
- $\mathsf{vars}(t) \cap \mathsf{vars}(\theta(y)) = \emptyset$ for each $t \in T$ and $y \in \mathsf{dom}(\theta)$.

As a result, we are particularly interested to *minimal* correct answers, according to the obvious point-wise ordering. Proofs corresponding to minimal answers are a sort of "non-redundant" proofs, where quantifiers are introduced only when they are really needed. In formulas, we denote by $\mathsf{mAns}_c(S)$ ($\mathsf{mAns}_i(S)$) the set of minimal correct answer for the sequent S w.r.t. classical (intuitionistic) logic.

Example 3.1.4

In the previous example, both classic and intuitionistic logic have the same set of minimal correct answers, i.e. those θ such that

- $\theta(y) = \{s^i(0)\}$ for some $i \in \mathbb{N}$,
- $\theta(x) = \{s^j(0) \mid j \in \{0, \dots, i-1\}\}$.

Note that we have a lot of information from these. We know that $p(s^i(0))$ is true for every $i \in \mathbb{N}$. Moreover, we know that, in order to prove $p(s^i(0))$, we need to apply several $\forall L$ introduction rules for the first binding with different terms, namely all the $s^j(0)$ with j from 0 to $i-1$.

In general, if $\mathsf{mAns}_c(S) \neq \mathsf{mAns}_i(S)$, it means that there is a proof of S which is "intrinsically" classical. We do not make precise this statement, since it requires further investigations. However, from an intuitive point of view, consider the following proof π of the sequent $p(a) \vee p(b) \twoheadrightarrow \exists x.p(x)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p(a) \twoheadrightarrow p(a), p(b)}\,id \qquad \dfrac{}{p(b) \twoheadrightarrow p(a), p(b)}\,id}{p(a) \vee p(b) \twoheadrightarrow p(a), p(b)}\,\vee L}{p(a) \vee p(b) \twoheadrightarrow \exists x.p(x), \exists x.p(x)}\,\exists R \text{ (2 times)}}{p(a) \vee p(b) \twoheadrightarrow \exists x.p(x)}\,\text{contraction}R \tag{3.1.7}$$

If we move the $\exists R$ rules upward, over the $\vee L$ rule, we can easily obtain an intuitionistic proof π' such that $\theta_{\pi'} = \theta_\pi = \{x/\{a, b\}\}$. However, consider the following proof π

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p(a), p(v) \twoheadrightarrow p(v)}\,id}{p(a), p(v) \twoheadrightarrow \exists x.p(x)}\,\exists R}{p(a), \exists y.p(y) \twoheadrightarrow \exists x.p(x)}\,\exists L}{p(a) \twoheadrightarrow \neg\exists y.p(y), \exists x.p(x)}\,\neg R}{p(a) \twoheadrightarrow \exists x.p(x) \vee \neg\exists y.p(y),\ \exists x.p(x) \vee \neg\exists y.p(y)}\,\vee R \text{ (2 times)}}{p(a) \twoheadrightarrow \exists x.p(x) \vee \neg\exists y.p(y)}\,\text{contraction}R \qquad (3.1.8)$$

Although the root sequent is intuitionistically provable, we are not able to write an intuitionistic proof π' such that $\theta_{\pi'} \leq \theta_\pi = \{x/v, y/v\}$. This is because the use of the contraction rule in π is essential to the effort of moving $\exists y.p(y)$ on the left side while keeping $\exists x.p(x)$ on the right side.

If we compare the “standard” correct answers for Horn clauses with our minimal correct answers, we still have a more general definition, since we also keep track of the occurrences of $\forall L$ introduction rules. However, if we restrict our answers to the existential quantifiers, we obtain a strong correspondence.

First of all, if ϕ is a propositional formula, we call *standard* correct answer for the sequent $\Gamma \twoheadrightarrow \exists x_1, \ldots, x_n.\phi$ in $\mathcal{L}_{hc}$ a substitution $\theta \in \mathsf{Subst}_W^{V \setminus V}$ such that $\Gamma \twoheadrightarrow \phi\theta$ is provable. Then, we can prove:

Lemma 3.1.5 *If the pure sequent $S = \Gamma \twoheadrightarrow \exists x.\phi$ has a correct answer θ such that $\theta(x) = \emptyset$, then $\Gamma \twoheadrightarrow$ is provable.*

Proof. If π is a final proof of S and $\theta_\pi(x) = \emptyset$, then there are no occurrences of $\exists R$ inference rules, with $\exists x.\phi$ as the principal formula, in π. However, looking at the form of inference rules in Figure 1.1, it is evident that we need a way to discard the existential quantifier from the right hand side of the sequent and the only way, other than an $\exists R$ rule, is a contraction rule.

Then, for each path in π from the root to the leaves, either $\exists x.\phi$ is never the principal formula of an introduction rule, or a contraction rule is applied on the right. We can obtain a new proof π' of $\Gamma \twoheadrightarrow$ just deleting every occurrence of $\exists x.\phi$ and of the now useless contraction rules. ■

Theorem 3.1.6 *If $S = \Gamma \twoheadrightarrow \exists x_1. \cdots \exists x_n.\phi$ is a pure sequent in $\mathcal{L}_{hc}$ and ϕ is propositional, then η is a standard correct answer for S iff there is a minimal correct answer θ such that $\theta(x_i) = \{x_i\eta\}$ for each $i \in \{1, \ldots, n\}$.*

Proof. Assume η is a standard correct answer for S. It means that $S' = \Gamma \twoheadrightarrow \phi\eta$ is provable. Let π' be a proof of S'. If we apply a sequence of $\exists R$ rules to π', we obtain a proof π for S. It is trivial that $\theta_\pi(x_i) = \{x_i\eta\}$ for each $i \in \{1, \dots, n\}$. Now, consider the set of all the correct answers θ' for S such that $\theta' \leq \theta_\pi$. If $\theta'(x_i) \neq \theta_\pi(x_i)$, then $\theta'(x_i) = \emptyset$, since $\theta_\pi(x_i)$ is a singleton. By the previous lemma, however, this would mean that Γ is inconsistent, and this is not possible for Horn clauses. Then, if we take θ to be a minimal θ', we prove half of the theorem.

Now, assume θ is a minimal correct answer for S. We want to prove that, if we define $\eta(x_i) = \theta(x_i)$ for $i \in \{1, \dots, n\}$, then $\Gamma \twoheadrightarrow \phi\eta$ is provable. If π is a proof such that $\theta = \theta_\pi$, we can think of permuting the inference rules to obtain a new proof π' with $\theta_{\pi'} = \theta$ and all the $\exists R$ rules applied just after the root. Since for Horn clauses we do not have essentially universal quantifiers, in π there are no occurrences of $\forall R$ or $\exists L$ rules. As a result, in π there are no eigenvariables. Therefore, rules in classical logic can be permuted freely to obtain π'. If we work in intuitionistic logics, not all the permutations are allowed, but Kleene in [42] shows that the only rules $\exists R$ does not permute with are $\vee L$ and $\exists L$. Neither of this can never be applied if Γ is made of Horn clauses, hence π' can be found in intuitionistic logic, too. If we drop the $\exists R$ rules from π', we remain with a proof of $\Gamma \twoheadrightarrow \phi\eta$ and the theorem is proved. ■

Note that we have not specified if the minimal correct answer should be considered w.r.t. intuitionistic or classical logic. Actually, if S is a pure sequent in $\mathcal{L}_{hc}$, it is the case that $\mathsf{mAns}_i(S) = \mathsf{mAns}_c(S)$.

3.1.3 Resultants

Another typical abstraction of SLD-derivations is those of resultants [33]. A resultant for a goal `G` in a program `P` is a pair made of a partial computed answer for `G` and a new goal `G'` which still needs to be refuted. We present an observable for proof skeletons which is inspired by this "standard" idea of resultant, although the relation here is more shallow than for correct answers.

Until now we have considered sequents as sequences of formulas. However, classical and intuitionistic logics are often presented by defining a sequent as a set of formulas. We use the term *set sequent* to refer to this alternative definition and we denote by *SetSeq* the collection of all the set sequents. If $S \in \mathit{Seq}$, we write $\bar{S}$ for the corresponding element in *SetSeq*.

We call *resultant* a pair $(\theta, \mathcal{S})$ where $\theta \in \mathsf{Ans}$ and $\mathcal{S}$ is a finite multi-set of set sequents. We denote by Res the set of all the resultants. For each proof skeleton $\pi : S_1, \dots, S_n \vdash S$, we define a corresponding $\mathsf{res}(\pi)$ as

$$\mathsf{res}(\pi) = (\mathsf{answer}(\pi), \llbracket \bar{S}_1, \dots, \bar{S}_n \rrbracket) \ , \tag{3.1.9}$$

where $\llbracket \_ \rrbracket$ denotes a multi-set. If π is a proof for the sequent S, then $\mathsf{res}(\pi)$ is a *correct resultant* for S. The set of correct resultants for S will be denoted by $\mathsf{CRes}(S)$, which is defined as

$$\mathsf{CRes}_c(S) = \{\mathsf{res}(\pi) \mid \pi \in \mathcal{D}(\mathcal{L}_c) \cap \mathsf{Sk}_S\} \ . \tag{3.1.10}$$

We write $\mathsf{CRes}_i(S)$ for the set of correct resultant in the case of intuitionistic logic. We define an order relation between two resultants, according to the following equation

$$(\theta, \mathcal{S}) \leq (\theta', \mathcal{S}') \text{ iff } \theta \leq \theta' \text{ and } \mathcal{S} \subseteq \mathcal{S}' \ . \tag{3.1.11}$$

Again, we talk of *minimal* correct resultants for the elements of $\mathsf{CRes}(S)$ which are minimal w.r.t. $\leq$. We denote the corresponding sets as $\mathsf{mRes}_c(S)$ and $\mathsf{mRes}_i(S)$.

Example 3.1.7

Let us consider the sequent $S = p(0) \twoheadrightarrow \exists x.p(x)$. The set $\mathsf{mRes}_c(S)$ contains all the pairs $(\theta, \mathcal{S})$ such that

- $\theta = \{x/0\}$ and $\mathcal{S} = \emptyset$, or
- $\theta = \{x/t\}$ for $t \neq 0$, $\mathsf{vars}(t) \cap W = \emptyset$ and $\mathcal{S} = \llbracket p(0) \twoheadrightarrow p(t) \rrbracket$, or
- $\theta = \{x/\emptyset\}$ and $\mathcal{S} = \llbracket p(0) \twoheadrightarrow \rrbracket$, or
- $\theta = \{x/\emptyset\}$ and $\mathcal{S} = \llbracket p(0) \twoheadrightarrow \exists x.p(x) \rrbracket$.

The same happens for $\mathsf{mRes}_i(S)$.

It is trivial to prove that the following correspondences hold between correct answers and correct resultants:

$$\mathsf{CAns}(S) = \{\theta \mid (\theta, \emptyset) \in \mathsf{CRes}(S)\} \ , \tag{3.1.12}$$

$$\mathsf{mAns}(S) = \{\theta \mid (\theta, \emptyset) \in \mathsf{mRes}(S)\} \ . \tag{3.1.13}$$

3.2 Observables

Now that we have defined what a correct answer is, we would like to find a bottom-up and a top-down construction for CAns and mAns. Following the abstract framework introduced in the previous chapter, we define the observable of candidate answers as a tuple $\langle \alpha_c, \gamma_c \rangle : \mathsf{Int} \rightleftharpoons [\mathit{Seq} \to \wp(\mathsf{Ans})]$ such that

$$\alpha_c(I)(S) = \{\mathsf{answer}(\pi) \mid \pi : \cdot \vdash S \in I\} \ . \tag{3.2.1}$$

It is trivial to show that $\alpha_c(\mathcal{D}(\mathcal{L}))(S)$ is exactly the set $\mathsf{CAns}(S)$ of all the correct answers for the sequent S. The optimal abstract operator corresponding to T is T_{α_c}. Assuming A in the image of α_c, it is

$$T_{\alpha_c}(A)(S) = A(S) \cup \bigcup \{r_{\alpha_c}(A) \mid r \in \mathcal{R}, \mathsf{root}(r) = S\} \ , \tag{3.2.2}$$

where, for each $r : S_1 \dots S_n \vdash S \in \mathcal{R}$,

$$r_{\alpha_c}(A) = \{r_{\alpha_c}(\theta_1, \dots, \theta_n) \mid \forall i \in \{1, \dots, n\}, \theta_i \in A(S_i)\} \ , \tag{3.2.3}$$

and

$$r_{\alpha_c}(\theta_1, \dots, \theta_n) = \begin{cases} \theta_1 \cup \{x/t\} & \text{if } r \text{ is an introduction rule for a quantifier} \\ & \text{which replaces the variable } x \text{ with } t, \\ \theta_1 \cup \dots \cup \theta_n & \text{otherwise.} \end{cases} \tag{3.2.4}$$

Here we write $\theta_1 \cup \theta_2$ for the candidate answer θ such that $\theta(x) = \theta_1(x) \cup \theta_2(x)$ for each $x \in V$. The intuitive meaning of r_{α_c} is evident: it collects all the bindings for the upper sequents, eventually adjoining the new binding which is created when it is a rule for a quantifier.

Theorem 3.2.1 *The observable α_c of candidate answers is denotational.*

Proof. We need to prove that

$$T_{\alpha_c}(\alpha_c(I)) \subseteq \alpha_c(T(I)) \ , \tag{3.2.5}$$

since the opposite disequality is trivial.

Assume $\theta \in \mathsf{Ans}$ and $\theta \in T_{\alpha_c}(\alpha_c(I))(S)$. We have two cases: $\theta \in \alpha_c(I)(S)$ or $\theta \in r_{\alpha_c}(\alpha_c(I))$ for some $r : S_1, \dots, S_n \vdash S \in \mathcal{R}$. If $\theta \in \alpha_c(I)(S)$, then $\theta \in \alpha_c(T(I))(S)$ follows trivially. Otherwise, it is $\theta = r_{\alpha_c}(\theta_1, \dots, \theta_n)$, with $\theta_i \in \alpha_c(I)(S_i)$ for each $i \in \{1, \dots, n\}$.

If r is an introduction rule for a quantifier, which replaces the variable x with the term t, then $\theta = \theta_1 \cup [x/t]$. Since $\theta_1 \in \alpha_c(I)(S_1)$, there exists a final proof π skeleton in I with $\theta_1 = \theta_\pi$. By applying the rule r to π, we obtain a new final proof skeleton $\pi' : \cdot \vdash S$ such that $\theta = \theta_{\pi'}$. Since $\pi' \in T(I)$, it is $\theta \in \alpha_c(T(I))(S)$.

If r is not an introduction rule for a quantifier, then $\theta = \theta_1 \cup \ldots \cup \theta_n$. For each θ_i, there exists a proof $\pi_i : \dots \vdash S_i$ in I. By applying the rule r to π, we can reason as in the previous case, and we prove the theorem. ■

We also have an observable for *resultants* which gives origin to complete bottom-up and top-down semantics. It is defined as the Galois connection $\langle \alpha_r, \gamma_r \rangle : Int \rightleftharpoons [SetSeq \to \wp(\mathsf{Res})]$ with the abstraction function

$$\alpha_r(I)(\bar{S}) = \{\mathsf{res}(\pi) \mid \pi \in I, \mathsf{root}(\pi) = S', \bar{S} = \bar{S'}\} \ . \tag{3.2.6}$$

The definition of the optimal bottom-up fixpoint operator is straightforward. With respect to the top-down fixpoint operator, assuming A in the image of α_r, we have

$$U_{\alpha_c}(A)(\bar{S}) = \bigcup_{\delta \in A(\bar{S})} \delta(\mathcal{R} \cup \epsilon) \tag{3.2.7}$$

where, if $\delta = (\theta, \llbracket \bar{S}_1, \dots, \bar{S}_n \rrbracket)$,

$$\delta(X) = \{\delta(\llbracket \pi_1, \dots, \pi_n \rrbracket) \mid \forall i \in \{1 \dots n\}, \pi_i \in X \cap \mathsf{Sk}_{S'_i} \text{ and } \bar{S}'_i = \bar{S}_i\} \ , \tag{3.2.8}$$

and

$$\delta(\llbracket \pi_1, \ldots, \pi_n \rrbracket) = \Big(\theta \cup \mathsf{answer}(\pi_1) \cup \cdots \mathsf{answer}(\pi_n), \overline{\mathsf{hyp}(\pi_1)}, \cdots, \overline{\mathsf{hyp}(\pi_n)} \rrbracket\Big) \quad . \tag{3.2.9}$$

Theorem 3.2.2 *The observable α_r of correct resultants is perfect.*

Proof. We need to prove that α_r is both operational and denotational. We only prove it is operational, since the other proof proceeds as for Theorem 3.2.1. Actually, we only need to check the disequality

$$U_{\alpha_r}(\alpha_r(I)) \subseteq \alpha_r(U(I)) \ , \tag{3.2.10}$$

since the opposite one is trivial.

If $\delta = (\theta, \llbracket \bar{S}_1, \ldots, \bar{S}_n \rrbracket) \in U_{\alpha_r}(\alpha_r(I))(\bar{S})$, there exists $\delta' = (\theta', \llbracket \bar{S'_1}, \ldots, \bar{S'_m} \rrbracket)$ in $\alpha_r(I)(\bar{S})$ such that $\delta = \delta'(\llbracket r_1, \ldots, r_m \rrbracket)$, where $r_i \in \mathcal{R} \cup \epsilon$ for each $i \in \{1, \ldots, m\}$. By the definition of α_r, there is a proof $\pi : Z_1, \ldots, Z_l \vdash Z$ in I with $Z \in Seq$, $\{Z_i\}_{i=1}^{l} \subseteq Seq$, $\bar{Z} = \bar{S}$, $\llbracket \bar{Z}_1, \ldots, \bar{Z}_l \rrbracket = \llbracket \bar{S'_1}, \ldots, \bar{S'_m} \rrbracket$ and $\theta_\pi = \theta'$.

The problem is that we cannot apply $r_1, \ldots, r_m$, to π. Actually, for each $i \in \{1, \ldots, l\}$, we know that there exists $\nu(i) \in \{1, \ldots, m\}$ such that $\bar{Z}_i = \bar{\mathsf{root}(r_{\nu(i)}}$. However, in general it is possible that $Z_i \neq \mathsf{root}(r_{\nu(i)})$. If we examine the form of the inference rules for $\mathcal{L}_c$, it is trivial to prove that, for each $i \in \{1, \ldots, l\}$, there exists an $r'_i \in \mathcal{R} \cup \epsilon$, which is a slight variation of $r_{\nu(i)}$ such that $Z_i = \mathsf{root}(r'_i)$, $\llbracket \bar{\mathsf{hyp}(r'_i)} \rrbracket = \llbracket Z_i \rrbracket$ and $\theta_{r'_i} = \theta_{r_{\nu(i)}}$. This is the reason while, sometimes, the calculus LK is presented with sequents made of multisets of formulas instead of sequences.

Then, we can glue $r'_1, \ldots, r'_l$ into π to to obtain a proof $\pi' : Z_1, \ldots, Z_r \vdash Z \in U(I)$ with $\llbracket \bar{Z}_1, \ldots, \bar{Z}_r \rrbracket = \llbracket \bar{S}_1, \ldots, \bar{S_m} \rrbracket$ and $\theta_{\pi'} = \theta$. This proves the theorem. ■

Since α_r is a perfect observable, we can build a top-down interpreter which computes correct resultants. However, the efficient implementation of such an interpreter is a very difficult task which is the realm of automatic deduction. Here, we are more interested in computing abstractions of correct resultants, which can be used for static analysis of logic languages.

3.2.1 Groundness

If θ is a candidate answer for the sequent S, we say that θ is *grounding* for the variable x when $\theta(x)$ is defined and, for each $t \in \theta(x)$, it is the case that $\mathsf{vars}(t) \subseteq \mathsf{FV}(S)$. Note that this is a natural extension of the standard definition for pure logic programs: in this latter case, we only have bound variables (although quantifications are implicit), hence the condition $\mathsf{vars}(t) \subseteq \mathsf{FV}(S)$ becomes equivalent to $\mathsf{vars}(t) = \emptyset$.

Let us define by GAns the set of functions $W \to \wp(\{\mathrm{g}, \mathrm{ng}\})$, which we call *groundness answers*. Given a candidate answer θ for the sequent S, we define a corresponding groundness answer $\beta = \mathsf{ground}_S(\theta)$ such that

- g $\in \beta(x)$ iff there exists $t \in \theta(x)$ such that $\mathsf{vars}(t) \subseteq \mathsf{FV}(S)$,
- ng $\in \beta(x)$ iff there exists $t \in \theta(x)$ such that $\mathsf{vars}(t) \not\subseteq \mathsf{FV}(S)$.

The candidate answer θ is grounding for x iff $\mathsf{ground}_S(\theta)(x) = \{\mathrm{g}\}$.

We can define a Galois connection $\langle \alpha_g, \gamma_g \rangle : [Seq \to \wp(\mathsf{Ans})] \rightleftharpoons [Seq \to \wp(\mathsf{GAns})]$ where

$$\alpha_g(A)(S) = \{\mathsf{ground}_S(\theta) \mid \theta \in A(S)\} \ . \tag{3.2.11}$$

Then, by composing α_g with α_c, we obtain an observable $\langle \alpha_g \circ \alpha_c, \gamma_c \circ \gamma_g \rangle : \mathsf{Int} \rightleftharpoons [Seq \to \wp(\mathsf{GAns})]$ for groundness answers. If $\beta \in \alpha_g(\mathsf{CAns}(S))$, then β is a *correct* groundness answer.

Example 3.2.3

Let us give some examples of sequents and their corresponding minimal correct groundness answers for intuitionistic logic.

sequent	**groundness answers**
$\forall y.p(y) \twoheadrightarrow \exists x.p(x)$	$\{x/g, y/g\}$ $\{x/ng, y/ng\}$
$\forall y.(p(a,y) \wedge p(y,b)) \twoheadrightarrow \exists x.p(x,x)$	$\{x/g, y/g\}$
$p(a), \forall y.(p(y) \supset p(y)) \twoheadrightarrow \exists x.p(x)$	$\{x/g, y/\emptyset\}$
$p(a) \vee r(b) \twoheadrightarrow \exists x.(p(x) \vee r(x))$	$\{x/g\}$
$r \wedge \neg r \twoheadrightarrow \exists x.p(x)$	$\{x/\emptyset\}$
$\forall y.p(y,y) \twoheadrightarrow \forall x_1.\exists x_2.p(x_1,x_2)$	$\{y/ng, x_1/ng, x_2/ng\}$
$\forall x_1.\ \exists x_2.\ p(x_1,x_2) \twoheadrightarrow \exists y.p(y,y)$	—
$\exists y.p(y) \twoheadrightarrow \exists x.p(x)$	$\{x/ng, y/ng\}$
$p(t(a)) \twoheadrightarrow \exists x.p(r(x))$	—
$p(a) \vee \exists x.r(x) \twoheadrightarrow \exists y.(p(y) \vee r(y))$	$\{x/ng, y/\{g, ng\}\}$

Note that if θ is a correct answer for S and x is a bound variable which only appears in essentially universal quantifiers, then θ is not grounding for x.

We may ask ourselves which is the correspondence between our observable and standard domains for analysis of groundness such as Pos [6]. It is possible to prove the following

Theorem 3.2.4 *Let* $\mathtt{P}$ *be a definite program and* $\mathtt{G}$ *a definite goal. We work in the realm of intuitionistic logic. Assume* $S = \Gamma \twoheadrightarrow \exists x_1, \ldots, x_n.\ G$ *is the corresponding pure Horn sequent. Consider* $x_1, \ldots, x_n$ *as propositional symbols and define the formula*

$$\Theta = \bigvee_{\beta \in \mathsf{GAns}(S)} \{\wedge_i x_i^{\beta(x_i)}\} \ , \tag{3.2.12}$$

where $x_i^{\{\mathrm{g}\}} = x_i$ *and* $x_i^{\{\mathrm{ng}\}} = \neg x_i$*. Then* Θ *is a positive formula.*

Moreover, if X *is the set of correct answers of* $\mathtt{G}$ *in* $\mathtt{P}$*, let* $\aleph = \alpha_{\mathsf{Pos}}(X)$*. Then* $\aleph$ *and* Θ *are equivalent formulas.*

Proof. First of all, since Horn clauses are always consistent, it follows from Lemma 3.1.5 that, if β is a correct groundness answer for S, then $\theta(x_i) \neq \emptyset$. Moreover, since we work in intuitionistic logic, each $\exists R$ inference rule can be applied only once for each variable, hence $\beta(x_i)$ is a singleton for each x_i. Therefore, $x_i^{\beta(x_i)}$ is well defined.

Now, consider an ordering $\leq$ on $\{g, ng\}$, such that $g \leq ng$ with the corresponding lifting to groundness answers. If $S = \Gamma \twoheadrightarrow \exists x_1, \ldots, x_n. G$ is a pure Horn sequent and β is a correct groundness answer, we call *existential* groundness answer the restriction of β to $\{x_1, \ldots, x_n\}$. We denote by $\mathsf{EAns}(S)$ the set of all the existential groundness answer for the sequent S. If $\beta \in \mathsf{EAns}(S)$, then $\beta' \in \mathsf{EAns}(S)$ for each $\beta' \leq \beta$. Therefore, consider the formula Θ. If S is a provable, there is an existential groundness answer β and, by the previous property, the answer $\beta' \leq \beta$ such that $\beta'(x_i) = \{g\}$ for each i is an element of EAns. Hence Θ is positive.

We still need to prove that Θ and $\aleph$ are equivalent. Given an assignment ν of truth values to $x_1, \ldots, x_n$, Θ is true iff there is an existential answer β such that $\beta(x_i) = \{g\}$ if $\nu(x_i) = \mathsf{true}$, $\beta(x_i) = \{ng\}$ otherwise. In turn, this means that there exists a correct answer θ such that $\mathsf{vars}(\theta(x_i)) \not\subseteq \mathsf{FV}(S)$ for each i with $\nu(x_i) = \mathsf{false}$. Since $\mathsf{FV}(S) = \emptyset$, it means that there is a correct answer θ such that $\theta(x_i)$ is not ground for each $x_i \in \nu^{-1}(\mathsf{false})$.

By the definition of α_{Pos}, we have that $\aleph$ is true under the assignment ν iff for each correct answers θ, $\theta(x_i)$ is ground if $\nu(x_i) = \mathsf{true}$. But this is equivalent to state that there exists a correct answer θ such that $\nu(x_i) = \mathsf{false}$ implies $\theta(x_i)$ not ground. Actually, this is the same statement which holds for Θ, then we have proved the required equivalence. ■

We can also build an abstraction for groundness analysis starting from resultants. Given a formula ϕ, we denote by $\alpha_v(\phi)$ an *abstract formula* obtained from ϕ by replacing each term with the set of free variables occurring in them. Let us denote by GSeq the set of *abstract set sequents* obtained from abstract formulas. A *groundness resultant* is a pair (β, ν) such that $\beta \in \mathsf{GAns}$ and ν is a multi-set of elements of GSeq. We write as GRes the set of groundness resultants. Then, we can define a Galois connection $\langle \alpha_{rg}, \gamma_{rg} \rangle : [\mathit{SetSeq} \to \wp(\mathsf{Res})] \rightleftharpoons [\mathsf{GSeq} \to \wp(\mathsf{GRes})]$, where

$$\begin{aligned}\alpha_{rg}(A)(\mathsf{S}) = \{(\mathsf{ground}_S(\theta), \{\alpha_v(\bar{S_1}), \ldots, \alpha_v(\bar{S_n}))\} \mid \\ (\theta, \{\bar{S_1}, \ldots, \bar{S_n}\}) \in A(\bar{S}), \mathsf{S} = \alpha_v(\bar{S})\} \ .\end{aligned} \tag{3.2.13}$$

We can compose α_{rg} with α_r to obtain a new observable which is well suited for a top-down analysis of *correct groundness resultants*.

Following the idea presented in the previous section, we have developed a prototypical abstract interpreter for intuitionistic logic, written in PROLOG, which is shown in Section 3.A.

Example 3.2.5

By applying our analyzer to the sequents in the Example 3.2.3 we obtain precisely the same set of minimal correct groundness answers, with the following exceptions:

sequent	**groundness answers**
$p(t(a)) \twoheadrightarrow \exists x.p(r(x))$	$\{x/ng\}$
$\forall x_1.\ \exists x_2.\ p(x_1, x_2) \twoheadrightarrow \exists y.p(y, y)$	$\{y/ng, x_1/ng, x_2/ng\}$

The previous example shows two different situations in which we loose precision. The first one is due to the fact that we abstract a term with the set of its free variables, discarding information about the functors. The second situations arises from the fact that the abstract domain is not enough powerful to keep track of the side condition for the $\exists L$ and the $\forall R$ introduction rules. To overcome this problem, we would need to improve the representation of abstract terms, by introducing a sort of *labeling* similar to what [56] does for hereditary Harrop formulas.

3.3 Conclusions

The usefulness of a general semantic framework strictly depends on its ability to be easily instantiated to well known cases while suggesting natural extensions to them. In the case of a framework which we want to use as a reference for the development of procedures for static analyses, we also require that theoretical descriptions can be implemented in a straightforward way.

We presented a semantic framework for sequent calculi modeled around the idea of the three semantics of Horn clauses and around abstract interpretation theory. With particular reference to groundness and correct answers, we have shown that well known concepts in the case of Horn clauses can be obtained as simple instances of more general definitions valid for much broader logics. This has two main advantages. First of all, we can instantiate the general concepts to computational logics other then Horn clauses, such as hereditary Harrop formulas. Moreover, the general definitions often make explicit the logical meaning of several constructions (such as correct answers), which are otherwise obscured by the use of small logical fragments. We think that, following this framework as a sort of guideline, it is possible to export most of the results for positive logic programs to the new logic languages developed following proof-theoretic methods.

Following this idea, we have presented a new definition for correct answers and correct resultants which can be applied to the full first order logic (both classical and intuitionistic). Moreover, we have shown that a well known abstraction of logic program semantics, namely groundness, can be easily re-introduced inside our framework. This definitions are so general that they can be reused with only slight changes for every logic system with standard quantifier rules, such as linear logic or modal logic.

Regarding the implementation of static analyzers from the theoretical description of the domains, not all the issues have been tackled. While a top-down analyzer can often be implemented in a straightforward manner, like our interpreter for groundness, the same does not hold for bottom-up analyzers. Since for a bottom-up analysis we have to build the entire abstract semantics of a logic, we need a way to isolate a finite number of "representative sequents" from which the semantics of all the others can easily be inferred: it is essentially a problem of compositionality.

We are actually studying this problem and we think that extending the notion of a logic $\mathcal{L}$ with the introduction of some *rules for the decomposition of sequents* will add to the theoretical framework the power needed to easily derive compositional $T_{\mathcal{L}}$ operators, thus greatly simplifying the implementation of bottom-up analyzers.

We also need a way to reduce non-determinism in abstract interpreters. This is a problem which has been tackled thoroughly in the field of automatic deduction. A standard solution is to use unification to reduce non-determinism in the introductions of quantifiers [13, 65].

Moreover, the problem of groundness analysis for intuitionistic logic could be further addressed. The precision we can reach with the proposed domain can be improved by refining the abstraction function, and the implementation of the analyzer could be reconsidered to make it faster. Finally, it should be possible to adapt the domain to work with linear logic. We would like to treat unification in our framework, and we want to do this without any major modification. We are working in the direction of defining an abstraction of proof skeletons using extra-logical variables such that the corresponding optimal abstract operators automatically computes the semantics trough unification.

We think that our approach to the problem of static analysis of logic programs is new. There are several papers focusing on logic languages other than Horn clauses [49] but, to the best of our knowledge, the problem has never been tackled before from the proof-theoretic point of view. An exception is [69], which, however, is limited to hereditary Harrop formulas and does not comes out with any real implementation of the theoretical framework. Also, in the field of automatic deduction, several papers have widened the concept of answer for resolution theorem provers [37, 16]. It would be interesting to check if there is some relation between general/hypothetical answers and correct answers. Note that an alternative presentation of correct answers can be found in [3]. In this paper, correct answers are presented by using model-theoretic concepts. We think that, w.r.t. [3], the new definitions of correct answers and groundness answers we introduce here give us more intuitive and accurate results and a much cleaner theory.

3.A The Text of the Analyzer

Here follows the PROLOG program which performs the analysis of groundness for the full intuitionistic logic. It is written in SWI-PROLOG 3.4 and does not work with previous versions, due to differences in some builtins. Since it is easier to implement, the program does not handle negation, whose expressive power is regained by the explicit introduction of the logical false. The analyzer is available online at http://www.di.unipi.it/~amato/papers.

```
/* Groundness analisys for full first-order intutionistic logic */

/* written by Gianluca Amato    */
/* developed in SWI-Prolog 3.4  */

/**** GENERIC PREDICATES *****/

%if S1 and S2 are ordered sets, subset_ordered(+S1,+S2) is true iff
% S1 is a subset of S2

subset_ordered([],_):-
        !.
subset_ordered([X|L],[X|R]):-
        !,
        subset_ordered(L,R).
subset_ordered(L,[_|R]):-
        subset_ordered(L,R).

% sort2(+X,+Y,-Z) is true iff Z is the ordered set whose
% elements are X and Y

sort2(X,X,[X]):-
!.
sort2(X,Y,[X,Y]):-
X@<Y, !.
sort2(X,Y,[Y,X]).

% minimal(X,K) is true if X is minimal in K

minimal(X,K):-
forall(member(Y,K), \+ subset(Y,X) ; X=Y).

% minimizeset(X,Y) takes a set of sets X and gives in Y only the
% minimal elements.

minimizeset(X,Y):-
minimizeset(X,X,Y).
minimizeset([],_,[]).
minimizeset([X|R],K,[X|R1]):-
minimal(X,K),
!,
minimizeset(R,K,R1).
minimizeset([_|R],K,R1):-
minimizeset(R,K,R1).

/**** LANGUAGE *****/

% atomic_formula(+F) is true iff F is an atomic formula

atomic_formula(Formula):-
Formula=..[Func|_],
\+(member(Func,[exists,and,or,forall,impl,false])).

% The abstraction function follows. They have the form
% absxxx(+Var,+X,-AbsX) where Var is the set of variables that should
% be considered bound, apart from those actually bound in X.

absterm(Vars,Term,[Term]):-
memberchk(Term,Vars),
!.

absterm(Vars,Term,AbsTerm):-
        Term=..[_|List],
        maplist(absterm(Vars),List,AbsList),
flatten(AbsList,AbsList2),
sort(AbsList2,AbsTerm).

absatom(Vars,Atom,AbsAtom):-
Atom=..[Symbol|List],
maplist(absterm(Vars),List,AbsList),
AbsAtom=..[Symbol|AbsList].

absformula(_Vars,false,false):-
!.
absformula(Vars,and(F1,F2),and(A1,A2)):-
!,
absformula(Vars,F1,A1), absformula(Vars,F2,A2).
absformula(Vars,or(F1,F2),or(A1,A2)):-
!,
absformula(Vars,F1,A1), absformula(Vars,F2,A2).
absformula(Vars,impl(F1,F2),impl(A1,A2)):-
!,
absformula(Vars,F1,A1),
absformula(Vars,F2,A2).
absformula(Vars,exists(Var,Formula),exists(Var,AbsFormula)):-
!,
union([Var],Vars,Vars2),
absformula(Vars2,Formula,AbsFormula).
absformula(Vars,forall(Var,Formula),forall(Var,AbsFormula)):-
!,
union([Var],Vars,Vars2),
absformula(Vars2,Formula,AbsFormula).
absformula(Vars,Atom,AbsAtom):-
absatom(Vars,Atom,AbsAtom).

% absseq(+Sequent,-AbsSequent) is true iff AbsSequent is the
% abstraction of Sequent. Note that the left-hand side of an abstract
% sequent is an ordered set.

absseq(Clauses->Goal,AbsClauses->AbsGoal):-
absformula([],Goal,AbsGoal),
maplist(absformula([]),Clauses,AbsClauses1),
sort(AbsClauses1,AbsClauses).

% absder(+Sequent,-Der) is true iff Der is the abstraction of the
% empty proofschema for Sequent.

absemptyder(Sequent,der([],[AbsSequent])):-
absseq(Sequent,AbsSequent).

/******** INFERENCE RULES *********/

% In groundterm(+Var,+Term,-Termg), Termg is the abstract term
% obtained by the abstract term Term by replacing Var with a
% ground term.

groundterm(Var,Term,Termg):-
        select(Var,Term,Termg),
!.
groundterm(_Var,Term,Term).

% In ngroundterm(+Var,+Term,-Termg), Termg is the abstract term
% obtained by the abstract term Term by replacing Var with a
% non-ground term.

ngroundterm(Var,Term,ng):-
member(Var,Term),
!.
ngroundterm(_Var,Term,Term).

replacevarinatom(_,Op,A,Ar):-
A=..[Symbol|List],
maplist(Op,List,Listr),
Ar=..[Symbol|Listr].

% replacevarinformula(+Var,+Op,+FormulaIn,-FormulaOut) gives in
% FormulaOut the result of replacing Var with a ground or non-ground
% term in FormulaIn, respectively if Op=groundterm or Op=ngroundterm.

replacevarinformula(Var,Op,and(F1,F2),and(Fr1,Fr2)):-
!,
replacevarinformula(Var,Op,F1,Fr1),
replacevarinformula(Var,Op,F2,Fr2).
replacevarinformula(Var,Op,or(F1,F2),or(Fr1,Fr2)):-
!,
replacevarinformula(Var,Op,F1,Fr1),
replacevarinformula(Var,Op,F2,Fr2).
replacevarinformula(Var,Op,impl(F1,F2),impl(Fr1,Fr2)):-
!,
replacevarinformula(Var,Op,F1,Fr1),
replacevarinformula(Var,Op,F2,Fr2).
replacevarinformula(Var,Op,exists(V,F),exists(V,Fr)):-
!,
replacevarinformula(Var,Op,F,Fr).
replacevarinformula(Var,Op,forall(V,F),forall(V,Fr)):-
!,
replacevarinformula(Var,Op,F,Fr).
replacevarinformula(_Var,_Op,false,false):-
!.
replacevarinformula(Var,Op,A,Ar):-
replacevarinatom(Var,Op,A,Ar).

% rule(?Name,+Sequent,-NewSequent,-NGround) gives the abstract
% version of the inference rules.

rule(id,Gamma->Goal,[],[]) :-
```

```
atomic_formula(Goal),
memberchk(Goal,Gamma).
rule(idfalse,Gamma->false,[],[]):-
memberchk(false,Gamma).
rule(falser,Gamma->G,[Gamma->false],[]):-
G\==false.
rule(implr,Gamma->impl(C,G),[NewGamma->G],[]):-
merge_set([C],Gamma,NewGamma).
rule(andr,Gamma->and(G1,G2),X,[]):-
sort2(Gamma->G1,Gamma->G2,X).
rule(orr,Gamma->or(G1,G2),[Gamma->G],[]):-
(
    G=G1;
    G=G2
).
rule(forallr,Gamma->forall(_,G),[Gamma->G2],[bind(Var,ng)]):-
replacevarinformula(Var,ngroundterm(Var),G,G2).
rule(existsr,Gamma->exists(Var,G),[Gamma->G2],[K]):-
(
    replacevarinformula(Var,groundterm(Var),G,G2),
    K=bind(Var,g);
    replacevarinformula(Var,ngroundterm(Var),G,G2),
    K=bind(Var,ng)
).
rule(andl,Gamma->G,[NewGamma->G],[]) :-
select(and(F1,F2),Gamma,Gamma0),
merge_set([F1],Gamma0,Gamma1),
merge_set([F2],Gamma1,NewGamma).
rule(orl,Gamma->G,X,[]):-
select(or(F1,F2),Gamma,Gamma0),
merge_set([F1],Gamma0,NewGamma1),
merge_set([F2],Gamma0,NewGamma2),
sort2(NewGamma1->G,NewGamma2->G,X).
rule(impll,Gamma->G,X,[]):-
select(impl(Body,Head),Gamma,Gamma0),
merge_set([Head],Gamma0,NewGamma),
sort2(Gamma0->Body, NewGamma->G, X).
rule(foralll,Gamma->G,[NewGamma->G],K):-
select(forall(Var,Clause),Gamma,Gamma0),
        replacevarinformula(Var,ngroundterm(Var),Clause,Clause1),
replacevarinformula(Var,groundterm(Var),Clause,Clause2),
(
    merge_set([Clause1],Gamma0,NewGamma),
    K=[bind(Var,ng)];
    merge_set([Clause2],Gamma0,NewGamma),
    K=[bind(Var,g)];
    merge_set([Clause1],Gamma0,Gamma1),
    merge_set([Clause2],Gamma1,NewGamma),
    K=[bind(Var,g), bind(Var,ng)]
).
rule(existsl,Gamma->G,[NewGamma->G],[bind(Var,ng)]):-
select(exists(Var,F),Gamma,Gamma0),
replacevarinformula(Var,ngroundterm(Var),F,F1),
merge_set([F1],Gamma0,NewGamma).


/******* ABSTRACT OPERATIONAL SEMANTICS ********/

% transitionstep(+AbsDerIn,-AbsDerOut)  is the implementation of the
% abstract transition system. It non deterministically gives
% in AbsDerOut the results of
% \alpha(\gamma(\{AbsDerInt\}) \glue (R \cup \epsilon))

transitionstep(der(NG,Tail),der(NG1,Tail1)):-
transitionstep(Tail,NG,Tail1,NG1,[]).

transitionstep([],NG,Acc,NG,Acc).
transitionstep([ASeq|Rest],NG,Tail1,NG1,Acc):-
rule(_,ASeq,ASeqSet,NGNew),
merge_set(NGNew,NG,NG2),
merge_set(ASeqSet,Acc,Acc1),
transitionstep(Rest,NG2,Tail1,NG1,Acc1).

%% operationalstep(+AbsIntIn,-AbsIntOut) is true iff
%% AbsIntOut=U_{L,\alpha}(AbsIntIn)

operationalstep([],[]).
operationalstep([Der|SetDers],NewSetDers):-
(setof(X,transitionstep(Der,X),NewListDers), ! ; NewListDers=[]),
operationalstep(SetDers,PartialSet),
merge_set(NewListDers,PartialSet,NewSetDers).

%% just some debugging stuff

operational(0,Der,Der).
operational(N,Der,NewDer):-
```

```
N>0,
        operationalstep(Der,NewDer1),
        N1 is N-1,
        operational(N1,NewDer1,NewDer).

%% omegaoperational(+AbsInt,-AbsIntOut) is true iff AbsIntOut is the
%% least fixpoint of the abstract U_L greater then AbsIntIn.

omegaoperational(SetDer,NewDer):-
operationalstep(SetDer,TempSetDer),
(TempSetDer=SetDer ->
    NewDer=SetDer
;
    omegaoperational(TempSetDer,NewDer)).

%% it is what we want.
%% answersemantics(+Seq,-Bindings) gives a correct approximation
%% of the correct groundness answers for Seq.

answersemantics(Seq,Binding):-
absemptyder(Seq,ADer),
omegaoperational([ADer],Semantics),
member(der(Binding,[]),Semantics).

%%
%% minimalsemantics(+Seq,-Bindings) gives the minimal correct
%% groundness answers fo Seq
%%

minimalsemantics(Seq,B):-
absemptyder(Seq,ADer),
omegaoperational([ADer],Semantics),
setof(Binding,member(der(Binding,[]),Semantics),List),
minimizeset(List,ListMin),
member(B,ListMin).

/******** EXTRA OPTIMIZED *******/

absproof(Seq,Bind):-
rule(_,Seq,ListSeq,B),
maplist(absproof,ListSeq,ListB),
flatten(ListB,FlatB),
append(B,FlatB,Bind).
```

```
sem(Seq,Bindings):-
absseq(Seq,ASeq),
setof(Bind,absproof(ASeq,Bind),Bindings).

/******** EXAMPLES  **********/

%% Here follows some examples. We mean by computed answer the best
%% possible groundness answer for the abstracted sequent.

% EXAMPLE 1. Best and computed answer: {{v1/g, v2/g},{v1/ng, v2/ng}}

example(1,[forall(v1, p(v1))]->exists(v2, p(v2))).

% EXAMPLE 2. Best and computed answer: {{v1/g, v2/g}}

example(2,[forall(v1,and(p(a,v1),p(v1,b)))]->exists(v2,p(v2,v2))).

% EXAMPLE 3. Best and computed answer: {{v2/g}}

example(3,[p(a),forall(v1,impl(p(v1),p(v1)))]->exists(v2,p(v2))).

% EXAMPLE 4. Best and computed answer: {{v1/g,v3/g},{v1/ng,v3/ng},
%                                        {v2/g,v3/g},{v2/ng,v3/ng}}

example(4,[forall(v1,p(t(v1))),forall(v2,p(s(v2)))]->
exists(v3,p(v3))).

% EXAMPLE 5. Best and computed answer:{{v1/g,v2/g,v3/g},
%        {v1/ng,v2/ng,v3/ng}}

example(5,[forall(v1,p(v1,v1))]->exists(v2,exists(v3,p(v2,v3)))).

% EXAMPLE 6. Best and computed answer: {{v1/ng,v2/g,v3/ng}}

example(6,[forall(v1,and(p(a,v1),p(v1,b)))]->
exists(v2,forall(v3,p(v2,v3)))).

% EXAMPLE 7. Best and computed answer: {{v1/g,v2/g},{v1/ng,v2/g}}

example(7,[forall(v1,and(p(a),r(b)))]->exists(v2,or(p(v2),r(v2)))).

% EXAMPLE 8. Best and computed answer: {{v1/g, v2/ng},{v1/ng,v2/g},
% {v1/g,v2/g}}

example(8,[and(p(a),r(b))]->
exists(v1,exists(v2,or(p(v1),r(v2))))).

% EXAMPLE 9. Best and computed answer: {{v1/g}}

example(9,[or(p(a),r(b))]->exists(v1,or(p(v1),r(v1)))).

% EXAMPLE 10. Best and computed answer: {{v1/ng,v2/ng}}

example(10,[exists(v1,p(v1))]->exists(v2,p(v2))).

% EXAMPLE 11. Best and computed answer: {{}}

example(11,[false]->exists(v1,p(v1))).

% EXAMPLE 12. Best and computed answer: {{v1/g},{v1/ng}}

example(12,[p(a)]->exists(v1,or(q(v1),p(a)))).

% EXAMPLE 13. Best and computed answer: {{v1/g},{v1/ng}}

example(13,[q(b),p(a)]->exists(v1,or(q(v1),p(a)))).

% EXAMPLE 14. Best and computed answer: {{v1/ng,v2/ng,v3/ng}}

example(14,[forall(v1,p(v1,v1))]->forall(v2,exists(v3,p(v2,v3)))).

% EXAMPLE 15.  Best and computed answer: {}

example(16,[]->exists(x,p(x))).

% EXAMPLE 16. Best answer {},
%             Computed Answer: {{v1/ng, v2/ng, v3/ng}}

example(16,[forall(v1,exists(v2,(p(v1,v2))))]->forall(v3,p(v3,v3))).

% EXAMPLE 17. Best answer {},
%             Computed Answer: { {v1/ng, v2/ng, v3/ng} }

example(17,[forall(v1,exists(v2,(p(v1,v2))))]->exists(v3,p(v3,v3))).
```

```
% EXAMPLE 18. Best and omputed Answer: { {v1/ng, v2/{g,ng}} }

example(18,[or(p(a),exists(v1,r(v1)))]->exists(v2,or(p(v2),r(v2)))).
```

Part II

Categorical Semantics for Logic Programs

Chapter 4

Categories

Abstract

In this chapter, we fix the notations for category theory which will be used in the rest of the thesis. The presentation is far from being complete, but we try to give enough details on those arguments which are more relevant for the thesis. For this reason, particular care is devoted to indexed categories, functors between them and premonoidal structures.

4.1 Basic Notations for Category Theory

Category theory studies *objects* and *morphisms* between them. Objects and morphisms can be viewed as a generalization of sets and functions. An important difference, which is the main point of all the theory of categories, is that any inspection on the structure of objects is forbidden and objects can be analyzed only by their interaction with other objects through morphisms. A complete account of category theory can be found in [7, 10, 32]. Here we only want to fix the basic terminology.

Given a category $\mathbb{C}$, we denote by $\mathrm{Ob}_{\mathbb{C}}$ and $\mathrm{Mor}_{\mathbb{C}}$ the corresponding collections of objects and morphisms (or *arrows*) respectively. Given an arrow f, $\mathsf{dom}(f)$ is the *domain* (or *source*) of f and $\mathsf{cod}(f)$ the *codomain* (or *target*). For every object A, id_A is the *identity* arrow for A. We write $f : A \to B$ to denote that f is an arrow with domain A and codomain B. With $f : A \rightarrowtail B$ we denote a monic arrow. Given $f : A \to B$ and $g : B \to G$, the *composition* of f and g is denoted by $g \circ f$ or by $f\,;g$ according to Freyd's [32] convention. We prefer the latter expression, since we think it is more intuitive. Given two objects A and B in $\mathbb{C}$, $Hom(A, B)$ is the collection of arrows with domain A and codomain B. A *functor* F from the category $\mathbb{C}$ to the category $\mathbb{D}$ is denoted by $F : \mathbb{C} \to \mathbb{D}$, while, if $F : \mathbb{C} \to \mathbb{D}$ and $G : \mathbb{C} \to \mathbb{D}$, then $\eta : F \to G$ is a *natural transformation* from F to G. With $\mathsf{Func}(\mathbb{C}, \mathbb{D})$ or $\mathbb{D}^{\mathbb{C}}$ we denote the category of functors from $\mathbb{C}$ to $\mathbb{D}$ and their natural transformations.

An extension of categories are *2-categories*. If $\mathbb{C}$ is a 2-category, we denote by $\mathrm{Cell}_{\mathbb{C}}$ the collections of *2-cells*. If η is a 2-cell with *vertical domain* $F = \mathsf{dom}(\eta)$ and *vertical codomain* $G = \mathsf{cod}(\eta)$, then we write $\eta : F \Rightarrow G$. The *vertical composition* of two cells $\eta : F \Rightarrow G$ and $\gamma : G \Rightarrow H$ is $\gamma \circ \eta : F \Rightarrow H$ or $\eta\, ;\gamma$. Given $\eta : F \Rightarrow G$, then $\mathsf{dom}(F) = \mathsf{dom}(G)$ is the *horizontal domain* of η and $\mathsf{cod}(F) = \mathsf{cod}(G)$ is the *horizontal codomain*. If $\eta : F \Rightarrow G$ and $\gamma : H \Rightarrow K$ are two 2-cells such that $\mathsf{cod}(G) = \mathsf{dom}(H)$, then the *horizontal composition* of η and γ is denoted by $\gamma \star \eta : H \circ F \Rightarrow K \circ G$. If $\eta = id_F$, we write γF as a short form of $\gamma \star id_F$. If $\gamma = id_H$, we write $H\eta$ instead of $id_H \star \eta$.

We denote with Cat the 2-category of small categories, functors and natural transformations, and with *Set* the category of sets and total functions. If $\mathbb{C}$ is category, $\mathbb{C}^{\mathrm{o}}$ is the *opposite* category of $\mathbb{C}$. If $F : \mathbb{C} \to \mathbb{D}$ is a functor, $F^{\mathrm{o}} : C^{\mathrm{o}} \to D^{\mathrm{o}}$ is the corresponding functor between opposite categories.

Given objects A and B in $\mathbb{C}$, we denote by $A \times B$, when it does exist, the binary product of A and B (which is unique up to iso), with projection morphisms given by $\pi_1^{A,B}$ and $\pi_2^{A,B}$. If $f : C \to A$ and $g : C \to B$, we denote by $\langle f, g\rangle$ the unique arrow from C to $A \times B$ such that $\langle f, g\rangle\, ; \pi_1^{A,B} = f$ and $\langle f, g\rangle\, ; \pi_2^{A,B} = g$. If $\mathbb{C}$ has products, we can extend $\times$ to a functor from $\mathbb{C} \times \mathbb{C}$ to $\mathbb{C}$. With 1 we denote the *terminator*, while $!_A$ is the only arrow from A to 1.

Dually, with $A + B$ we denote the binary coproduct of A and B, with $in_1^{A,B} : A \to A + B$ and $in_2^{A,B} : B \to A + B$ the injection arrows. If $f : A \to C$ and $g : B \to C$, with $[f, g]$ we denote the unique arrow such that $in_1^{A,B}\, ; [f, g] = f$ and $in_2^{A,B}\, ; [f, g] = g$. The coterminator is denoted by 0.

In general, if $\{X_j\}_{j\in J}$ is a collection of objects in $\mathbb{C}$, we denote by $\wedge_{j\in J} X_j$ and by $\vee_{j\in J} X_j$ respectively the product and the coproduct of the family of objects. Projections and injections of the i-th component are denoted by π_i and in_i. If $\{f_j\}_{j\in J}$ is a family of arrows $f_j : X \to X_j$, we denote by $\langle \{f_j\}_{j\in J}\rangle$ the unique arrow from X to $\wedge_{j\in J} X_j$ such that $\langle \{f_j\}_{j\in J}\rangle ; \pi_i = f_i$ for each $i \in J$. An analogous notation is used for coproducts. We call *finite product category* or *FP category* a category which has all the products of the kind $\wedge_{j\in J} X_j$ with J finite.

If I and $\mathbb{C}$ are categories, a *diagram* in $\mathbb{C}$ with *shape* I is a functor $\Gamma : I \to \mathbb{C}$. A *cone* for a diagram Γ is written as

$$(l, \{\eta_i\}_{i\in \mathrm{Ob}_I})\,, \tag{4.1.1}$$

where $l \in \mathrm{Ob}_{\mathbb{C}}$ is the *vertex* of the cone and $\eta_i : \Gamma(i) \to l$ are arrows in $\mathbb{C}$. The same notation we use for *cocones*. We denote by $\lim \Gamma$ and $\mathsf{colim}\, \Gamma$ the limit and colimit of Γ.

4.2 Some Categorical Properties

In the previous section we have summarized the basic terminology of category theory. There are other basic concepts we will use in the rest of the thesis which are not well standardized. Therefore, we think they deserve an explicit definition and some words of explanation. To this task is devoted the current section.

Definition 4.2.1 (Maximal Element) *A* maximal element *in a category $\mathbb{C}$ is an object A such that, for each arrow f with $\mathsf{dom}(f) = A$, f is an iso.*

Theorem 4.2.2 *If $\mathbb{C}$ has a terminator, then either all the terminators are maximal elements, or $\mathbb{C}$ has no maximal element at all.*

Proof. Assume A is a terminal element in $\mathbb{C}$ which is not maximal. It means there exists $f : A \to A'$ which is not an iso. Given B object of $\mathbb{C}$, consider the arrow $!_B\,; f$. This is an arrow from B which is not an iso. Actually, assume $h : A' \to B$ such that $h;!_\rho\,; f = id_{A'}$. Since $h;!_\rho =!_{A'}$, we have $!_{A'}\,; f = id_{A'}$. However, $f;!_{\sigma'} = id_A$, hence f is an iso, and this is an absurd. ■

Assume $\{\Gamma_j\}$ with $j \in J$ a collection of diagrams of shape I in $\mathbb{C}$. If $\mathbb{C}$ has enough products, it is possible to build a diagram $\Gamma = \bigotimes_{j\in J} \Gamma_j$ such that $\Gamma(i) = \wedge_{j\in J}\Gamma_j(i)$ for each $i \in \mathrm{Ob}_I$. In the same way, given cones $\{L_j\}_{j\in J}$ for the Γ_j's, we can build a cone $\bigotimes_{j\in J} L_j$ for Γ. If

$$L_j = (l_j, \{\eta_{j,i}\}_{i\in \mathrm{Ob}_I}) \tag{4.2.1}$$

then

$$\bigotimes_{j\in J} L_j = \Big(\wedge_{j\in J} l_j, \{\wedge_{j\in J}\eta_{j,i}\}_{i\in \mathrm{Ob}_I}\Big) \tag{4.2.2}$$

On the contrary, given $L = (l, \{\eta_i\}_{i\in\mathrm{Ob}_I})$ a cone for Γ, for each $j \in J$ it induces a cone $L_j = (l, \{\eta_i\,; \pi_j\}_{i\in\mathrm{Ob}_I})$ for Γ_j, where π_j is the projection on the j-th component of $\Gamma(i)$.

Theorem 4.2.3 *Given a collection of diagrams $\{\Gamma_j\}_{j\in J}$ with corresponding limits $\{L_j\}_j$, then*

$$\bigotimes_{j\in J} L_j = \lim \bigotimes_{j\in J} \Gamma_j$$

Proof. Assume $L' = (l', \{\eta'_i\}_{i\in I})$ is another cone for $\bigotimes_{j\in J} \Gamma_j$. Then, for each j, L'_j is a cone for Γ_j. Since L_j is a limiting cone, there exist an unique arrow $\delta_j : L'_j \to L_j$. It is obvious that $\langle\{\delta_j\}_{j\in J}\rangle$ is the unique arrow from L' to $\bigotimes_{j\in J} L_j$. ■

4.3 Categories and First Order Languages

Categories with finite products (FP categories) can be used to give semantics to first order languages. The basic translation almost always follows the general pattern described in [48]. The main idea is that objects, arrows and monics are the categorical counterpart of sorts, terms and predicates. Let us examine how all this proceeds.

Assume given an FP category $\mathbb{C}$, a many sorted first order signature (Σ, Π) and a set of sorted variables V. We call $\mathbb{C}$-structure on (Σ, Π) a function M that maps each sort σ to an object $M(\sigma) \in \mathrm{Ob}_{\mathbb{C}}$ and each function symbol f of arity n, input sorts $\sigma_1, \ldots, \sigma_n$ and output sort ρ to an arrow $M(f) : M(\sigma_1) \times \cdots \times M(\sigma_n) \to M(\rho)$. Constants are functions of arity 0: for each constant c of sort ρ, we have $M(c) : 1 \to M(\rho)$ where 1 is the terminator of $\mathbb{C}$.

A $\mathbb{C}$-structure M induces an interpretation for all the terms in $T_\Sigma(V)$. Given a sequence $\vec{x} = x_1, \ldots, x_n$ of variables of sorts $\vec{\sigma} = \sigma_1, \ldots, \sigma_n$ respectively, we define $M(\vec{x}) = M(\vec{\sigma})$ as $M(\sigma_1) \times \cdots \times M(\sigma_n)$. Given a term t of sort ρ having all the variables among $\vec{x}$, we define an arrow $M_{\vec{x}}(t) : M(\vec{x}) \to M(\rho)$ as follows:

- $\mathbf{t} = \mathbf{x_i}$] $M_{\vec{x}}(x_i)$ is the projection $\pi_i : M(\vec{x}) \to M(\rho_i)$.
- $\mathbf{t} = \mathbf{c}$] For a constant c of sort ρ, $M_{\vec{x}}(c)$ is defined as the following composition
$$M(\vec{x}) \xrightarrow{!_{M(\vec{x})}} 1 \xrightarrow{M(c)} M(\rho) \ .$$
- $\mathbf{t} = \mathbf{f}(\mathbf{t_1}, \ldots, \mathbf{t_m})$] if each t_i is of sort σ_i, then $M_{\vec{x}}(t)$ is defined as the following composition:
$$M(\vec{x}) \xrightarrow{\langle M_{\vec{x}}(t_1), \ldots, M_{\vec{x}}(t_m) \rangle} M(\vec{\sigma}) \xrightarrow{M(f)} M(\rho) \ .$$

Note that most of this definitions are only meaningful up to iso. To overcome this problem we can work in a category with some sort of canonical products, such as a τ-category [32].

It is possible to interpret in $\mathbb{C}$ atomic formulas of first order logic. For every predicate symbol R in Π with arity n and sorts $\sigma_1, \ldots, \sigma_n$, let us fix a monic $M(R) \rightarrowtail M(\vec{\sigma})$. For an atomic formula $\phi = R(t_1, \ldots, t_m)$ with all the variables among $\vec{x}$, we define $M_{\vec{x}}(\phi)$ as the pullback of the monic $M(R) \rightarrowtail M(\vec{\sigma})$ along the arrow $\langle M_{\vec{x}}(t_1), \ldots, M_{\vec{x}}(t_m) \rangle$. The formula ϕ is considered true when $M_{\vec{x}}(\phi)$ is isomorphic to $M(\vec{x})$.

In this context, substitutions and unification have a direct counterpart, too. Let $\theta = [x_1/t_1, \ldots, x_n/t_n]$ be an idempotent substitution and assume that all the variables in $\mathsf{range}(\theta)$ are in the sequence $\vec{y}$. Then can define a corresponding categorical substitution $\Theta_{\vec{y}}$ as the morphism:

$$M(\vec{y}) \xrightarrow{\langle M_{\vec{y}}(t_1), \ldots, M_{\vec{y}}(t_n) \rangle} M(\vec{x})$$

It is possible to prove that, given a term s with all the variables among $\vec{x}$, $M_{\vec{y}}(s\theta) = \Theta_{\vec{y}} \,;\, M_{\vec{x}}(s)$.

Given two terms s and t of the same sort ρ with all the variables in $\vec{x}$, if θ is an unifier then $\Theta_{\vec{x}}$ equalizes $M_{\vec{x}}(s)$ and $M_{\vec{x}}(t)$. If $\mathbb{C}$ is the Lawvere algebraic category [44] for Σ, if θ is a most general unifier, then $\Theta_{\vec{x}}$ is an equalizer.

Given this interpretation of classical first order languages, the successive abstraction step is considering the FP category itself as the language, without relying on any interpreted syntactic objects. This is common in categorical logic and it has the great advantage of working with syntax without having to worry about variables and consequent name clashes. We will talk therefore of objects as sorts, arrows as terms or substitutions, equalizers as mgu's and monics as predicates. A particular notation must be explained to this regard: if X is a subobject of σ and $t : \rho \to \sigma$ an arrow, with $X(t)$ we mean the pullback of X along t. This stresses the fact that pullback is the categorical counterpart of instantiation of predicates symbols.

4.4 Fixpoints of Functors

In the field of semantics of programming languages, one of the most widely used tools is the Knaster-Tarski theorem [11] on fixpoints of functions in partial ordered sets. Since categories can often be regarded as generalized posets, we should expect something similar for functors.

Given an endofunctor $F : \mathbb{C} \to \mathbb{C}$, an *algebra* is a pair (σ, t) where $\sigma \in \mathrm{Ob}_{\mathbb{C}}$ and $t : F\sigma \to \sigma$ is an arrow. A *fixpoint* for F is an algebra (σ, t) where t is an isomorphism. If we think to an arrow $t : \rho \to \sigma$ as a generalization of the property $\rho \leq \sigma$, algebras correspond to generalizations of pre-fixpoints. It is possible to define the dual notion of coalgebra (post-fixpoints).

A morphism from the algebra (σ, t) to (ρ, r) is an arrow $s : \sigma \to \rho$ such that the following diagram commute:

$$\begin{tikzcd}
\sigma \arrow[d, "s"'] & F\sigma \arrow[l, "t"'] \arrow[d, "Fs"] \\
\rho & F\rho \arrow[l, "r"]
\end{tikzcd}$$

An initial algebra for F is a fixpoint. Moreover, it is a *least fixpoint*. Hence, least fixpoints are unique up to iso.

Now, assume that $\mathbb{C}$ has colimits of ω-chains, F preserves such colimits and $(\perp, \iota)$ is a coalgebra for F. We obtain the following:

$$\perp \xrightarrow{\iota} F\perp \xrightarrow{F\iota} \cdots \xrightarrow{F^n\iota} F^n\perp \cdots$$

We can take the colimit $F^\omega\perp$ of this diagram. It is possible to prove that there exists $\delta : F(F^\omega\perp) \to F^\omega\perp$ such that $(F^\omega\perp, \delta)$ is a fixpoint for F. Moreover, assume (V, χ) is an algebra and $\eta : \perp \to V$ an arrow such that the following diagram commute:

$$\begin{tikzcd}
\perp \arrow[r, "\iota"] \arrow[d, "\eta"'] & F\perp \arrow[d, "F\eta"] \\
V & F(V) \arrow[l, "\chi"']
\end{tikzcd}$$

We say in this case that (V, χ) extends $(\perp, \iota)$ with η. We obtain as a result that there exists an unique arrow from the above fixpoint to (V, χ). It is possible to rephrase this property stating that $(F^\omega\perp, \delta)$ is the least algebra that extends $(\perp, \iota)$.

4.5 Monoidal Structures

A *monoidal structure* for a category is a construction that allows us to combine objects and arrows together with some properties of associativity. There are several kinds of monoidal structures, according to the possible forms of combining which are allowed. The fundamental concept, however, is that of *monoidal category*.

Definition 4.5.1 (Monoidal category) *A* monoidal category *is a category* $\mathbb{C}$ *together with a functor* $\otimes : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$*, an identity element* $\top \in \mathrm{Ob}_{\mathbb{C}}$*, and natural isomorphisms*

$$\alpha_{A,B,C} : A \otimes (B \otimes C) \to (A \otimes B) \otimes C \tag{4.5.1}$$

$$\lambda_A : \top \otimes A \to A \tag{4.5.2}$$

$$\rho_A : A \otimes \top \to A \tag{4.5.3}$$

subject to the following coherence conditions:

$$\begin{tikzcd}
A \otimes (B \otimes (C \otimes D)) \arrow[r, "\alpha"] \arrow[d, "id\otimes\alpha"] & (A \otimes B) \otimes (C \otimes D) \arrow[r, "\alpha"] & ((A \otimes B) \otimes C) \otimes D \\
A \otimes ((B \otimes C) \otimes D) \arrow[rr, "\alpha"] & & (A \otimes (B \otimes C)) \otimes D \arrow[u, "\alpha\otimes id"']
\end{tikzcd} \tag{4.5.4}$$

$$\begin{tikzcd}
A \otimes (\top \otimes B) \arrow[rr, "\alpha"] \arrow[dr, "id\otimes\lambda"'] & & (A \otimes \top) \otimes B \arrow[dl, "\rho\otimes id"] \\
& A \otimes B &
\end{tikzcd} \tag{4.5.5}$$

A strict monoidal category *is a monoidal category such that the natural isomorphisms c, l and r are identities.*

Here, the coherence conditions guarantee that every diagram built from α, γ, ρ commutes (see [41]). A detailed account of monoidal categories and their properties can be found in [7].

4.5.1 Premonoidal Categories

Now, we consider another kind of monoidal structure with weaker properties. It gives origin to *premonoidal categories*, which have been introduced in [62] as a tool for analyzing side effects in the denotational semantics of programming languages.

Definition 4.5.2 (Binoidal category) *A* binoidal category *is a category* $\mathbb{C}$ *with a pair of functors* $\otimes_l : \mathbb{C} \times \mathrm{Ob}_{\mathbb{C}} \to \mathbb{C}$ *and* $\otimes_r : \mathrm{Ob}_{\mathbb{C}} \times \mathbb{C} \to \mathbb{C}$ *such that* $A \otimes_l B = A \otimes_r B$ *for each pair of objects in* $\mathbb{C}$.

Since there are no ambiguities, we can denote both functors with $\otimes$, writing things such as $A \otimes B$, $A \otimes g$ or $f \otimes B$. In general, it does not make sense to write $f \otimes g$, unless f or g is a central morphism.

Definition 4.5.3 (Central morphism) *A morphism $f : A \to A'$ in a binoidal category is* central *if, for every morphism $g : B \to B'$,*

$$(B' \otimes f) \circ (g \otimes A) = (g \otimes A') \circ (B \otimes f) \tag{4.5.6}$$

$$(f \otimes B') \circ (A \otimes g) = (A' \otimes g) \circ (f \otimes B) \tag{4.5.7}$$

In this case, we use the notations $g \otimes f$ and $f \otimes g$ respectively.

Definition 4.5.4 (Premonoidal category) *A* premonoidal category *is a binoidal category with an object $\top \in \mathrm{Ob}_{\mathbb{C}}$ and central natural isomorphisms*

$$\alpha_{A,B,C} : A \otimes (B \otimes C) \to (A \otimes B) \otimes C \tag{4.5.8}$$

$$\lambda_A : \top \otimes A \to A \tag{4.5.9}$$

$$\rho_A : A \otimes \top \to A \tag{4.5.10}$$

subject to the same coherence conditions of Definition 4.5.1. A premonoidal category is strict when the natural isomorphisms are identities.

Obviously, when we work with premonoidal categories we are interested in functors which preserve the premonoidal structure.

Definition 4.5.5 (Premonoidal functor) *If $\mathbb{C}$ and $\mathbb{D}$ are two premonoidal categories, a (strict)* premonoidal functor *$F : \mathbb{C} \to \mathbb{D}$ is a functor which sends central maps to central maps and preserve the monoidal structure on the nose, i.e.*

$$F(f \otimes A) = F(f) \otimes F(A)$$
$$F(A \otimes f) = F(A) \otimes F(f)$$
$$F(\top) = \top$$
$$F(\alpha_{A,B,C}) = \alpha_{F(A),F(B),F(C)}$$
$$F(\lambda_A) = \lambda_{F(A)}$$
$$F(\rho_A) = \rho_{F(A)}$$

Finally, we can also define a notion of natural transformation between premonoidal functors.

Definition 4.5.6 (Premonoidal natural transformation) *If $F, G : \mathbb{C} \to \mathbb{D}$ are two premonoidal functors, a premonoidal natural transformation $\eta : F \Rightarrow G$ is a central natural transformation such that $\eta_{A \otimes B} = \eta_A \otimes \eta_B$ and $\eta_\top = id_\top$.*

If we denote by mCat the category of monoidal categories and monoidal functor and by pmCat the category of premonoidal categories and premonoidal functors, we have obvious forgetful functors $U_m : \mathsf{mCat} \to \mathsf{pmCat}$ and $U_{pm} : \mathsf{pmCat} \to \mathsf{Cat}$. The corresponding left adjoints give a way to build the free monoidal/premonoidal category generated by a category $\mathbb{C}$.

4.6 Indexed Categories

Indexed categories and the related notion of *fibration* have been used extensively in categorical logic. A broad coverage of these topics can be found in [39].

Definition 4.6.1 (Indexed categories) *An* indexed category *over a* base category $\mathbb{C}$ *(i-category) is a functor* $\mathbb{C}^{\mathrm{o}} \to \mathsf{Cat}$.

If $\mathcal{P}$ is an indexed category over $\mathbb{C}$ and f is an arrow in the base category, $\mathcal{P}f$ is called *reindexing functor*. We will write $f^\sharp$ instead of $\mathcal{P}f$ when this does not cause ambiguities.

What we call indexed category is well known in literature as *strict* indexed category. Since we do not deal with the non-strict variant, we omit the prefix.

Definition 4.6.2 (Indexed functors) *Given $\mathcal{P}$ and $\mathcal{Q}$ indexed categories over $\mathbb{C}$ and $\mathbb{D}$ respectively, an* indexed functor *(i-functor) from $\mathcal{P}$ to $\mathcal{Q}$ is given by*

- *a* change of base *functor $F : \mathbb{C} \to \mathbb{D}$,*
- *a natural transformation $\tau : \mathcal{P} \to (F^{\mathrm{o}} ; \mathcal{Q})$.*

If $H = (F, \tau)$ is a functor of indexed categories, we will write $H\sigma$ in the place of τ_σ for all $\sigma \in \mathrm{Ob}_{\mathbb{C}}$.

In the following, when we have an indexed functor $H = (F, \tau) : \mathcal{P} \to \mathcal{Q}$, we will often use $u^\sharp$ for $\mathcal{P}u$ and $u^\#$ for $\mathcal{Q}(Fu)$. Indexed categories and their functors form a new category ICat. The identity morphism on $\mathcal{P}$ is

$$id_{\mathcal{P}} = (id_{\mathbb{C}}, \{id_{\mathcal{P}\sigma}\}_{\sigma \in \mathrm{Ob}_{\mathbb{C}}}) \tag{4.6.1}$$

while composition of $H = (F, \tau)$ and $H' = (F', \tau')$ is defined as

$$H ; H' = \big(F ; F', \tau ; (F \star \tau')\big) \tag{4.6.2}$$

In the same way as for Cat we can define natural transformations of indexed functors, so that ICat becomes a 2-category.

Definition 4.6.3 (Indexed natural transformations) *Given (F, τ) and (F', τ') two indexed functors from $\mathcal{P}$ to $\mathcal{Q}$, an* indexed natural transformation *(i-nat) between them is given by a pair (χ, δ) such that*

- *$\chi : F \to F'$ is a natural transformation,*
- *δ is a $\mathbb{C}$-indexed family of natural transformations*

$$\delta_\sigma : \tau_\sigma \to \tau'_\sigma ; \mathcal{Q}(\chi_\sigma)$$

where $\mathbb{C}$ is the base category of $\mathcal{P}$ and, for each $f : \sigma \to \sigma'$ in $\mathbb{C}$, we have

$$\mathcal{P}(f) \star \delta_\sigma = \delta_{\sigma'} \star \mathcal{Q}(Ff)$$

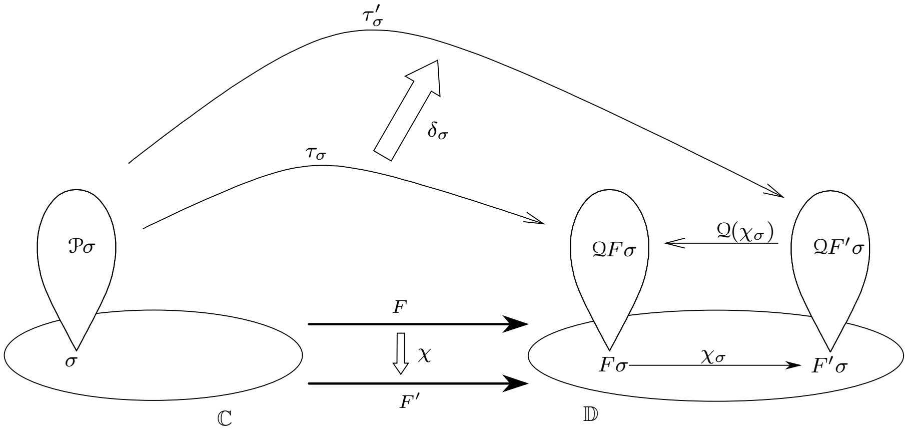

Figure 4.1: Indexed categories diagram

The identity natural transformation for $H = (F, \tau)$ is

$$id_H = (id_F, \{id_{\tau_\sigma}\}_{\sigma \in \mathrm{Ob}_\mathbb{C}}) \tag{4.6.3}$$

while composition of $\Delta = (\chi, \delta)$ and $\Delta' = (\chi', \delta')$ is

$$\Delta\,;\Delta' = \big(\chi\,;\chi', \{\delta_\sigma\,;(\delta'_\sigma \star \chi'_\sigma)\}_{\sigma \in \mathrm{Ob}_\mathbb{C}}\big) \tag{4.6.4}$$

Figure 4.1 depicts the relationships between indexed categories, functors and natural transformations. We call $\mathsf{IFunc}(\mathcal{P}, \mathcal{Q})$ the category of i-functors between i-categories $\mathcal{P}$ and $\mathcal{Q}$ and their i-nats. Often we will restrict our attention to the subcategory $\mathsf{IFunc}_F(\mathcal{P}, \mathcal{Q})$ whose objects are i-functors (F, τ) with F fixed and whose arrows are pairs (id_F, δ). When it is clear from the context that we are talking of objects and arrows in $\mathsf{IFunc}_F(\mathcal{P}, \mathcal{Q})$, we will only write τ and δ for i-functors and i-nats.

4.7 Limits of Categories of Indexed Functors

Given $\mathbb{C}, \mathbb{D} \in \mathrm{Ob}_{\mathsf{Cat}}$, the functor category $\mathsf{Func}(\mathbb{C}, \mathbb{D})$ and $\sigma \in \mathrm{Ob}_\mathbb{C}$, we define the *evaluation functor* $\mathsf{Ev}_\sigma : \mathsf{Func}(\mathbb{C}, \mathbb{D}) \to \mathbb{D}$ given by

$$\mathsf{Ev}_\sigma(F) = F\sigma \tag{4.7.1}$$

$$\mathsf{Ev}_\sigma(\eta) = \eta_\sigma \tag{4.7.2}$$

If $\Gamma : I \to \mathsf{Func}(\mathbb{C}, \mathbb{D})$ is a diagram and $L = (F, \{\alpha_i\}_{i \in \mathrm{Ob}_I})$ a cone, for each $\sigma \in \mathrm{Ob}_\mathbb{C}$ we obtain a diagram in $\mathbb{D}$, i.e. $\Gamma\,;\mathsf{Ev}_\sigma$, and a corresponding instance cone $L_\sigma = (F\sigma, \{(\alpha_i)_\sigma\}_{i \in \mathrm{Ob}_I})$.

What follows is a well known result regarding limits in functional categories. We will see that a similar theorem holds when $\mathsf{Func}(\mathbb{C}, \mathbb{D})$ is replaced by $\mathsf{IFunc}_F(\mathcal{P}, \mathcal{Q})$.

Proposition 4.7.1 *Given categories $\mathbb{C}$ and $\mathbb{D}$ and a diagram $\Gamma : I \to \mathsf{Func}(\mathbb{C}, \mathbb{D})$, assume that $\Gamma \,;\mathsf{Ev}_\sigma$ has a limit L_σ for each $\sigma \in \mathrm{Ob}_\mathbb{C}$. Then*

- *Γ has a limit L with L_σ as instances;*
- *if L' is a limit of Γ, L'_σ is a limit of $\Gamma \,;\mathsf{Ev}_\sigma$ for all $\sigma \in \mathrm{Ob}_\mathbb{C}$.*

For each diagram in $\mathbb{D}$ it is possible to fix, when it exists, a *canonical* limit. Then, Property 4.7.1 brings an obvious definition of canonical limits in $\mathsf{Func}(\mathbb{C}, \mathbb{D})$ as the cones whose instances are canonical limits. As we should expect, there is at most one canonical limit for each diagram.

If $\Gamma : I \to \mathsf{Func}(\mathbb{C}, \mathbb{D})$ is a diagram and $F : \mathbb{D} \to \mathbb{B}$ is a functor, we define the new diagram $\Gamma^F : I \to \mathsf{Func}(\mathbb{C}, \mathbb{B})$ given by

$$\Gamma^F(i) = \Gamma(i) \,; F \tag{4.7.3}$$

$$\Gamma^F(f) = \Gamma(f) \star F \tag{4.7.4}$$

for $i \in \mathrm{Ob}_I$ and $f : i \to j$ in I. If $F : \mathbb{B} \to \mathbb{C}$ we define Γ_F in the same way but reversing the order of compositions.

Lemma 4.7.2 *Given a diagram Γ in $\mathsf{Func}(\mathbb{C}, \mathbb{D})$ and $\sigma \in \mathrm{Ob}_\mathbb{C}$, we have the following identities:*

1. *if $F : \mathbb{D} \to \mathbb{B}$, then $\Gamma^F \,; \mathsf{Ev}_\sigma = \Gamma \,; \mathsf{Ev}_\sigma ;F$,*
2. *if $F : \mathbb{B} \to \mathbb{C}$, then $\Gamma_F \,; \mathsf{Ev}_\sigma = \Gamma \,; \mathsf{Ev}_{F\sigma}$.*

Proof. Trivially follows by definition of Γ^F, Γ_F and Ev_σ. ■

Using the results of Property 4.7.1 and subsequent observation, it is possible to prove the following

Corollary 4.7.3 *Let $\mathbb{C}$ and $\mathbb{D}$ be two categories and $\Gamma : I \to \mathsf{Func}(\mathbb{C}, \mathbb{D})$ a diagram. Assume, for each $\sigma \in \mathrm{Ob}_\mathbb{C}$, the existence of a (canonical) limit for $\Gamma \,; \mathsf{Ev}_\sigma$ and that Γ itself has (canonical) limit $(G, \{\eta_i\}_{i \in \mathrm{Ob}_I})$. We have the following results:*

1. *if $F : \mathbb{B} \to \mathbb{C}$ is a functor, then $(F \,; G, \{F \star \eta_i\}_{i \in \mathrm{Ob}_I}\})$ is a (canonical) limit for the diagram Γ_F,*
2. *if $F : \mathbb{D} \to \mathbb{B}$ is a functor that preserves (canonical) limits of $\Gamma \,; \mathsf{Ev}_\sigma$ for all σ's, then $(G \,; F, \{\eta_i \star F\}_{i \in \mathrm{Ob}_I})$ is a (canonical) limit for the diagram Γ^F.*

Now we move our attention to limits of diagrams over indexed functors, since they will be used extensively later, in the section regarding the fixpoint semantics. Given $\mathcal{P}$ and $\mathcal{Q}$ indexed categories over base categories $\mathbb{C}$ and $\mathbb{D}$ respectively and a functor $F : \mathbb{C} \to \mathbb{D}$, we define, for each $\sigma \in \mathrm{Ob}_{\mathbb{C}}$, the *fiber selection functor* $\mathsf{Fib}_\sigma : \mathsf{IFunc}_F(\mathcal{P}, \mathcal{Q}) \to \mathsf{Func}(\mathcal{P}\sigma, \mathcal{Q}(F\sigma))$, such that

$$\mathsf{Fib}_\sigma(\tau) = \tau_\sigma \tag{4.7.5}$$

$$\mathsf{Fib}_\sigma(\eta) = \eta_\sigma \tag{4.7.6}$$

If $\Gamma : I \to \mathsf{IFunc}_F(\mathcal{P}, \mathcal{Q})$ is a diagram and $L = (\tau, \{\eta_i\}_{i \in \mathrm{Ob}_I})$ a cone, for each $\sigma \in \mathrm{Ob}_{\mathbb{C}}$ we obtain a diagram $\Gamma ; \mathsf{Fib}_\sigma$ in $\mathsf{Func}(\mathcal{P}\sigma, \mathcal{Q}(F\sigma))$ and a corresponding instance cone $L_\sigma = (\tau_\sigma, \{(\eta_i)_\sigma\}_{i \in \mathrm{Ob}_I})$.

Theorem 4.7.4 *Let $\mathcal{P}$ and $\mathcal{Q}$ be indexed categories over the base categories $\mathbb{C}$ and $\mathbb{D}$, $F : \mathbb{C} \to \mathbb{D}$ a functor and $\Gamma : I \to \mathsf{IFunc}(\mathcal{P}, \mathcal{Q})$ a diagram. Assume $\Gamma ; \mathsf{Fib}_\sigma ; \mathsf{Ev}_{\mathbf{G}}$ has canonical limit $L_{\sigma,\mathbf{G}}$ for each $\sigma \in \mathrm{Ob}_{\mathbb{C}}$ and $\mathbf{G} \in \mathrm{Ob}_{\mathcal{P}\sigma}$ that are preserved by $\mathcal{Q}(Ff)$ for each $f \in \mathbb{C}$. Then*

- *Γ has a canonical limit L with $L_{\sigma,\mathbf{G}}$ as instances;*
- *if L' is another limit of L, then L'_σ is a limit of $\Gamma ; \mathsf{Fib}_\sigma$ for all σ's.*

Proof. First of all, according to Property 4.7.1, $\Gamma ; \mathsf{Fib}_\sigma$ has a canonical limiting cone $L_\sigma = (\tau_\sigma, \{(\eta_i)_\sigma\}_{i \in \mathrm{Ob}_I})$ for all $\sigma \in \mathbb{C}$. Then we can define

$$L = (\tau, \{\eta_i\}_{i \in \mathrm{Ob}_I})$$

and prove that it is a limiting cone for Γ.

However, at first we need to prove that τ and η are well defined. This amounts to show that for each $f : \sigma \to \sigma'$ in $\mathbb{C}$ and $i \in \mathrm{Ob}_I$ it is

- $\mathcal{P}(f) ; \tau_\sigma = \tau_{\sigma'} ; \mathcal{Q}(Ff)$,
- $\mathcal{P}(f) \star (\eta_i)_\sigma = (\eta_i)_{\sigma'} \star \mathcal{Q}(Ff)$.

We know from Corollary 4.7.3 that $(\mathcal{P}(f) ; \tau_\sigma, \{\mathcal{P}(f) \star (\eta_i)_\sigma\}_{i \in \mathrm{Ob}_I})$ is the canonical limit of $\Gamma' = (\Gamma ; \mathsf{Fib}_\sigma)_{\mathcal{P}(f)}$. Note that

$$\Gamma'(i) = \mathcal{P}(f) ; \Gamma(i)(\sigma) = \Gamma(i)(\sigma') ; \mathcal{Q}(Ff)$$

since $\Gamma(i)$ is a functor of indexed categories. In the same way

$$\Gamma'(g) = \mathcal{P}(f) \star \Gamma(g)_\sigma = \Gamma(g)_{\sigma'} \star \mathcal{Q}(Ff)$$

for each $g : i \to j$ in I, hence $\Gamma' = (\Gamma ; \mathsf{Fib}_{\sigma'})^{\mathcal{Q}(Ff)}$. Since $\mathcal{Q}(Ff)$ preserves canonical limits, by the same corollary we have that another canonical limit of Γ' is $(\tau_{\sigma'} ; \mathcal{Q}(Ff), \{(\eta_i)_{\sigma'} \star \mathcal{Q}(Ff)\}_{i \in \mathrm{Ob}_I})$. Therefore the required equalities hold by uniqueness of canonical limits.

Now we want to prove that L is a limit. Assume we have another cone

$$L' = (\tau', \{\eta'_i\}_{i\in \mathrm{Ob}_I}) \tag{4.7.7}$$

For each $\sigma \in \mathrm{Ob}_{\mathbb{C}}$, since L_σ is a limiting cone for $\Gamma\,;\mathsf{Fib}_\sigma$, we have an unique arrow $\delta_\sigma : L'_\sigma \to L_\sigma$. We want to prove that the collection $(\delta_\sigma)_{\sigma\in\mathrm{Ob}_{\mathbb{C}}}$ is an indexed natural transformation. It is enough to prove that $\mathcal{P}(f) \star \delta_\sigma = \delta_{\sigma'} \star \mathcal{Q}(Ff)$ for all $f : \sigma \to \sigma'$ in $\mathbb{C}$.

If we consider the diagram $(\Gamma\,;\mathsf{Fib}_\sigma)_{\mathcal{P}f}$, $\mathcal{P}f \star \delta_\sigma$ is the unique arrow between cones $(\mathcal{P}f\,;\tau'_\sigma, \{\mathcal{P}f\star(\eta'_i)_\sigma\}_{i\in\mathrm{Ob}_I})$ and $(\mathcal{P}f\,;\tau_\sigma, \{\mathcal{P}f\star(\eta_i)_\sigma\}_{i\in\mathrm{Ob}_I})$, since the latter is a limiting cone by Corollary 4.7.3. However, we know that these cones are respectively equals to $(\tau'_{\sigma'}\,;\mathcal{Q}(Ff), \{(\eta'_i)_{\sigma'} \star \mathcal{Q}(Ff)\}_{i\in\mathrm{Ob}_I})$ and $(\tau_{\sigma'}\,;\mathcal{Q}(Ff), \{(\delta_i)_{\sigma'} \star \mathcal{Q}(Ff)\}_{i\in\mathrm{Ob}_I})$. Since $\delta_{\sigma'} \star \mathcal{Q}(Ff)$ is another arrow between them, then we have $\mathcal{P}(f) \star \delta_\sigma = \delta_{\sigma'} \star \mathcal{Q}(Ff)$.

Now the proof of the first point is concluded. The second point is trivial considering that two limiting cones must be isomorphic and that isomorphic cones have isomorphic instances. ∎

A trivial corollary is that, if we have a cone L such that $L_{\sigma,\mathbf{G}}$ is a limit for each $\sigma \in \mathrm{Ob}_{\mathbb{C}}$ and $\mathbf{G} \in \mathrm{Ob}_{\mathcal{P}\sigma}$, then L is a limit. In fact, since L is a cone, its instances are naturally preserved by reindexing functors. Therefore, we can apply the first point of previous theorem.

Note that, by duality, everything we said regarding limits can be restated for colimits.

Chapter 5

The categorical framework

Abstract

First of all, we briefly sketch some of the works in the categorical treatment of logic programs which are related to the thesis. Then we introduce the categorical framework we are going to use in this and the following chapters. The presentation we give here is quite simple and it is not able to cope with the most essential structure of logic languages which is the logical "and". In the following chapter, using premonoidal structures, we will extend the framework with the ability to treat conjunctions of goals.

In Part I, we have discussed a framework which can be used for the semantic analysis of those logic languages based on sequent calculi. Typically, these languages are defined as suitable fragments of well known logic systems, from where they inherit the proof theory. We have shown that it is possible to reconstruct common semantic concepts, such as correct answers, resultants, groundness properties, for these purely proof-theoretical languages. Moreover, we have shown that practical tasks, such as static analysis, can be performed in this context.

However, there is another class of logic languages, whose syntactic structure is essentially that of Horn clauses, but where formulas are not interpreted in a purely syntactical model. For example, in CLP , constraints are interpreted in a given *constraint system*. Although it is possible to rearrange the operational semantics of a CLP language so that it becomes an instance of our framework, the set of inference rules we obtain lacks any good property of symmetry, which is the strong point of sequent calculi.

It seems that the approach we have followed in the first part of the thesis is not well suited for such kind of logic languages. In this part of the thesis, we will present another approach to the semantics of logic programs which works well for these model-theoretic based languages. In the conclusion of the thesis, we will discuss about possible interactions between these two approaches.

5.1 Logic Programming with Categories

Categorical approaches to logic programming appeared with the categorical treatment of unification given by Rydeheard and Burstall in [63]. Building on this work, in [8] the syntax of Horn clause logic is formalized using categorical tools, and a topos-theoretic semantics is given. In [22], following some basic ideas already developed in [23], a categorical analysis of logic program transitions and models is given using indexed monoidal category. The framework that we propose in the following, in particular, has many similarities with the work in this paper.

All these different approaches are focused on the operational or model theoretic side of the matter. On the contrary, they lack any bottom-up denotational semantics like the T_P operator of van Emden and Kowalski [68]. However, the immediate consequence operator seems to be an important cornerstone of logic programming, since it appears, in one form or another, across several semantic treatments of logic programs [9]. As we have discussed in the first part of the thesis, most of the studies in the semantics of logic programming are heavily based on the existence of some fixpoint construction. For these reasons, it seems to us that further investigations of a categorical framework which includes a form of bottom-up semantics is advisable.

5.1.1 An Approach with a Bottom-Up Semantics

A first work in this directions has been [30]. In this paper a categorical syntax over finite τ-categories [32] is defined, as discussed in Section 4.3. It is the starting point for introducing both a notion of categorical SLD derivation and a denotational semantics that resembles the correct answer semantics for Horn logic programs. This semantics can be computed with a fixpoint construction and it is proved to agree with categorical derivations. One of the advantages of this framework is that the category of terms needs not to be the Lawvere algebraic category [44] for a given set of function symbols. This roughly means that it is possible for terms like $r(\vec{x})$ and $s(\vec{y})$ to have unifiers, although they have different root function symbols. In this way, several kinds of constraints on terms can be enforced. On the contrary, predicates remain completely free as in classical Horn logic programming.

We give here a brief account of the principal ideas we find in [30], without using τ-categories. For this reason, several definitions are only meaningful up to iso.

Given a finite product category $\mathbb{C}$ of terms, we know that monic arrows can be seen as predicates. Assume we want to write a logic program using a set of predicates $X_1, \dots X_n$ of sorts $\sigma_1, \dots, \sigma_n$. The idea pursued in the paper is having a syntactic category where the meaning of these predicates is completely arbitrary and an *interpretation* (i.e. a functor) that maps the syntactic category to a semantic category, in such a way that it is compatible with the clauses that compose our program. When we speak of completely arbitrary meaning of X_i in the syntactic category, we intend that, for each term $t : \rho \to \sigma_i$, the pullback of X_i along t is a proper subobject of ρ. Otherwise, this would mean that $X_i(t)$ is true, independently from the clauses that compose our program.

In general, if we identify X_i with a subobject of σ_i, we cannot be sure that the above property is satisfied. Hence, we need to *freely adjoin* a subobject of σ_i for every X_i in such a way that all its instances exist and are proper subobjects of the right sort. We obtain a new category, that we call $\mathbb{C}[X_1, \dots, X_n]$.

Then, a notion of categorical derivation is defined. A goal is a sequence of atomic goals, and a clause is a pair made of a goal $\mathbf{Tl}$ (tail) and an atomic goal (head) $X_i(t)$, written as $X_i(t) \leftarrow \mathbf{Tl}$. An SLD step is a step in a labeled transition system $\rightsquigarrow$. If $\mathbf{G}_1$ and $\mathbf{G}_2$ are goals, ϕ is a substitution and c a clause $X_k(u) \leftarrow \mathbf{Tl}$, then

$$\mathbf{G}_1 \overset{\phi,c}{\rightsquigarrow} \mathbf{G}_2 \tag{5.1.1}$$

when

- $\mathbf{G}_1 = X_{j_1}(t_1), \dots, X_k(t), \dots, X_{j_m}(t_m)$ for some arrow $t : A_1 \to \sigma_k$,
- there is an arrow θ such that the following diagram commute:

$$\begin{tikzcd} A_2 \arrow[r, "\phi"] \arrow[d, "\theta"'] & {} \arrow[d, "u"] \\ A_1 \arrow[r, "t"'] & \sigma_k \end{tikzcd} \tag{5.1.2}$$

  Here the pair (θ, ϕ) is an unifier of t and u. We have a couple of arrows instead of a single one because we are unifying arrows with different sources (that corresponds to renamed apart terms).

- $\mathbf{G}_2 = \theta^\sharp X_{j_1}(t_1), \dots, \phi^\sharp \mathbf{Tl}, \dots, \theta^\sharp X_{j_m}(t_m)$.

An SLD derivation is a derivation in the transition system $\rightsquigarrow$. An SLD refutation is a derivation that ends with an empty goal. Given a derivation $d = \mathbf{G}_1 \overset{\theta_1,c_1}{\rightsquigarrow} \cdots \overset{\theta_n,c_n}{\rightsquigarrow} \mathbf{G}_n$, the *computed answer* of d is defined as the composition $\theta_n ; \cdots ; \theta_1$.

An interpretation is a finite product preserving functor

$$\llbracket \_ \rrbracket : \mathbb{C}[X_1, \dots, X_n] \to \mathit{Set}^{\mathbb{C}^{\mathrm{o}}} \tag{5.1.3}$$

that extends the Yoneda embedding (i.e. such that $\llbracket \sigma \rrbracket = \mathit{Hom}(\_, \sigma)$ for each $\sigma \in \mathrm{Ob}_{\mathbb{C}}$) and assigns a subobject of $\mathit{Hom}(\_, \sigma_i)$ to X_i. It is possible to prove that, given an assignment of subobjects to the X_i's, there is only one interpretation that extends this assignment. Moreover, interpretations form a complete lattice.

Now, let us consider the operator on interpretations E_P, parametric with respect to the program P

$$\mathsf{E}_P(\llbracket \_ \rrbracket)(X_i) = \bigcup_{X_i(t) \leftarrow \mathbf{Tl} \in P} \Im_{\llbracket t \rrbracket}(\llbracket \mathbf{Tl} \rrbracket) \tag{5.1.4}$$

where $\Im_f(X)$ is the *image* of the monic X along the arrow f. It results that E_P is continuous, hence it has a least fixpoint, the interpretation $[\![\_]\!]^*$. The fundamental relation between SLD derivations and the fixpoint semantics is that $[\![\mathbf{G}]\!]^*$ is true (i.e., it is isomorphic to $Hom(\_, \sigma)$ for the right σ) if and only if there exists an SLD refutation of $\mathbf{G}$ with identity computed answer. More in general, $[\![\mathbf{G}]\!]^*$ corresponds to the set of *correct* answers for $\mathbf{G}$.

This methodology has been proved strong enough to be easily adapted to weak hereditary Harrop formulas [31] and abstract data types [46, 50].

5.2 The New Approach

Our framework takes origin from the ideas just discussed. However, we redefine the fundamental categorical structures with the hope to generalize the methods in three main directions:

- in the ability to treat other kinds of semantics other than the "correct answer" one. The concept of interpretation must be far broader than that of [30], allowing for semantic domains different from $Set^{\mathbb{C}^o}$;

- in the ability to treat programs with constraints between goals. This means that we must provide a new E_P operator that works with whatever syntactic category $\mathbb{C}$, not only with $\mathbb{C}[X_1, \ldots, X_n]$. In this way, goals need not to be freely generated objects;

- in the adaptability to different logic languages. In particular, we would like to treat languages such as CLP or add some control operators to the pure logic programming.

To pursue this goals, we move to a more general categorical interpretation of logic. Instead of taking goals to be monics in the category $\mathbb{C}$, we use an indexed category over $\mathbb{C}$. An object on the fiber $\sigma \in \mathrm{Ob}_{\mathbb{C}}$ will be the categorical counterpart of a goal of sort σ. It is the standard categorical interpretation of full first order logic, as pursued in [64].

5.2.1 Syntax

In the following we introduce several kind of indexed categories and we call them doctrines [43]. This is actually an abuse of notations, since a doctrine is generally understood to be an indexed category where reindexing functors have left adjoints, and this property does not always holds for our doctrines. However, we have chosen this terminology to emphasize the relation between indexed categories used for the syntax and the semantics (which are actually doctrines).

Definition 5.2.1 (Logic programming doctrine) *An* LP doctrine *(logic programming doctrine) is an indexed category $\mathcal{P}$ over a base category $\mathbb{C}$. For each $\sigma \in \mathrm{Ob}_{\mathbb{C}}$, objects and arrows in $\mathcal{P}\sigma$ are called* formulas *and* proofs *(of sort σ) respectively. We use the term* goal *as a synonymous for* formula*. Given a goal $\mathbf{G}$ of sort σ and $f : \rho \to \sigma$ in $\mathbb{C}$, $f^\sharp\mathbf{G} = \mathbf{G}(f)$ is an* instance *of $\mathbf{G}$.*

We write $\mathbf{G} : \sigma$ and $f : \sigma$ as a short form for $\mathbf{G} \in \mathrm{Ob}_{\mathbb{P}\sigma}$ and $f \in \mathrm{Mor}_{\mathbb{P}\sigma}$. Given an LP doctrine $\mathcal{P}$, a *clause* (of sort σ) is an object cl, with an associated pair $(\mathbf{Tl}, \mathbf{Hd})$ of goals of sort σ, that we write as $\mathbf{Hd} \overset{cl}{\leftarrow} \mathbf{Tl}$.

Definition 5.2.2 (Logic program) *A* logic program *is a pair $(P, \mathcal{P})$ where $\mathcal{P}$ is an LP doctrine and P a set of clauses. We often say that P is a program over $\mathcal{P}$.*

It is possible to see a logic program as an indexed category P over $\mathrm{Ob}_{\mathbb{C}}$ such that $P(\sigma)$ is the category of objects of sort σ and arrows $cl : \mathbf{Hd} \to \mathbf{Tl}$ of clauses of sort σ.

The idea underlying the framework is that the base category represents the world of all possible states to which program execution can lead. At each state, the corresponding fiber represents a set of deductions that can be performed. These deductions are always the same, whatever is the actual program. A clause, on the contrary, is a new deduction that we want to consider in addition to the proofs in the fibers.

What we mean with state here is a very broad concept: it can be the value of some global storage or the current tuple of free variables (as in the standard hyperdoctrinal semantics for logic). We now give a look at some examples of this latter kind.

Example 5.2.3 (Binary logic programs)

Assume given a first order signature Σ composed of a set F of function symbols and a set P of predicate symbols, with respective arities. We build the category $\mathbb{S}_\Sigma$ as the algebraic category freely generated by F. Objects of $\mathbb{S}_\Sigma$ are natural numbers (with zero), while arrows from n to m are m-tuples of terms built from the set of variables $\{v_1, \ldots, v_n\}$. In formulas

$$\mathrm{Ob}_{\mathbb{S}_\Sigma} = \mathbb{N} \tag{5.2.1}$$

$$\mathbb{S}_\Sigma(n, m) = T_\Sigma(\{v_1, \ldots, v_n\})^m \tag{5.2.2}$$

Then, we build a syntactic category $\mathcal{P}_\Sigma$ over $\mathbb{S}$ given by

- for each $n \in \mathbb{N}$, $\mathcal{P}_\Sigma(n)$ is the discrete category of atomic goals built from variables $v_1, \ldots, v_n$;
- for $\mathbf{t} = \langle t_1, \ldots, t_m\rangle : n \to m$, $\mathcal{P}_\Sigma(\mathbf{t})$ is the functor mapping an atomic goal A to $A[v_1/t_1, \ldots, v_m/t_m]$.

Therefore, a clause in $\mathcal{P}_\Sigma$ is a pair of atomic goals. This is essentially the definition of a binary clause in the standard logic programming terminology. As a result, we have a syntactic doctrine which exactly mimics the syntax of binary logic programs, if we exclude the fact that we do not have a direct translation for axioms, i.e. clauses with empty tails. It is quite straightforward to extend the previous definitions to work with a many-sorted signature Σ.

The advantage of using categories is that we do not need to restrict our interest to syntactical terms and goals. Actually, we can choose the base category we desire and build binary logic programs where terms are interpreted at the syntactic level.

Example 5.2.4 (General binary logic programs)

Assume $\mathbb{C}$ is a finite product category. We can think at $\mathbb{C}$ as a non-free model of an appropriate many-sorted signature. We can build a syntactic doctrine for binary logic programs where terms are elements of this category. We need to fix a signature Π for predicates over $\mathbb{C}$, i.e. a set of predicate symbols with associated sort among $\mathrm{Ob}_\mathbb{C}$. We write $p : \sigma$ when p is a predicate symbol of sort σ. Then, we define an indexed category $\mathcal{P}_\Pi$ over $\mathbb{C}$ such that

- $\mathcal{P}_\Pi(\sigma)$ is the discrete category whose objects are pairs $\langle p, f\rangle$ such that $p : \rho$ in Π and $f : \sigma \to \rho$ is an arrow in $\mathbb{C}$. To ease notation, we write $p(f)$ instead of $\langle p, f\rangle$;
- $\mathcal{P}_\Pi(f)$ where $f : \rho \to \sigma$ is the functor mapping $p(t) \in \mathrm{Ob}_{\mathcal{P}_\Pi(\sigma)}$ to $p(f\, ; t)$.

Note that, if $\mathbb{C}$ is the free algebraic category for the signature Σ and Π is the restriction of Σ to predicate symbols, then $\mathcal{P}_\Pi$ and $\mathcal{P}_\Sigma$ are isomorphic.

The interesting point here is that terms are treated semantically. For example, assume $\mathbb{C}$ is the full subcategory of *Set* whose objects are the sets $\mathbb{N}^i$ for every natural i and $p : \mathbb{N}$ is a predicate symbol. If succ and fact are the functions for the successor and factorial of a natural number, then $(\mathsf{succ}\, ; \mathsf{succ}\, ; \mathsf{succ})^\sharp(p(3)) = \mathsf{fact}^\sharp(p(3)) = p(6)$.

In the previous examples, the fibers of the syntactic doctrine were always discrete categories freely generated by a set of predicate symbols. When we will define the concept of model for a program, it will be manifest that this means we are not imposing any constraint on the meaning of predicates. In general, we can use more complex categories for fibers.

Example 5.2.5 (Symmetric closure of predicates)

In the hypotheses of Example 5.2.4, assume we have two predicate symbols p and *refp* of sort $\rho \times \rho$, and we want to encode in the syntactic doctrine the property that *refp* is the symmetric closure of p. Then, we freely adjoin to $\mathcal{P}_\Pi$ the following two arrows in the fiber $\rho \times \rho$:

$$r_1 : p \to \mathit{refp} \ ,$$
$$r_2 : p \to \mathit{refp}(\langle \pi_2, \pi_1 \rangle) \ .$$

We call $\mathcal{P}_\sigma^{\mathit{refp}}$ the new indexed doctrine we obtain. The intuitive meaning of the adjoined arrows is evident. We will see in the following sections how they formally affect the semantics of a program in $\mathcal{P}_\sigma^{\mathit{refp}}$.

5.2.2 Models

A main effort of this treatment is to consider extensions to definite logic programs without loosing the declarative point of view. Therefore, first of all we are going to define a straightforward notion of model for a program.

Functors of LP doctrines will be named *interpretations*. If $[\![\_]\!] = (F, \tau)$ is an interpretation from $\mathcal{P}$ to $\mathcal{Q}$, we write $F(x)$ as $[\![x]\!]$ for every object or arrow x in $\mathbb{C}$. Moreover, for every goal or proof x in the fiber σ, we write $\tau_\sigma(x)$ as $[\![x]\!]_\sigma$. We also use $[\![x]\!]$ when the fiber of x is clear from the context.

Definition 5.2.6 (Models) *Given a program P over the doctrine $\mathcal{P}$, a* model *of P is a pair $([\![\_]\!], \iota)$ where $[\![\_]\!] : \mathcal{P} \to \mathcal{Q}$ is an interpretation and ι is a function that maps a clause $\mathbf{Hd} \overset{cl}{\leftarrow} \mathbf{Tl} \in P$ to an arrow $[\![\mathbf{Hd}]\!] \xleftarrow{\iota(cl)} [\![\mathbf{Tl}]\!]$.*

If we consider a program as an indexed category, then ι is a functor from P to $U(\mathcal{Q})$, where $U : \mathsf{ICat} \to \mathsf{ICat}$ is a functor which maps and indexed category over $\mathbb{C}$ to an indexed category over $\mathrm{Ob}_\mathbb{C}$ by forgetting the arrows in the base category. Formally, if $\mathcal{Q} : \mathbb{C} \to \mathsf{Cat}$, we have $U(\mathcal{Q}) : \mathrm{Ob}_\mathbb{C} \to \mathsf{Cat}$ such that $U(\mathcal{Q})(\sigma) = \mathcal{Q}(\sigma)$.

In the following, a model $M = ([\![\_]\!], \iota)$ will be used as an alias for its constituent parts. Hence, $M(cl)$ will be the same of $\iota(cl)$ and $M_\sigma(\mathbf{G})$ the same of $[\![\mathbf{G}]\!]_\sigma$. Moreover, we define the composition of M with an interpretation N as the model $([\![\_]\!] \, ; N, \iota \, ; N)$.

A model $M : \mathcal{P} \to \mathcal{Q}$ for a program P can be seen as a sort of goal-independent semantics for P. Given a goal $\mathbf{G}$ of sort σ, the corresponding semantics, in the Heyting style, is the class of arrows targeted at $M_\sigma(\mathbf{G})$ in $\mathcal{Q}$.

Example 5.2.7 (Ground answers for binary logic programs)
Consider the LP doctrine $\mathcal{P}_\Sigma$ defined in Example 5.2.3, a program P and the indexed category $\mathcal{Q}$ over $\mathbb{S}_\Sigma$ such that

- $\mathcal{Q}(n) = \wp(T_\Sigma(\emptyset)^n)$, which is an ordered set viewed as a category;
- $\mathcal{Q}(\mathbf{t})(X) = \{\langle t'_1, \ldots, t'_n \rangle \mid \mathbf{t}[v_1/t'_1, \ldots, v_n/t'_n] \in X\}$. In other words, the reindexing functor yields the factorizations of all the tuples of term in X through $\mathbf{t}$.

An interpretation $[\![\_]\!]$ maps an atomic goal in $\mathcal{P}_\sigma(n)$ (i.e. an atomic goal with n free variables) to a set of n-uples of ground terms. In particular, if we define $[\![A]\!]_\sigma$ as the set of partial correct ground answer for A in the program P, the it is trivially possible to extend $[\![\_]\!]$ to a model by mapping a clause $A_1 \overset{cl}{\leftarrow} A_2$ to the inclusion map $[\![A_1]\!] \subseteq [\![A_2]\!]$.

As Example 5.2.3 has been generalized in Example 5.2.4, it is possible to generalize the interpretations presented in the previous example to work with a generic syntactic doctrine as $\mathcal{P}_\Pi$.

Example 5.2.8 (Ground answers for generalized binary logic programs)
Consider the LP doctrine $\mathcal{P}_\Pi$ defined in Example 5.2.4 and the indexed category $\mathcal{Q}$ over $\mathbb{C}$ such that

- for each $\sigma \in \mathrm{Ob}_\mathbb{C}$, $\mathcal{Q}(\sigma) = \wp(\mathit{Hom}(1, \mathbb{C}))$, which is an ordered set viewed as a category;
- for each $f \in \mathit{Hom}_\mathbb{C}(\sigma, \rho)$, $\mathcal{Q}(f)(X) = \{r \in \mathit{Hom}(1, \sigma) \mid r \,;\, f \in X\}$.

An interpretation $[\![\_]\!]$ maps an atomic goal of sort σ to a set of arrows from the terminal object of $\mathbb{C}$ to σ. These arrows are indeed the categorical counterpart of ground terms.

Two interesting models are given by the interpretations which map every goal $\mathbf{G}$ of sort σ to $\mathit{Hom}(1, \sigma)$ or to $\emptyset$. Clauses and arrows are obviously mapped to identities. If we see $\mathit{Hom}(1, \sigma)$ as the true value and $\emptyset$ as false, they correspond to the interpretations where everything is true or everything is false. Later, we will introduce syntactic doctrine with truth, which have a distinguished "true" goal which has to be mapped to a true semantic value.

By the way, there are a lot of other models, probably more interesting. Once we will define the concept of most general answer for a logic program, the map from a goal to the corresponding set of ground most general answers will turn out to be a model.

When the syntactic doctrine is discrete, as in the previous two examples, an interpretation from $\mathcal{P}$ to $\mathcal{Q}$ can map every object in $\mathcal{P}$ to every object in $\mathcal{Q}$, provided this mapping is well-behaved w.r.t. reindexing. However, in the general case, other restrictions are imposed.

Example 5.2.9
Assume we are in the hypotheses of Example 5.2.5. Consider the LP doctrine $\mathcal{Q}$ as defined in Example 5.2.8. An interpretation $[\![\_]\!]$ from $\mathcal{P}$ to $\mathcal{Q}$ is forced to map the arrows r_1 and r_2 to arrows in $\mathcal{Q}$. This means that $[\![\mathit{refp}]\!] \supseteq [\![p]\!]$ and $[\![\mathit{refp}]\!] \supseteq [\![p(\langle \pi_2, \pi_1 \rangle)]\!]$, i.e. $[\![\mathit{refp}]\!] \supseteq [\![p]\!] \,;\, \langle \pi_2, \pi_1 \rangle$. In other words, the interpretation of *refp* must contain both the interpretation of p and its symmetric.

One of the way to obtain a model of a program P in $\mathcal{P}$ is *freely adjoining* the clauses of P to the fibers of $\mathcal{P}$. We obtain the *free model* of P.

Definition 5.2.10 (Free model) *Given a program P over $\mathcal{P}$ we call a* free model *of P, if it exists, a model $M : \mathcal{P} \to \mathcal{Q}$ such that for every other model M' of P there exists an unique interpretation N such that $M' = M \,; N$.*

It is easy to prove by universality properties that, if M and M' are both free models for a program P in two different logic doctrines $\mathcal{Q}$ and $\mathcal{R}$, then $\mathcal{Q}$ and $\mathcal{R}$ are isomorphic.

5.2.3 Operational Semantics

The model theoretic point of view we have just exposed is not the only possible for our logic programs: they also have a quite straightforward operational meaning. Given a goal $\mathbf{G}$ of sort σ in a program $(P, \mathcal{P})$, we want to reduce $\mathbf{G}$ using both arrows in the fibers of $\mathcal{P}$ and clauses. This means that, if $x : \mathbf{G} \leftarrow \mathbf{Tl}$ is a clause or an arrow in $\mathcal{P}$, we want to perform a reduction step from $\mathbf{G}$ to $\mathbf{Tl}$.

In this way, the only rewritings we can immediately apply to $\mathbf{G}$ are given by rules (arrows or clauses) of sort σ. It is possible to rewrite using a clause *cl* of another sort ρ only if we find a common "ancestor" α of σ and ρ, i.e. a span

$$\begin{tikzcd}
& \sigma \\
\alpha \arrow[ur, "t"] \arrow[dr, "s"'] & \\
& \rho
\end{tikzcd} \qquad (5.2.3)$$

such that $\mathbf{G}$ and the head of *cl* become equals once they are reindexed to the fiber α.

Definition 5.2.11 (Unifier) *Given two goals $\mathbf{G}_1 : \sigma_1$ and $\mathbf{G}_2 : \sigma_2$ in an LP doctrine $\mathcal{P}$, an* unifier *for them is a span $\langle t_1, t_2 \rangle$ of arrows of the base category such that $t_1 : \alpha \to \sigma_1$, $t_2 : \alpha \to \sigma_2$ and ${t_1}^\sharp \mathbf{G}_1 = {t_2}^\sharp \mathbf{G}_2$*

Unifiers for a pair of goals form a category $\mathit{Unif}_{\mathbf{G}_1, \mathbf{G}_2}$ where arrows from $\langle t_1, t_2 \rangle$ to $\langle r_1, r_2 \rangle$ are given by the common notion of arrow between spans, i.e. a morphism $f : \mathsf{dom}(t_1) \to \mathsf{dom}(r_1)$ such that $f; r_1 = t_1$ and $f; r_2 = t_2$.

Definition 5.2.12 (MGU) *An mgu (*most general unifier*) for goals $\mathbf{G}_1 : \sigma_1$ and $\mathbf{G}_2 : \sigma_2$ in an LP doctrine $\mathcal{P}$ is a maximal element of $\mathit{Unif}_{\mathbf{G}_1, \mathbf{G}_2}$.*

Example 5.2.13 (Standard mgu)

Consider the indexed category $\mathcal{P}_\Sigma$ as in the Example 5.2.3. Given goals $p_1(t_1) : \sigma_1$ and $p_2(t_2) : \sigma_2$, an unifier is a pair of arrow $r_1 : \alpha \to \sigma_1$ and $r_2 : \alpha \to \sigma_2$ such that the following diagram commute:

$$\begin{tikzcd} \alpha \arrow[r, "r_1"] \arrow[d, "r_2"'] & \sigma_1 \arrow[d, "t_1"] \\ \sigma_2 \arrow[r, "t_2"'] & \sigma_2 \end{tikzcd} \tag{5.2.4}$$

This is exactly the definition of unifier for renamed apart terms t_1 and t_2 given in [8], which corresponds to unifiers in the standard syntactic sense. Moreover, the span $\langle r_1, r_2\rangle$ is maximal when (5.2.4) is a pullback diagram, i.e. a most general unifier.

Note that there is no unifier between goals $p_1(t_1)$ and $p_2(t_2)$ if $p_1 \neq p_2$. However, this does not hold in general. Actually, in the very general case, we do not have a notion of predicate symbol at all.

In the same way, it is possible to reduce a goal $\mathbf{G} : \sigma$ with an arrow $f : \mathbf{Hd} \leftarrow \mathbf{Tl}$ in the fiber ρ, iff there exists a arrow $r : \rho \to \sigma$ such that $r^\sharp \mathbf{G} = \mathbf{Hd}$. We call a pair $\langle r, f\rangle$ with such properties a *reduction pair*. Reduction pairs form a category such that $t \in \mathrm{Mor}_\mathbb{C}$ is an arrow from $\langle r_1, f_1\rangle$ to $\langle r_2, f_2\rangle$ if $r_1 = t; r_2$ and $t^\sharp f_2 = f_1$. A *most general* reduction pair is a maximal reduction pair.

Following these ideas, it is possible to define a categorical form of SLD derivation. Furthermore, we will see that derivations give us a way to constructively build the free model of a program.

Definition 5.2.14 (Categorical derivation) *Given an program $(P, \mathcal{P})$, we define a labeled transition system $(\biguplus_{\sigma \in \mathrm{Ob}_\mathbb{C}} \mathrm{Ob}_{\mathcal{P}\sigma}, \rightsquigarrow)$ with goals as objects, according to the following rules:*

backchain-clause) $\mathbf{G} \overset{\langle r,t,cl\rangle}{\rightsquigarrow} t^\sharp \mathbf{Tl}$ *if cl is a clause* $\mathbf{Hd} \overset{cl}{\leftarrow} \mathbf{Tl}$ *and $\langle r, t\rangle$ is an unifier for* $\mathbf{G}$ *and* $\mathbf{Hd}$ *(i.e.* $r^\sharp \mathbf{G} = t^\sharp \mathbf{Hd}$*);*

backchain-arrow) $\mathbf{G} \overset{\langle r,f\rangle}{\rightsquigarrow} \mathbf{Tl}$ *iff* $\mathbf{G}$ *is a goals in the fiber σ,* $f : \mathbf{Hd} \leftarrow \mathbf{Tl}$ *is an arrow in the fiber ρ and $\langle r, f\rangle$ is a reduction pair for* $\mathbf{G}$.

A categorical derivation *is a (possibly empty) derivation in this transition system.*

If we restrict SLD-steps to the use of most general unifiers and most general reduction pairs, we have a new transition system $(\biguplus_{\sigma \in \mathrm{Ob}_\mathbb{C}} \mathrm{Ob}_{\mathcal{P}\sigma}, \rightsquigarrow_g)$ and a corresponding notion of *most general* (m.g.) categorical derivation. In the following, when not otherwise stated, everything we say about categorical derivation can be applied to m.g. ones.

If there are goals $\mathbf{G}_0, \ldots, \mathbf{G}_i$ and labels $l_0, \ldots, l_{i-1}$ with $i \geq 0$ such that

$$\mathbf{G}_0 \overset{l_0}{\rightsquigarrow} \mathbf{G}_1 \cdots \mathbf{G}_{i-1} \xrightarrow{l_{i-1}} \mathbf{G}_i \tag{5.2.5}$$

we write $\mathbf{G}_0 \overset{d}{\rightsquigarrow}^* \mathbf{G}_i$ where $d = l_0 \cdots l_{i-1}$ is the string obtained concatenating all the labels. Note that $d \neq \lambda$ uniquely induces the corresponding sequence of goals. We will write $\epsilon_{\mathbf{G}}$ for the empty derivation starting from goal $\mathbf{G}$.

Given a derivation d, we call *answer* of d (and we write $\mathsf{answer}(d)$) the arrow in $\mathbb{C}$ defined by induction on the length of d as follows

$$\begin{aligned} &\mathsf{answer}(\epsilon_{\mathbf{G}}) = id_\sigma && \text{if } \mathbf{G} : \sigma \\ &\mathsf{answer}(\langle r, f \rangle \cdot d) = \mathsf{answer}(d) \,;\, r \\ &\mathsf{answer}(\langle r, t, a \rangle \cdot d) = \mathsf{answer}(d) \,;\, r \end{aligned}$$

In particular, we call *most general* answers the answers corresponding to m.g. derivations.

Example 5.2.15 (Standard SLD derivations)

Consider a program P in the syntactic doctrine $\mathcal{P}_\Sigma$ and a goal $p(t_1)$ of arity n. Given a clause $p(t_2) \overset{cl}{\leftarrow} q(t)$, and an mgu $\langle r_1, r_2 \rangle$ for $p(t_1)$ and $p(t_2)$, we have a most general derivation step

$$p(t_1) \xrightarrow{\langle r_1, r_2, cl \rangle}_g q(r_2; t) \ . \tag{5.2.6}$$

This strictly correspond to a step of the standard SLD derivation procedure for binary clauses and atomic goals.

However, in the categorical derivation, it is possible to reduce w.r.t. one of the identity arrows of the fibers. Therefore, if $p(t)$ is a goal of arity n,

$$p(t) \xrightarrow{\langle id_n, id_{p(t)} \rangle}_g p(t) \tag{5.2.7}$$

is an identity step which does not have a counterpart in the standard resolution procedure. However, these steps have an identity answer. Therefore, fixed a goal $p(t)$, the set

$$\mathsf{answer}\{d \mid \exists p_2(t_2).\ d : p(t) \rightsquigarrow_g^* p_2(t_2)\} \tag{5.2.8}$$

is the set of partial computed answer for the goal $p(t)$.

If we consider a derivation in $\rightsquigarrow$ instead of $\rightsquigarrow_g$, it is the case that, at every step, we can perform an arbitrary instantiation of the goal we are resolving. It is like having an SLD resolution where we can use a generic unifier instead of the mgu. We have that

$$\mathsf{answer}\{d \mid \exists p_2(t_2).\ d : p(t) \rightsquigarrow^* p_2(t_2)\} \tag{5.2.9}$$

is the set of partial correct answer for the goal $p(t)$.

We can use categorical derivations to build several interesting models for logic programs. In particular, with the answer function we can build models which are the general counterpart of partial computed answers, correct answer and ground answers.

Example 5.2.16 (Model for ground answers)

Consider a program P in $\mathcal{P}_\Pi$ and an interpretation $[\![\_]\!]$ in the LP doctrine $\mathcal{Q}$ defined in Example 5.2.8, such that

$$[\![p(t)]\!] = \{\mathsf{answer}(d) \mid d : p(t) \rightsquigarrow^* \mathbf{G} \text{ is a m.g. ground categorical derivation}\} \ . \tag{5.2.10}$$

where a *ground categorical derivation* is a derivation whose last goal is in the fiber $\mathcal{P}(1)$. Now, for each clause $p_1(t_1) \overset{cl}{\leftarrow} p_2(t_2)$, if d is a m.g. ground derivation of $p_2(t_2)$, then

$$d' = p_1(t_1) \xrightarrow{\langle id,id,cl\rangle} p_2(t_2) \cdot d \tag{5.2.11}$$

is a m.g. ground derivation for $p_1(t_1)$ with $\mathsf{answer}(d') = \mathsf{answer}(d)$. Therefore, $[\![p_1(t_1)]\!] \supseteq [\![p_2(t_2)]\!]$ and this gives an obvious mapping ι from clauses to arrows in the fibers of $\mathcal{Q}$. It turns out that $([\![\_]\!], \iota)$ is a model for P.

We can give a categorical treatment of derivations. If $\mathcal{P}$ has base category $\mathbb{C}$ and P is a program over $\mathcal{P}$, we define the category $\mathcal{SLD}_P$ as follows:

- $\mathrm{Ob}_{\mathcal{SLD}_P} = \biguplus_{\sigma\in\mathrm{Ob}_\mathbb{C}} \mathrm{Ob}_{\mathcal{P}\sigma}$
- $\mathcal{SLD}_P(\mathbf{G}, \mathbf{G}')$ is the set of categorical derivations from $\mathbf{G}'$ to $\mathbf{G}$
- $id_\mathbf{G} = \epsilon_\mathbf{G}$
- $d\,;d' = d' \cdot d$.

We have chosen to reverse the direction of arrows w.r.t. derivations since, from a logical viewpoint, a derivation step $\mathbf{G} \rightsquigarrow \mathbf{G}'$ is an entailment of (some instance of) $\mathbf{G}$ from $\mathbf{G}'$. In this way, it possible to extend answer to a functor between $\mathcal{SLD}_P$ and $\mathbb{C}$ just defining $\mathsf{answer}(\mathbf{G}) = \sigma$ if $\mathbf{G} \in \mathrm{Ob}_{\mathcal{P}\sigma}$.

It turns out that the process of resolution can be seen as a formal counterpart of the process of building the free model of a program.

5.2.4 Free Models and Derivations

Assume given a program P over the LP doctrine $\mathcal{P} : \mathbb{C}^\mathrm{o} \to \mathbb{C}at$. In some sense $\mathcal{SLD}_P$ is a free construction, which we would like to turn in a free model of P.

Definition 5.2.17 (Flat derivations) *A derivation in $\mathcal{SLD}_P$ is called* flat *(on the fiber σ) when all the r fields in the labels of the two backchain rules are identities (on σ).*

If $d : \mathbf{G} \rightsquigarrow^* \mathbf{G}'$ is a flat derivation on the fiber σ and $k : \rho \to \sigma$ an arrow in $\mathbb{C}$, we define a new flat derivation $k^\sharp d : k^\sharp \mathbf{G} \rightsquigarrow^* k^\sharp \mathbf{G}'$ on the fiber ρ as follows:

$$\begin{aligned} &k^\sharp(\epsilon_G) = \epsilon_{k^\sharp \mathbf{G}} \\ &k^\sharp(\langle id_\sigma, f\rangle \cdot d) = \langle id_\rho, k^\sharp(f)\rangle \cdot k^\sharp(d) \\ &k^\sharp(\langle id_\sigma, t, cl\rangle) = \langle id_\rho, k\,;t, cl\rangle \cdot k^\sharp(d) \end{aligned}$$

We also define an operator "flatten" on derivations which, from $d : \mathbf{G} \rightsquigarrow^* \mathbf{G}'$ with computed answer θ, gives as a result a flat derivation $\mathsf{flatten}(d) : \theta^\sharp \mathbf{G} \rightsquigarrow^* \mathbf{G}'$:

$$\begin{aligned} &\mathsf{flatten}(\epsilon_{\mathbf{G}}) = id_{\mathbf{G}} \\ &\mathsf{flatten}(d \cdot \langle r, f\rangle) = r^\sharp(\mathsf{flatten}(d)) \cdot \langle id_\rho, f\rangle \quad \text{if } r : \rho \to \sigma \\ &\mathsf{flatten}(d \cdot \langle r, t, cl\rangle) = r^\sharp(\mathsf{flatten}(d)) \cdot \langle id_\rho, t, cl\rangle \quad \text{if } r : \rho \to \sigma. \end{aligned}$$

Definition 5.2.18 (Simple derivations) *A derivation is called* simple *when*

- *there are no two consecutive* backchain-arrow *steps,*
- *there are no* backchain-arrow *steps with identity arrows.*

We can define an operator simplify that, from a derivation $d : \mathbf{G} \rightsquigarrow^* \mathbf{G}'$, builds a simple derivation $\mathsf{simplify}(d) : \mathbf{G} \rightsquigarrow^* \mathbf{G}'$. The operator works in two steps:

- first of all, it repeatedly collapses all the subderivations of the kind $\langle t_1, f_1\rangle \cdot \langle t_2, f_2\rangle$ to $\langle t_2\,; t_1, f_2\,; {t_2}^\sharp(f_1)\rangle$;
- then, it removes every remaining backchain-arrow step with identity arrows.

Note that, if d is flat, $\mathsf{simplify}(d)$ is flat, too. We can use the operator simplify to define a new category for simple derivations.

Definition 5.2.19 (Category of simple derivations) *The category $s\mathcal{SLD}_P$ of simple derivations is defined as*

- $\mathrm{Ob}_{s\mathcal{SLD}_P} = \mathrm{Ob}_{\mathcal{SLD}_P}$*;*
- $\mathcal{SLD}_P(\mathbf{G}, \mathbf{G}')$ *is the set of simple derivations from* $\mathbf{G}'$ *to* $\mathbf{G}$*;*
- $id_{\mathbf{G}} = \epsilon_{\mathbf{G}}$*;*
- $d\,; d' = \mathsf{simplify}(d' \cdot d)$.

The proof that ; is associative is trivial. Moreover, since simplify *preserves flatness of derivations, it happens that simple and flat derivations form a subcategory of $s\mathcal{SLD}_P$.*

Using flat and simple derivations, we can define a new LP doctrine $\mathcal{Q} = \mathbb{C}^{\mathrm{o}} \to \mathbb{C}at$ such that

- $\mathcal{Q}\sigma$ is the subcategory of $s\mathcal{SLD}_P$ of all the simple flat derivations of sort σ;
- given $f : \sigma \to \rho$ in $\mathbb{C}$, we define $\mathcal{Q}f$ as follows
  - on objects, $\mathcal{Q}f(\mathbf{G}) = f^\sharp(\mathbf{G})$;
  - on arrows, $\mathcal{Q}f(d) = \mathsf{simplify}(f^\sharp d)$;

Now, we can define an interpretation $[\![\_]\!] = (id_{\mathbb{C}}, \tau)$ from $\mathcal{P}$ to $\mathcal{Q}$ and a choose function ι such that:

- $\tau_\sigma(\mathbf{G}) = \mathbf{G}$;
- $\tau_\sigma(f : \mathbf{G} \to \mathbf{G}') = \mathbf{G}' \xrightarrow{\langle id_\sigma, f\rangle}\!\!\!\!\rightsquigarrow \mathbf{G}$;
- $\iota(\mathbf{Hd} \overset{cl}{\leftarrow} \mathbf{Tl}) = \mathbf{Hd} \xrightarrow{\langle id_\sigma, id_\sigma, cl\rangle}\!\!\!\!\rightsquigarrow \mathbf{Tl}$

It results that $([\![\_]\!], \iota)$ is a model of P. We need to prove that it is a free model. Assume $M = ([\![\_]\!]', \iota')$ is another model of P in the doctrine $\mathcal{R}$. If $[\![\_]\!]' = (F, \tau')$, we try to define an interpretation $N = (H, \tau'')$ from $\mathcal{Q}$ to $\mathcal{R}$ such that $[\![\_]\!] ; N = [\![\_]\!]'$ and $\iota ; N = \iota'$. Since $[\![\_]\!] = (id_{\mathbb{C}}, \tau)$, it must be $H = F$. For the same reason $\tau''_\sigma(\mathbf{G}) = \tau'_\sigma(\mathbf{G})$ for each $\sigma \in \mathrm{Ob}_{\mathbb{C}}$ and $\mathbf{G} \in \mathrm{Ob}_{\mathcal{P}\sigma} = \mathrm{Ob}_{\mathcal{Q}\sigma}$.

It remains to define τ''_σ on arrows. It is obvious that τ'' must satisfy the following constraints:

- $\tau''_\sigma(\mathbf{G} \xrightarrow{\langle id_\sigma, f\rangle}\!\!\!\!\rightsquigarrow \mathbf{G}') = \tau'_\sigma(f)$
- $\tau''_\sigma(\mathbf{G} \xrightarrow{\langle id_\sigma, id_\sigma, cl\rangle}\!\!\!\!\rightsquigarrow \mathbf{G}') = \iota'(cl)$

The proof that this constrains uniquely define τ'' is trivial.

In the following we will write the LP doctrine $\mathcal{Q}$ just defined as $\mathcal{F}_P$ and the model $([\![\_]\!], \iota)$ as F_P. In conclusion, we have proved the following:

Theorem 5.2.20 *Given a program P in $\mathcal{P}$, F_P is a free model of P in $\mathcal{F}_P$.*

From this, the following soundness and completeness result can be easily obtained.

Corollary 5.2.21 (Completeness theorem) *Assume given a program P in $\mathcal{P}$, a free model $M : \mathcal{P} \to \mathcal{Q}$ and goals $\mathbf{G}$, $\mathbf{G}'$ of sort σ. If there is an arrow $f : M(\mathbf{G}) \to M(\mathbf{G}')$ in the fiber $M(\sigma)$ of $\mathcal{Q}$, then there is a simple flat derivation $\mathbf{G}' \overset{d}{\rightsquigarrow}{}^* \mathbf{G}$.*

Proof. Since M is a free model, we can write $F_P = M ; N$ for some interpretation $N : \mathcal{Q} \to \mathcal{F}_P$. Hence, $N(M(f))$ is a simple flat derivation d in $\mathcal{F}_P$ from $N(M(\mathbf{G})) = \mathbf{G}$ to $N(M(\mathbf{G}')) = \mathbf{G}'$. It follows that $d : \mathbf{G}' \rightsquigarrow^* \mathbf{G}$. ■

Corollary 5.2.22 (Soundness theorem) *Assume given a program P in $\mathcal{P}$, a goal $\mathbf{G}$ and a model $M = ([\![\_]\!], \iota) : \mathcal{P} \to \mathcal{Q}$. If d is a derivation of $\mathbf{G}$ to $\mathbf{G}'$ with computed answer θ, there exists an arrow $\theta^{\#} [\![\mathbf{G}]\!] \xleftarrow{p} [\![\mathbf{G}']\!]$ in $\mathcal{Q}$, where $p = \mathsf{arrow}(d)$ is defined by induction:*

$$\begin{aligned}
&\mathsf{arrow}(\epsilon_{\mathbf{G}}) = id_{\mathbf{G}}\\
&\mathsf{arrow}(d \cdot \langle r, f\rangle) = r^{\sharp}(\mathsf{arrow}(d))\ ;\ f\\
&\mathsf{arrow}(d \cdot \langle r, t, cl\rangle) = r^{\sharp}(\mathsf{arrow}(d))\ ;\ t^{\#}(\iota(cl))
\end{aligned}$$

Proof. By Theorem 5.2.20, there exists $N : \mathcal{F}_P \to \mathcal{Q}$ such that $M = F_P\ ;\ N$. Consider the simple flat derivation $d' = \mathsf{simplify}(\mathsf{flatten}(d)) : \theta^{\sharp}\mathbf{G} \rightsquigarrow^{*} \mathbf{G}'$. We know that d' is an arrow in $\mathcal{F}_P(\sigma)$, where σ is the sort of $\mathbf{G}'$. It is easy to show that $\mathsf{arrow}(d) = N_{\sigma}(d')$. ■

5.3 Fixpoint Semantics

Assume we have a program $(P, \mathcal{P})$. We have just defined the notions of SLD derivations. Now, as promised in Section 5.1.1, we start looking for a fixpoint semantic construction, similar in spirit to the immediate consequence operator T_P of van Emden and Kowalski. Starting from an interpretation $[\![\_]\!] : \mathcal{P} \to \mathcal{Q}$, our version of T_P gives as a result a new interpretation $[\![\_]\!]' : \mathcal{P} \to \mathcal{Q}$ which, in some way, can be extended to a model of P with more hopes of success than $[\![\_]\!]$. According to what stated in the preliminaries, we expect to obtain a T_P operator which essentially builds SLD derivations bottom-up.

Our long term objective is the ability to give fixpoint semantics to all of the possible programs in our framework. However, in this thesis we will restrict our attention to a subclass of programs that have particular freeness properties.

Definition 5.3.1 (Goal Free Logic Program) *A logic program $(\mathcal{P}, P)$ is called* goal free *when there is a set $\{X_1 : \sigma_1, \ldots, X_n : \sigma_n\}$ of sorted* generic goals *with the following properties:*

- *$\mathcal{P}$ is obtained from an LP doctrine $\bar{\mathcal{P}}$ by freely adjoining the generic goals to the right fibers of $\bar{\mathcal{P}}$;*
- *there are no clauses targeted at a goal in $\bar{\mathcal{P}}$.*

An instance of a generic goal is called *dynamic goal.* If $t : \rho \to \sigma_i$, we will write $t^{\sharp}X_i$ also as $X_i(t)$, to resemble notation used in section 4.3. We want to stress here that only the meaning of dynamic goals is modified step after step by the fixpoint construction, while all the goals in $\bar{\mathcal{P}}$ have a fixed meaning. Note that, given $[\![\_]\!] : \mathcal{P} \to \mathcal{Q}$, the interpretation of all the dynamic goals only depends from the interpretation of generic goals.

Example 5.3.2

If P is a program over the syntactic doctrine $\mathcal{P}_\Sigma$ or $\mathcal{P}_\Pi$ of Example 5.2.3 and Example 5.2.4, then it is goal free. Actually, we can define $\bar{\mathcal{P}} : \mathbb{C}^{\mathrm{o}} \to \mathsf{Cat}$ such that

- for each $\sigma \in \mathrm{Ob}_\mathbb{C}$, $\bar{\mathcal{P}}(\sigma) = 0$, i.e. the empty category, which is the initial element in Cat;
- for each $t \in \mathrm{Mor}_\mathbb{C}$, $\bar{\mathcal{P}}(t) = id_0$.

Then $\mathcal{P}_\Pi$ is obtained by $\bar{P}$ freely adjoining a goal $p(id_\sigma)$ for each $p : \sigma \in \Pi$. The same happens for $\mathcal{P}_\Sigma$.

However, if we consider the syntactic doctrine in Example 5.2.5, then a program P must not have any arrow targeted at p or *refp* if it wants to be goal free.

In order to define a fixpoint operator with reasonable properties, we require a more complex categorical structure in the target doctrine $\mathcal{Q}$ than in $\mathcal{P}$. We introduce the notion of

Definition 5.3.3 (Semantic doctrine) *A semantic LP doctrine $\mathcal{Q}$ is an LP doctrine where*

- *fibers have coproducts and canonical colimits of ω-chains;*
- *each reindexing functor $\mathcal{Q}t$ has a left-adjoint $\exists^\mathcal{Q}_t$ and preserves on the nose canonical colimits of ω-chains.*

We will drop the superscript $\mathcal{Q}$ from $\exists^\mathcal{Q}_t$ when it is clear from the context. If we only work with finite programs, it is enough for fibers to have only finite coproducts.

Example 5.3.4

Given a finite product category $\mathbb{C}$, consider the indexed category $\mathcal{Q}$ as defined in Example 5.2.8. It is possible to turn $\mathcal{Q}$ into a semantic LP doctrine. Actually:

- each fiber is a complete lattice, hence it has coproducts given by intersection and canonical colimits of ω-chains given by union;
- we can define $\exists^\mathcal{Q}_f$, with $f : \rho \to \sigma$ as the function which maps an $X \subseteq Hom_\mathbb{C}(1, \rho)$ to
$$\{t\,;f \mid t \in X\} \tag{5.3.1}$$
which is a subset of $Hom_\mathbb{C}(1, \sigma)$. The function is monotone, hence it can be trivially turned into a functor.
- more accurately, $\exists^\mathcal{Q}_t$ is additive, i.e. it preserves on the nose all the colimits.

- finally, $\exists_t^{\mathcal{Q}}$ is the left adjoint of $\mathcal{Q}(t)$, since

$$(\mathcal{Q}(t) \,;\, \exists_t^{\mathcal{Q}})(X) = \{f \in X \mid f \text{ factors trough } t\} \subseteq X \ , \tag{5.3.2}$$

$$(\exists_t^{\mathcal{Q}} \,;\, \mathcal{Q}(t))(X) = \{f \mid \exists g \in X . f \,;\, t = g \,;\, t\} \supseteq X \ . \tag{5.3.3}$$

In the following, we will use this semantic doctrine to give a fixpoint semantics to programs in $\mathcal{P}_\Pi$.

Now, assume we have an interpretation $[\![\_]\!] = (F, \tau)$ from $\mathcal{P}$ to $\mathcal{Q}$, where $\mathcal{Q}$ is a semantic LP doctrine. We want to build step by step a modified interpretation which is also a model of P. With a single step we move from $[\![\_]\!]$ to $\mathbf{E}_P([\![\_]\!]) = (F, \tau')$ where

$$\tau'_{\sigma_i}(X_i) = [\![X_i]\!] \vee \bigvee_{X_i(t) \leftarrow \mathbf{Tl} \in P} \exists_{Ft} \, [\![\mathbf{Tl}]\!] \ , \tag{5.3.4}$$

$$\tau'_\sigma(X_i(t)) = t^\# \big(\tau'_{\sigma_i}(X_i)\big) \ , \tag{5.3.5}$$

while $\tau' = \tau$ restricted to $\bar{\mathcal{P}}$. We should define τ' on arrows but, since there are only identities between dynamic goals, the result is obvious.

It is clear from the definition that (F, τ') is an indexed functor. However, if we want $\mathbf{E}_P$ to be a functor on interpretation, we must define its behavior on indexed natural transformations. If δ is an arrow between interpretations, we have $\mathbf{E}_P(\delta) = \delta'$ where

$$\delta'_{\sigma_i, X_i} = \delta_{\sigma_i, X_i} \vee \bigvee_{X_i(t:\rho \to \sigma_i) \leftarrow \mathbf{Tl} \in P} \exists_{Ft} \delta_{\rho, \mathbf{Tl}} \ , \tag{5.3.6}$$

$$\delta'_{\sigma, X_i(t)} = t^\# \big(\delta'_{\sigma_i, X_i}\big) \ , \tag{5.3.7}$$

$$\delta'_{\sigma, \mathbf{G}} = \delta_{\sigma, \mathbf{G}} \text{ if } \mathbf{G} \in \mathrm{Ob}_{\bar{\mathcal{P}}} \ . \tag{5.3.8}$$

Since the only non-trivial arrows are in $\bar{\mathcal{P}}$ and δ_σ is a natural transformation for each σ, the same can be said of δ'_σ. Moreover, it is obvious from (5.3.7) that δ' satisfies the special condition in definition 4.6.3 regarding indexed natural transformations. It follows that $\mathbf{E}_P$ is well defined.

Theorem 5.3.5 *An interpretation $[\![\_]\!]$ for a program P can be extended to a model if and only if it can be extended to an algebra for $\mathbf{E}_P$.*

Proof. We will separately prove the two implications. Let us assume that $[\![\_]\!]$ can be extended to an algebra for $\mathbf{E}_P$, i.e. there exists an arrow $\eta : \mathbf{E}_P([\![\_]\!]) \to [\![\_]\!]$. Given a clause $X_i(t) \leftarrow \mathbf{Tl}$, there is an arrow

$$\begin{aligned} [\![\mathbf{Tl}]\!] \xrightarrow{unit} t^\# \exists_{Ft} \, [\![\mathbf{Tl}]\!] \xrightarrow{t^\# in} t^\# \Big([\![X_i]\!] \vee \bigvee_{X_i(r) \leftarrow \mathbf{G} \in P} \exists_{Ft} \, [\![\mathbf{G}]\!] \Big) = \\ = t^\# \, \mathbf{E}_P([\![\_]\!])(X_i) = \mathbf{E}_P([\![\_]\!])(X_i(t)) \xrightarrow{\eta_{X_i(t)}} [\![X_i(t)]\!] \ . \end{aligned} \tag{5.3.9}$$

On the contrary, assume that $([\![\_]\!], \iota)$ is a model of P. Fixed a generic goal X_i, for each clause $cl = X_i(t) \leftarrow \mathbf{Tl}$ we have an arrow

$$a_{cl} = \exists_{Ft} [\![\mathbf{Tl}]\!] \xrightarrow{\exists_{Ft}\iota(cl)} \exists_{Ft} [\![X_i(t)]\!] \xrightarrow{counit} [\![X_i]\!] \quad . \tag{5.3.10}$$

Using properties of coproducts, we can define

$$\eta_{\sigma_i,X_i} : \mathsf{E}_P([\![\_]\!])(X_i) \xrightarrow{[id,\{a_{cl}\}_{cl}]} [\![X_i]\!] \quad , \tag{5.3.11}$$

$$\eta_{\sigma_i,X_i(t)} : t^{\#}\eta_{\sigma_i,X_i} \quad , \tag{5.3.12}$$

$$\eta_{\sigma,\mathbf{G}} = id_{\mathbf{G}} \text{ if } \mathbf{G} \in \mathrm{Ob}_{\bar{\mathcal{P}}} \quad , \tag{5.3.13}$$

and this is an indexed natural transformation $\eta : \mathsf{E}_P([\![\_]\!]) \to [\![\_]\!]$. Hence $([\![\_]\!], \eta)$ is an algebra for E_P. ■

It is interesting to observe that there is a canonical transformation ν between $[\![\_]\!]$ and $\mathsf{E}_P([\![\_]\!])$ given by:

$$\nu_{\sigma_i,X_i} = [\![X_i]\!]_{\sigma_i} \xrightarrow{in} \mathsf{E}([\![\_]\!])_{\sigma_i}(X_i) \quad , \tag{5.3.14}$$

$$\nu_{\sigma,X_i(t)} = t^{\#}(\iota_{\nu_i,X_i}) \quad , \tag{5.3.15}$$

$$\nu_{\sigma,\mathbf{G}} = id_{\mathbf{G}} \text{ if } \mathbf{G} \in \mathrm{Ob}_{\bar{\mathcal{P}}} \quad , \tag{5.3.16}$$

where *in* is the appropriate injection. This means that $([\![\_]\!], \nu)$ is a coalgebra for E_P. Moreover, by theorem 4.7.4, ω-chains have canonical colimits.

Theorem 5.3.6 *The immediate consequence operator* E_P *preserves colimits of* ω*-chains in* $\mathsf{IFunc}_F(\mathcal{P}, \mathcal{Q})$.

Proof. Let Γ be a diagram in $\mathsf{IFunc}_F(\mathcal{P}, \mathcal{Q})$ with shape I. Assume $([\![\_]\!], \{\eta_i\}_{i\in \mathrm{Ob}_I})$ is the canonical colimit of Γ. We want to prove that $(\mathsf{E}_P([\![\_]\!]), \{\mathsf{E}_P(\eta_i)\}_{i\in\mathrm{Ob}_I})$ is a colimit of $\Gamma ; \mathsf{E}_P$.

According to theorem 4.7.4, it is enough to prove that, for each $\sigma \in \mathrm{Ob}_{\mathbb{C}}$ and $\mathbf{G} \in \mathrm{Ob}_{\mathcal{P}\sigma}$,

$$\big(\mathsf{E}_P([\![\_]\!])_\sigma(\mathbf{G}), \{(\mathsf{E}_P(\eta_i)_{\sigma,\mathbf{G}})\}_{i\in\mathrm{Ob}_I}\big) = \mathsf{colim}\,\Gamma ; \mathsf{E}_P ; \mathsf{Fib}_\sigma ; \mathsf{Ev}_{\mathbf{G}} \quad . \tag{5.3.17}$$

We know that

$$([\![\mathbf{G}]\!]_\sigma, \{(\eta_i)_{\sigma,\mathbf{G}}\}_{i\in\mathrm{Ob}_I}\}) = \mathsf{colim}\,\Gamma ; \mathsf{Fib}_\sigma ; \mathsf{Ev}_{\mathbf{G}} \quad . \tag{5.3.18}$$

If $\mathbf{G}$ is not a dynamic goal, left and right side of this equation are equals to left and right side of equation 5.3.17, which is trivially satisfied. If $\mathbf{G}$ is the generic goal X_k, since $\exists_{Ft}$ is a left adjoint (hence preserves colimits) we have

$$(\exists_{Ft} [\![\mathbf{Tl}]\!]_\sigma, \{\exists_{Ft}(\eta_i)_{\sigma,\mathbf{Tl}}\}_{i\in\mathrm{Ob}_I}) = \mathsf{colim}\,\Gamma ; \mathsf{Fib}_\sigma ; \mathsf{Ev}_{\mathbf{Tl}} ;\exists_{Ft} \quad , \tag{5.3.19}$$

for each clause $X_k(t) \leftarrow \mathbf{Tl}$ with $t : \sigma \to \sigma_k$. Thanks to theorem 4.2.3, we finally have

$$\begin{aligned}
\Big(\mathsf{E}_P(\llbracket\_\rrbracket)_{\sigma_k,X_k}, \{(\mathsf{E}_P(\eta_i)_{\sigma_k,X_k})\}_{i\in\mathrm{Ob}_I}\}\Big) &= \\
\Big(\llbracket X_k\rrbracket \vee \bigvee_{X_k(t)\leftarrow\mathbf{Tl}} \exists_{Ft}\,\llbracket\mathbf{Tl}\rrbracket_\sigma\,, \big\{(\eta_i)_{\sigma,\mathbf{Tl}} \vee \bigvee_{X_k(t)\leftarrow\mathbf{Tl}} (\exists_{Ft}(\eta_i)_{\sigma,\mathbf{Tl}})\big\}_{i\in\mathrm{Ob}_I}\Big) &= \\
\mathsf{colim}\Big(\Gamma\,;\mathsf{Fib}_{\sigma_k}\,;\mathsf{Ev}_{X_k} \vee \bigvee_{X_k(t)\leftarrow\mathbf{Tl}} (\Gamma\,;\mathsf{Fib}_\sigma\,;\mathsf{Ev}_{\mathbf{Tl}}\,;\exists_{Ft})\Big) &\ .
\end{aligned} \tag{5.3.20}$$

Since the last diagram is equal to $\Gamma\,;\mathsf{E}\,;\mathsf{Fib}_\sigma\,;\mathsf{Ev}_{X_k}$, the theorem holds for generic goals.

In case of dynamic goals of the kind $X_k(t)$, the proof directly descends from the fact that reindexing functors in $\mathcal{Q}$ preserve colimits of ω-chains. ■

We have now all the pieces to apply the results mentioned in Section 4.4. We obtain a fixpoint $(\mathsf{E}^\omega_P(\llbracket\_\rrbracket), \delta)$, hence a model of P, which is the least model that extends $(\llbracket\_\rrbracket\,, \iota)$.

Theorem 5.3.7 *Given a program $(P, \mathcal{P})$, a semantic LP doctrine $\mathcal{Q}$ and an interpretation $\llbracket\_\rrbracket : \mathcal{P} \to \mathcal{Q}$, then E_P has a least fixpoint greater then $\llbracket\_\rrbracket$. Such fixpoint is a model of P in $\mathcal{Q}$.*

To ease notation, we will write in the following $\llbracket\_\rrbracket^n$ for $\mathsf{E}^n_P(\llbracket\_\rrbracket)$. This is true also for $n = \omega$.

Example 5.3.8

If we write the definition of E_P in all the details for the syntactic doctrine in Example 5.3.4, we obtain

$$\mathsf{E}_P(\llbracket\_\rrbracket)_{\sigma_i}(X_i) = \llbracket X_i\rrbracket \cup \bigcup_{X_i(t)\leftarrow X_j(r)} \{f\,;t \mid f\,;r \in \llbracket X_j\rrbracket\}\ , \tag{5.3.21}$$

$$\mathsf{E}_P(\llbracket\_\rrbracket)_{\sigma_i}(X_i(t)) = \{f \mid f;t \in \llbracket\mathbf{Tl}\rrbracket\}\ . \tag{5.3.22}$$

If we work with $\mathbb{C}$ defined by the free algebraic category for a signature Σ, then $\mathsf{E}_P(\llbracket\_\rrbracket)$ becomes equivalent to the standard T_P semantics for logic programs.

Assume $\mathbb{C} = \mathit{Set}$. Moreover, assume we have two predicate symbols $p : \mathbb{N}$ and $\mathsf{true} : 1$ and two clauses $p(\mathsf{succ} \circ \mathsf{succ}) \overset{cl_1}{\leftarrow} p(id_\mathbb{N})$ and $p(0) \leftarrow \mathsf{true}$. Let $\llbracket\_\rrbracket$ be the unique interpretation which maps true to $\mathit{Hom}(1, 1)$ (isomorphic the set of natural numbers) and p to $\emptyset$. Then, we can compute successive steps of the E_P operator starting from $\llbracket\_\rrbracket$, obtaining

$$\begin{aligned}
\llbracket p\rrbracket^0 &= \emptyset \\
\llbracket p\rrbracket^1 &= \{0\} \\
\llbracket p\rrbracket^2 &= \{0, 2\} \\
&\ \vdots \\
\llbracket p\rrbracket^n &= \{0, 2, \dots, 2(n-1)\}
\end{aligned} \tag{5.3.23}$$

where we are denoting an arrow $f : 1 \to \mathbb{N}$ with $f(\cdot)$, i.e. f applied to the only element of its domain. If we take the colimit of this ω-chain, we have

$$[\![p]\!]^{\omega} = \{f : 1 \to \mathbb{N} \mid f(\cdot) \text{ is even } \} \tag{5.3.24}$$

which is what we would expect by the intuitive meaning of the program P.

5.3.1 Truth Completeness

Up to now, we have given a fixpoint construction that builds a model of a program given a starting interpretation for the syntactic doctrine. We can ask ourselves what is the relationship between this model and the operational semantics.

First of all, consider that, in general, we do not want a full completeness results. This means that, if $[\![\_]\!]$ is the fixpoint semantics and there is an arrow from $[\![\mathbf{G}]\!]$ to $[\![\mathbf{G}']\!]$, we do not expect to find an SLD derivation from $\mathbf{G}'$ to $\mathbf{G}$. For example, consider a program that independently defines two predicates `div2` and `div6` on natural numbers. In an appropriate fixpoint semantics, $[\![\texttt{div2}]\!]$ and $[\![\texttt{div6}]\!]$ are respectively the set of natural numbers divisible by two and six; arrows in the semantic category are just set inclusions. Hence, it is normal to have an arrow $[\![\texttt{div2}]\!] \to [\![\texttt{div6}]\!]$ without a corresponding SLD derivation.

However, assume we have a particular goal that represents the truth value "true". We think useful that, for each arrow from true to a certain goal $\mathbf{G}$, there is a corresponding SLD derivation of $\mathbf{G}$. We will call *truth completeness* this property.

Definition 5.3.9 (LP doctrine with truth) *An LP doctrine with truth is an LP doctrine $\mathcal{P}$ where each fiber $\mathcal{P}\sigma$ has a distinguished element $\top_\sigma$ named* true. *Moreover, truth values are preserved by reindexing functors.*

Comparing with classical logic programming, $\top$ plays the same role of the empty goal. We call *interpretations with truth* an interpretation between LP doctrines with truth such that true values are preserved.

Example 5.3.10

Indexed categories defined in Example 5.2.3, Example 5.2.4 and Example 5.2.5 can be trivially turned into LP doctrines with truth by freely adjoining a new object $\top$ to each fiber and defining the reindexing functors so that they preserve the new objects. We call $\mathcal{P}_\Sigma^\top$ and $\mathcal{P}_\Pi^\top$ the new LP doctrines we obtain by this procedure. A program in $\mathcal{P}_\Sigma^\top$ is a real binary logic program when there are no clauses targeted at $\top$ and when we consider clauses of the kind $p(t) \leftarrow \top$ as the equivalent for the unary clause `p(t)`.

Moreover, the semantic doctrine $\mathcal{Q}$, as defined in Example 5.3.4, is already a doctrine with truth if, for each fiber σ, we consider $Hom(1, \sigma)$ as the $\top_\sigma$. Note that reindexing functors preserve truth objects.

Note that, once we consider doctrines with truth, the trivial interpretation which maps every goal $\mathbf{G}$ in $\mathcal{P}_\Pi$ to $\emptyset$ in $\mathcal{Q}$ is not valid anymore. Actually, every interpretation with truth from $\mathcal{P}^\top_\Pi$ to $\mathcal{Q}$ has to map $\top_\sigma$ in $\mathcal{P}^\top_\Pi$ in $Hom(1,\sigma)$. However, we still have a trivial model mapping every goal $\mathbf{G}$ of sort σ to $Hom(1,\sigma)$.

Definition 5.3.11 (Truth completeness) *Given a program P in $\mathcal{P}$ and an interpretation with truth $[\![\_]\!] : \mathcal{P} \to \mathcal{Q}$, we say $[\![\_]\!]$ is* truth complete *when, for each arrow $[\![\top]\!]_\sigma \to [\![\mathbf{G}]\!]_\sigma$, there is an SLD derivation $\mathbf{G} \leadsto^* \top_\sigma$ with id as computed answer.*

Given a program P in an LP doctrine with truth $\mathcal{P}$, assume $[\![\_]\!]$ is an interpretation of $\mathcal{P}$ into a semantic LP doctrine $\mathcal{Q}$ which is also an LP doctrine with truth. We want to know if $\mathbf{E}^\omega_P([\![\_]\!])$ enjoys the truth completeness property.

Definition 5.3.12 (Goal-free program with truth) *A* goal-free program with truth *in a syntactic doctrine with truth $\mathcal{P}$ is a goal-free program such that $\top_\sigma \in \bar{\mathcal{P}}\sigma$.*

If P is a goal-free program with truth and we apply the $\mathbf{E}_P$ operator to an interpretation with truth $[\![\_]\!]$, we obtain another interpretation with truth. This is true for every $[\![\_]\!]^n$ for $n \leq \omega$. What we would like to prove is that $[\![\_]\!]^\omega$ is truth complete. First of all, it is obvious that, for this to be true, $[\![\_]\!]$ itself must be truth complete. Moreover, there are other conditions which must be satisfied:

1. **(coprimality)** for each arrow $t : \sigma \to \rho$ in $\mathbb{C}$ and each collection $\{O_j\}_{j\in J}$ where $O_j \in \mathrm{Ob}_{\mathcal{Q}(F\rho)}$, if $\exists_{Ft}\top_{F\sigma} \to \bigvee_{j\in J} O_j$ there exists $l \in J$ such that $\exists_{Ft}\top_{F\sigma} \to O_l$;
2. for each arrow $t : \sigma \to \rho$ and ω-chain $\Gamma : I \to \mathcal{Q}\sigma$ with colimit L, if $\exists_{Ft}\top_{F\sigma} \to L$, there exists $i \in \mathrm{Ob}_I$ such that $\exists_{Ft}\top_{F\sigma} \to \Gamma(i)$;
3. for each couple of arrows $t : \sigma \to \rho$, $r : \sigma' \to \rho$ and $O \in \mathrm{Ob}_{\mathcal{Q}(F\sigma')}$, if $\exists_{Ft}\top_{F\sigma} \to \exists_{Fr}O$, there exists $k : \sigma \to \sigma'$ in $\mathbb{C}$ such that $t = k\,;r$ and $\top_{F\sigma} \to k^\# O$.

When the truth values satisfy these conditions, we say that they are *well-behaved*. It is noteworthy that the first and third conditions, although they can appear rather obscure, have a precise meaning from the logical point of view. If we take $t = id$, they corresponds to the disjunctive and existential properties of intuitionistic logic.

Example 5.3.13

Consider the semantic doctrine with truth $\mathcal{Q}$ defined in Example 5.3.10. Truth values in $\mathcal{Q}$ are not well-behaved. Actually if $t : \rho \to \sigma$ is an arrow in $\mathbb{C}$, $\{O_j\}_{j\in J}$ is a finite collection of objects in $\mathcal{Q}\sigma$ and

$$\{f\,; t \mid f \in Hom(1,\rho)\} \subseteq \bigcup_{j\in J} O_j \tag{5.3.25}$$

it is not the case that

$$\{f\,; t \mid f \in Hom(1,\rho)\} \subseteq O_j \tag{5.3.26}$$

for some j. Think at a category $\mathbb{C}$ with just two arrows t_1 and t_2 from 1 to ρ. Then, for each $t : \rho \to \sigma$, if we define $O_1 = \{t_1 ; t\}$ and $O_2 = \{t_2 ; t\}$, equation (5.3.25) holds but (5.3.26) does not. We will see later an example of semantic doctrine with well behaved truth.

Lemma 5.3.14 *Given a truth complete interpretation $[\![\_]\!]$ of an LP doctrine with truth $\mathcal{P}$ in a semantic LP doctrine with well-behaved truth $\mathcal{Q}$, a goal-free program P over $\mathcal{P}$ and $n \in \mathbb{N}$, if $\top \to [\![\mathbf{G}]\!]^n$, there exists a categorical derivation $\mathbf{G} \rightsquigarrow^* \top$.*

Proof. The proof proceeds by induction on n. If $n = 1$ the property trivially follows by truth completeness of $[\![\_]\!]$. Now, assume we have proved the property for $n = m - 1$ and let it write a proof for $n = m$.

If $\mathbf{G} = X_i(t)$ is a dynamic goal, we have

$$\top \to [\![X_i(t)]\!]^m = t^{\#}\Big([\![X_i]\!]^{m-1} \vee \bigvee_{X_i(r)\leftarrow\mathbf{Tl}} \exists_{Fr}\, [\![\mathbf{Tl}]\!]^{m-1}\Big)$$

By coprimality, there are two possible cases: either there is an arrow $\top \to t^\sharp\, [\![X_i]\!]^{m-1}$ or there is a clause $X_i(r) \leftarrow \mathbf{Tl}$ such that $\exists_{Ft}\top \to \exists_{Fr}\, [\![\mathbf{Tl}]\!]^{m-1}$. In the former case, the lemma follows by inductive hypothesis, while in the latter, since truth values are well-behaved, there exists k such that $t = k ; r$ and

$$\top \to k^{\#}\, [\![\mathbf{Tl}]\!]^{m-1} = [\![k^\sharp\mathbf{Tl}]\!]^{m-1}$$

By inductive hypothesis, there is an SLD derivation $k^\sharp\mathbf{Tl} \overset{d}{\rightsquigarrow}^* \top$. Moreover, since $t = k ; r$, there is an SLD step $X_i(t) \xrightarrow{\langle id,k,X_i(r)\leftarrow\mathbf{Tl}\rangle} k^\sharp\mathbf{Tl}$. Putting together, we have a derivation $X_i(t) \rightsquigarrow^* \top$.

If $\mathbf{G}$ is not a dynamic goal, we have that $\top \to [\![\mathbf{G}]\!]^m$ iff $\top \to [\![\mathbf{G}]\!]$ since interpretation of $\mathbf{G}$ does not change. Since $[\![\_]\!]$ is truth complete, we have a derivation $[\![\mathbf{G}]\!] \rightsquigarrow \top$ of length one. ■

Theorem 5.3.15 *Given a truth complete interpretation $[\![\_]\!]$ of an LP doctrine with truth $\mathcal{P}$ in a semantic LP doctrine with well-behaved truth $\mathcal{Q}$ and a goal free program P over $\mathcal{P}$, if $\top \to [\![\mathbf{G}]\!]^\omega$ there exists a categorical derivation $\mathbf{G} \rightsquigarrow^* \top$ with id as computed answer.*

Proof. If $\top \to [\![\mathbf{G}]\!]^\omega$, since truth in $\mathcal{Q}$ is well-behaved, there exists $n \in \mathbb{N}$ such that $\top \to [\![\mathbf{G}]\!]^n$. Applying the previous lemma we have the result. ■

5.4 An Example: Yoneda Semantics

We have proved it is possible to define several kinds of fixpoint semantics for our logic programming framework, according to the chosen semantic doctrine and initial interpretation.

Now, we will focus our attention to one of these, that we name *Yoneda semantics*. We will show that this semantics has particular completeness properties w.r.t. declarative semantics.

Before going one, note that a subobject of $Hom(\_, \sigma)$ in $Set^{\mathbb{C}^{o}}$ can be thought of as a left-closed set of arrows targeted at σ. Moreover, it is possible to define a notion of canonical subobject, where $H \overset{m}{\to} F$ is a canonical subobject of F when m_σ is a set inclusion for each $\sigma \in \mathrm{Ob}_{\mathbb{C}}$.

Definition 5.4.1 (Yoneda doctrine) *Given a category $\mathbb{C}$, the* Yoneda doctrine *over $\mathbb{C}$ is the strict indexed category $\mathcal{Y}_{\mathbb{C}} : \mathbb{C}^{o} \to \mathsf{Cat}$, such that*

- *$\mathcal{Y}_{\mathbb{C}}(\sigma)$ is the complete lattice of canonical subobject of $Hom(\_, \sigma)$;*
- *$\mathcal{Y}_{\mathbb{C}}(t : \sigma \to \rho)$ takes the left-closed span $H \subseteq Hom(\_, \rho)$ to the set of arrows*

$$\{f \in \mathrm{Mor}_{\mathbb{C}} \mid f \,;\, t \in H\} \ . \tag{5.4.1}$$

It can be proved that $\mathcal{Y}$ is a semantic LP doctrine with truth. First of all, $\mathcal{Y}t$ has a well-known left adjoint $\exists_t$ such that, if $t : \rho \to \sigma$ and $H \subseteq Hom(\_, \rho)$

$$\exists_t H = \{m \,;\, t \mid m \in H\} \tag{5.4.2}$$

Since fibers are complete lattices, they have all the limits and colimits, given respectively by intersection and union of left-closed spans of arrows. Moreover, for each diagram, there is only one limit and one colimit. It is easy to prove that $\mathcal{Y}_{\mathbb{C}}(t)$ preserves colimits. Given $t : \rho \to \sigma$ and H_i for $i \in I$ canonical subobject of $Hom(\_, \sigma)$, we have

$$\begin{aligned} \mathcal{Y}_{\mathbb{C}}(t)\left(\bigcup_{i \in I} H_i\right) &= \{f \in \mathrm{Mor}_{\mathbb{C}} \mid f \,;\, t \in \bigcup_{i \in I} H_i\} \\ &= \{f \in \mathrm{Mor}_{\mathbb{C}} \mid \exists i \in I . f \,;\, t \in H_i\} \\ &= \bigcup_{i \in I} \{f \in \mathrm{Mor}_{\mathbb{C}} \mid f \,;\, t \in H_i\} \\ &= \bigcup_{i \in I} \mathcal{Y}_{\mathbb{C}}(t)(H_i) \ . \end{aligned} \tag{5.4.3}$$

Finally, we can take $Hom(\_, \sigma)$ as the true object $\top_\sigma$, since it is preserved by reindexing functors.

Given a goal-free logic program $(P, \mathcal{P})$ on the same base category $\mathbb{C}$, consider $[\![\_]\!] = (id_{\mathbb{C}}, \tau)$ such as

$$[\![\mathbf{G}]\!]_\sigma = \{t : \rho \to \sigma \mid \text{ there exists } t^\sharp \mathbf{G} \leftarrow \top_\rho in \mathcal{P}(\rho)\} \tag{5.4.4}$$

Note that, for a dynamic goal $X(t)$ of sort σ, this means $[\![X(t)]\!] = \emptyset$ (the empty span), initial element of $\mathcal{Y}\sigma$. This interpretation is truth complete. If $[\![\mathbf{G}]\!]_\sigma \leftarrow$

$Hom(\_,\sigma)$ is an arrow in $\mathcal{Y}_{\mathbb{C}}(\sigma)$, then $[\![\mathbf{G}]\!]_\sigma = Hom(\_,\sigma)$, i.e. $id_\sigma \in [\![\mathbf{G}]\!]_\sigma$ and therefore there exists an arrow $\mathbf{G} \leftarrow \top_\sigma$ in $\mathcal{P}(\sigma)$. As a result, we also have a derivation $\mathbf{G} \xrightarrow{\langle id_\sigma, \mathbf{G}\leftarrow\top_\sigma\rangle} \top_\sigma$.

We can build a fixpoint semantics $[\![\_]\!]^\omega$ using the result of Theorem 5.3.7. However, if we want $[\![\_]\!]^\omega$ to be truth complete, we need to prove that $\mathcal{Y}_{\mathbb{C}}$ has well-behaved truth.

Theorem 5.4.2 *The semantic doctrine $\mathcal{Y}_{\mathbb{C}}(\sigma)$ has well-behaved truth.*

Proof. We need to prove the three main properties. Let us start with coprimality. Given $f : \sigma \to \rho$, assume there is an arrow $\exists_f \top_\sigma \to \bigcup_{j\in J} O_j$, i.e.

$$\{t\,;f \mid \mathsf{cod}(t) = \sigma\} \subseteq \bigcup_{j\in J} O_j \quad . \tag{5.4.5}$$

In particular, $f \in \bigcup_{j\in J} O_j$, therefore there exists $j \in J$ such that $f \in O_j$. Since O_j is a left-closed span, we have the required property $\exists_f \top_\sigma \subseteq O_j$. The proof for the second condition on ω-chains is similar.

Now, assume given $t : \sigma \to \rho$, $r : \sigma' \to \rho$, and a canonical subobject O of $Hom(\_,\sigma')$ such that

$$\{f\,;t \mid \mathsf{cod}(f) = \sigma\} \subseteq \{f\,;r \mid f \in O\} \quad . \tag{5.4.6}$$

In particular, we have $t \in \{f\,;r \mid f \in O\}$. Therefore, there exists $k : \sigma \to \sigma' \in O$ such that $t = k\,;r$. We have

$$k^\sharp O = \{s \in \mathrm{Mor}_{\mathbb{C}} \mid s\,;k \in O\} \quad , \tag{5.4.7}$$

and, since $k \in O$ and O is left closed, then

$$k^\sharp O = Hom(\_,\sigma) = \top_\sigma \quad . \tag{5.4.8}$$

Therefore, $\mathcal{Y}_{\mathbb{C}}$ has well-behaved truth. ∎

We apply the results of Theorem 5.3.5 to obtain a model $([\![\_]\!]^\omega, \iota)$ which we call *Yoneda semantics* and we write as Y_P. We can prove the following:

Theorem 5.4.3 *Given a program P in $\mathcal{P}$, its Yoneda semantics Y_P and a goal $\mathbf{G}$ of sort σ, if $t : \rho \to \sigma$ is an arrow in $Y_P(\mathbf{G})$, there exists a categorical derivation $t^\sharp\mathbf{G} \rightsquigarrow^* \top_\rho$ with the identity answer.*

Proof. Since $t \in Y_P(\mathbf{G})$ and $Y_P(\mathbf{G})$ is left-closed, it is obvious that, for each s targeted at ρ, $s \circ t \in Y_P(\mathbf{G})$. As a result, $t^\# Y_P(\mathbf{G}) = Y_P(t^\sharp\mathbf{G}) = Hom(\_,\rho) = \top_\rho$. Applying theorem 5.3.15 we obtain the required derivation $t^\sharp\mathbf{G} \rightsquigarrow^* \top_\rho$. ∎

5.5 Conclusions

In this chapter, we have introduced a categorical framework to handle several extensions of logic programming, those based on the idea of relying on Horn clause, but interpreted in a context which is not necessarily the Herbrand universe. Typical examples of these languages are CLP [40] and logic programs with builtin or embedded data types [46].

With respect to the stated intentions in Section 5.2, we have not tackled the problem of programs with constraints on goals when it comes to fixpoint semantics. From this point of view, the only advantage offered by our framework is the ability to include pre-defined goal with a fixed and immutable interpretation. Goals whose semantics can be modified by clauses (i.e. dynamic goals) must be freely generated as in [30].

The main problem in extending the fixpoint construction to work without restrictions is to define the E_P operator in such a way that it is a functor, which maps natural transformations to natural transformations. We think it is a goal which is important to pursue, since it would give a bottom-up semantic construction for really a large class of logic languages.

Chapter 6

Monoidal Structures and Examples

Abstract

We introduce the use of premonoidal structures on the fibers of the syntactic and semantic doctrines. In this way, we are able to cope with languages which have a conjunction operator, such as pure logic programming. We also show some examples of logic languages which can be treated inside the framework.

6.1 Modeling Conjunctions

If we examine the framework we have developed till now, it is evident that clauses and arrows in the fibers have a poor structure. In pure logic programming we are accustomed with clauses that have a body made of sequences of atomic goals. This does not yet have a counterpart in our framework.

We could define an LP doctrine $\mathcal{P}_\Sigma$ like that in Example 5.2.3 but where objects are sequences of atomic goals. In this way, we define an implicit monoidal structure on the fibers given by the concatenation of goal. However, this is not enough, since all the remaining semantic constructs we have introduced so far do not take into account this structure. For example, models do not have to preserve monoidal structures, hence they are not and-compositional. Moreover, given a clause $p(t) \leftarrow \mathbf{G}$ and a goal $p(t), q(r)$, our categorical derivation cannot perform the obvious step $p(t) \leftarrow \mathbf{G}, q(r)$. The same problem is reflected on the fixpoint semantics. Therefore, we need to rephrase the definitions and results of the previous chapter to take into account added structures on the fibers.

6.1.1 Programming with Premonoidal Structures

We choose to use premonoidal structures on the fibers instead of monoidal ones. We will see that, operationally, premonoidal structures strictly corresponds to the standard SLD resolution without selection rule. If we choose the more powerful monoidal structure, we have a corresponding operational model given by parallel SLD resolution. We think that, since we are not interested in parallel computation, there is no interest in considering parallel SLD resolutions.

Definition 6.1.1 (Premonoidal indexed category) *A (strict)* premonoidal indexed category *is an indexed category in which each fiber $\mathcal{P}\sigma$ has a premonoidal structure $(\otimes_\sigma, \top_\sigma, \alpha_\sigma, \lambda_\sigma, \rho_\sigma)$ which is preserved on the nose by reindexing functors.*

Note that a premonoidal indexed category is an LP doctrine with truth if we take the monoidal identities as truth values. Given a premonoidal indexed category as a syntactic doctrine for programs, goals acquire a premonoidal structure. Hence, although clauses are still characterized only by a pair of goals, the new inner structure is enough to mimic classic logic programming.

Example 6.1.2 (Pure logic programming)

Consider the finite product category $\mathbb{S}_\Sigma$ defined in Example 5.2.3. We can build a premonoidal indexed category over $\mathbb{S}_\Sigma$, denoted by $\mathcal{P}^\otimes_\Sigma$ which is the free indexed premonoidal category generated by $\mathcal{P}_\Sigma$ of Example 5.2.3.

Spelling out the definition, we have

- for each $n \in \mathbb{N}$, $\mathcal{P}^\otimes_\Sigma(n)$ is the discrete category whose objects are finite sequences of atomic goals built from variables $v_1, \ldots, v_n$;
- for each $n \in \mathbb{N}$, the premonoidal structure in the fiber n is given by defining:
  - $\mathbf{G}_1 \otimes \mathbf{G}_2 = \mathbf{G}_1 \cdot \mathbf{G}_2$;
  - $\top_n = \lambda$, the empty sequence of goals;
  - λ_n, ρ_n and α_n as identities.

  Actually, this is a monoidal structure.
- for each $\mathbf{t} = \langle t_1, \ldots, t_m \rangle : n \to m$, $\mathcal{P}^\otimes_\Sigma(\mathbf{t})$ is the functor mapping a goal $\mathbf{G}$ to $\mathbf{G}[v_1/t_1, \ldots, t_m/v_m]$.

A clause in $\mathcal{P}^\otimes_\Sigma$ is a an object $\mathbf{G}_1 \overset{cl}{\leftarrow} \mathbf{G}_2$.

If we start from the category $\mathcal{P}_\Pi$ defined in Example 5.2.4, we obtain a premonoidal indexed category $\mathcal{P}^\otimes_\Pi$. Programs over $\mathcal{P}^\otimes_\Pi$ are essentially equivalent to the kind of logic programs considered in [30] over a category $\mathbb{C}[X_1, \ldots, X_n]$. We can write everything more formally in the following:

Example 6.1.3

Consider a finite product category $\mathbb{C}$ and a predicate signature Π over $\mathbb{C}$. We can build a premonoidal indexed category over $\mathbb{C}$, denoted by $\mathcal{P}_\Pi^\otimes$ which is the free premonoidal indexed category generated by $\mathcal{P}_\Pi$ as defined in Example 5.2.4.

Spelling out the definition, we have

- for each $\sigma \in \mathrm{Ob}_\mathbb{C}$, $\mathcal{P}_\Pi^\otimes(\sigma)$ is the discrete category whose objects are finite sequences of atomic goals $p(t)$ where if $p : \rho$ then $t : \sigma \to \rho$;
- for each $\sigma \in \mathrm{Ob}_\mathbb{C}$, the premonoidal structure in the fiber σ is given by defining
  - $\mathbf{G}_1 \otimes_\sigma \mathbf{G}_2 = \mathbf{G}_1 \cdot \mathbf{G}_2$;
  - $\top_\sigma = \lambda$, the empty sequence of goals;
  - $\lambda_\sigma, \rho_\sigma$ and α_σ as identities.

  Note that it is actually a monoidal structure since all the maps are central.
- for each $f : \rho \to \sigma$ in $\mathrm{Mor}_\mathbb{C}$, $\mathcal{P}_\Pi^\otimes(f)$ is the functor mapping a goal $\mathbf{G} = p_1(t_1), \ldots, p_n(t_n)$ to $p_1(f;t_1), \ldots, p_n(f;t_n)$.

Now, we want to rephrase everything we have seen in the previous chapter to work in the 2-category of premonoidal indexed categories, which we are going to define:

Definition 6.1.4 (Premonoidal indexed functors) *Given premonoidal indexed categories $\mathcal{P}$ over $\mathbb{C}$ and $\mathcal{Q}$ over $\mathbb{D}$, a* premonoidal indexed functor $[\![\_]\!] : \mathcal{P} \to \mathcal{Q}$ *is an indexed functor (F, τ) such that, for each $\sigma \in \mathrm{Ob}_\mathbb{C}$, $\tau_\sigma : \mathcal{P}\sigma \to \mathcal{Q}\sigma$ is a strict premonoidal functor.*

Definition 6.1.5 (Premonoidal indexed natural transformation) *Given two premonoidal indexed functors (F, τ) and (F', τ') from $\mathcal{P}$ to $\mathcal{Q}$, a* premonoidal indexed natural transformation *is an indexed natural transformation (ξ, δ) such that, for each σ in the base category of $\mathcal{P}$, δ_σ is a premonoidal natural transformation.*

We use the terms *premonoidal LP doctrines* and *premonoidal interpretations* as a synonymous for premonoidal indexed categories and premonoidal indexed functors.

Definition 6.1.6 (Premonoidal model) *Given a program P built over the premonoidal LP doctrine $\mathcal{P}$, a premonoidal model of P is a model $([\![\_]\!], \iota)$ where $[\![\_]\!]$ is a premonoidal interpretation.*

Example 6.1.7

Consider the indexed category $\mathcal{Q}$ over $\mathbb{C}$ defined in the Example 5.2.8. It is possible to provide each fiber $\mathcal{Q}\sigma$ with a premonoidal structure by defining:

- $X_1 \otimes_\sigma X_2 = X_1 \cap X_2$ for each $X_1, X_2 \subseteq \mathit{Hom}(1, \mathbb{C})$;
- if $X_1 \subseteq X'_1$ and $X_2 \subseteq X'_2$, then $X_1 \cap X_2 \subseteq X'_1 \cap X'_2$, hence $\otimes$ can be easily extended to arrows in $\mathcal{Q}\sigma$;
- $\top_\sigma$ is $\mathit{Hom}(1, \mathbb{C})$;
- $\alpha_\sigma, \lambda_\sigma$ and ρ_σ are identities.

Then, it is immediate to see that reindexing functors preserve the premonoidal structure (which is actually a monoidal one). Hence, $\mathcal{Q}$ can be turned into a premonoidal indexed category.

A trivial premonoidal model for a program P in $\mathcal{Q}$ is given by the interpretation $[\![\_]\!]$ such that

$$[\![\mathbf{G}]\!]_\sigma = \mathit{Hom}(1, \sigma) \tag{6.1.1}$$

for each goal $\mathbf{G}$ and by the mapping ι such that

$$\iota(cl) = id_{\mathit{Hom}(1,\sigma)} \tag{6.1.2}$$

for each clause cl of sort σ. It corresponds to the maximal Herbrand model for pure logic programming. We will see later other examples of models with more interesting logical properties.

If the syntactic doctrine $\mathcal{P}$ is premonoidal, we can have both premonoidal and non premonoidal models. The former corresponds to AND-compositional semantics of classic logic programming, while the second to non-compositional semantics. In the following we will focus on the compositional case.

6.1.2 Operational Semantics

It is possible to adapt our categorical resolution procedure to the case of premonoidal structures on the fibers. First of all, we need a new concept of unifier.

Definition 6.1.8 (Premonoidal unifiers) *Given a premonoidal LP doctrine $\mathcal{P}$ over $\mathbb{C}$ and goals $\mathbf{G} : \sigma$ and $\mathbf{G}' : \rho$, a* premonoidal unifier *of $\mathbf{G}'$ into $\mathbf{G}$ is a tuple $\langle r, t, \mathbf{G}_1, \mathbf{G}_2 \rangle$ such that there exists $\alpha \in \mathrm{Ob}_\mathbb{C}$ with $r : \alpha \to \sigma$, $t : \alpha \to \rho$, $\mathbf{G}_1 : \alpha$, $\mathbf{G}_2 : \alpha$ and*

$$\mathbf{G}_1 \otimes_\alpha t^\sharp \mathbf{G}' \otimes_\alpha \mathbf{G}_2 = r^\sharp \mathbf{G} \ . \tag{6.1.3}$$

Given unifiers $\langle r, t, \mathbf{G}_1, \mathbf{G}_2 \rangle$ and $\langle r', t', \mathbf{G}'_1, \mathbf{G}'_2 \rangle$, an arrow from the first to the second is an arrow f in $\mathbb{C}$ such that $\mathsf{dom}(f) = \mathsf{dom}(r)$, $\mathsf{cod}(f) = \mathsf{dom}(r')$, $f\,; r' = r$, $f\,; t' = t$, $f^\sharp \mathbf{G}'_1 = \mathbf{G}_1$ and $f^\sharp \mathbf{G}'_2 = \mathbf{G}_2$. Therefore, premonoidal unifiers of $\mathbf{G}$ and $\mathbf{G}'$ form a category. Maximal elements of this category are called *most general premonoidal unifiers* (mgpu).

Example 6.1.9

Given the category $\mathcal{P}_\Pi^\otimes$ in Example 6.1.3, consider a goal $\mathbf{G}' = p_1(f_1), \ldots, p_n(f_n)$ of sort σ and an atomic goal $\mathbf{G} = p_i(f)$ of sort ρ. If

$$\begin{tikzcd}
\alpha \arrow[r, "r"] \arrow[d, "t"'] & \sigma \arrow[d, "f_i"] \\
\rho \arrow[r, "f"'] & \cdot
\end{tikzcd} \tag{6.1.4}$$

is a pullback diagram, the premonoidal unifiers of $\mathbf{G}'$ into $\mathbf{G}$ have a maximal element given by $\langle r, t, r^\sharp(p_1(f_1), \ldots, p_{i-1}(f_{i-1})), r^\sharp(p_{i+1}(f_{i+1}), \ldots, p_n(f_n))\rangle$. Note that, if $p_i = p_j$ for some j and f and f_j have an mgu $\langle r', t'\rangle$, then the maximal element in not unique (up to iso), since $\langle r', t', r'^\sharp(p_1(f_1), \ldots, p_{j-1}(f_{j-1})), r^\sharp(p_{j+1}(f_{j+1}), \ldots, p_n(f_n))\rangle$ is maximal but it is not isomorphic to the previous one.

If we take the transition system for categorical derivations we already know and we just consider premonoidal unifiers instead of unifiers in the backchain-clause rule, we obtain a new transition system for premonoidal derivations.

Definition 6.1.10 (Premonoidal derivation) *Given a premonoidal LP doctrine $\mathcal{P}$, we define a labeled transition system $(\biguplus_{\sigma\in\mathrm{Ob}_\mathbb{C}} \mathrm{Ob}_{\mathcal{P}\sigma}, \rightsquigarrow)$ with goals as objects, according to the following rules:*

backchain-clause) $\mathbf{G} \overset{\langle r,t,\mathbf{G}_1,\mathbf{G}_2,cl\rangle}{\rightsquigarrow} \mathbf{G}_1 \otimes t^\sharp\mathbf{Tl} \otimes \mathbf{G}_2$ *if cl is a clause* $\mathbf{Hd} \leftarrow \mathbf{Tl}$ *and* $\langle r, t, \mathbf{G}_1, \mathbf{G}_2\rangle$ *is a premonoidal unifier of* $\mathbf{Hd}$ *into* $\mathbf{G}$*;*

backchain-arrow) $\mathbf{G} \overset{\langle r,f\rangle}{\rightsquigarrow} \mathbf{Tl}$ *if* $\langle r, f : \mathbf{Hd} \leftarrow \mathbf{Tl}\rangle$ *is a reduction pair for* $\mathbf{G}$*.*

A premonoidal derivation *is a (possibly empty) derivation in this transition system.*

Again, if we restrict the rule "backchain-clause" to the case where $\langle r, t\rangle$ forms an mgu of $\mathbf{Hd}$ into $\mathbf{G}$ and the rule "backchain-arrow" to the case where $\langle r, f\rangle$ is a most general reduction pair, we have a *most general* premonoidal categorical derivation. The *answer* for a derivation is defined in the obvious way, i.e.

$$\begin{aligned}
\mathsf{answer}(\epsilon_\mathbf{G}) &= id_\sigma \qquad \text{if } \mathbf{G} \in \mathrm{Ob}_{\mathcal{P}\sigma} \\
\mathsf{answer}(\langle r, f\rangle \cdot d) &= \mathsf{answer}(d) \mathbin{;} r \\
\mathsf{answer}(\langle r, t, \mathbf{G}_1, \mathbf{G}_2, cl\rangle \cdot d) &= \mathsf{answer}(d) \mathbin{;} r
\end{aligned} \tag{6.1.5}$$

Example 6.1.11 (Standard SLD derivation)

We have seen in the Example 5.2.15 that, if we work in the LP doctrine $\mathcal{P}_\Sigma$, the most general categorical derivations correspond to the standard notion of SLD resolution for atomic goals and binary clauses, but for the existence of identity steps of the kind $p(t) \rightsquigarrow p(t)$.

Now, let us consider $\mathcal{P}^{\otimes}_{\Sigma}$ and premonoidal derivations. Assume P is a program over $\mathcal{P}^{\otimes}_{\Sigma}$ with the property that, for each clause $\mathbf{G} \overset{cl}{\leftarrow} \mathbf{G}'$, $\mathbf{G}$ is an atomic goal $p(t)$ (hence, we are excluding both conjunction of atomic goals and the empty goal $\top$). It is evident that most general premonoidal categorical derivations in this setting correspond to standard SLD resolution. Moreover, if d is a premonoidal derivation, then $\mathsf{answer}(d)$ is a correct answer in the standard sense. If d is most general, then $\mathsf{answer}(d)$ is a computed answer.

If we consider more general clauses, things are different. A clause of the kind $p_1(t_1), p_2(t_2) \overset{cl}{\leftarrow} \mathbf{G}$ does not have a counterpart in the standard SLD resolution. Be careful that this is not equivalent to the pair of clauses $p_1(t_1) \overset{cl}{\leftarrow} \mathbf{G}$ and $p_2(t_2) \overset{cl}{\leftarrow} \mathbf{G}$. If we had chosen finite products as the structure for managing conjunction, then we would have projection arrows $\pi_i : p_1(t_1), p_2(t_2) \to p_i(t_i)$. Then, we could resolve a goal $p_1(t_1)$ with π_1 and then with cl to obtain $\mathbf{G}$. But, in a premonoidal structure, we do not have projection arrows. Then, we cannot reduce $p_1(t_1)$ with the clause cl.

Derivations can be used to build a free model of a program P in a premonoidal LP doctrine $\mathcal{P}$. We first need to define flat and simple derivations. A *flat* derivation is a derivation such that, for each step $\mathbf{G} \overset{\langle r,t,c,\mathbf{G}_1,\mathbf{G}_2\rangle}{\rightsquigarrow} \mathbf{G}'$ or $\mathbf{G} \overset{\langle r,f\rangle}{\rightsquigarrow} \mathbf{G}'$, it is $r = id_\sigma$ when $\mathbf{G}$ is in the fiber of σ. A derivation is *simple* when there are no two consecutive backchain-arrow steps and no backchain-arrow steps with an identity arrow f.

Given a flat $d : \mathbf{G} \rightsquigarrow^* \mathbf{G}'$ in the fiber σ and an arrow $k : \rho \to \sigma$ in $\mathbb{C}$, we define a new flat derivation $k^\sharp d : \mathbf{G}^\sharp \rightsquigarrow^* \mathbf{G}'^\sharp$ on the fiber ρ by replacing

- each step $\mathbf{G}_a \overset{\langle id_\sigma,f\rangle}{\rightsquigarrow} \mathbf{G}_b$ with $k^\sharp(\mathbf{G}_a) \overset{\langle id_\rho,k^\sharp(f)\rangle}{\rightsquigarrow} k^\sharp(\mathbf{G}_b)$;
- each step $\mathbf{G}_1 \overset{\langle id_\sigma,t,\mathbf{G}_1,\mathbf{G}_2,cl\rangle}{\rightsquigarrow} \mathbf{G}_2$ with $k^\sharp(\mathbf{G}_a) \overset{\langle id_\rho,k;t,k^\sharp(\mathbf{G}_1),k^\sharp(\mathbf{G}_2),cl\rangle}{\rightsquigarrow} k^\sharp(\mathbf{G}_b)$;

Moreover, if $d : \mathbf{G}_1 \rightsquigarrow \mathbf{G}_2$ is a derivation with $\mathbf{G}_1$ of sort σ, $\mathsf{answer}(d) = f$, then, for each $\mathbf{G}$ of sort σ, we define a new derivation $\mathbf{G} \otimes d : \mathbf{G} \otimes \mathbf{G}_1 \rightsquigarrow^* f^\sharp\mathbf{G} \otimes \mathbf{G}_2$ by induction on the structure of d:

$$\begin{aligned}
&\mathbf{G} \otimes \epsilon_{\mathbf{G}_1} = \epsilon_{\mathbf{G}\otimes\mathbf{G}_1} \\
&\mathbf{G} \otimes (\langle r, f\rangle \cdot d) = \langle r, r^\sharp\mathbf{G} \otimes f\rangle \cdot (r^\sharp\mathbf{G} \otimes d) \qquad (6.1.6)\\
&\mathbf{G} \otimes (\langle r, t, \mathbf{G}_a, \mathbf{G}_b, cl\rangle \cdot d) = \langle r, t, cl, r^\sharp\mathbf{G} \otimes \mathbf{G}_a, \mathbf{G}_b\rangle \cdot (r^\sharp\mathbf{G} \otimes d)
\end{aligned}$$

In the same way, it is possible to define a new derivation $d \otimes \mathbf{G} : \mathbf{G}_1 \otimes \mathbf{G} \rightsquigarrow^* \mathbf{G}_2 \otimes f^\sharp\mathbf{G}$ by induction on the structure of d.

If $\mathcal{P}$ is strict premonoidal, it is easy to build a free model from the operational semantics, following the same basic ideas of Section 5.2.4. In other words, we build a premonoidal indexed category $\mathcal{Q}$ such that:

- $\mathcal{Q}\sigma$ is the subcategory of simple premonoidal derivations of sort σ, with domain and codomain reversed;

- the premonoidal structure in $\mathcal{Q}\sigma$ is given by $\otimes$ as defined in (6.1.6) and the identity element $\top_\sigma$ of $\mathcal{P}\sigma$. The natural transformations α, λ and ρ are identities;
- for each $f : \sigma \to \rho$ in $\mathbb{C}$, the reindexing functor $\mathcal{Q}f$ maps a derivation d in $f^\sharp d$. It preserves premonoidal structures on the nose.

Then, we define a premonoidal interpretation $[\![\_]\!] = (id_{\mathbb{C}}, \tau)$ from $\mathcal{P}$ to $\mathcal{Q}$ and a choice function ι for clauses in P such that

- $[\![f : \mathbf{G} \to \mathbf{G}']\!]_\sigma = \mathbf{G}' \xrightarrow{\langle id_\sigma, f\rangle}\!\!\!\!\!\rightsquigarrow \mathbf{G}$;
- $\iota(\mathbf{Hd} \overset{cl}{\leftarrow} \mathbf{Tl}) = \mathbf{Hd} \xrightarrow{\langle id_\sigma, id_\sigma, \top_\sigma, \top_\sigma, cl\rangle}\!\!\!\!\!\rightsquigarrow \mathbf{Tl}$,

and it turns out that $([\![\_]\!], \iota)$ forms a premonoidal free model of $(P, \mathcal{P})$.

It is possible to adapt the same construction to the case where $\mathcal{P}$ is not strict premonoidal. In this case, for each σ, the family of arrows λ_A in $\mathcal{Q}\sigma$ is given by the family of derivations $A \xrightarrow{\langle id_\sigma, \lambda_A\rangle}\!\!\!\!\!\rightsquigarrow \top \otimes A$, where the λ_A in the reduction pairs is the natural iso in $\mathcal{P}\sigma$. However, since this family of derivations has to be natural in A, we need to consider an equivalence relation on arrows of $\mathcal{Q}\sigma$, such that, for example

$$A \otimes B \xrightarrow{\langle id_\sigma, \lambda_A \otimes B\rangle}\!\!\!\!\!\rightsquigarrow (A \otimes \top) \otimes B \xrightarrow{\langle r, t, A\otimes\top, \lambda, cl\rangle}\!\!\!\!\!\rightsquigarrow (A \otimes \top) \otimes t^\sharp \mathbf{Tl} \qquad (6.1.7)$$

is equivalent to

$$A \otimes B \xrightarrow{\langle r, t, A, \lambda, cl\rangle}\!\!\!\!\!\rightsquigarrow A \otimes t^\sharp \mathbf{Tl} \xrightarrow{\langle id_\sigma, \lambda_A \otimes t^\sharp \mathbf{Tl}\rangle}\!\!\!\!\!\rightsquigarrow (A \otimes \top) \otimes t^\sharp \mathbf{Tl} \ . \qquad (6.1.8)$$

Similar equivalences need to be considered for the natural transformations ρ and α.

Example 6.1.12

Consider the premonoidal LP doctrine $\mathcal{P}^{\otimes}_{\Pi}$ as defined in Example 6.1.3 and the semantic LP doctrine $\mathcal{Q}$ in Example 6.1.7. We can build a model $([\![\_]\!], \iota)$ for *ground answers* by defining, for every object $\mathbf{G}$:

$$[\![\mathbf{G}]\!]_\sigma = \{\mathsf{answer}(d) \mid d : \mathbf{G} \rightsquigarrow^* \mathbf{G}' \text{ with } \mathbf{G}' : 1\} \ . \qquad (6.1.9)$$

If $f : \mathbf{G}_1 \to \mathbf{G}_2$ is an arrow in the fiber σ and d is a derivation for $\mathbf{G}_1$ with $\mathsf{answer}(d) \in Hom(1, \sigma)$, then $\langle id_\sigma, f\rangle \cdot d$ is a derivation of $\mathbf{G}_2$ with $\mathsf{answer}(\langle id_\sigma, f\rangle \cdot d) = \mathsf{answer}(d) \in Hom(1, \sigma)$. Therefore, $[\![\mathbf{G}_2]\!]_\sigma \supseteq [\![\mathbf{G}_1]\!]_\sigma$ and we can extend $[\![\_]\!]$ to a indexed functor by defining

$$[\![f : \mathbf{G}_1 \to \mathbf{G}_2]\!]_\sigma : [\![\mathbf{G}_1]\!]_\sigma \subseteq [\![\mathbf{G}_2]\!]_\sigma \ . \qquad (6.1.10)$$

The same is true for a clause $\mathbf{Hd} \overset{cl}{\leftarrow} \mathbf{Tl}$, hence we can define

$$\iota(cl) = [\![\mathbf{Hd}]\!]_\sigma \subseteq [\![\mathbf{Tl}]\!]_\sigma \ , \qquad (6.1.11)$$

which gives the required model.

6.1.3 Fixpoint Semantics

Definition 6.1.13 (Goal-free programs) *A* premonoidal goal-free logic program *is a goal-free logic program where*

- *$\bar{\mathcal{P}}$ and $\mathcal{P}$ are premonoidal LP doctrines;*
- *clauses can only have atomic goals as heads.*

An *atomic goal* is an instance of a generic goal. A *dynamic goal* is an atomic goal or a premonoidal composition of dynamic goals.

Definition 6.1.14 (Premonoidal semantic LP doctrines) *A* premonoidal semantic LP doctrine *is a premonoidal LP doctrine such that*

- *it is a semantic LP doctrine,*
- *every morphism on the fibers is central,*
- *for each σ, object of the base category, the two premonoidal functors $\otimes_\sigma$ preserve canonical colimits of ω-chains.*

Actually, the second condition is equivalent to require that the fibers have a monoidal structure given by $f \otimes g = (f \otimes \mathbf{G}_2);(\mathbf{G}'_1 \otimes g) = (\mathbf{G}_1 \otimes g)\,;(f \otimes \mathbf{G}'_2)$ for each $f : \mathbf{G}_1 \to \mathbf{G}'_1$ and $g : \mathbf{G}_2 \to \mathbf{G}'_2$. It would be possible to give more detailed restrictions on what arrows are needed to be central, but it does not seem that the expressive power of the framework could really gain from this greater generality.

Applying the same procedure of Section 5.3 when $\mathcal{P}$ and $\mathcal{Q}$ are premonoidal LP doctrines and $[\![\_]\!]$ is a premonoidal interpretation, we would like to obtain a chain of premonoidal interpretations with a premonoidal colimit. Then, we want to extend the colimit to a premonoidal model.

According to this idea, we define a new $\mathbf{E}_P$ operator in almost the same way as done in Section 5.3. However, the following cases must be added for $\tau' = \mathbf{E}_P(\tau)$ and $\delta' = \mathbf{E}_P(\delta)$ when $\mathbf{G}_1$ and $\mathbf{G}_2$ are dynamic goals:

$$\tau'_\sigma(\mathbf{G}_1 \otimes_\sigma \mathbf{G}_2) = \tau'_\sigma(\mathbf{G}_1) \otimes_{F\sigma} \tau'_\sigma(\mathbf{G}_2) \tag{6.1.12}$$

$$\delta'_{\sigma,\mathbf{G}_1 \otimes_\sigma \mathbf{G}_2} = \delta'_{\sigma,\mathbf{G}_1} \otimes_\sigma \delta'_{\sigma,\mathbf{G}_2} \tag{6.1.13}$$

Since the only arrows between dynamic goals are trivial arrows (identities or the ρ, λ, α components of premonoidal structures), it is immediate how to define τ' on arrows such that it becomes a monoidal interpretation. An analogous case addition is needed for the canonical morphism of interpretations $\nu : [\![\_]\!] \to \mathbf{E}_P([\![\_]\!])$:

$$\nu_{\sigma,\mathbf{G}_1 \otimes_\sigma \mathbf{G}_2} = \nu_{\sigma,\mathbf{G}_1} \otimes_\sigma \nu_{\sigma,\mathbf{G}_2} \tag{6.1.14}$$

It happens that Theorem 5.3.5 can be rephrased changing the definition of η in its second part, adding the case

$$\eta_{\sigma,\mathbf{G}_1\otimes\mathbf{G}_2} = \eta_{\sigma,\mathbf{G}_1} \otimes \eta_{\sigma,\mathbf{G}_2}$$

when $\mathbf{G}_1$ and $\mathbf{G}_2$ are dynamic goals.

Also Theorem 5.3.6 can be rewritten almost identically, thanks to the fact the monoidal operators preserve colimits of ω-chains. The last thing that need to be proved is that, once built the chain

$$[\![\_]\!]^0 \xrightarrow{\nu} [\![\_]\!]^1 \xrightarrow{\mathsf{E}_P(\nu)} \cdots \xrightarrow{\mathsf{E}_P^n(\nu)} [\![\_]\!]^n \to \cdots$$

the colimit $([\![\_]\!]^\omega, \{\eta_i\}_{i\in\mathbb{N}})$ is a monoidal colimit. In particular, we need to prove that, for each $\sigma \in \mathrm{Ob}_{\mathbb{C}}$ and $\mathbf{G}_1, \mathbf{G}_2 \in \mathrm{Ob}_{\mathcal{P}\sigma}$:

$$[\![\mathbf{G}_1 \otimes \mathbf{G}_2]\!]^\omega = [\![\mathbf{G}_1]\!]^\omega \otimes [\![\mathbf{G}_2]\!]^\omega \tag{6.1.15}$$

$$(\eta_i)_{\sigma,\mathbf{G}_1\otimes\mathbf{G}_2} = (\eta_i)_{\sigma,\mathbf{G}_1} \otimes (\eta_i)_{\sigma,\mathbf{G}_2} \tag{6.1.16}$$

If Γ is the previous chain of interpretations, we know that $[\![\mathbf{G}_1 \otimes \mathbf{G}_2]\!]^\omega$ is the colimit of $\Gamma' = \Gamma \circ \mathsf{Fib}_\sigma \circ \mathsf{Ev}_{\mathbf{G}_1\otimes\mathbf{G}_2}$. This means that $\Gamma'(i) = \Gamma(i)_\sigma(\mathbf{G}_1 \otimes \mathbf{G}_2) = \Gamma(i)_\sigma(\mathbf{G}_1) \otimes \Gamma(i)_\sigma(\mathbf{G}_2)$ since each $\Gamma(i)$ is a monoidal interpretation. The thesis follows by the fact that $\otimes$ preserves canonical colimits of ω-chains.

Example 6.1.15

Consider the premonoidal LP doctrine $\mathcal{P}_\Pi^\otimes$ as defined in Example 6.1.3 and the premonoidal semantic doctrine $\mathcal{Q}$ in Example 6.1.7. Moreover, consider the premonoidal interpretation $[\![\_]\!]$ such that

$$[\![\mathbf{G}]\!]_\sigma = \begin{cases} \mathit{Hom}(1,\sigma) & \text{if } \mathbf{G} = \top_\sigma, \\ \emptyset & \text{otherwise.} \end{cases} \tag{6.1.17}$$

Note that, if $[\![\_]\!]$ has to be premonoidal, we are forced to map the monoidal unit $\top_\sigma$ in $\mathcal{P}_\Pi^\otimes$ to the monoidal unit $\mathcal{Q}$.

If we compute the fixpoint semantics of a program P starting from this $[\![\_]\!]$, we obtain the model of the Example 6.1.12.

6.1.4 Truth Completeness and Yoneda Semantics

In the previous chapter, we have introduced the Yoneda semantics, which maps a goal $\mathbf{G}$ to its corresponding set of correct answers. It is interesting to observe that $\mathcal{Y}$ can be easily turned into a premonoidal LP doctrine by defining, for each σ:

- $O_1 \otimes_\sigma O_2 = O_1 \cap O_2$, since the intersection of left-closed span is still a left closed span;

- $\top_\sigma = \mathit{Hom}(\_, \sigma)$, just as it was defined in Section 5.4;
- $\alpha_\sigma, \lambda_\sigma, \rho_\sigma$ are identities.

Therefore, we can use the fixpoint construction to build models of programs in a Yoneda doctrine.

However, we cannot directly apply the results on truth completeness of Theorem 5.3.15, since they work with a different $\mathbf{E}_P$ operator.

Definition 6.1.16 (Well-behaved premonoidal doctrine) *A premonoidal semantic doctrine $\mathcal{Q}$ is well-behaved when the monoidal identities $\top_\sigma$ are well-behaved and the existence of an arrow $\top_\sigma \to \mathbf{G}_1 \otimes_\sigma \mathbf{G}_2$ in the fiber σ implies the existence of $\top_\sigma \to \mathbf{G}_1$ and $\top_\sigma \to \mathbf{G}_2$.*

Note that, when the premonoidal structures are actually finite products, the extra condition we have for premonoidal doctrines are automatically satisfied thanks to the existence of the project arrows from $G_1 \otimes G_2$ to $\mathbf{G}_i$.

Theorem 6.1.17 *If P is a goal-free premonoidal program over $\mathcal{P}$, $\mathcal{Q}$ is a well-behaved premonoidal semantic doctrine, $[\![\_]\!] : \mathcal{P} \to \mathcal{Q}$ is a truth-complete premonoidal interpretation, then $\top \to [\![\mathbf{G}]\!]^\omega$ implies the existence of a premonoidal categorical derivation $\mathbf{G} \leadsto^* \top$ with identity answer.*

Proof. The proof proceed by induction on the premonoidal structure of $\mathbf{G}$. If $\mathbf{G}$ is not a dynamic goal, the property easily follows from the fact that $[\![\mathbf{G}]\!]^\omega = [\![\mathbf{G}]\!]$. If $\mathbf{G}$ is an atomic goal, the proof proceed as in Lemma 5.3.14. Otherwise, if $\mathbf{G} = \mathbf{G}_1 \otimes \mathbf{G}_2$ and $\top \to [\![\mathbf{G}]\!]$, since $\mathcal{Q}$ is well-behaved, it is $\top \to [\![\mathbf{G}_1]\!]$ and $\top \to [\![\mathbf{G}_2]\!]$. By induction hypothesis, there are two derivations $d_1 : \mathbf{G}_1 \leadsto^* \top$ and $d_2 : \mathbf{G}_2 \leadsto^* \top$ with identity answers. From these, we can obtain the required flat derivation

$$d = (d_1 \otimes \mathbf{G}_2) \cdot (\top \otimes d_2) \cdot \langle id, \lambda \rangle \tag{6.1.18}$$

from $\mathbf{G}_1 \otimes \mathbf{G}_2$ to $\top$. ■

It is self-evident that $\mathcal{Y}$ is well-behaved. Actually, in each fiber σ, if $\top_\sigma = \mathit{Hom}(\_, \sigma) \subseteq O_1 \cap O_2$, then $\top_\sigma \subseteq O_1$ and $\top_\sigma \subseteq O_2$. Then, we can prove:

Theorem 6.1.18 *Given a goal-free premonoidal program P in $\mathcal{P}$, its Yoneda semantics Y_P and a goal $\mathbf{G}$ of sort σ, if $t : \rho \to \sigma$ is an arrow in $Y_P(\mathbf{G})$, there exists a categorical flat derivation $t^\sharp\mathbf{G} \leadsto^* \top_\rho$.*

Proof. The proof proceed as for Theorem 5.3.15. ■

6.2 Examples

Now that we have examined the general abstract framework, it is the moment to give a look to some specific instances. We have already viewed pure logic programming over a generic base category, the Yoneda semantics and the semantics of ground correct answers. We present here other examples.

6.2.1 Abduction Semantics

We will show how, given an LP doctrine $\mathcal{P}$, it is possible to build an abductive semantics for categorical resolutions over $\mathcal{P}$. With abductive semantics we mean that, fixed a program P, we map each goal $\mathbf{G}$ to a set of pairs $\langle P', f\rangle$ such that, if we adjoin P' to the current program, f is a correct answer for $P \cup P'$.

We have seen that a *program*, in our framework, is a set of clauses, i.e. objects $\mathbf{Hd} \overset{cl}{\leftarrow} \mathbf{Tl}$ where $\mathbf{Hd}$ and $\mathbf{Tl}$ have the same sort. While in the previous sections we have considered the possibility of having several clauses with the same head and tail, we are not interested in this generality for the abduction semantics. Hence, in this section, clauses (of sort σ) will be just pairs $\mathbf{Hd} \leftarrow \mathbf{Tl}$ of goals (of sort σ). If $f : \rho \to \sigma$ is an arrow in the base category of $\mathcal{P}$ and $\mathbf{Hd} \leftarrow \mathbf{Tl}$ is an arrow of sort σ, we define by $f^\sharp(\mathbf{Hd} \leftarrow \mathbf{Tl})$ the clause $f^\sharp\mathbf{Hd} \leftarrow f^\sharp\mathbf{Tl}$.

We can build a partial order category of programs $\mathbb{P}_{\mathcal{P}}$ with the standard assumption that $P_1 \to P_2$ iff $P_1 \subseteq P_2$. We can also restrict ourselves to particular subcategories of $\mathbb{P}_{\mathcal{P}}$. For example, if $\mathcal{P} = P^{\otimes}_{\Pi}$, we can choose to only consider those clauses whose codomain is an atomic goal.

Then, we build another syntactic doctrine $\mathcal{R}_{\mathcal{P}}$ over the category of *contexts* $\mathbb{P}_{\mathcal{P}}{}^{\mathrm{o}} \times \mathbb{C}$. For each object $\langle P, \sigma\rangle$ in $\mathbb{P}_{\mathcal{P}}{}^{\mathrm{o}} \times \mathbb{C}$, we have a category $\mathcal{R}(P, \sigma)$ which is obtained by $\mathcal{P}\sigma$ by freely adjoining new arrows $f^\sharp(\mathbf{Hd}) \overset{f^\sharp(cl)}{\leftarrow} f^\sharp(\mathbf{Tl})$ if $cl = \mathbf{Hd} \leftarrow \mathbf{Tl} \in P$ of sort ρ and $f : \sigma \to \rho$

Assume $\langle \supseteq, f\rangle$ is an arrow from $\langle P_1, \sigma_1\rangle$ to $\langle P_2, \sigma_2\rangle$ in the base category. We define the reindexing functor $\langle \supseteq, f\rangle^\sharp$ as

$$\begin{aligned}
&\langle \supseteq, f\rangle^\sharp(\mathbf{G}) = f^\sharp(\mathbf{G}) \ , \\
&\langle \supseteq, f\rangle^\sharp(t : \mathbf{G} \to \mathbf{G}') = f^\sharp(t) : f^\sharp\mathbf{G} \to f^\sharp\mathbf{G}' \ , \\
&\langle \supseteq, f\rangle^\sharp(cl : \mathbf{G} \to \mathbf{G}') = f^\sharp(cl) : f^\sharp\mathbf{G} \to f^\sharp\mathbf{G}' \ ,
\end{aligned} \tag{6.2.1}$$

where t is an arrow in $\mathcal{P}(\sigma_2)$ and cl a clause of sort σ_2. The reindexing for all the other arrows follows by composition. We are not considering any premonoidal structure on the fiber to keep the presentation simple.

The idea is that, given a goal $\mathbf{G}$ in a context $\langle P, \sigma\rangle$, we can reduce $\mathbf{G}$ with the standard clauses and arrows in P, or we can move to a context $\langle P', \rho\rangle$ with $P' \supseteq P$.

To be more formal, assume we work with an empty program. Then, given a goal $\mathbf{G}$ in the fiber $\langle P, \sigma\rangle$, we can apply the following most general derivation steps:

- $\mathbf{G} \overset{\langle\langle id_P, r\rangle, f\rangle}{\leadsto} \mathbf{Tl}$ if $\langle r, f\rangle$ is a most general reduction pair in $\mathcal{P}$. In this case, we are backchaining w.r.t. an arrow in $\mathcal{P}$;

- $\mathbf{G} \overset{\langle\langle id_P, r\rangle, t^\sharp(cl)\rangle\rangle}{\leadsto} t^\sharp(\mathbf{Tl})$ if $cl = \mathbf{Hd} \leftarrow \mathbf{Tl} \in P$ and $\langle r, t\rangle$ is an mgu of $\mathbf{G}$ and $\mathbf{Hd}$. In this case, we are doing a back-chain w.r.t. a clause in P, without changing the program in the context;

- $\mathbf{G} \overset{\langle\langle P\cup\{cl\}, r\rangle, t^\sharp(cl)\rangle\rangle}{\leadsto} t^\sharp(\mathbf{Tl})$ if $cl = \mathbf{Hd} \to \mathbf{Tl}$ and $\langle r, t\rangle$ is an mgu of $\mathbf{G}$ and $\mathbf{Hd}$. This is just like the previous case, but we also add a new clause cl to the program.

Now, if f is an answer for the goal $\mathbf{G}$ of sort σ with the program P in $\mathcal{P}$, then $\langle P, f\rangle$ is an answer for the goal $\mathbf{G}$ in the fiber $\langle \emptyset, \sigma\rangle$ in $\mathcal{R}_{\mathcal{P}}$.

6.2.2 CLP

We are going to show that our categorical framework can handle quite easily the broad class of languages known with the name of *constraint logic programming* [40]. It is evident we need a categorical counterpart of a *constraint system* . We refer to the definition which appear in [59].

Definition 6.2.1 (Constraint system) *A* constraint system *over the finite product category $\mathbb{C}$ is an indexed category over $\mathbb{C}$ such that each fiber is a meet semilattice and reindexing functors have left adjoints.*

Now, given a constraint system $\mathcal{D}$ over $\mathbb{C}$, let us denote by $\mathbb{D}$ the corresponding category we obtain by the Grothendieck [39] construction. To be more precise:

- objects of $\mathbb{D}$ are pairs $\langle \sigma, c\rangle$ where $\sigma \in \mathrm{Ob}_{\mathbb{C}}$ and $c \in \mathrm{Ob}_{\mathcal{D}(\sigma)}$;

- arrows in $\mathbb{D}$ from $\langle \sigma_1, c_1\rangle$ to $\langle \sigma_2, c_2\rangle$ are given by arrows $f : \sigma_1 \to \sigma_2$ in $\mathbb{C}$ such that $c_1 \leq \mathcal{D}(f)(c_2)$.

Given a predicate signature Π over $\mathbb{C}$, we define a new strict premonoidal LP doctrine $\mathcal{P}_{\Pi,\mathbb{D}}$ over $\mathbb{D}$ as follows:

- for each $\langle \sigma, c\rangle$ in $\mathbb{D}$, objects are freely generated by the premonoidal structure starting from atomic goals $p(t)$ for $p : \rho$ in Π and $t : \sigma \to \rho$ and from constraints $c' \in \mathrm{Ob}_{\mathcal{D}\sigma}$;

- for each $\langle \sigma, c\rangle$ in $\mathbb{D}$, arrows are freely generated by composition and premonoidal structure starting from
  - an arrow $c' \to c''$ for each $c', c'' \in \mathcal{D}\sigma$ such that $c' \leq c''$. These arrows do compose as they were in $\mathcal{D}\sigma$;

  - an arrow the $\mathsf{true}_c : \top_\sigma \to c$ where $\top_\sigma$ is the identity element of the monoidal structure.

- for each arrow $\langle \sigma_1, c_1 \rangle \xrightarrow{f} \langle \sigma_2, c_2 \rangle$ in $\mathbb{D}$, the reindexing functor $\mathcal{P}_{\Pi,\mathbb{D}}(f)$ maps
  - an atomic goal $p(t)$ to $p(f\,;t)$;
  - a constraint c to $\mathcal{D}(f)(c)$;
  - an arrow $c \to c'$ to $f^\sharp c \to f^\sharp d$;
  - the arrow true_c to $\mathsf{true}_{\mathcal{D}(f)(c)}$.

  For all the other arrows and objects the reindexing functor is uniquely defined by composition and by the premonoidal structure.

Now, we fix a program P such that, for each clause $cl : \mathbf{Hd} \leftarrow \mathbf{Tl}$, $\mathbf{Hd}$ is an atomic goal. If we analyze the most general derivation steps we can perform, we obtain the following ones

- for an atomic goal $p(f)$ in the (σ, c), a step

$$p(f) \xrightarrow{\langle (\alpha, r^\sharp c) \xrightarrow{r} (\sigma, c), (\alpha, r^\sharp c) \xrightarrow{t} (\sigma', \mathsf{true}), \top_\alpha, \top_\alpha, cl \rangle} t^\sharp \mathbf{Tl} \quad , \tag{6.2.2}$$

  when $cl : \mathbf{Hd} \leftarrow \mathbf{Tl}$ of sort σ' and $\langle r, t \rangle$ is an mgu of $\mathbf{G}$ and $\mathbf{Hd}$ in $\mathcal{P}_\Pi$;

- for a constraint c' of sort σ, a step

$$c' \xrightarrow{\langle (\sigma, c \sqcap_\sigma c') \overset{id_\sigma}{\to} (\sigma, c)), \mathsf{true}_{c'} \rangle} \top_{(c \sqcap c', \sigma)} \quad , \tag{6.2.3}$$

  where $\sqcap_\sigma$ is the meet operator in $\mathcal{D}\sigma$.

- for all the other goals, possible derivations are given by the premonoidal structure.

Therefore, we have a m.g. refutation d of $\mathbf{G}$ from the context (σ, c) with $\mathsf{answer}(d) = f : \langle \rho, c' \rangle \to \langle \sigma, c \rangle$ when the execution of $\mathbf{G}$ succeeds with computed constraint c' and computed answer f.

6.3 Selection Rules and Left-Monoidal Structures

Although in pure logic programming we are generally concerned with properties, such as correct answers, which do not depend on a particular selection rule, this is not always the case. If we want to observe call patterns or resultants, for example, the choice of the selection rule is essential.

Therefore, it would be very useful to be able to express selection rules in our framework. Actually, this is made possible by weakening the premonoidal structure we use to represent conjunctions. We will show, in particular, how to force a leftmost selection rule with the use of left-monoidal structures.

Definition 6.3.1 (Left-monoidal structure) *A* left-monoidal structure *in a category $\mathbb{C}$ is given by a functor $\otimes : \mathbb{C} \times \mathrm{Ob}_{\mathbb{C}} \to \mathbb{C}$, an identity element $\top \in \mathrm{Ob}_{\mathbb{C}}$ and natural isomorphisms:*

$$\lambda_A : \top \otimes A \to A \tag{6.3.1}$$

$$\rho_A : A \otimes \top \to A \tag{6.3.2}$$

Note that for λ is a natural transformation in $\mathrm{Ob}_{\mathbb{C}}$ and ρ is a natural transformation in $\mathbb{C}$.

As usual, we call *left-monoidal category* a category endowed with a left-monoidal structure. Given $\mathbb{C}$ and $\mathbb{D}$ left-monoidal categories, a (strict) *left-monoidal functor* is a functor $F : \mathbb{C} \to \mathbb{D}$ which preserves the left-monoidal structure on the nose.

Definition 6.3.2 (Left-monoidal functor) *Given left-monoidal categories $\mathbb{C}$ and $\mathbb{D}$, a* left-monoidal functor *$F : \mathbb{C} \to \mathbb{D}$ is a functor such that*

$$F(f \otimes B) = F(f) \otimes F(B)$$
$$F(\lambda_A) = \lambda_{F(A)}$$
$$F(\rho_A) = \rho_{F(A)}$$

for each $A, B \in \mathrm{Ob}_{\mathbb{C}}$ and $f \in \mathrm{Mor}_{\mathbb{C}}$.

It is easy to prove that left-monoidal categories and functors form a category with the standard definition of composition of functors and identities.

Then, we define left-monoidal indexed categories, indexed functors, LP doctrines, interpretations and semantic doctrines as usual, following the guideline of what we have done for premonoidal structures.

We can define an analogous notion of left-monoidal categorical derivation.

Definition 6.3.3 (Left-monoidal derivations) *Given a left-monoidal LP doctrine $\mathcal{P}$, we define a labeled transition system $(\biguplus_{\sigma \in \mathrm{Ob}_{\mathbb{C}}} \mathrm{Ob}_{\mathcal{P}\sigma}, \leadsto)$ with goals as objects, according to the following rules:*

backchain-clause) $\mathbf{G} \overset{\langle r,t,\mathbf{G}',cl\rangle}{\leadsto} t^\sharp \mathbf{Tl} \otimes \mathbf{G}'$ *if cl is a clause $\mathbf{Hd} \leftarrow \mathbf{Tl}$ and there are arrows t and r in $\mathbb{C}$ such that $r^\sharp \mathbf{G} = t^\sharp \mathbf{Hd} \otimes \mathbf{G}'$.*

backchain-arrow) $\mathbf{G} \overset{\langle r,f\rangle}{\leadsto} \mathbf{Tl}$ *if $\langle r, f : \mathbf{Hd} \leftarrow \mathbf{Tl}\rangle$ is a reduction pair for $\mathbf{G}$.*

A left-monoidal derivation *is a (possibly empty) derivation in this transition system.*

Starting from left-monoidal derivations we can build the free left-monoidal model with the same procedure we have seen to work for premonoidal derivations.

6.4 Conclusions

In this chapter we have introduced premonoidal structures in the fibers of our categorical structures, so that we are now able to treat conjunctions in goals. While this works in our case, a major drawback of this approach is that every time we want to extend the structure of fibers (hence the syntax of goals), we need to proceed by hand. We would like a sort of meta-framework that works for all the LP doctrines with internal structures that enjoy some particular

We need to find a way to formally describe properties of categorical structures. We think that internal categories could be used with profit in this case, as done, for example, in [22].

It is important to observe that premonoidal structures give origin to new problems in the definition of a fixpoint semantics. If we allow clauses with non-atomic heads, like $\mathbf{G}_1 \otimes \mathbf{G}_2 \leftarrow \mathbf{Tl}$, how it is possible to define a fixpoint semantics that is compositional on $\otimes$? It should be $[\![\mathbf{G}_1 \otimes \mathbf{G}_2]\!] = [\![\mathbf{G}_1]\!] \otimes [\![\mathbf{G}_2]\!]$, but this is independent from the clauses of the program. We can think that the previous clause should have an influence on the semantics of $\mathbf{G}_1$ and $\mathbf{G}_2$ independently. With finite product structures this would be easy: thanks to projections, we could obtain derived clauses $\mathbf{G}_1 \leftarrow \mathbf{Tl}$ and $\mathbf{G}_2 \leftarrow \mathbf{Tl}$, but there are no similar constructions for simple premonoidal structures.

Conclusions

Part I

The usefulness of a general semantic framework strictly depends on its ability to be easily instantiated to well known cases while suggesting natural extensions to them. In the case of a framework which we want to use as a reference for the development of procedures for static analyses, we also require that theoretical descriptions can be implemented in a straightforward way.

In Part I we presented a semantic framework for sequent calculi modeled around the idea of the three semantics of Horn clauses and around abstract interpretation theory. With particular reference to groundness and correct answers, we have shown that well known concepts in the case of Horn clauses can be obtained as simple instances of more general definitions valid for much broader logics. This has two main advantages. First of all, we can instantiate the general concepts to computational logics other then Horn clauses, such as hereditary Harrop formulas. Moreover, the general definitions often make explicit the logical meaning of several constructions (such as correct answers), which are otherwise obscured by the use of small logical fragments. We think that, following this framework as a sort of guideline, it is possible to export most of the results for positive logic programs to the new logic languages developed following proof-theoretic methods.

Following this idea, we have presented a new definition for correct answers and correct resultants which can be applied to the full first order logic (both classical and intuitionistic). Moreover, we have shown that a well known abstraction of logic program semantics, namely groundness, can be easily re-introduced inside our framework. This definitions are so general that they can be reused with only slight changes for every logic system with standard quantifier rules, such as linear logic or modal logic.

Regarding the implementation of static analyzers from the theoretical description of the domains, not all the issues have been tackled. While a top-down analyzer can often be implemented in a straightforward manner, like our interpreter for groundness, the same does not hold for bottom-up analyzers. Since for a bottom-up analysis we have to build the entire abstract semantics of a logic, we need a way to isolate a finite number of "representative sequents" from which the semantics of all the others can easily be inferred: it is essentially a problem of compositionality.

We are actually studying this problem and we think that extending the notion of a logic $\mathcal{L}$ with the introduction of some *rules for the decomposition of sequents* will add to the theoretical framework the power needed to easily derive compositional $T_{\mathcal{L}}$ operators, thus greatly simplifying the implementation of bottom-up analyzers.

We also need a way to reduce non-determinism in abstract interpreters. This is a problem which has been tackled thoroughly in the field of automatic deduction. A standard solution is to use unification to reduce non-determinism in the introductions of quantifiers [13, 65].

Moreover, the problem of groundness analysis for intuitionistic logic could be further addressed. The precision we can reach with the proposed domain can be improved by refining the abstraction function, and the implementation of the analyzer could be reconsidered to make it faster. Finally, it should be possible to adapt the domain to work with linear logic. We would like to treat unification in our framework, and we want to do this without any major modification. We are working in the direction of defining an abstraction of proof skeletons using extra-logical variables such that the corresponding optimal abstract operators automatically computes the semantics trough unification.

We think that our approach to the problem of static analysis of logic programs is new. There are several papers focusing on logic languages other than Horn clauses [49] but, to the best of our knowledge, the problem has never been tackled before from the proof-theoretic point of view. An exception is [69], which, however, is limited to hereditary Harrop formulas and does not come out with any real implementation of the theoretical framework.

An alternative presentation of correct answers can be found in [3]. In that paper, correct answers are presented by using model-theoretic concepts. We think that, w.r.t. [3], the new definitions of correct answers and groundness answers we introduce here give us more intuitive and accurate results and a much cleaner theory.

Part II

In Part II, we have introduced a categorical framework to handle several extensions to logic programming. If we compare the intentions stated in Section 5.2 with the current results, it is evident that we are far away from what we had planned. In particular, two appear the main problems: knowledge:

- we have almost not tackled the problem of programs with constraints on goals when it comes to fixpoint semantics. From this point of view, the only advantage offered by our framework is the ability to include pre-defined goal with a fixed and immutable interpretation. Goals whose semantics can be defined by clauses (i.e. dynamic goals) must be freely generated as in [30];

- every time we want to extend the structure of fibers (hence the syntax of goals), we need to proceed by hand. We would like a sort of meta-framework that works for all the LP doctrines with internal structures that enjoy some particular properties.

For the first problem, the main concern is about defining E_P in such a way that it is a functor, mapping natural transformation to natural transformations. To overcome the second problem, we need to find a way to formally describe properties of categorical structures. We think that internal categories could be used with profit in this case, as done, for example, in [22].

It is important to observe that these two different problems are not unrelated. There are particular structures for fibers that impose restrictions on the type of allowed clauses. For example, if we use monoidal structures and allow clauses with non-atomic heads, like $\mathbf{G}_1 \otimes \mathbf{G}_2 \leftarrow \mathbf{Tl}$, how it is possible to define a fixpoint semantics that is compositional on $\otimes$? It should be $[\![\mathbf{G}_1 \otimes \mathbf{G}_2]\!] = [\![\mathbf{G}_1]\!] \otimes [\![\mathbf{G}_2]\!]$, but this is independent from the clauses of the program. We can think that the previous clause should have an influence on the semantics of $\mathbf{G}_1$ and $\mathbf{G}_2$ independently. With finite product structures this would be easy: thanks to projections, we could obtain derived clauses $\mathbf{G}_1 \leftarrow \mathbf{Tl}$ and $\mathbf{G}_2 \leftarrow \mathbf{Tl}$, but there are no similar constructions for simple monoidal structures.

Interactions between the two Frameworks

The two parts of the thesis are oriented toward different kinds of logic programming languages. Actually, Part I is concerned with languages which extend Horn clauses with a richer logical structure, while Part II is concerned with languages with a richer structure of models. These two extensions are, for a good way, orthogonal, hence it would very interesting to merge the two framework we have presented into a single general framework which could handle almost all of the pure declarative extension for logic programming which have been proposed. This probably would require a categorical framework, where the structure of the fibers are given by the proof theoretical properties of the language. So, for example, if the language has essentially universal quantifiers, we need to consider right adjoints of reindexing functors. In the general case of full first order logic, we probably need an hyperdoctrine as in [64].

However, also if we do not care about such a generalization, it could be interesting to reconstruct, in the categorical framework, the idea of abstraction that is central in the first part of the thesis. It seems that the natural counterpart of Galois connection could be fibered adjunctions.

Bibliography

[1] G. Amato. Correct Answers for First Order Logic. In the electronic proceedings of the *APPIA-GULP-PRODE 2000 Joint Conference on Declarative Programming*, `http://w3.dm.univaq.it/agp00/general.html`, 2000.

[2] G. Amato and G. Levi. Properties of the lattice of observables in logic programming. In M. Falaschi and M. Navarro, editors, *Proceedings of the APPIA-GULP-PRODE'97 Joint Conference on Declarative Programming*, pages 175–187, 1997.

[3] G. Amato and G. Levi. Abstract Interpretation Based Semantics of Sequent Calculi. In J. Palsberg, editor, *Static Analysis Symposium 2000*, volume 1824 of *Lecture Notes in Computer Science*, pages 38–57. Springer-Verlag, Berlin, 2000.

[4] J. M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[5] K. R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 495–574. Elsevier and The MIT Press, 1990.

[6] T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Boolean functions for dependency analysis: Algebraic properties and efficient representation. In B. Le Charlier, editor, *Proc. Static Analysis Symposium, SAS'94*, volume 864 of *Lecture Notes in Computer Science*, pages 266–280. Springer-Verlag, 1994.

[7] A. Asperti and G. Longo. *Categories, Types, and Structures.* Foundations of Computing Series. The MIT Press, Cambridge, Massachussetts, 1991.

[8] A. Asperti and S. Martini. Projections instead of variables. In G. Levi and M. Martelli, editors, *Proc. Sixth Int'l Conf. on Logic Programming*, pages 337–352. The MIT Press, 1989.

[9] R. Barbuti, R. Giacobazzi, and G. Levi. A General Framework for Semantics-based Bottom-up Abstract Interpretation of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(1):133–181, 1993.

[10] M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice Hall, New York, 1990.

[11] G. Birkhoff. *Lattice Theory*, volume XXV of *AMS Colloquium Publications*. AMS, third edition, 1967.

[12] A. Bossi, M. Gabbrielli, G. Levi, and M. C. Meo. A Compositional Semantics for Logic Programs. *Theoretical Computer Science*, 122(1–2):3–47, 1994.

[13] K.A. Bowem. Programming with full first-order logic. *Machine Intelligence*, pages 421–440, 1982.

[14] A. Brogi. *Program Construction in Computational Logic*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 1993.

[15] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Modular logic programming. *ACM Transactions on Programming Languages and Systems*, 16(4):1361–1398, July 1994.

[16] D. T. Burhans and S. C. Shapiro. Expanding the notion of answer in rule-based systems. Technical Report 99-07, Department of Computer Science and Engineering, SUNY Buffalo, November 1999.

[17] K. L. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[18] M. Comini, G. Levi, and M. C. Meo. A theory of observables for logic programs. *Information and Computation*, 1999. To appear.

[19] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 39(1-3):43–93, 1999.

[20] M. Comini and M. C. Meo. Compositionality properties of *SLD*-derivations. *Theoretical Computer Science*, 211(1 & 2):275–309, 1999.

[21] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press and McGraw-Hill Book Company, 6th edition, 1992.

[22] A. Corradini and A. Asperti. A categorical model for logic programs: Indexed monoidal categories. In *Proceedings REX Workshop '92*. Springer Lectures Notes in Computer Science, 1992.

[23] A. Corradini and U. Montanari. An algebraic semantics for structured transition systems and its application to logic programs. *Theoretical Computer Science*, 103(1):51–106, August 1992.

[24] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. Fourth ACM Symp. Principles of Programming Languages*, pages 238–252, 1977.

[25] P. Cousot and R. Cousot. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming*, 13(2 & 3):103–179, 1992.

[26] P. Cousot and R. Cousot. Temporal abstract interpretation. In *Conference Record of the Twentyseventh Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–25, Boston, Mass., January 2000. ACM Press, New York, NY.

[27] S. K. Debray. Formal bases for dataflow analysis of logic programs. In G. Levi, editor, *Advances in logic programming theory*, pages 115–182. Clarendon Press, Oxford, 1994.

[28] A. Di Pierro, M. Martelli, and C. Palamidessi. Negation as Instantiation: a new rule for the treatment of negation in Logic Programming. In K. Furukawa, editor, *Proc. Eighth Int'l Conf. on Logic Programming*, pages 32–45. The MIT Press, 1991.

[29] S. Feferman. Lectures on proof theory. In M. Lob, editor, *Proceedings of the Summer School in Logic, Leeds '67*, volume 70 of *Lecture Notes in Mathematics*, pages 1–107, Berlin, 1968. Springer-Verlag.

[30] S. Finkelstein, P. Freyd, and J. Lipton. Logic programming in tau categories. In *Computer Science Logic '94*, volume 933 of *Lecture Notes in Computer Science*, pages 249–263. Springer Verlag, Berlin, 1995.

[31] S. Finkelstein, P. Freyd, and J. Lipton. A new framework for declarative programming. To appear in *Theoretical Computer Science*, 2000.

[32] P. J. Freyd and A. Scedrov. *Categories, Allegories.* North-Holland, Elsevier Publishers, Amsterdam, 1990.

[33] M. Gabbrielli, G. Levi, and M. C. Meo. Resultants semantics for PROLOG. *Journal of Logic and Computation*, 6(4):491–521, 1996.

[34] G. Gentzen. Untersuchungen über das logische Schliessen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in [35], pp. 68–131.

[35] G. Gentzen. *Collected Papers of Gerhard Gentzen.* North Holland, Amsterdam, 1969. Edited by M.E. Szabo.

[36] L. Giordano and A. Martelli. Structuring logic programs: a modal approach. *Journal of Logic Programming*, 21 (2):59–94, 1994.

[37] C. Green. Theorem-proving by resolution as a basis for question-answering systems. In Bernard Meltzer, Donald Michie, and Michael Swann, editors, *Machine Intelligence 4*, pages 183–205. Edinburgh University Press, Edinburgh, Scotland, 1969.

[38] J.S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Journal of Information and Computation*, 110(2):327–365, May 1994.

[39] B. Jacobs. *Categorical Logic and Type Theory*. Studies in Logic and the Foundations of Mathematics. North Holland, Elsevier, 1999.

[40] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19–20:503–581, 1994.

[41] G.M. Kelly. On Mac Lane's condition for coherence of natural associativities, commutativities, etc. *Journal of Algebra*, 1:397–402, 1964.

[42] S.C. Kleene. Permutability of inferences in Gentzen's calculi LK and LJ. *Memoris of the AMS*, 10, 1952.

[43] A. Kock and G. E. Reyes. Doctrines in categorical logic. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 283–313. North Holland, 1977.

[44] F. W. Lawvere. *Functorial Semantics of Algebraic Theories*. PhD thesis, Columbia University, 1963.

[45] G. Levi and P. Volpe. Derivation of Proof Methods by Abstract Interpretation. In *Proceedings of the Joint International Coneference ALP/PLILP'98*, 1998.

[46] J. Lipton and R. McGrail. Encapsulating data in logic programming via categorical constraints. In C. Palamidessi, H.Glaser, and K. Meinke, editors, *Principles of Declarative Programming*, volume 1490 of *Lecture Notes in Computer Science*, pages 391–410. Springer Verlag, Berlin, 1998.

[47] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987. Second edition.

[48] M. Makkai and G. E. Reyes. *First Order Categorical Logic*, volume 611 of *Lecture Notes in Mathematics*. Springer-Verlag, 1977.

[49] F. Malésieux, O. Ridoux, and P. Boizumault. Abstract compilation of λProlog. In J. Jaffar, editor, *Joint International Conference and Symposium on Logic Programming*, pages 130–144, Manchester, United Kingdom, June 1998. MIT Press.

[50] R. McGrail. *Modules, Monads and Control in Logic Programming*. PhD thesis, Wesleyan University, 1998.

[51] D. Miller. Hereditary Harrop Formulas and Logic Programming. In *Proceedings of the VIII International Congress of Logic, Methodology and Philosophy of Science*, pages 153–156, Moscow, Russia, August 1987.

[52] D. Miller. A Logical Analysis of Modules in Logic Programming. *Journal of Logic Programming*, 6:79–108, 1989.

[53] D. Miller. FORUM: a multiple-conclusion specification logic. *Theoretical Computer Science*, 165(1):201–232, 1996.

[54] D. Miller, F. Pfenning, G. Nadathur, and A. Scedrov. Uniform proofs as a foundation for Logic Programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[55] B. Möller. On the Algebraic Specification of Infinite Objects – Ordered and Cntinuous Models of Algebraic Types. *Acta Informatica*, 22:537–578, 1985.

[56] G. Nadathur. A Proof Procedure for the Logic of Hereditary Harrop Formulas. *Journal of Automated Reasoning*, 11:115–145, 1993.

[57] G. Nadathur and D. Miller. An Overview of λProlog. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Fifth International Logic Programmiong Conference*, pages 810–827. MIT Press, 1988.

[58] G. Nadathur and D. Miller. An overview of λProlog. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming*, pages 810–827. The MIT Press, 1988.

[59] P. Panangaden, V. J. Saraswat, P. J. Scott, and R. A. G. Seely. A Hyperdoctrinal View of Concurrent Constraint Programming. In J. W. de Bakker et al, editor, *Semantics: Foundations and Applications*, volume 666 of *Lecture Notes in Computer Science*, pages 457–475. Springer-Verlag, 1993.

[60] F. Pfenning. Dependent types in logic programming. In Frank Pfenning, editor, *Types in Logic Programming*, chapter 10, pages 285–311. MIT Press, Cambridge, Massachusetts, 1992.

[61] G. Plotkin. Pisa Notes (On Domain Theory). available at `http://www.dcs.ed.ac.uk/home/gdp/publications`, 1983.

[62] J. Power and E. Robinson. Premonoidal categories and notions of computation. *Mathematical Structures in Computer Science*, 7(5):453–468, October 1997.

[63] D.E. Rydeheard and R.M. Burstall. A categorical unification algorithm. In *Category Theory and Computer Programming, LNCS 240*, pages 493–505, Guildford, 1985. Springer Verlag.

[64] R.A.G. Seely. Hyperdoctrines, Natural Deduction and the Beck Condition. *Zeitschrift für Math. Logik Grundlagen der Math.*, 29(6):505–542, 1983.

[65] N. Shankar. Proof search in the intuitionistic sequent calculus. In M.E. Stickel, editor, *10th Int. Conf. on Automated Deduction*, volume 449 of *Lecture Notes in Artifical Intelligence*, pages 522–536. Springer-Verlag, 1990.

[66] L. Sterling and E. Y. Shapiro. *The Art of Prolog.* The MIT Press, 1986.

[67] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5:285–309, 1955.

[68] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[69] P. Volpe. Abstractions of uniform proofs. In M. Hanus and M. Rodriguez-Artalejo, editors, *Algebraic and Logic Programming, Proc. 5th International Conference, ALP '96*, volume 1139 of *Lecture Notes in Computer Science*, pages 224–237. Springer-Verlag, 1996.

Index