Logic Functors : a Framework for Developing Embeddable Customized Logics

Sébastien Ferré, Olivier Ridoux

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Logic functors: a framework for developing embeddable customized logics

Sébastien Ferré — Olivier Ridoux

N° 4457

Mai 2002

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--4457--FR+ENG

Logic functors: a framework for developing embeddable customized logics

Sébastien Ferré* , Olivier Ridoux

Thème 2 — Génie logiciel
et calcul symbolique
Projet Lande

Rapport de recherche n° 4457 — Mai 2002 — 32 pages

Abstract: Logic-based applications often use *customized logics* which are composed of several logics. These customized logics are also often *embedded* as a black-box in an application. So, implementing them requires the specification of a well-defined interface with common operations such as a parser, a printer, and a theorem prover. In order to be able to compose these logic, one must also define composition laws, and prove their properties. We present the principles of *logic functors* and their compositions for constructing logics that are *ad-hoc*, but sound. An important issue is how the operations of different sublogics inter-operate. We propose a formalization of the logic functors, their semantics, implementations, proof-theoretic properties, and their composition.

Key-words: applied logic, software component

* This author is supported by a scholarship from CNRS and Région Bretagne

Les foncteurs logiques — un cadre pour le développement de composant logiques enfouis et ad-hoc

Résumé : Les applications logicielles de la logique utilisent le plus souvent des logiques *ad hoc* résultants de la composition de plusieurs logiques. Ces logiques ad hoc sont aussi souvent enfouies dans les applications en tant que boîtes noires. Les mettre en oeuvre demande donc de spécifier une interface bien définie avec des opérations communes comme l'analyse syntaxique, l'affichage ou la démonstration de théorèmes. Afin de les combiner, nous devons aussi définir des lois de compositions, et en déterminer les propriétés. Nous présentons la théorie des *foncteurs logiques* et de leur composition pour construire des logiques *ad hoc* mais consistantes. Une question importante est de comprendre comment les opérations des différentes logiques entrant dans une composition interfèrent. Nous proposons une formalisation des foncteurs logiques, de leur sémantique, de leur implémentation, de leurs propriétés vis-à-vis de la démonstration, et de leur composition.

Mots-clés : logique appliquée, composant logiciel

1 Introduction

We present a framework for building embeddable automatic theorem provers for *customized logics*. The framework defines *logic functors* as logic components, e.g., propositional logic or intervals. Logic functors can be composed to form new logics, e.g., propositional logic on intervals.

Each logic functor has its own proof-theory, which can be implemented as a theorem prover. Our goal is that the proof-theory and theorem prover of a composition of logic functors result from an automatic composition of the proof-theory and theorem prover of each logic functor.

All logic functors and their compositions implement a common *interface*. This makes it possible to program generic applications that can be instantiated with a logic component. Conversely, customized logics built using the logic functors can be *embedded* in an application that respects this interface.

In summary, logic functors specify off-the-shelf software components, the validation of the composition of which reduces to a form of type-cheking, and their composition automatically results in an automatic theorem prover. Logic functors can be assembled by laymen, and used routinely in system-level programming: e.g., compilers, operating systems, file-systems, information systems.

This article is organized as follows: Section 2 develops our motivations, Section 3 introduces the notions of logics and logic functors, and several logic functor properties like *completeness* and *correctness*, Section 4 introduces a simple nullary logic functor as an example, and a more sophisticated unary logic functor that raises important questions on the properties of logics that result from a composition of logic functors, and Section 5 answers these questions by introducing a new property, called *reducedness*. In Section 6, we compare this work with other works, and we conclude this article. Appendix A presents more nullary logic functors, and Appendix B presents more n-ary logic functors.

2 Motivations

2.1 Logic-based information processing systems

In [FR00b, FR01], we have proposed a Logical Information System that is built on a variant of Formal Concept Analysis [GW99, FR00a]. The framework is generic in the sense that any logic whose deduction relation forms a lattice can be plugged-in. However, leaving the logic totally undefined sets a too large responsibility on the end-users, or even on a knowledge-base administrator. It is unlikely they can design such a logical component themselves. Using the framework developed in this article, one can design a toolbox of logical components, and only leave the user the responsibility of composing them. The design of this Logical Information System is the main motivation for this research.

However, we believe the application scope of this research goes beyond our logical information system. Several information processing domains have a logic-based variant in which logic plays a crucial role: e.g., logic-based information retrieval [SM83, vRCL98], logic-based diagnosis [Poo88], logic-based programming [Llo87, MS98], logic-based program analysis [SFRW98, AMSS98, CSS99]. These variants not only model an information processing domain in logic, but they bring to the front solutions in which logic is the main engine, even at run-time. This can be illustrated by the difference between using a logic of programs and programming in logic.

Even in information processing domains where traditionally logic does not play a crucial role it has been proposed to embed a logic component in otherwise not logic-based systems. For instance, in [IB96] the authors propose to model quality of service (QoS) conditions in logic, and to make applications check dynamically that the platform on which they run enforces the condition for a specified quality of service.

The logic in use in these system is often not defined by a single pure deduction system, but rather combines several logics and concrete domains. The designer of the application will have to make an *ad hoc* proof of consistency and an *ad hoc* implementation (a theorem prover) every time he imagines a new *ad hoc* logic. Since these logics are often variants of a more standard logic we call them *customized logics*.

In order to favour separation of concerns, which is recognized as a good practice in software engineering, it is important that the application that is based on a logic engine, and the logic engine itself, be designed separately. So doing, one can assemble an application and a logic engine at will. This implies that the interface of the logic engine does not depend on the logic itself. This is what we call *embeddability* of the logical component.

For instance, practical query-answering systems often use a mixture of logic and concrete computations. Queries are built with logical connectives, and with purely operational constructs like wild-cards. This usually causes no harm because the query-answering system is not actually a theorem prover, and thus does not actually implement a logic. Indeed, in most cases a boolean query is used to filter concrete strings that contain no wild-cards, and no boolean connectives.

However, one can imagine a logic-based query-answering systems in which queries and data actually use the same language. For instance, one may use the full language with connectives and wild-cards both for describing entries in an information system, and for querying them (e.g., see [FR00b] and Sections 2.4 and 6.2). Then the query-answering system must decide something which can be written as $description \models query$ (where $\models$ means logical consequence of the logic used in descriptions and queries), and it must have the full capacity of a theorem prover for a logic whose syntax is the description/querying language. The choice of a particular logic depends on the application, but query-answering is generic and depends solely on the existence of $\models$.

Designing separately the application and the logical components that it uses raises the question of who really is to be the developer of an embedded logic component?

2.2 The actors of the development of an information processing system

In this section, we present our views on the Actors of the development on an information processing system. Note that Actors are not necessarily incarnated in one person each; each Actor gathers several persons possibly not living at the same time. In short, Actors are roles, rather than persons. Sometimes, Actors may even be incarnated in computer programs.

What follows is rather standard in the information system (especially data-base) community because it has adopted organisation standards of industry and administration, but we think it is not widely accepted by the academic community for other kinds of information processing systems, where it tends to follow more academic standards in which several Actors are collapsed into one, the Researcher.

The first Actor is the Theorist; he invents an abstract framework, like, say relational algebra, lattice theory, or logic (remember that Actors need not be single persons).

Sometimes the abstract framework finds applications, and a second Actor, the System Programmer, implements (part of) the theory in a *generic* system for a range of applications. This results in systems like data-bases, static analysers, or logic programming systems.

Then the third Actor, the Application Designer, applies the abstract framework to a concrete goal by *instantiating* a generic system. This can be done by composing a data-base schema, or a program property, or a logic program.

Finally, the User, the fourth Actor, maintains and uses an application. He queries a data-base, he analyses programs, or he runs logic programs. The User is often incarnated in programs.

Certainly, the User could be analysed further in Administrators, End-Users, etc. However, we stop here because it is the relation between the System Programmer and the Application Designer that interests us; the first one creates a generic system, and the second one instantiates it.

2.3 Genericity and instantiation

Genericity is often achieved by offering a language: e.g., a data-base schema language, a lattice operation language, and a programming language. Symmetrically, instantiation is done by programming and composing: e.g., drawing a data-base schema, composing an abstract domain for static analysis, or composing a logic program.

We propose to do the same for logic-based tools. Indeed, the System Programmer is competent for building a logic subsystem, but he does not know the application; he only knows a range of applications. On the other side the Application Designer knows the application, but is generally not competent for building a logic subsystem. In this article, we will act as System Programmers by providing elementary components for safely building a logic subsystem, and also as Theorists by giving formal results on the composition laws of these components.

More specifically, we explore the systematic building of logics using basic components that we call *logic functors*. By "construction of a logic" we mean the definition of its syntax, its semantics, and its abstract implementation as a deduction system. All the logic functors we describe in this article have also a concrete implementation as an actual program. Moreover, a *logic composer* that takes the description of a customized logic and builds a concrete logic component is implemented. So, we define a process that goes from the description of a logic in terms of logic functors to a concrete program that implements it.

2.4 Customized logics

The range of logic functors can be very large. We consider in this article only products and sums of logics, propositions (on arbitrary formulas), intervals, sets, valued attributes (abstracted w.r.t. values), strings (e.g., "begin with", "contains"), and $\mathcal{ONL}$ (a modal epistemic logic functor [Lev90]).

The whole framework development is geared towards manipulating logics as lattices, as in abstract interpretation. So, deduction is considered as a relation between formulas, and we study the conditions under which this relation is a partial order. This excludes non-monotonic logics though they still can be used as nullary logic functors. Note that non-monotonicity is seldom a goal in itself, and that notoriously non-monotonic features have a monotonic rendering; e.g., Closed World Assumption can be reflected in the monotonic modal logic $\mathcal{ONL}$. Note that even in our framework not all logics are lattices (nor their deduction relation is a partial order), but the most interesting ones (to be defined) can always be completed in a lattice.

We will consider as a motivating example an application for managing bibliographic entries. Each entry has a description made of its author name(s), publishing date, etc. The user navigates among a set of entries by comparing descriptions with queries that are written in the same language. The application answers navigation queries by lists of entries that match the queries, and lists of subqueries that can complement the current one to form a more precise query. So doing, we have a logic-based notion of navigation where matching a query is being in some place, and subqueries are links to other places. For instance, let us assume the following entry set:

- `descr(entry`$_1$`) =`
  `[author:"Kipling"/ title:"The Jungle Book"/ paper-back/ publisher:"Penguin"/`
  `year: 1985]`,
- `descr(entry`$_2$`) =`
  `[author:"Kipling"/ title:"The Jungle Book"/ hard-cover/ publisher:"Century Co."/`
  `year: 1908]`,
- `descr(entry`$_3$`) =`
  `[author:"Kipling"/ title:"Just So Stories"/ hard-cover/ publisher:""/ year: 1902]`.

An answer to a query `title: contains "Jungle"` might be

`hard-cover`	`publisher:"Penguin"`	`year: 1900..1950`
`paper-back`	`publisher:"Century Co."`	`year: 1950..2000`

because several entries (`entry`$_1$ and `entry`$_2$) have a description that entails the query (i.e., they are possible answers), and the application asks the user to make his query more precise by suggesting some relevant refinements. Note that `author:"Kipling"` is not a relevant refinement because it is true of all matching entries. For every possible answer `entry` we have `descr(entry)`$\models$`query`, and for every relevant refinement `x` the following holds

1. there exists a possible answer `e`$_1$ such that `descr(e`$_1$`)`$\models$`x`, and
2. there exists a possible answer `e`$_2$ such that `descr(e`$_2$`)`$\not\models$`x`.

In other words, a refinement must restrict the set of possible answers while avoiding to make it empty. We will not go any further in the description of this application (see [FR00b, FR01]). The essential is to note that

1. descriptions, queries, and answers belong to the same logical language, which combines logical symbols and concrete expressions like strings, numbers, or intervals, and
2. a similar application with a different logic can be imagined, e.g., for manipulating software components. So, it is important that all different logics share a common interface for being able to program separately the navigation system and the logic subsystem it uses.

2.5 Summary

We define tools for building *fully automatic* theorem provers for *customized logics*. This is because the User is not a sophisticated logic actor. Note also that the User may be a program itself: e.g., a mobil agent proving "quality of service" properties as it tries to execute on some host system [IB96]. This rules out interactive theorem provers.

Validating a theorem prover built upon our tools must be as simple as possible. Some kind of type-checking would be ideal. Again, this is because the Application designer, though more sophisticated than the User, is not a logic actor.

Finally, the resulting theorem provers must have a common interface so that they can be *embedded* in generic applications. Deduction is decidable in all the logic components that we define. So, they can be safely embedded as black-boxes in applications.

3 Logics and logic functors

If an Application Designer is to define a customized logic by the means of composing primitive components, these components should be of a rather high-level so that the resulting logic subsystem can be proven to be correct. Indeed, if the primitive components are too low-level, proving the correctness of the result is similar to proving the correctness of a program. So, we decided to define logical components that are very close to be logics themselves. So doing, we abandon computational expressivity, but we will see that proving correctness is little more than type-checking (i.e., using a rule like $\frac{A:\alpha\to\beta \quad B:\alpha}{A(B):\beta}$).

Our idea is to consider that a logic interprets its formulas as functions of their atoms. By abstracting atomic formulas from the language of a logic we obtain what we call a *logic functor*. A logic functor can be applied to a logic to form a new logic. For instance, if propositional logic is abstracted over its atomic formulas, we obtain a logic functor called *prop*, which we can apply to, say, a logic on intervals *interv*, to form propositional logic on intervals, $prop(interv)$.

3.1 Logics

We formally define the class of logics as structures, whose axioms are merely type axioms. Section 4 and Appendix A present examples of logics.

Definition 1 *A* syntax *AS is a denumerable set of (abstract syntax tree of) formulas.*

A *semantics* associates to each formula a subset of an *interpretation domain* where the formula is true of all elements. This way of treating formulas as unary predicate is akin to description logics [DLNS96].

Definition 2 *Given a syntax AS, a* semantics *S based on AS is a pair $(I, \models)$, where*

- *I is the* interpretation domain*,*
- *$\models \in \mathcal{P}(I \times AS)$, (where $\mathcal{P}(X)$ denotes the power-set of set X), is a* satisfaction relation *between interpretations and formulas.*

$i \models f$ reads "i is a model *of f". For every formula $f \in AS$, $M(f) = \{i \in I \mid i \models f\}$ denotes the set of all models of formula f. For every formulas $f, g \in AS$, an* entailment relation *is defined as "f entails g" iff $M(f) \subseteq M(g)$.*

Entailment is never used formally in this article, but we believe it gives a good intuition of our very frequent usage of the inclusion of sets of models.

The formulas define the logic language, the semantics defines its interpretation, and an *implementation* defines how the logic implements an *interface* that is common to all logics. This common interface comprises a deduction relation, a conjunction, a disjunction, a tautology, and a contradiction.

Definition 3 *Given a syntax AS and a symbol $'undef' \notin AS$, an* implementation *P based on AS is a 5-tuple $(\sqsubseteq, \sqcap, \sqcup, \top, \bot)$, where*

- *$\sqsubseteq \ \in \mathcal{P}(AS \times AS)$ is the* deduction relation*,*
- *$\sqcap, \sqcup \in AS \times AS \to AS \cup \{undef\}$ are the* conjunction *and the* disjunction*,*
- *$\top, \bot \in AS \cup \{undef\}$ are the* tautology *and the* contradiction*.*

Operations $\sqsubseteq$, $\sqcap$, $\sqcup$, $\top$, $\bot$ are all defined on the syntax of some logic, though they are not necessarily connectives of the logic, simply because the connectives of a logic may be different from these operations. Similarly, the syntax and the semantics may define quantifiers, though they are absent from the interface.

Note that this common interface can be implemented partially (by using $undef$) if it is convenient. Because the interface is the same for every logic, generic logic-based systems can be designed easily.

Definition 4 *A logic L is a triple (AS_L, S_L, P_L), where AS_L is (the abstract syntax of) a set of formulas, S_L is a semantics based on AS_L, and P_L is an implementation based on AS_L.*

When necessary, the satisfaction relation $\models$ of a logic L will be written $\models_L$, the interpretation domain I will be written I_L, the models $M(f)$ will be written $M_L(f)$, and each operation op will be written op_L.

In object oriented terms, this forms a class $\mathbb{L}$, which comprises a slot for the type of an internal representation, and several methods for a deduction relation, a conjunction, a disjunction, a tautology, and a contradiction. A logic L is simply an instance of this class.

Definition 3 shows that operations $\sqcap$, $\sqcup$ can be partially defined, and that operations $\top$, $\bot$ can be undefined.

Definition 5 *A logic is* partial *if either operations $\sqcap$ or $\sqcup$ or both are partially defined. It is unbounded if either operations $\top$ or $\bot$ or both is undefined.*

In the opposite case, a logic is respectively called total *and* bounded.

When necessary, we make it precise for which operation a logic is total/partial.

Total logics are usually preferred, because they make applications simpler, since they do no have to test for $undef$. Section 4.2 shows that the propositional logic functor applied to a partial logic always constructs a total logic.

There is no constraint, except for their types, on what $\sqsubseteq$, $\sqcap$, $\sqcup$, $\top$, $\bot$ can be. So, we define a notion of *consistency* and *completeness* that relates the semantics and the implementation of a logic. These notions are defined respectively for each operation of an implementation, and only for the defined part of them.

Definition 6 *Let L be a logic. An implementation P_L is* consistent *(resp.* complete*) in operation $op \in \{\sqsubseteq, \top, \bot, \sqcap, \sqcup\}$ w.r.t. a semantics S_L iff*
for all $f, g \in AS_L$

- $(op = \sqsubseteq)$ $f \sqsubseteq g \Longrightarrow M_L(f) \subseteq M_L(g)$ *(resp.* $M_L(f) \subseteq M_L(g) \Longrightarrow f \sqsubseteq g$*),*
- $(op = \top)$ $\top$ *is defined* $\Longrightarrow$ *always consistent (resp.* $M_L(\top) = I$*),*
- $(op = \bot)$ $\bot$ *is defined* $\Longrightarrow$ $M_L(\bot) = \emptyset$ *(resp. always complete),*
- $(op = \sqcap)$ $f \sqcap g$ *is defined* $\Longrightarrow$ $M_L(f \sqcap g) \subseteq M_L(f) \cap M_L(g)$
  (resp. $f \sqcap g$ *is defined* $\Longrightarrow$ $M_L(f \sqcap g) \supseteq M_L(f) \cap M_L(g)$*),*
- $(op = \sqcup)$ $f \sqcup g$ *is defined* $\Longrightarrow$ $M_L(f \sqcup g) \subseteq M_L(f) \cup M_L(g)$
  (resp. $f \sqcup g$ *is defined* $\Longrightarrow$ $M_L(f \sqcup g) \supseteq M_L(f) \cup M_L(g)$*).*

We say that an implementation is consistent *(resp.* complete*) iff it is consistent (resp. complete) in the five operations. We abbreviate "P_L is complete/consistent in op w.r.t. S_L" in "op_L is complete/consistent".*

Note that it is easy to make an implementation consistent and complete for the last four operations $\sqcap$, $\sqcup$, $\top$, $\bot$, by keeping them undefined, but then the implementation is of little use. Note also that a consistent $\sqsubseteq$ can always be extended into a partial order because it is contained in $\subseteq$.

In general, consistent and complete logics are preferred to ensure that expected answers, specified by the semantics, and actual answers, specified by the implementation, match. So, in these preferred logics deduction can be extended into a partial order. However, some logics defined on concrete domains are definitely not complete. So, an important issue is how to build complete logics with components that are not complete.

To the five operations of an implementation, we must add at least a parser and a printer for handling the concrete syntax of formulas. Indeed, an application may have to input/output formulas in a readable format. However, we do not consider them further as they do not cause any logical problem. On the contrary, the five logical operations (deduction, conjunction, disjunction, tautology, and contradiction) are at the core of a constructed logic.

3.2 Logic functors

Logic functors also have a syntax, a semantics, and an implementation, but they are all abstracted over one or more logics that are considered as formal parameters. We formally define the class of logic functors as structures. Section 4 and Appendix B presents examples of logic functors.

Given $\mathbb{L}$ the class of logics, logic functors are functions of type $\mathbb{L}^n \rightarrow \mathbb{L}$. In object oriented terms, this defines a *template* $\mathbb{F}$. For the sake of uniformity, logics are considered as logic functors with arity 0 (a.k.a. atomic functors, or nullary logic functors).

Assuming the class of all syntaxes is written $\mathbb{AS}$, that of all semantics is written $\mathbb{S}$, and that of all implementations is written $\mathbb{P}$ the syntax of a logic functor is simply a function of the syntaxes of the logics that are passed to it and returns the syntax of the resulting logic.

Definition 7 *Assuming the class of all syntaxes is written $\mathbb{AS}$, that of all semantics is written $\mathbb{S}$, and that of all implementations is written $\mathbb{P}$, a logic functor F is a triple (AS_F, S_F, P_F) where*

- *the abstract syntax AS_F is a function of type $\mathbb{AS}^n \to \mathbb{AS}$, such that $AS_{F(L_1,..,L_n)} = AS_F(AS_{L_1}, .., AS_{L_n})$;*
- *the semantics S_F is a function of type $\mathbb{S}^n \to \mathbb{S}$, such that $S_{F(L_1,..,L_n)} = S_F(S_{L_1}, .., S_{L_n})$;*
- *the implementation P_F is a function of type $\mathbb{P}^n \to \mathbb{P}$, such that $P_{F(L_1,..,L_n)} = P_F(P_{L_1}, .., P_{L_n})$.*

A logic functor in itself is neither partial or total, unbounded or bounded, complete or uncomplete, nor consistent or inconsistent. It is the logics that are built with a logic functor that can be qualified this way. However, it is possible to state that if a logic L has some property, then $F(L)$ has some other property. In the following, the definition of every new logic functor is accompanied with theorems stating under which conditions the resulting logic is total, consistent, or complete.

These theorems have all the form *hypothesis on L $\Rightarrow$ conclusion on $F(L)$*. We consider them as type assignments, F : *hypothesis* $\to$ *conclusion*. Similarly, totality/consistency/completeness properties on logics are considered as type assignments, L : *properties*, so that proving that $F(L)$ has some property regarding totality, consistency, or completeness, is simply to type-check it.

4 Composition of logic functors

We define a nullary logic functor and a propositional unary logic functor, and we observe that completeness may not propagate well when we compose them. We introduce a new property, called *reducedness*, that helps completeness propagate via composition of logic functors. From now on, the definitions are definitions of instances of $\mathbb{L}$ or $\mathbb{F}$.

4.1 Atoms

One of the most simple logic we can imagine is the logic of unrelated atoms *atom*. These atoms usually play the role of atomic formulas in most of known logics: propositional, first-order, description, etc.

Definition 8 *AS_{atom} is a set of atom names.*

Definition 9 *S_{atom} is $(I, \models)$ where $I = \mathcal{P}(AS_{atom})$ and $i \models a$ iff $a \in i$.*

The implementation reflects the fact that the atoms being unrelated they form an anti-chain for the deduction relation (a set where no pair of elements can be ordered).

Definition 10 *P_{atom} is $(\sqsubseteq, \sqcap, \sqcup, \top, \bot)$ where for every $a, b \in AS_{atom}$*
– $a \sqsubseteq b$ iff $a = b$
– $a \sqcap b = a \sqcup b = \begin{cases} a & \text{if } a = b \\ undef & \text{otherwise} \end{cases}$
– $\top$ and $\bot$ are undefined.

Theorem 11 *P_{atom} is consistent and complete in $\sqsubseteq$, $\top$, $\bot$, $\sqcap$, $\sqcup$ w.r.t. S_{atom}.*

Proof: Let a, b be atoms in AS_{atom}.

(deduction)
- either $a = b$: $a \sqsubseteq b$ is true, and $M(a) \subseteq M(b)$,
- or $a \neq b$: $a \sqsubseteq b$ is false, and $\{a\} \in M(a), \{a\} \notin M(b) \Longrightarrow M(a) \not\subseteq M(b)$.

(tautology and contradiction) true because $\top$ and $\bot$ are undefined.

(conjunction and disjunction) If $a \sqcap b$ is defined,
then $a \sqcap b = a$ and $a = b \Longrightarrow M(a \sqcap b) = M(a) = M(a) \cap M(b)$.
Proof for disjunction is similar. ■

In summary, P_{atom} is not bounded and is partial in both conjunction and disjunction, but it is consistent and complete w.r.t. S_{atom}.

4.2 Propositional logic abstracted over atoms

Let us assume that we use a logic as a description/querying language. Since it is almost always the case that we want to express conjunction, disjunction, and negation, the choice of propositional logic is natural. For instance, the / used to separate description fields in the bibliographical application (see Section 2) can be interpreted as conjunction. Similarly, disjunction and negation could be used, especially to express information like "published in 1908 or 1985". Propositional logic, *Prop*, is defined by taking a set of *atoms* A, and by forming a set of propositional formulas $Prop(A)$ by the closure of A for the three boolean connectives, $\wedge$, $\vee$, and $\neg$: the *boolean closure*.

Prop is usually considered as a free boolean algebra, since there is no relation between atoms. I.e., they are all pairwise incomparable for the deduction order. However in applications, atoms are often comparable. For instance, boolean queries based on string matching use atoms whose meaning is *contains* s, *is* s, *begins with* s, and *ends with* s where s is a character string. In this example, the atom *is* `"The Jungle Book"` implies *ends with* `"Jungle Book"`, which implies *contains* `"Jungle"`.

This leads to considering the boolean closure as a logic functor *prop*. So doing, the atoms can come from another logic where they have been endowed with a deduction order.

Definition 12 (Syntax) *The syntax AS_{prop} of the logic functor prop maps the syntax AS_A of a logic of atoms A to its syntactic closure by the operators $\wedge$, $\vee$, and the operator $\neg$.*

The interpretation of these operators is that of the connectives with the same names. It is defined by induction on the structure of the formulas. For atomic formulas of $AS_{prop(A)}$ (i.e., AS_A) the semantics is the same as in the logic A.

Definition 13 (Semantics) *S_{prop} is $(I_A, \models_A) \mapsto (I_A, \models)$ such that*

$$i \models f \text{ iff } \begin{cases} i \models_A f & \text{if } f \in AS_A \\ i \not\models f_1 & \text{if } f = \neg f_1 \\ i \models f_1 \text{ and } i \models f_2 & \text{if } f = f_1 \wedge f_2 \\ i \models f_1 \text{ or } i \models f_2 & \text{if } f = f_1 \vee f_2. \end{cases}$$

Definition 14 *P_{prop} is $(\sqsubseteq_A, \sqcap_A, \sqcup_A, \top_A, \bot_A) \mapsto (\sqsubseteq, \sqcap, \sqcup, \top, \bot)$ such that*

- *$f \sqsubseteq g$ is true iff there exists a proof of the sequent $\vdash \neg f \vee g$ in the sequent calculus of Table 1 (inspired from leanTAP [BP95, Fit98]).*

  In the rules, Δ is always a set of literals (i.e., atomic formulas or negations of atomic formulas), Γ is a sequence of propositions, L is a literal, X is a proposition, β is the disjunction of β_1 and β_2, α is the conjunction of α_1 and α_2, and $\overline{L}$ denotes the negation of L ($\overline{a} := \neg a$ and $\overline{\neg a} := a$).
- $f \sqcap g = f \wedge g$,
- $f \sqcup g = f \vee g$,
- $\top = a \vee \neg a$, *for any* $a \in AS_A$,
- $\bot = a \wedge \neg a$, *for any* $a \in AS_A$.

Rules $\top$-Axiom, $\bot$-Axiom, $\sqsubseteq$-Axiom, $\sqcap$-Rule, and $\sqcup$-Rule play the role of the ordinary axiom rule. The first two axioms are variants of the third one when either a or b is missing. Rules $\sqcap$-Rule, and $\sqcup$-Rule interpret the propositional connectives in the logic of atoms.

Note that the logic has a connective $\neg$ though its implementation has no corresponding operation. However, the deduction relation takes care of it. This is an example of how more connectives or quantifiers can be defined in a logic or a logic functor, though the interface does not know them. A logic functor for the predicate calculus could be defined in the same way, but since this theory is not decidable, the resulting logic functor would be of little use to form embeddable logic components. Instead of the full predicate calculus, it would be better to define a logic functor for a decidable fragment of it, like the fragments in the family of description logics [DLNS96].

Definition 15 *A sequent $\Delta \vdash \Gamma$ is called* valid *in $S_{prop(A)}$ iff it is* true *for every interpretation. It is* true *for an interpretation $i \in I$ iff there is an element in Δ that is false for i, or there is an element in Γ that is true for i.*

$\top$-Axiom:	$\neg b, \Delta \vdash \Gamma$	if $\top_A$ is defined and $\top_A \sqsubseteq_A b$
$\bot$-Axiom:	$a, \Delta \vdash \Gamma$	if $\bot_A$ is defined and $a \sqsubseteq_A \bot_A$
$\sqsubseteq$ -Axiom:	$a, \neg b, \Delta \vdash \Gamma$	if $a \sqsubseteq_A b$
$\sqcap$-Rule:	$\dfrac{a \sqcap_A b, \Delta \vdash \Gamma}{a, b, \Delta \vdash \Gamma}$	if $a \sqcap_A b$ is defined
$\sqcup$-Rule:	$\dfrac{\neg(a \sqcup_A b), \Delta \vdash \Gamma}{\neg a, \neg b, \Delta \vdash \Gamma}$	if $a \sqcup_A b$ is defined
$\neg\neg$-Rule:	$\dfrac{\Delta \vdash X, \Gamma}{\Delta \vdash \neg\neg X, \Gamma}$	literal-Rule: $\dfrac{\overline{L}, \Delta \vdash \Gamma}{\Delta \vdash L, \Gamma}$
β-Rule:	$\dfrac{\Delta \vdash \beta_1, \beta_2, \Gamma}{\Delta \vdash \beta, \Gamma}$	α-Rule: $\dfrac{\Delta \vdash \alpha_1, \Gamma \qquad \Delta \vdash \alpha_2, \Gamma}{\Delta \vdash \alpha, \Gamma}$

Table 1: Sequent calculus for deduction in propositional logic.

Lemma 16 *A sequent $\Delta \vdash \Gamma$ is valid in $S_{prop(A)}$ iff $\bigcap_{\delta\in\Delta} M(\delta) \subseteq \bigcup_{\gamma\in\Gamma} M(\gamma)$.*

Proof: $\Delta \vdash \Gamma$ is valid
$\iff \forall i \in I : (\exists \delta \in \Delta : i \not\models \delta) \vee (\exists \gamma \in \Gamma : i \models \gamma)$ Definition 15
$\iff \forall i \in I : (\forall \delta \in \Delta : i \models \delta) \Rightarrow (\exists \gamma \in \Gamma : i \models \gamma)$
$\iff \forall i \in I : i \in \bigcap_{\delta\in\Delta} M(\delta) \Rightarrow i \in \bigcup_{\gamma\in\Gamma} M(\gamma)$
$\iff \bigcap_{\delta\in\Delta} M(\delta) \subseteq \bigcup_{\gamma\in\Gamma} M(\gamma)$. ■

4.3 Properties of $prop(A)$

We present the properties of $prop(A)$ w.r.t. the properties of A.

Theorem 17 (Consistency) *$P_{prop(A)}$ is consistent in $\sqsubseteq$, $\top$, $\bot$, $\sqcap$, $\sqcup$ w.r.t. $S_{prop(A)}$ if P_A is consistent in $\sqsubseteq$, $\bot$, $\sqcup$ and complete in $\top$, $\sqcap$ w.r.t. S_A.*

Proof:

(deduction) • Let us show that $\sqsubseteq$-Axiom is valid:
$a \sqsubseteq_A b \Longrightarrow M_A(a) \subseteq M_A(b)$ $\sqsubseteq_A$ consistent
$\Longrightarrow M(a) \subseteq M(b) \Longrightarrow \forall i \in I : i \models a \Rightarrow i \models b$ Definition 13
$\Longrightarrow \forall i \in I : i \not\models a \vee i \not\models \neg b$ Semantics of negation
$\Longrightarrow$ sequent $a, \neg b, \Delta \vdash \Gamma$ is valid.

The $\top$-Axiom and $\bot$-Axiom are valid as a corollary: replace a by $\top_A$ for $\top$-Axiom, and b by $\bot_A$ for $\bot$-Axiom, along with completeness of $\top_A$ and consistency of $\bot_A$.

- Let us show that the $\sqcap$-Rule preserves validity. Assume that $a \sqcap_A b$ is defined and the sequent $a \sqcap_A b, \Delta \vdash \Gamma$ is valid, then for all $i \in I$
  - either $\exists X \in \Gamma : i \models X \Longrightarrow$ the sequent $a, b, \Delta \vdash \Gamma$ is valid
  - or $\exists L \in \Delta : i \not\models L \Longrightarrow$ the sequent $a, b, \Delta \vdash \Gamma$ is valid
  - or $i \not\models a \sqcap_A b \Longrightarrow i \notin M(a \sqcap_A b) \Longrightarrow i \notin M_A(a \sqcap_A b)$ Definition 13
    $\Longrightarrow i \notin M_A(a) \cap M_A(b)$ $\sqcap_A$ complete
    $\Longrightarrow i \notin M_A(a) \Longrightarrow i \notin M(a) \Longrightarrow i \not\models a$
    $\Longrightarrow$ the sequent $a, b, \Delta \vdash \Gamma$ is valid.
- Similarly, the $\sqcup$-Rule preserves validity. It is easy to recognize that inference rules $\neg\neg$-Rule, β-Rule, α-Rule and literal-Rule also preserve validity.

  As a consequence, every provable sequent is valid. In particular, if $\vdash X$ can be proved, it is valid, and X is a tautology. This proves that $\sqsubseteq_{prop(A)}$ is consistent.

(tautology) $\top$ is always complete by definition.

(contradiction) $\bot$ is defined and $\forall a \in A : M(a \wedge \neg a) = M(a) \cap \overline{M(a)} = \emptyset$.

(conjunction) $\sqcap$ is totally defined and $\forall f, g \in AS_{prop(A)} : M(f \sqcap g) = M(f \wedge g) = M(f) \cap M(g)$.

(disjunction) $\sqcup$ is totally defined and $\forall f, g \in AS_{prop(A)} : M(f \sqcup g) = M(f \vee g) = M(f) \cup M(g)$. ■

There is no such lemma for completeness. In fact, the logic of atoms is not necessarily total, and thus not all sequent $a_1, a_2, \Delta \vdash \Gamma$ can be interpreted as $a_1 \sqcap_A a_2, \Delta \vdash \Gamma$. So, there is a risk of incompleteness.

For instance, imagine 3 atoms a_1, a_2, b such that $M(a_1) \cap M(a_2) \subseteq M(b)$, and $M(a_i) \neq \emptyset$, $M(b) \neq I$, and $M(a_i) \not\subseteq M(b)$. If the implementation is complete, then the sequent $a_1, a_2, \neg b \vdash$ should be provable. However, if the logic of atoms is only partial, the conjunction $a_1 \sqcap_A a_2$ may not be defined, and rule $\sqcap$-Rule does not apply. In this case, the sequent would not be provable. One can build a similar example for disjunction.

5 Reducedness

5.1 Formal presentation

We define a property of an atomic logic A, which is distinct from completeness, is relative to the definedness of the logic operations, and helps in ensuring the completeness of $prop(A)$.

Definition 18 *A sequent $\Delta \vdash \Gamma$ is called* open *in $P_{prop(A)}$ iff it is the conclusion of no deduction rule, and it is not an axiom. Otherwise, it is called* closed.

An open sequent is a node of a proof tree that cannot be developed further, but is not an axiom. In short, it is a failure in a branch of a proof search.

Lemma 19 *A sequent $\Delta \vdash \Gamma$ is open according to implementation P_A iff*

- *Γ is empty,*
- *$\forall a \in \Delta : a \not\sqsubseteq_A \bot_A$ (when $\bot_A$ is defined),*
- *$\forall \neg b \in \Delta : \top_A \not\sqsubseteq_A b$ (when $\top_A$ is defined),*
- *$\forall a, \neg b \in \Delta : a \not\sqsubseteq_A b$,*
- *$\forall a \neq b \in \Delta : a \sqcap_A b$ is undefined,*
- *$\forall \neg a \neq \neg b \in \Delta : a \sqcup_A b$ is undefined.*

Proof: Case inspection of the sequent calculus defined in Table 1. ■

So, an open sequent $\Delta \vdash \Gamma$ can be characterized by a pair (A, B), where $A \subseteq AS_A$ is the set of positive literals of Δ, and $B \subseteq AS_A$ is the set of negative literals of Δ (let us recall that Γ is empty). The advantage of noting open sequents by such a pair is that they are then properly expressed in terms of the logic of atoms.

Incompleteness arises when an open sequent is valid; the proof cannot be developed further though the semantics tells the sequent is true.

Lemma 20 *An open sequent (A, B) is valid in the atom semantics S_A iff $\bigcap_{a \in A} M_A(a) \subseteq \bigcup_{b \in B} M_A(b)$.*

Proof: The open sequent (A, B) is valid
$\iff$ $A, \neg B \vdash$ is valid
$\iff$ $\bigcap_{a \in A} M(a) \cap \bigcap_{b \in B} M(\neg b) \subseteq \emptyset$ Lemma 16
$\iff$ $\bigcap_{a \in A} M_A(a) \cap \overline{\bigcup_{b \in B} M_A(b)} \subseteq \emptyset$
$\iff$ $\bigcap_{a \in A} M_A(a) \subseteq \bigcup_{b \in B} M_A(b)$. ■

Definition 21 *A* family *of open sequents $((A_i, B_i))_{i \in I}$ is valid in atom semantics S_A iff every open sequent (A_i, B_i) is valid in S_A.*

Definition 22 *An implementation P_A is* reduced *on a set F of open sequent families, w.r.t. a semantics S_A, iff every non-empty family of F is not valid.*

Theorem 23 (Completeness) *$P_{prop(A)}$ is* complete *in $\sqsubseteq$ on a subset of pairs of formulas $\Pi \subseteq AS_{prop(A)} \times AS_{prop(A)}$, w.r.t. $S_{prop(A)}$, if $\sqcap_A$ is consistent and $\sqcup_A$ is complete, and P_A is reduced on open sequent families of all $\neg f \vee g$ formula proof trees (where $(f,g) \in \Pi$) w.r.t. S_A. It is also complete in $\top$, $\bot$, $\sqcap$, $\sqcup$ w.r.t. $S_{prop(A)}$.*

Proof:

(deduction) • We first prove that the backward-chaining interpretation of the above inference system terminates for all root sequent. For this proof we need a total ordering of sequents. Every formula being either a literal L, a double negation $\neg\neg X$, a conjunction α or a disjunction β, one defines an integral measure m for every formula in $AS_{prop(A)}$ as follows: $m(\alpha) = 1 + m(\alpha_1) + m(\alpha_2)$, $m(\neg\neg X) = 1 + m(X)$, $m(\beta) = 1 + m(\beta_1) + m(\beta_2)$, and $m(L) = 1$.

This measure is extended to sequences of propositions Γ, to sequences of literals Δ, and to full sequents $\Delta \vdash \Gamma$ as follows: $m(\Gamma) = \sum_{X\in\Gamma} m(X)$, $m(\Delta) = \sum_{L\in\Delta} m(L)$, $m(\Delta \vdash \Gamma) = (m(\Gamma), m(\Delta))$.

Finally, sequents are totally ordered according to a lexicographic ordering $<$ on $\mathbb{N}^2$. We observe that for every deduction rule $\frac{Seq_1}{Seq_2}$, $m(Seq_1) < m(Seq_2)$ holds. So, every proof tree is finite. In other words, the backward-chaining procedure always terminates.

- Now, one proves that $\sqcap$-Rule preserves non-validity. Let us assume that $a \sqcap_A b, \Delta \vdash \Gamma$ is not valid . Then $\exists i \in I : (i \models a \sqcap_A b$ and $\forall L \in \Delta : i \models L$ and $\forall X \in \Gamma : i \not\models X)$
  $\Longrightarrow \exists i \in I : (i \models a$ and $i \models b$ and $\forall L \in \Delta : i \models L$ and $\forall X \in \Gamma : i \not\models X$ $\quad\sqcap_A$ consistent
  $\Longrightarrow$ the sequent $a, b, \Delta \vdash \Gamma$ is not valid.

  Similarly, the $\sqcup$-Rule preserves non-validity if P_A is complete in $\sqcup$. It is easy to check that $\neg\neg$-Rule, α-Rule, β-Rule and literal-Rule also preserve non-validity.
- Let $X = \neg f \vee g$ where (f,g) is any pair of formulas in Π. Let us assume that for every proof tree of sequent $\vdash X$, there is at least an open sequent, this is a non-empty open sequent family. As P_A is reduced on all these families, they are all not valid (Definition 22). So, in every proof tree of sequent $\vdash X$, there is an open sequent that is not valid. Now, as each rule preserves non-validity, the root sequent $\vdash X$ is not valid in all proof trees.

  The contrapositive is that if $\vdash X$ is a valid sequent, then there is a proof tree without open sequent, this is $\vdash X$ is a provable sequent. So, if $M(f) \subseteq M(g)$ (see Lemma 16) then $f \sqsubseteq g$. In other words, this proves that $\sqsubseteq_{prop(A)}$ is complete on Π.

(tautology) $\top$ is defined and $\forall a \in A : M(a \vee \neg a) = M(a) \cup \overline{M(a)} = I$.

(contradiction) $\bot$ is always complete by definition.

(conjunction) $\sqcap$ is totally defined and $\forall f, g \in AS_{prop(A)} : M(f \sqcap g) = M(f \wedge g) = M(f) \cap M(g)$.

(disjunction) $\sqcup$ is totally defined and $\forall f, g \in AS_{prop(A)} : M(f \sqcup g) = M(f \vee g) = M(f) \cup M(g)$. ■

Theorem 23 is somewhat complicated to allow the proof of completeness on a subset of $prop(A)$. In some logics, it is possible to show that every open sequent is not valid. Then every non empty open sequent family is not valid, and so, atom implementation P_A is reduced on every set of open sequent families. In such a case, we merely say that P_A is *reduced* w.r.t. S_A.

5.2 Application to $prop(atom)$

The following lemma shows that the nullary logic functor *atom* is reduced. So, the implementation of logic $prop(atom)$ is complete.

Lemma 24 *P_{atom} is reduced w.r.t. S_{atom}.*

Proof: We prove this Lemma by showing that every open sequent is not valid.
Let (A, B) be an open sequent. By Lemma 19, it follows that
$\forall a \in A, b \in B : a \not\sqsubseteq b \Longrightarrow \forall a \in A, b \in B : a \neq b \Longrightarrow A \cap B = \emptyset$.

Now, assume sequent (A, B) is valid
$\Longrightarrow \bigcap_{a \in A} M(a) \subseteq \bigcup_{b \in B} M(b)$ Lemma 20
$\Longrightarrow \forall i \in I : (\forall a \in A : i \in M(a)) \Rightarrow (\exists b \in B : i \in M(b))$
$\Longrightarrow \forall i \in I : (\forall a \in A : a \in i) \Rightarrow (\exists b \in B : b \in i)$ Definition 9
$\Longrightarrow \forall i \in I : A \subseteq i \Rightarrow i \cap B \neq \emptyset$
$\Longrightarrow A \cap B \neq \emptyset$ (take $i = A$). contradiction ■

Corollary 25 *$P_{prop(atom)}$ is totally defined, and complete and consistent w.r.t. $S_{prop(atom)}$.*

Proof: See Definition 14, and theorems 17 and 23.

5.3 Discussion

All this leads to the following methodology. Nullary logic functors are defined for tackling concrete domains like intervals and strings. They must be designed carefully, so that they are consistent and complete, and reduced. More sophisticated logics can also be built using non-nullary logic functors (e.g., see Appendix B). Then, they can be composed with logic functor *prop* in order to form a total, consistent and complete logic. The resulting logic is also reduced because any total, consistent and complete logic is trivially reduced. Furthermore, its implementation forms a lattice because totality, consistency and completeness make the operations of the implementation isomorphic to set operations on the models.

At first sight, reducedness formalizes a notion of being defined enough. So it seems that it is a coherence relation between the semantics and the implementation, and that a notion of maximaly defined implementation could be defined and useful.

Definition 26 (Maximal definedness) *An implementation $(\sqsubseteq, \sqcap, \sqcup, \top, \bot)$ is* maximally defined *w.r.t. a semantics $(I, \models)$ iff*

- $\forall f, g : \forall h : M(h) = M(f) \cap M(g) \Rightarrow f \sqcap g = h$
- $\forall f, g : \forall h : M(h) = M(f) \cup M(g) \Rightarrow f \sqcup g = h$
- $\forall h : M(h) \supseteq I \Rightarrow h \sqsupseteq \top$
- $\forall h : M(h) \subseteq \emptyset \Rightarrow h \sqsubseteq \bot$
- $\forall h : M(h) \subseteq M(g) \Rightarrow h \sqsubseteq g$

An implementation obeying this definition would be consistent, and complete, and it seems it would be reduced.

However, it is more subtle than that. Reducedness is more fundamentaly a property of the semantics itself. One can build atomic logic functors whose semantics is such that no definition of its implementation makes it reduced. In fact, the problem comes when intersection of models can be empty and no formula has an empty model. For instance, in logic *atom* no intersection of models can be empty.

We will describe more nullary reduced logic functors in Appendix A, and more n-ary logic functors Appendix B.

6 Conclusion

We propose *logic functors* to construct logics that are used very concretely in logic-based applications. This makes the development of logic-based systems safer and more efficient because the constructed logic can be *compiled* to produce a fully automatic theorem prover. We have listed a number of logic functors, but many others can be built.

6.1 Related works

Our use of the word *functor* is similar to ML's one for designating *parameterized modules* [Mac88]. However, our logic functors are very specialized though ML's are general purpose (in short, we have fixed the signature), and they carry a semantic component. Both the specialization and the semantic component allow us to express composition conditions that are out of the scope of ML's functor. We could have implemented logic functors in a programming language that offers ML-like functors, but we did not, mainly for the sake of compatibility with the rest of our application that was already written in λProlog. Yet, it could be interesting to implement a library of logic functors in these languages.

The theory of *institutions* [GB92] shares our concern for *customized logics*, and also uses the word *functor*. However, the focus and theoretical ground are different. Institutions focus on the relation between notations and semantics, whereas we focus on the relation between semantics and implementations. In fact, the implementation class $\mathbb{P}$ is necessary for us to enforce *embeddability*. We consider the notation problem in the printing and parsing operations of an implementation, but it is marginal in our work. The theory of institutions is developed using category theory, whence comes their use of the word functor; e.g., there are functors from signatures to formulas, from signatures to models, or from institutions to institutions. In fact, our logic functors correspond to parameterized institutions.

An important work which shares our motivations is LeanTap [BP95, BP96]. The authors of LeanTap have also recognized the need for embedding customized logics in applications, and the need for offering the Application Designer some means to design a correct logic subsystem. To this end, they propose a very concise style of theorem proving, which they call *lean theorem proving*, and they claim that a theorem prover written in this style is so concise that it is very easy to modify it in order to accomodate a different logic. And indeed, they have proposed a theorem prover for first-order logic, and several variants of it for modal logic, etc. Note that the first-order theorem prover is less than 20 clauses of Prolog. We think that their claim does not take into account the fact that the System Programmer and the Application Designer are really different Actors. There is no doubt that modifying their first-order theorem prover was easy for these authors, but we also think it could have been undertaken by few others. A hint for this is that it takes a full journal article to revisit and justify the first-order lean theorem prover [Fit98]. So, we think lean theorem proving is an interesting technology, and we have used it to define logic functor *prop*, but it does not actually permit the Application Designer to build a customized logic.

Our main concern is to make sure that logic functors can be composed in a way that preserves their logical properties. This led us to define technical properties that simply tell us how logic functors behave: total/partial, consistent/complete, and reduced/unreduced. This concern is complementary to the concern of actually implementing customized logics, e.g., in logical frameworks like Isabelle [Pau94], Edinburgh LF [HHP93], or Open Mechanized Reasoning Systems [GPT96], or even using a programming language. These frameworks allow users to implement a customized logic, but do not help users in proving the completeness and consistency of the resulting theorem prover. Note that one must not be left with the impression that these frameworks do not help at all. For instance, axiomatic types classes have been introduced in Isabelle [Wen97] in order to permit automatic admissibility check. Another observation is that these frameworks mostly offer environments for interactive theorem proving, which is incompatible with the objective of building fully automatic embeddable logic components. Note finally that our implementation is written in λProlog, which is sometimes considered as a logical framework.

In essence, our work is more similar to works on static program analysis toolbox (e.g., PAG [AM95]) where authors assemble known results of lattice theory to combine domain operators like product, sets of, and lists in order to build abstract domains and derive automatically a fixed-point solver for these domains. The fact that in the most favourable cases (e.g., $prop(A)$), our deduction relations form (partial) lattices is another connection with these works. However, our framework is more flexible because it permits to build lattices from domains that are not lattices. In particular, logic functor *prop* acts as a lattice completion operator on every other reduced logic. Moreover, we believe that non-lattice logics like *interval* (see Appendix A.1) can be of interest for program analysis.

Figure 1 summaries our analysis of these related works. The dark shade of System Programmer task is essentially to implement a Turing-complete programming language (recall that Actors are roles not single persons). The light shade of System Programmer task is to implement a very specific programming language for one Application Designer task. In this respect, we should have mentionned the studies on Domain Specific Languages (DSL) as related works, but we know no example of a DSL with similar aims. Note also that what remains of the task of the Application Designer is more rightly called *gluing* than *programming* when the System Programmer has gone far enough in the Application Designer's direction.

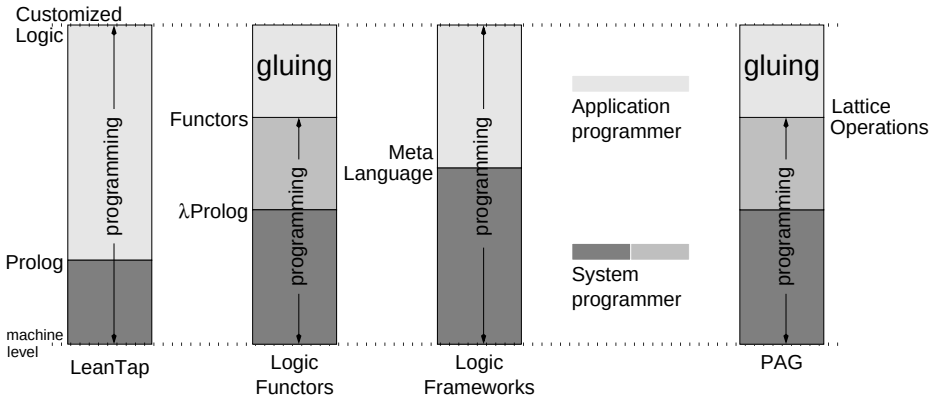

Figure 1: Several related works and the respective tasks of the System Programmer and the Application Designer

6.2 Summary of results and further works

Our logic functors specify logical "components off-the-shelf" (COTS). As such, the way they behave w.r.t. composition is defined for every logic functor.

The principle of composing logic functors has been implemented in a prototype. It comprises a logic composer that reads logic specifications such as $sum(prop(atom), prop(interv))$ (sums of propositions on atoms and propositions on intervals) and produces automatically a printer, a parser, and a theorem prover. The theorem prover is built by instantiating the theorem prover associated to each logic functor at every occurrence where it is used. The logic composer, each logic functor implementation, and the resulting implementations are written in λProlog.

This work suggests a software architecture for logic-based systems, in which the system is generic in the logic, and the logic component can be defined separately, and plugged in when needed. We have realized a prototype Logical Information System along these lines [FR00b].

Coming back to the bibliography example of the introduction, we construct a dedicated logic with logic functors defined in this article:

$$prop(aik(prop(sum(atom, valattr(sum(interv, string)))))).$$

According to results of this article, the composition of these logic functors is such that the generated implementation is total, bounded, and consistent and complete in all five operations of the implementation. It allows to build descriptions and queries such as

```
descr(entry1) =
    [author:  is "Kipling" ∧ title:  is "The Jungle Book" ∧ paper-back ∧
    publisher:  is "Penguin" ∧ year:  1985],

query = title:  contains "Jungle" ∧ year:  1950..  ∧ (paper-back ∨ hard-cover).
```

Note that entry_1 is a possible answer to the query because

$$\texttt{descr(entry}_1\texttt{)} \sqsubseteq_{prop(aik(prop(sum(atom,valattr(sum(interv,string))))))} \texttt{query},$$

which is automatically proved using the generated implementation.

We plan to validate the use of logic functors within the Logical Information System. This application will also motivate the study of other logic functors like, e.g., modalities or taxonomies, since it happens that they are useful to make queries and answers more precise and compact.

Another possible continuation of this work is to vary the type of logic functors and their composition. In the present situation, all logic functors have type $\mathbb{L}^n \to \mathbb{L}$. It means that the only possibility is to choose the atomic formulas of a logic. However, one may wish to act on the interpretation domain, or on the quantification domain. So, one may want to have a class $\mathbb{D}$ of domains, and logic functors that take a domain as argument, e.g., $\mathbb{D} \to \mathbb{L}$. At a similar level, one may wish to act on the interface, either to pass new operations through it, e.g., negation or quantification, or to pass new structures, e.g., specific sets of models. The extension to

higher-order logic functors, e.g., $(\mathbb{L} \to \mathbb{L}) \to \mathbb{L}$, would make it possible to define a fixed-point logic functor, μ, with which we could construct a logic as $L = \mu F$ where F is a unary logic functor.

Finally, we plan to develop new logic functors for the purpose of program analysis. For instance, in [RBM99, RB01] we have proposed to combine the domain of boolean values with the domain of types to form a logic of positive functions that extends the well-known domain *Pos* [CSS99]. We called this *typed analysis*. The neat result is to compute at the same time the properties of groundness and of properness [O'K90]. Our project is to define logic functors for every type constructors, and to combine them according to the types inferred/checked in the programs (e.g., $list(list(bool))$, where *bool* is simply $\{true, false\}$). This will make it possible to redo what we have done on typed analysis, but also to explore new static analysis domains by combining the logic functors for types with other nullary logic functors than *bool*.

References

[AM95] M. Alt and F. Martin. Generation of efficient interprocedural analyzers with PAG. In *Static Analysis Symp.*, LNCS 983, pages 33–50, 1995.

[AMSS98] T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Two classes of boolean functions for dependency analysis. *Science of Computer Programming*, 31:3–45, 1998.

[BP95] B. Beckert and J. Posegga. leanT^AP: Lean, tableau-based deduction. *J. Automated Reasoning*, 11(1):43–81, 1995.

[BP96] B. Beckert and J. Posegga. Logic programming as a basis for lean automated deduction. *J. Logic Programming*, 28(3):231–236, 1996.

[CSS99] M. Codish, H. Søndergaard, and P.J. Stuckey. Sharing and groundness dependencies in logic programs. *ACM TOPLAS*, 21(5):948–976, 1999.

[DLNS96] F. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. Reasoning in description logics. In G. Brewka, editor, *Principles of Knowledge Representation and Reasoning*, Studies in Logic, Language and Information, pages 193–238. CLSI Publications, 1996.

[Fit98] M. Fitting. leanTAP revisited. *Journal of Logic and Computation*, 8(1):33–47, February 1998.

[FR00a] S. Ferré and O. Ridoux. A file system based on concept analysis. In Y. Sagiv, editor, *Int. Conf. Rules and Objects in Databases*, LNCS 1861, pages 1033–1047. Springer, 2000.

[FR00b] S. Ferré and O. Ridoux. A logical generalization of formal concept analysis. In G. Mineau and B. Ganter, editors, *Int. Conf. Conceptual Structures*, LNCS 1867, pages 371–384. Springer, 2000.

[FR01] S. Ferré and O. Ridoux. Searching for objects and properties with logical concept analysis. In H. S. Delugach and G. Stumme, editors, *Int. Conf. Conceptual Structures*, LNCS 2120, pages 187–201. Springer, 2001.

[GB92] J.A. Goguen and R.M. Burstall. Institutions: Abstract model theory for specification and programming. *J. ACM*, 39(1):95–146, 1992.

[GPT96] F. Giunchiglia, P. Pecchiari, and C. Talcott. Reasoning theories - towards an architecture for open mechanized reasoning systems. In F. Baader and K. U. Schulz, editors, *1st Int. Workshop: Frontiers of Combining Systems*, pages 157–174. Kluwer Academic Publishers, March 1996.

[GW99] B. Ganter and R. Wille. *Formal Concept Analysis — Mathematical Foundations.* Springer, 1999.

[HHP93] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *JACM*, 40(1):143–184, January 1993.

[IB96] V. Issarny and Ch. Bidan. Aster: A framework for sound customization of distributed runtime systems. In *16th Int. Conf. Distributed Computing Systems*, 1996.

[Lev90] H. Levesque. All I know: a study in autoepistemic logic. *Artificial Intelligence*, 42(2), March 1990.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Symbolic computation — Artificial Intelligence. Springer, Berlin, 1987.

[Mac88] D.B. MacQueen. An implementation of Standard ML modules. In *LISP and Functional Programming*, pages 212–223, 1988.

[MS98] K. Marriott and P.J. Stuckey. *Programming with Constraints: An Introduction.* The MIT Press, 1998.

[O'K90] R.A. O'Keefe. *The Craft of Prolog.* MIT Press, 1990.

[Pau94] L. C. Paulson. *Isabelle: a generic theorem prover.* LNCS 828. Springer, New York, NY, USA, 1994.

[Poo88] D. Poole. Representing knowledge for logic-based diagnosis. In *Int. Conf. Fifth Generation Computer Systems*, pages 1282–1290. Springer, 1988.

[RB01] O. Ridoux and P. Boizumault. Typed static analysis: Application to the groundness analysis of typed prolog. *Journal of Functional and Logic Programming*, 2001(4), 2001.

[RBM99] O. Ridoux, P. Boizumault, and F. Malésieux. Typed static analysis: Application to groundness analysis of Prolog and λProlog. In *Fuji Int. Symp. Functional and Logic Programming*, pages 267–283, 1999.

[SFRW98] M. Sagiv, N. Francez, M. Rodeh, and R. Wilhelm. A logic-based approach to program flow analysis. *Acta Informatica*, 35(6):457–504, June 1998.

[SM83] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, 1983.

[vRCL98] C.J. van Rijsbergen, F. Crestani, and M. Lalmas, editors. *Information Retrieval: Uncertainty and Logics. Advanced models for the representation and retrieval of information.* Kluwer Academic Publishing, Dordrecht, NL, 1998.

[Wen97] M. Wenzel. Type classes and overloading in higher-order logic. In E.L. Gunter and A. Felty, editors, *Theorem proving in higher-order logics, LNCS 1275*, pages 307–322. Springer-Verlag, 1997.

A More nullary reduced logic functors

Ad-hoc logics are often designed for representing concrete observations on a domain. They serve as a language to write atomic formulas. In the bibliographical application atomic formulas could be *between* 1900 *and* 1910 or *contains* `"Kipling"`. In order to serve as arguments to the logic functor *prop* (or other similar boolean logic functors if available), they must be equipped with a "natural" conjunction and "natural" disjunction, i.e., they must be consistent and complete (cf. Definition 6). However, these operations can usually be only partially defined. For instance, the "natural" disjunction of two intervals is only defined if the intervals overlap.

By definition, applying the logic functor *prop* to such an atomic logic produces a logic that is always total and bounded (Definition 14). It also provides a consistent and complete implementation if the atom logic also has a consistent, complete, and reduced implementation (Lemmas 17 and 23).

So, for every nullary logic functor presented in this section, we prove that its implementation is consistent, complete, and reduced w.r.t. its semantics.

A.1 Intervals

Intervals are often used to express incomplete knowledge either in the data-base or in the queries. For instance, in the bibliographical application, `year: 1900..1910` may express an interval of dates between 1900 and 1910. We can also express open intervals such that `year: ..1910`, which means "before 1910".

Definition 27 $AS_{interval} = \{[x, y] \mid x, y \in \mathbb{R} \uplus \{-, +\}\}$.

The symbol $-$ denotes the negative infinity (smaller than any real number), and the symbol $+$ denotes the positive infinity (greater than any real number). So, $\mathbb{R} \uplus \{-, +\}$ is a totally ordered set bounded by $-$ and $+$.

Definition 28 $S_{interval}$ *is* $(I, \models)$ *where* $I = \mathbb{R}$ *and* $i \models [x, y] \iff x \leq i \leq y$.

For simplifying further proofs it should be noted that models of interval formulas are intervals of the real numbers. In particular, $M_{interval}([+,-]) = \emptyset$.

Property 29 *$M_{interval}(f) =$ interval f (recall that formulas are only syntax, so "interval f" is the interval ordinarily written f).*

Proof: $M_{interval}([x,y]) = \{i \mid x \leq i \wedge i \leq y\} = \text{interval } f$ ■

Definition 30 *$P_{interval}$ is $(\sqsubseteq, \sqcap, \sqcup, \top, \bot)$*
where for every $[x_1,y_1],[x_2,y_2] \in AS_{interval}$

- $[x_1,y_1] \sqsubseteq [x_2,y_2]$ *iff* $x_2 \leq x_1$ *and* $y_1 \leq y_2$,
- $[x_1,y_1] \sqcap [x_2,y_2] = [max(x_1,x_2), min(y_1,y_2)]$,
- $[x_1,y_1] \sqcup [x_2,y_2] = \begin{cases} [min(x_1,x_2), max(y_1,y_2)] & \text{if } x_2 \leq y_1 \text{ and } x_1 \leq y_2 \\ undef & \text{otherwise} \end{cases}$,
- $\top = [-,+]$,
- $\bot = [+,-]$.

Note that conjunction is defined for every pair of intervals, but disjunction is only defined for pairs of overlapping intervals.

Theorem 31 *$P_{interval}$ is consistent and complete in $\sqsubseteq$, $\top$, $\bot$, $\sqcap$, $\sqcup$ w.r.t. $S_{interval}$.*

Proof: Let $[x_1,y_1],[x_2,y_2] \in AS_{interval}$.

(deduction) $[x_1,y_1] \sqsubseteq [x_2,y_2]$
$\iff x_2 \leq x_1$ and $y_1 \leq y_2$ — Definition 30
$\iff M([x_1,y_1]) \subseteq M([x_2,y_2])$ — Property 29

(conjunction) $\sqcap$ is always defined and
$M([x_1,y_1] \sqcap [x_2,y_2]) = M([max(x_1,x_2), min(y_1,y_2)])$ — Definition 30
$= [max(x_1,x_2), min(y_1,y_2)]$ — Property 29
$= [x_1, y-1] \cap [x_2,y_2] = M([x_1,y_1]) \cap M([x_2,y_2])$ — Property 29

(disjunction) If $[x_1,y_1] \sqcup [x_2,y_2]$ is defined, then $x_2 \leq y_1$ and $x_1 \leq y_2$ and
$M([x_1,y_1] \sqcup [x_2,y_2]) = M([min(x_1,x_2), max(y_1,y_2)])$ — Definition 30
$= [min(x_1,x_2), max(y_1,y_2)]$ — Property 29
$= [x_1,y_1] \cup [x_2,y_2] = M([x_1,y_1]) \cup M([x_2,y_2])$ — Property 29

(tautology) $\top$ is defined by $[-,+]$, and $M([-,+]) = \{i \in I \mid - \leq i \leq +\} = I$.

(contradiction) $\bot$ is defined by $[+,-]$, and $M([+,-]) = \{i \in I \mid + \leq i \leq -\} = \emptyset$. ■

$P_{interval}$ is partial in disjunction, but it is consistent and complete. Furthermore, the following lemma shows that it is reduced, and so, it can serve as argument of the logic functor *prop*.

Lemma 32 *$P_{interval}$ is reduced w.r.t. $S_{interval}$.*

Proof: We show that every open sequent is not valid.
Let (A,B) be an open sequent, and let $B = \{[x_1,y_1],\ldots,[x_n,y_n]\}$, then (1), (2) and (3) hold

(1) $\forall 1 \leq i < j \leq n : [x_i,y_i] \sqcup [x_j,y_j]$ is undefined — Lemma 19
$\implies \forall 1 \leq i < j \leq n : y_i < x_j$ or $y_j < x_i$ — Definition 30
$\implies \forall 1 \leq i < j \leq n : M([x_i,y_i]) \cap M([x_j,y_j]) = \emptyset$ (i.e., the intervals do not overlap),

(2) $\forall [x_i, y_i] \in B : [-,+] \not\sqsubseteq [x_i, y_i]$ Lemma 19
$\Longrightarrow \forall [x_i, y_i] \in B : x_i \neq -$ or $y_i \neq +$ Definition 30

(3) $\forall [x, y] \in A, [x_i, y_i] \in B : [x, y] \not\sqsubseteq [x_i, y_i]$ Lemma 19
$\forall [x, y] \in A, [x_i, y_i] \in B : x < x_i$ or $y > y_i$ Definition 30

As the conjunction is always defined, we have

- either $A = \emptyset \Longrightarrow \bigcap_{[x,y]\in A} M([x, y]) = I$
  but (2) and the fact that intervals in B do not overlap
  $\Longrightarrow \bigcup_{[x_i,y_i]\in B} M([x_i, y_i]) \neq \mathcal{I}$
  $\Longrightarrow \bigcap_{[x,y]\in A} M([x, y]) \not\subseteq \bigcup_{[x_i,y_i]\in B} M([x_i, y_i])$,
- or $A = \{[x, y]\}$: (2) and (3) and the fact that intervals in B do not overlap
  $\Longrightarrow \exists z \in M([x, y]) : \forall 1 \leq i \leq n : z \notin M([x_i, y_i])$
  $\Longrightarrow \exists z \in I : z \in \bigcap_{a\in A} M(a)$ and $z \notin \bigcup_{b\in B} M(b)$
  $\Longrightarrow \bigcap_{a\in A} M(a) \not\subseteq \bigcup_{b\in B} M(b)$.

Hence, the open sequent (A, B) is not valid. ■

A.2 Strings

Often, descriptions and queries contain string specifications, like *is*, *start with* and *contains*. Moreover, these specification can be ordered by an entailment relation. For instance, the atom *is* `"The Jungle Book"` entails *ends with* `"Jungle Book"`, which entails *contains* `"Jungle"`.

Definition 33 *$AS_{string} = {}^{\wedge 0|1}\Sigma^*\$^{0|1} \uplus \{\#\}$, where Σ is some (infinite) signature such that $\{\hat{}, \$, \#\} \cap \Sigma = \emptyset$.*

The optional symbol $\hat{}$ denotes the beginning of a string; it is the left bound of a string. The optional symbol \$ denotes the end of a string; it is the right bound of a string. So, "*contains s*" is written s, "*starts with s*" is written $\hat{}s$, and *is s* is written $\hat{}s\$$. The symbol # denotes the empty language (matched by no string).

Definition 34 *S_{string} is $(I, \models)$ where $I = \hat{}\Sigma^*\$$ and $i \models f \iff i = \alpha f \beta$.*

So, models are made of complete strings. More precisely,

Property 35 *$M_{string}(f)$ is $\hat{}\Sigma^* f \Sigma^*\$$ if f is not bounded, $f\Sigma^*\$$ if f is only left-bounded, $\hat{}\Sigma^* f$ if f is only right-bounded, and f if f is bounded.*

Proof: Inspection of the four cases. ■

Note also that only formula # has an empty model.

Definition 36 *P_{string} is $(\sqsubseteq, \sqcap, \sqcup, \top, \bot)$ where for every $f, g \in AS_{string}$*

- *$f \sqsubseteq g$ iff $f = \alpha g \beta$,*
- $$f \sqcap g = \begin{cases} f & \text{if } f \sqsubseteq g \\ g & \text{if } g \sqsubseteq f \\ \# & \text{if } f \not\sqsubseteq g \text{ and } g \not\sqsubseteq f \text{ and both } f \text{ and } g \text{ are} \\ & \text{either left-bounded or right-bounded, or one of them is bounded} \\ undef & \text{otherwise} \end{cases}$$
- *$\sqcup$ is undefined,*
- *$\top = \epsilon$,*
- *$\bot = \#$.*

Theorem 37 *P_{string} is consistent and complete in $\sqsubseteq$, $\top$, $\bot$, $\sqcap$, and $\sqcup$ w.r.t. S_{string}.*

Proof: Let $f, g \in AS_{string}$.

(deduction) $f \sqsubseteq g$
$\iff f = \alpha g \beta$ Definition 36
$\iff \forall i \in I : i = \alpha' f \beta' \Rightarrow i = \alpha' \alpha g \beta \beta'$
$\iff M(f) \subseteq M(g)$. Definition 34

(conjunction) if $f \sqcap g$ is defined, then

- either $f \sqsubseteq g \Longrightarrow M(f) \subseteq M(g)$ $\sqsubseteq$ consistent
  hence $M(f \sqcap g) = M(f) = M(f) \cap M(g)$,
- or $g \sqsubseteq f \Longrightarrow M(g) \subseteq M(f)$ $\sqsubseteq$ consistent
  hence $M(f \sqcap g) = M(g) = M(f) \cap M(g)$,
- or $f \not\sqsubseteq g$ and $g \not\sqsubseteq f$ and both f and g are either left-bounded or right-bounded, or one of them is bounded
  $\Longrightarrow M(f) \cap M(g) = \emptyset$ (by inspection of each case) $\sqsubseteq$ complete
  $\Longrightarrow M(f \sqcap g) = M(\#) = \emptyset = M(f) \cap M(g)$.

(disjunction) $\sqcup$ is never defined.

(tautology) $\top$ is defined as ϵ, and $M(\epsilon) = \{i \in I \mid i = \alpha\epsilon\beta\} = I$.

(contradiction) $\bot$ is defined as $\#$, and $M(\#) = \emptyset$. Definition 34 ■

P_{string} is partial, but it is consistent and complete. Furthermore, the following lemma shows that it is reduced, and so, the composition *prop*(*string*) is also consistent and complete.

Lemma 38 *P_{string} is reduced w.r.t. S_{string}.*

Proof: We show that every open sequent is not valid.
Let (A, B) be an open sequent, then $\forall a \in A : a \not\sqsubseteq \#$ and $\forall b \in B : \epsilon \not\sqsubseteq b$ and $\forall a \in A, b \in B : a \not\sqsubseteq b$
and $\forall a_1 \neq a_2 \in A : a_1 \sqcap a_2$ is undefined Definition 18
$\Longrightarrow \# \notin A$ and $\epsilon \notin B$ and $\forall a \in A, b \in B : a \neq \alpha b \beta$ and $\forall a_1 \neq a_2 \in A : a_1 \neq \epsilon, a_2 \neq \epsilon$ Definition 36
$\Longrightarrow A$ can take 3 forms, and we prove for each of them that (A, B) is not valid, that is $\bigcap_{a \in A} M(a) \not\subseteq \bigcup_{b \in B} M(b)$:

1. $A = \{\epsilon\}$: let $c \in \Sigma$ a character that does not appear in B and $i = \hat{}c\$$, then
   $i \in \bigcap_{a \in A} M(a) = M(\epsilon) = I$ and $i \notin \bigcup_{b \in B} M(b)$ (because $\epsilon \notin B$),
2. $A = \{\hat{}\alpha\$\}$: let $i = \hat{}\alpha\$$, then $\bigcap_{a \in A} M(a) = M(\hat{}\alpha\$) = \{i\}$
   now suppose that $i \in \bigcup_{b \in B} M(b)$
   $\Longrightarrow \exists b \in B : \{i\} \subseteq M(b) \Longrightarrow \exists b \in B : M(\hat{}\alpha\$) \subseteq M(b)$
   $\Longrightarrow \exists a \in A, b \in B : a \sqsubseteq b$ with completeness of $\sqsubseteq$, which leads to a contradiction,
3. $A = \{\hat{}\alpha_1, \alpha_2, \ldots, \alpha_{n-1}, \alpha_n\$\}$: let $c \in \Sigma$ a character that does not appear in B
   and $i = \hat{}\alpha_1 c \alpha_2 c \ldots c \alpha_{n-1} c \alpha_n\$$, then $i \in \bigcap_{a \in A} M(a)$.
   Now suppose that $i \in \bigcup_{b \in B} M(b) \Longrightarrow \exists b \in B : i \in M(b)$
   $\Longrightarrow \exists a \in A, b \in B : a = \alpha b \beta$ because c does not appear in B
   $\Longrightarrow \exists a \in A, b \in B : a \sqsubseteq b$, which leads to a contradiction.

Hence, the open sequent (A, B) is not valid. ■

B More n-ary logic functors

We present in this appendix more n-ary logic functors. Some of them produce reduced logics that are not necessarily total. In this case, partialness is not a problem, since it is enough to wrap them in logic functor *prop*. A few others produce logics that are not reduced, but that are total. They are useful anyway, but only as the outermost logic functor of a composition. Using them in, say, the logic functor *prop* would produce an incomplete logic, which is seldom desired, and difficult to repair.

In each case, we present the syntax, the semantics, the implementation and results about consistency and completeness, and reducedness.

B.1 Complete knowledge

The logic "All I Know" [Lev90] represents knowledge judgements in a modal way, instead of by an extra-logical rule as with closed world assumption. Note also that it is a monotonous logic.

Definition 39 *AS_{aik} is the optional wrapping of the syntax of some logic by the All I Know modality. We will use square brackets [and] as a concrete syntax.*

The syntax of *aik* operates on *descriptions* expressed as logical formulas. For any description f_d, $[f_d]$ represents its closure in a complete description (f_d is all that is true), f_d represents a positive fact, and if *aik* is composed with *prop*, $\neg f_d$ represents a negative fact.

Definition 40 *S_{aik} is $(I_d, \models_d) \mapsto (I, \models)$ such that*

$$I = \mathcal{P}(I_d) \setminus \{\emptyset\} \text{ and } \begin{cases} i \models f_d & \text{iff } i \subseteq M_d(f_d) \\ i \models [f_d] & \text{iff } i = M_d(f_d) \end{cases}$$

Definition 41 *P_{aik} is $(\sqsubseteq_d, \sqcap_d, \sqcup_d, \top_d, \bot_d) \mapsto (\sqsubseteq, \sqcap, \sqcup, \top, \bot)$ such that*

- *the deduction $\sqsubseteq$ is defined according to Table 2*
- $f \sqcap g = \begin{cases} f_d \sqcap_d g_d & \text{if } f = f_d \text{ and } g = g_d \\ f & \text{if } f \sqsubseteq g \\ g & \text{if } g \sqsubseteq f \\ \bot & \text{if } f = [f_d] \not\sqsubseteq g \\ \bot & \text{if } g = [g_d] \not\sqsubseteq f \end{cases}$
- $f \sqcup g = undef$
- $\top = \top_d$
- $\bot = \bot_d$.

$\sqsubseteq$	g_d	$[g_d]$
f_d	$f_d \sqsubseteq_d g_d$	$f_d \sqsubseteq_d \bot_d$
$[f_d]$	$f_d \sqsubseteq_d g_d$	$f_d \equiv_d g_d$ or $f_d \sqsubseteq_d \bot_d$

Table 2: Definition of logical deduction in logic functor *aik*.

Theorem 42 *P_{aik} has the following completeness and consistence properties:*

The tautology, $\top$, is defined (resp. complete) if the description tautology, $\top_d$, is defined (resp. complete). The case of the contradiction, $\bot$, is similar w.r.t. to consistency.

Conjunction $\sqcap$ is consistent and complete if the description conjunction $\sqcap_d$ is consistent and complete. Disjunction $\sqcup$ is always consistent and complete because it is undefined.

The deduction $\sqsubseteq$ is consistent and complete if the description deduction $\sqsubseteq_d$ is consistent and complete, and no formula in AS_{L_d} has only 1 model (which is usually the case).

Proof:

(tautology) Assume $\top_d$ is defined and complete, $i \in I$ and $i_d \in i$, then $i_d \models_d \top_d \Longrightarrow i \models \top_d \Longrightarrow M(\top_d) = I$

(contradiction) Assume $\bot_d$ is defined and consistent, $i_d \in I_d$, if $i_d \in I_d$, then $i_d \not\models_d \bot_d$ (1)
Assume $i \models \bot$ then $i \models \bot_d \Rightarrow \forall i_d \in i : i_d \models_d \bot_d$
$\Rightarrow \exists i_d \in i : i_d \models_d \bot_d$, because $\emptyset \notin I$, contradiction with (1)
so $M(\bot) = \emptyset$

(conjunction) Assume $f \sqcap g$ is defined, then either

- $f = f_d$ and $g = g_d$
  $\Longrightarrow i \models f \sqcap g \Longrightarrow i \models f_d \sqcap_d g_d \Longrightarrow \forall i_d \in i : i_d \models_d f_d \sqcap_d g_d$
  $\Longrightarrow \forall i_d \in i : i_d \models_d f_d$ and $i_d \models_d g_d$ — $\sqcap_d$ consistent and complete
  $\Longrightarrow \forall i_d \in i : i_d \models_d f_d$ and $\forall i_d \in i : i_d \models_d g_d$
  $\Longrightarrow i \models f_d$ and $i \models g_d \Longrightarrow i \models f$ and $i \models g$
  hence, $M(f \sqcap g) = M(f) \cap M(g)$
- $f \sqsubseteq g$ (or symetrically $g \sqsubseteq f$)
  $\Longrightarrow M(f) \subseteq M(g) \Longrightarrow M(f) \cap M(g) = M(f) = M(f \cap g)$
- $f = [f_d]$ and $f \not\sqsubseteq g$ (or symetrically $g = [g_d]$ and $g \not\sqsubseteq f$)
  $\Longrightarrow M(f) = \{M_d(f_d)\}$ and $M(f) \not\subseteq M(g)$
  $\Longrightarrow M_d(f_d) \notin M(g) \Longrightarrow M(f) \cap M(g) = \emptyset = M(\bot)$

(deduction) Case analysis of the four cases of table 2:

$(f_d \sqsubseteq g_d) \iff M_d(f_d) \subseteq M_d(g_d)$ — $\sqsubseteq_d$ is consistent and complete
$\iff M_d(f_d) \subseteq M_d(g_d)$
$\iff \mathcal{P}(M_d(f_d)) \subseteq \mathcal{P}(M_d(f_d))$
$\iff \mathcal{P}(M_d(f_d)) \setminus \{\emptyset\} \subseteq \mathcal{P}(M_d(f_d)) \setminus \{\emptyset\}$
$\iff M(f_d) \subseteq M(g_d)$

$(f_d \sqsubseteq [g_d]) \iff M(f_d) \sqsubseteq M([g_d])$
$\iff \mathcal{P}(M_d(f_d)) \setminus \{\emptyset\} \subseteq \{M_d([g_d])\} \setminus \{\emptyset\}$
$\iff M_d(f_d) = \emptyset$, — because L_d has no formula with a single model in I_d,
$\iff f_d \sqsubseteq_p \bot_d$

$([f_d] \sqsubseteq g_d) \iff M([f_d]) \subseteq M(g_d)$
$\iff \{M_d(f_d)\} \setminus \{\emptyset\} \subseteq \mathcal{P}(M_d(g_d)) \setminus \{\emptyset\}$
$\iff M_d(f_d) \in \mathcal{P}(M_d(g_d))\} \setminus \{\emptyset\}$ if $M_d(f_d) \neq \emptyset$,
or $M_d(f_d) = \emptyset$
$\iff M_d(f_d) \in \mathcal{P}(M_d(g_d))\} \setminus \{\emptyset\}$
$\iff M_d(f_d) \subseteq M_d(g_d)$
$\iff f_d \sqsubseteq_d g_d$

$([f_d] \sqsubseteq [g_d]) \iff M([f_d]) \subseteq M([g_d])$
$\iff \mathcal{P}(M_d(f_d)) \setminus \{\emptyset\} \subseteq \mathcal{P}(M_d(g_d)) \setminus \{\emptyset\}$
$\iff M_d(f_d) = \emptyset$ or $M_d(f_d) = M_d(g_d)$
$\iff f_d \sqsubseteq_d \bot_d$ or $f_d \equiv_d g_d$ ■

Lemma 43 *$P_{aik(L_d)}$ is reduced for open sequent families included in $S = \{(A, B) \mid A \subseteq AS_{aik(L_d)}, B \subseteq L_d\}$, if $\sqsubseteq_d$ is consistent and complete, $\top_d$ is defined and complete, $\bot_d$ is defined and consistent, and $\sqcap_d$ is totally defined.*

Proof: We prove this lemma by showing that every open sequent $(A, B) \in S$ is not valid. If $\sqcap_d$ is totally defined, A is a singleton $\{a\}$ or the empty set $\emptyset$.
In the case where $A = \emptyset$, $\bigcap_{a \in A} M(a) = I = M(\top)$, if $\top_d$ is defined and complete. So, in any case, $A = \{a\}$ where $a \in AS_{aik(L_d)}$, that is it exists $a_d \in AS_d$ such that either $a = a_d$, or $a = [a_d]$.

First, suppose $M_d(a_d) = \emptyset$,
then $M(a_d) = \emptyset$ and $M([a_d]) = \emptyset$ — Definition 40
$\Longrightarrow M(a) = \emptyset \Longrightarrow a \sqsubseteq \bot$, if $\bot_d$ is defined and consistent, and $\sqsubseteq$ is complete
$\rightarrow$ contradicts that (A, B) is an open sequent. So, $M_d(a_d) \neq \emptyset$.

Second, suppose $M_d(a_d) \in \bigcup_{b \in B} M(b)$,
then $M_d(a_d) \in \bigcup_{b \in B} (\mathcal{P}(M_d(b)) \setminus \emptyset)$ — because $B \subseteq L_d$
$\Longrightarrow \exists b \in B : M_d(a_d) \subseteq M_d(b)$
$\Longrightarrow \exists b \in B : a_d \sqsubseteq_d b$ — if $\sqsubseteq_d$ is complete
$\Longrightarrow \exists b \in B : a_d \sqsubseteq b$ and $[a_d] \sqsubseteq b$ — Definition 41
$\Longrightarrow \exists b \in B : a \sqsubseteq b$, which contradicts that (A, B) is an open sequent.
So, $M_d(a_d) \notin \bigcup_{b \in B} M(b)$.

Hence, $\{M_d(a_d)\} \not\subseteq \bigcup_{b\in B} M(b)$ and $\mathcal{P}(M_d(a_d)) \setminus \emptyset \not\subseteq \bigcup_{b\in B} M(b)$
$\Longrightarrow M(a) \not\subseteq \bigcup_{b\in B} M(b)$ Definition 41
$\Longrightarrow \bigcap_{a\in A} M(a) \not\subseteq \bigcup_{b\in B} M(b) \Longrightarrow (A, B)$ is not valid. ■

To summarize, logic functor *prop* can be applied to a logic $aik(L_d)$ if $\top_d$ is defined and complete, $\bot_d$ is defined and consistent, $\sqsubseteq_d$ and $\sqcap_d$ are consistent, complete, and total for $\sqcap_d$. In this case, $\sqsubseteq_{prop(aik(L_d))}$ is consistent and complete when the right argument has no closed formula $[g_d]$ among its atoms. This is satisfying when used in a logical information system, because closed formulas appear only in object descriptions, and so as left argument of deduction $\sqsubseteq$.

B.2 Valued attributes

Valued attributes are useful for attaching several properties to objects. For instance, a bibliographical reference has several attributes, like `author`, `year`, or `title`, each of which has a value. We want to express some conditions on these values, and for this, we consider a logic L_V, whose semantics is in fact the domain of values for the attributes. Attributes themselves are taken in an infinite set *Attr* of distinct symbols. Thus, a logic of valued attributes is built with the logic functor *valattr*, whose argument is the logic of values, and that is defined as follows:

Definition 44 (Syntax) *Given a set Attr of attribute name, $AS_{valattr}$ is the product of Attr with the syntax of some logic:*
$AS_{valattr(L)} = \{a : f \mid f \in L \wedge a \in Attr\}$

Definition 45 (Semantics) *$S_{valattr}$ is $(I_V, \models_V) \mapsto (I, \models)$ such that $I = A \to I_V \uplus \{undef\}$ and $i \models a : v$ iff $i(a) \neq undef$ and $i(a) \models_V v$.*

Definition 46 (Implementation) *$P_{valattr}$ is*
$(\sqsubseteq_V, \sqcap_V, \sqcup_V, \top_V, \bot_V) \mapsto (\sqsubseteq, \sqcap, \sqcup, \top, \bot)$ such that

- *$a : v \sqsubseteq b : w$ iff $v \sqsubseteq_V w$ and $(a = b$ or $v \sqsubseteq_V \bot_V)$,*
- $a : v \sqcap b : w = \begin{cases} a : (v \sqcap_V w) & \text{if } a = b \\ a : \bot_V & \text{if } v \sqsubseteq_V \bot_V \text{ or } w \sqsubseteq_V \bot_V \\ undef & \text{otherwise} \end{cases}$,
- $a : v \sqcup b : w = \begin{cases} a : (v \sqcup_V w) & \text{if } a = b \\ a : v & \text{if } w \sqsubseteq_V \bot_V \\ b : w & \text{if } v \sqsubseteq_V \bot_V \\ undef & \text{otherwise} \end{cases}$,
- *$\top$ and $\bot$ are undefined.*

Theorem 47 *$P_{valattr(V)}$ is consistent and complete in $\sqsubseteq$, $\top$, $\bot$, $\sqcap$, $\sqcup$ w.r.t. $S_{valattr(V)}$ if P_V is consistent and complete w.r.t. S_V.*

Proof: Let $a : v, b : w \in AS_{valattr(L_V)}$.

(deduction) If $a : v \sqsubseteq b : w$ is true, then $v \sqsubseteq_V w$ and $(a = b$ or $v \sqsubseteq_V \bot_V)$ Definition 46
$\Longrightarrow M_V(v) \subseteq M_V(w)$ $\sqsubseteq_V$ consistent
$\Longrightarrow \forall i_V \in I_V : i_V \in M_V(v) \Rightarrow i_V \in M_V(w)$
$\Longrightarrow \forall x \in Attr : \forall i \in I : i(x) \in M_V(v) \Rightarrow i(x) \in M_V(w)$.
We have

- either $a = b \Longrightarrow \forall i \in I : i(a) \in M_V(v) \Rightarrow i(b) \in M_V(w)$
  $\Longrightarrow M(a : v) \subseteq M(b : w)$,
- or $v \sqsubseteq_V \bot_V \Longrightarrow M_V(v) = \emptyset \Longrightarrow M(a : v) = \emptyset \Longrightarrow M(a : v) \subseteq M(b : w)$.

If $a : v \sqsubseteq b : w$ is false, then

- either $v \not\sqsubseteq_V w \Longrightarrow M_V(v) \not\subseteq M_V(w)$ — $\sqsubseteq_V$ complete
  $\Longrightarrow \exists i_V \in I_V : i_V \in M_V(v)$ and $i_V \notin M_V(w)$
  $\Longrightarrow \exists i \in I : i(a) \in M_V(v)$ and $i(b) \notin M_V(w)$ (i is defined such that $i(a) = i(b) = i_V$)
  $\Longrightarrow M(a : v) \not\subseteq M(b : w)$ — Definition 45
- or $a \neq b$ and $v \not\sqsubseteq_V \bot_V$
  $\Longrightarrow M_V(v) \neq \emptyset$ — $\sqsubseteq_V$ complete
  $\Longrightarrow \exists i_v \in I_V : i_v \in M_V(v)$
  $\Longrightarrow \exists i \in I : i(a) \in M_V(v)$ and $i(b) \notin M_V(w)$ (i is defined such that $i(a) = i_v$ and $i(b) = undef$)
  $\Longrightarrow \exists i \in I : i(a) \in M_V(v)$ and $i(b) \notin M_V(w)$
  $\Longrightarrow M(a : v) \not\subseteq M(b : w)$ — Definition 45

(tautology) true because $\top$ is undefined.

(contradiction) true because $\bot$ is undefined.

(conjunction) If $a : v \sqcap b : w$ is defined, then

- either $a = b$ (and $v \sqcap_V w$ is defined):
  $i \in M(a : v \sqcap b : w) \iff i \in M(a : (v \sqcap_V w))$
  $\iff i(a) \in M_V(v \sqcap_V w)$ — Definition 45
  $\iff i(a) \in M_V(v) \cap M_V(w)$ — $\sqcap_V$ consistent and complete
  $\iff i(a) \in M_V(v)$ and $i(b) \in M_V(w)$ — $a = b$
  $\iff i \in M(a : v) \cap M(b : w)$ — Definition 45
- or $v \sqsubseteq_V \bot_V$ (and $\bot_V$ is defined) $\Longrightarrow M_V(v) = \emptyset$ — $\sqsubseteq_V, \bot_V$ consistent
  $\Longrightarrow M(a : v) = \emptyset$ — Definition 45
  Now, $M(a : v \sqcap b : w) = M(a : \bot_V) = \emptyset = M(a : v) \cap M(b : w)$ — $\bot_V$ consistent,
- or $w \sqsubseteq_V \bot_V$ (and $\bot_V$ is defined): idem.

(disjunction) If $a : v \sqcup b : w$ is defined, then

- either $a = b$ (and $v \sqcup_V w$ is defined):
  $i \in M(a : v \sqcup b : w) \iff i \in M(a : (v \sqcup_V w))$
  $\iff i(a) \in M_V(v \sqcup_V w)$ — Definition 45
  $\iff i(a) \in M_V(v) \cup M_V(w)$ — $\sqcup_V$ consistent and complete
  $\iff i(a) \in M_V(v)$ or $i(b) \in M_V(w)$ — $a = b$
  $\iff i \in M(a : v) \cup M(b : w)$ — Definition 45
- or $w \sqsubseteq_V \bot_V$ (and $\bot_V$ is defined) $\Longrightarrow M_V(w) = \emptyset$ — $\sqsubseteq_V, \bot_V$ consistent
  $\Longrightarrow M(b : w) = \emptyset$ — Definition 45
  Now, $i \in M(a : v \sqcup b : w) \Longrightarrow i \in M(a : v) \Longrightarrow i \in M(a : v) \cup M(b : w)$,
- or $v \sqsubseteq_V \bot_V$ (and $\bot_V$ is defined): idem. ■

$P_{valattr}$ is partially defined in both conjunction and disjunction, but it is consistent and complete provided that its implementation argument is. Furthermore, the following lemma shows that $P_{valattr(V)}$ is reduced provided that its argument is. So, the logic functor *prop* can be applied to logic functor *valattr* to form a complete and consistent logic.

Lemma 48 *$P_{valattr(V)}$ is reduced w.r.t. $S_{valattr(V)}$ if P_V is reduced w.r.t. S_V.*

Proof: Let (A, B) an open sequent for $P_{valattr(V)}$. From Lemma 19, we have

(1) $\forall a_1 : v_1, a_2 : v_2 \in A : a_1 : v_1 \sqcap a_2 : v_2$ is undefined,

(2) $\forall b_1 : w_1, b_2 : w_2 \in B : b_1 : w_1 \sqcup b_2 : w_2$ is undefined,

(3) $\forall a : v \in A, b : w \in B : a : v \not\sqsubseteq b : w$.

Now, consider any attribute $x \in Attr$, and the following notations:

- $A(x) = \{v \in AS_{L_V} \mid x : v \in A\}$,

- $B(x) = \{w \in AS_{L_V} \mid x : w \in B\}$.

We show that for every $x \in Attr$, $(A(x), B(x))$ is an open sequent for P_V, using Lemma 19.

- Suppose that $\exists v_1 \neq v_2 \in A(x) : v_1 \sqcap_V v_2$ is defined
  $\Longrightarrow \exists x : v_1 \neq x : v_2 \in A : x : v_1 \sqcap x : v_2$ is defined, which contradicts the fact that (A, B) is an open sequent Lemma 19
  Hence, $\forall v_1 \neq v_2 \in A(x) : v_1 \sqcap_V v_2$ is undefined,
- Similarly, $\forall w_1 \neq w_2 \in B(x) : w_1 \sqcup_V w_2$ is undefined,
- $(3) \Longrightarrow \forall a : v \in A : v \not\sqsubseteq_V \bot_V \Longrightarrow \forall v \in A(x) : v \not\sqsubseteq_V \bot_V$,
- $(3) \Longrightarrow \forall b : w \in B : \top_V \not\sqsubseteq_V b : w \Longrightarrow \forall w \in B(x) : \top_V \not\sqsubseteq_V w$,
- $(3) \Longrightarrow \forall a : v \in A, b : w \in B : a \neq b$ or $v \not\sqsubseteq_V w$
  $\Longrightarrow \forall v \in A(x), w \in B(x) : v \not\sqsubseteq_V w$.

Now, because P_V is reduced w.r.t. S_V, the open sequent $(A(x), B(x))$ is not valid for every $x \in Attr$
$\Longrightarrow \bigcap_{v \in A(x)} M_V(v) \not\subseteq \bigcup_{w \in B(x)} M_V(w)$
$\Longrightarrow \bigcap_{v \in A(x)} M_V(v) \setminus \bigcup_{w \in B(x)} M_V(w) \neq \emptyset$ Lemma 20
So, we can find an interpretation $i \in I$ such that
$\forall x \in Attr : i(x) \in \bigcap_{v \in A(x)} M_V(v) \setminus \bigcup_{w \in B(x)} M_V(w)$
$\Longrightarrow \forall a : v \in A : i(a) \in M_V(v)$ and $\forall b : w \in B : i(b) \notin M_V(w)$
$\Longrightarrow i \in \bigcap_{a:v \in A} M(a : v)$ and $i \notin \bigcup_{b:w \in B} M(b : w)$
$\Longrightarrow \bigcap_{a:v \in A} M(a : v) \not\subseteq \bigcup_{b:w \in B} M(b : w)$
$\Longrightarrow (A, B)$ is not valid. Therefore, $P_{valattr(V)}$ is reduced w.r.t. $S_{valattr(V)}$. ■

B.3 Sums of logics

The sum of two logics allows one to form descriptions/queries about objects that belong to different domains. Objects from one domain are described by formulas of a logic L_1, while other objects use logic L_2. A special element '?' represents the absence of information, and the element '#' represents a contradiction. For instance, the bibliographical application may be part of a larger knowledge base whose other parts are described by completely different formulas. Even inside the bibliographical part, journal articles use facets that are not relevant to conference article (and vice-versa).

We write *sum* the logic functor used for constructing the sum of 2 logics. Note that *sum* could easily be generalized to arbitrary arities.

Definition 49 *AS_{sum} forms the disjoint union of two logics plus formulas ? and #.*

Definition 50 *S_{sum} is $(I_{L_1}, \models_{L_1}), (I_{L_2}, \models_{L_2}) \mapsto (I, \models)$ such that*

$$I = I_{L_1} \uplus I_{L_2} \text{ and } i \models f = \begin{cases} i \models_{L_1} f & \text{if } i \in I_{L_1}, f \in AS_{L_1} \\ i \models_{L_2} f & \text{if } i \in I_{L_2}, f \in AS_{L_2} \\ true & \text{if } f = ? \\ false & \text{otherwise} \end{cases}$$

We will prove that $P_{sum(L_1,L_2)}$ is reduced w.r.t. $S_{sum(L_1,L_2)}$ if P_{L_1} and P_{L_2} are reduced w.r.t. S_{L_1} and S_{L_2}, making the logic functor *sum* usable inside the logic functor *prop*. The development of this logic functor is rather complex but we could not find simpler but reduced definitions for *sum*.

Definition 51 *P_{sum} is $(\sqsubseteq_{L_1}, \sqcap_{L_1}, \sqcup_{L_1}, \top_{L_1}, \bot_{L_1}), (\sqsubseteq_{L_2}, \sqcap_{L_2}, \sqcup_{L_2}, \top_{L_2}, \bot_{L_2}) \mapsto (\sqsubseteq, \sqcap, \sqcup, \top, \bot)$ such that*

- $f \sqsubseteq g = \begin{cases} f \sqsubseteq_{L_1} g & \text{if } f, g \in AS_{L_1} \\ f \sqsubseteq_{L_2} g & \text{if } f, g \in AS_{L_2} \\ true & \text{if } f \sqsubseteq_{L_1} \bot_{L_1} \text{ or } f \sqsubseteq_{L_2} \bot_{L_2} \text{ or } f = \# \text{ or } g = ? \\ false & \text{otherwise} \end{cases}$

- $f \sqcap g \;=\; \begin{cases} f \sqcap_{L_1} g & \text{if } f,g \in AS_{L_1} \\ f \sqcap_{L_2} g & \text{if } f,g \in AS_{L_2} \\ f & \text{if } g =? \\ g & \text{if } f =? \\ \# & \text{otherwise} \end{cases}$
- $f \sqcup g \;=\; \begin{cases} f \sqcup_{L_1} g & \text{if } f,g \in AS_{L_1} \\ f \sqcup_{L_2} g & \text{if } f,g \in AS_{L_2} \\ g & \text{if } f = \# \text{ or } f \sqsubseteq_{L_1} \bot_{L_1} \text{ or } f \sqsubseteq_{L_2} \bot_{L_2} \\ f & \text{if } g = \# \text{ or } g \sqsubseteq_{L_1} \bot_{L_1} \text{ or } g \sqsubseteq_{L_2} \bot_{L_2} \\ ? & \text{if } f =? \text{ or } g =? \\ ? & \text{if } f \in AS_{L_1}, g \in AS_{L_2} \text{ and } \top_{L_1} \sqsubseteq_{L_1} f \text{ and } \top_{L_2} \sqsubseteq_{L_2} g \\ ? & \text{if } f \in AS_{L_2}, g \in AS_{L_1} \text{ and } \top_{L_1} \sqsubseteq_{L_1} g \text{ and } \top_{L_2} \sqsubseteq_{L_2} f \\ undef & \text{otherwise} \end{cases}$
- $\top \;=\; ?$ $\qquad\qquad \bot \;=\; \#$

Theorem 52 *$P_{sum(L_1,L_2)}$ is consistent and complete in $\sqsubseteq$, $\top$, $\bot$, $\sqcap$, $\sqcup$ w.r.t. $S_{sum(L_1,L_2)}$ if P_{L_1} and P_{L_2} are consistent and complete w.r.t. S_{L_1} and S_{L_2}.*

Proof: Let f, g be formulas in $AS_{sum(L_1,L_2)}$.

(deduction) according to Definition 51, we have the following cases

- $f, g \in AS_{L_1}$: $f \sqsubseteq g \iff f \sqsubseteq_{L_1} g \iff M_{L_1}(f) \subseteq M_{L_1}(g)$ (if $\sqsubseteq_{L_1}$ is consistent ($\Rightarrow$ part) and complete ($\Leftarrow$ part)),
  $\iff M(f) \subseteq M(g)$ Definition 50
- $f, g \in AS_{L_2}$: similarly, $f \sqsubseteq g \iff M(f) \subseteq M(g)$ (if $\sqsubseteq_{L_2}$ is consistent ($\Rightarrow$ part) and complete ($\Leftarrow$ part)),
- $f \sqsubseteq_{L_1} \bot_{L_1}$ or $f \sqsubseteq_{L_2} \bot_{L_2}$ or $f = \#$ or $g =?$: $f \sqsubseteq g$ is true, and $M(f) \subseteq M(g)$ is also true,
- otherwise: $f \sqsubseteq g$ is false, and it can be verified that $M(f) \subseteq M(g)$ is also false.

(tautology) $\top$ is always defined, and $M(\top) = M(?) = I$ Definition 50

(contradiction) $\bot$ is always defined, and $M(\bot) = M(\#) = \emptyset$ Definition 50

(conjunction) If $f \sqcap g$ is defined, then

- either $f, g \in AS_{L_1}$ and $f \sqcap_{L_1} g$ is defined
  $\Longrightarrow M(f) \cap M(g) = M_{L_1}(f) \cap M_{L_1}(g) = M_{L_1}(f \sqcap_{L_1} g)$ $\sqcap_{L_1}$ consistent and complete
  $\Longrightarrow M(f) \cap M(g) = M(f \sqcap_{L_1} g)$,
- or $f, g \in AS_{L_2}$ and $f \sqcap_{L_2} g$ is defined: idem,
- or $f =? \Longrightarrow M(f) \cap M(g) = I \cap M(g) = M(g)$,
- or $g =?$: idem,
- or $f = \# \Longrightarrow M(f) \cap M(g) = \emptyset \cap M(g) = \emptyset = M(\#)$,
- or $g = \#$: idem,
- or $f \in AS_{L_1}, g \in AS_{L_2} \Longrightarrow M(f) = M_{L_1}(f) \subseteq I_{L_1}$ and $M(g) = M_{L_2}(g) \subseteq I_{L_2}$
  $\Longrightarrow M(f) \cap M(g) = \emptyset = M(\#)$ (because $I_{L_1} \cap I_{L_2} = \emptyset$),
- or $f \in AS_{L_2}, g \in AS_{L_{L_1}}$: idem.

(disjunction) If $f \sqcup g$ is defined, then

- either $f, g \in AS_{L_1}$ and $f \sqcup_{L_1} g$ is defined
  $\Longrightarrow M(f) \cup M(g) = M_{L_1}(f) \cup M_{L_1}(g) = M_{L_1}(f \sqcup_{L_1} g)$ $\sqcup_{L_1}$ consistent and complete
  $\Longrightarrow M(f) \cup M(g) = M(f \sqcup_{L_1} g)$,
- or $f, g \in AS_{L_2}$ and $f \sqcup_{L_2} g$ is defined: idem,
- or $f = \#$ or $f \sqsubseteq_{L_1} \bot_{L_1}$ or $f \sqsubseteq_{L_2} \bot_{L_2}$: $M(f) \cup M(g) = \emptyset \cup M(g) = M(g)$,

- or $g = \#$ or $g \sqsubseteq_{L_1} \bot_{L_1}$ or $g \sqsubseteq_{L_2} \bot_{L_2}$: idem,
- or $f = ?$: $M(f) \cup M(g) = I \cup M(g) = I = M(?)$,
- or $g = ?$: idem,
- or $f \in AS_{L_1}, g \in AS_{L_2}$ and $\top_{L_1} \sqsubseteq_{L_1} f \top_{L_2} \sqsubseteq_{L_2} g$: $M(f) \cup M(g) = I_{L_1} \cup I_{L_2} = I = M(?)$,
- or $f \in AS_{L_2}, g \in AS_{L_1}$ and $\top_{L_1} \sqsubseteq_{L_1} g \top_{L_2} \sqsubseteq_{L_2} f$: $M(f) \cup M(g) = I_{L_2} \cup I_{L_1} = I = M(?)$. ■

Lemma 53 *$P_{sum(L_1,L_2)}$ is reduced w.r.t. $S_{sum(L_1,L_2)}$ if P_{L_1} and P_{L_2} are reduced and consistent in $\top, \bot$ w.r.t. S_{L_1} and S_{L_2}.*

Proof: Let (A, B) an open sequent for $P_{sum(L_1,L_2)}$. From Lemma 19, it follows that

(1) $\# \notin A$ and $\forall a \in A \cap AS_{L_1} : a \not\sqsubseteq_{L_1} \bot_{L_1}$ and $\forall a \in A \cap AS_{L_2} : a \not\sqsubseteq_{L_2} \bot_{L_2}$,

(2) $? \notin B$,

(3) $\forall a \in A, b \in B : a \not\sqsubseteq b$,

(4) $\forall a_1, a_2 \in A : a_1 \sqcap a_2$ is undefined,

(5) $\forall b_1, b_2 \in B : b_1 \sqcup b_2$ is undefined.

With (1), (4), and definition of $\sqcap$, we can say that A has one of the following forms: $A = \emptyset$, $A = \{?\}$, $\emptyset \subsetneq A \subseteq AS_{L_1}$, or $\emptyset \subsetneq A \subseteq AS_{L_2}$.

With (2), (5), and definition of $\sqcup$, we can say that B has one of the following forms: $B = \{\#\}$, or $B = B_1 \uplus B_2$ where $B_1 \subseteq AS_{L_1}$ and $B_2 \subseteq AS_{L_2}$.

We show that each form of A and B satisfies $\bigcap_{a \in A} M(a) \not\sqsubseteq \bigcup_{b \in B} M(b)$:

- $A = \emptyset$ or $A = \{?\}$: $\bigcap_{a \in A} M(a) = I$ — $\top_{sum}$ consistent
  - $B = \{\#\}$: $\bigcup_{b \in B} M(b) = \emptyset$ — $\bot_{sum}$ consistent,
  - $B = B_1 \uplus B_2$: with (2), (5), and Lemma 19
    we have that $(\emptyset, B_1)$ and $(\emptyset, B_2)$ are open sequents for P_{L_1} and P_{L_2} respectively
    $\Longrightarrow$ $(\emptyset, B_1)$ and $(\emptyset, B_2)$ are not valid open sequents because P_{L_1} and P_{L_2} are reduced
    $\Longrightarrow$ $I_{L_1} \not\sqsubseteq \bigcup_{b \in B_1} M_{L_1}(b)$ and $I_{L_2} \not\sqsubseteq \bigcup_{b \in B_2} M_{L_2}(b)$ — Lemma 20
    $\Longrightarrow$ $\bigcup_{b \in B} M(b) \neq I$ — Definition 50
- $\emptyset \subsetneq A \subseteq AS_{L_1}$: first, $\bigcap_{a \in A} M(a) \subseteq I_{L_1}$;
  second, $(A, \emptyset)$ is an open sequent because of (1), (4), and Lemma 19
  $\Longrightarrow$ $(A, \emptyset)$ is not a valid open sequent — P_{L_1} reduced
  $\Longrightarrow$ $\bigcap_{a \in A} M_{L_1}(a) \not\sqsubseteq \emptyset \Longrightarrow \bigcap_{a \in A} M(a) \neq \emptyset$.
  - $B = \{\#\}$: $\bigcup_{b \in B} M(b) = \emptyset$ — $\bot_{sum}$ consistent,
  - $B = B_1 \uplus B_2$: (A, B_1) is an open sequent for P_{L_1} because of (1), (2), (3), (4), (5), and Lemma 19
    $\Longrightarrow$ (A, B_1) is not valid — P_{L_1} reduced
    $\Longrightarrow$ $\bigcap_{a \in A} M_{L_1}(a) \not\sqsubseteq \bigcup_{b \in B_1} M_{L_1}(b)$
    $\Longrightarrow$ $\bigcap_{a \in A} M(a) \not\sqsubseteq \bigcup_{b \in B} M(b)$ — $(I_{L_1} \cap I_{L_2} = \emptyset)$,
- $\emptyset \subsetneq A \subseteq AS_{L_1}$: idem.

Hence, the open sequent (A, B) is not valid — Lemma 20 ■

B.4 An alternative definition of sums of logics

The sum of logics considered in this section is the same as in Section B.3 for the syntax AS_{sum} (Definition 49) and semantics S_{sum} (Definition 50). What is changed is its simpler implementation of $\sqcup$. As a consequence, this implementation is not reduced, but it is total.

Definition 54 *P'_{sum} is the same as in Definition 51 except for the definition of $\sqcup$ which is:*

$$f \sqcup g \;=\; \begin{cases} f \sqcup_{L_1} g & \text{if } f,g \in AS_{L_1} \\ f \sqcup_{L_2} g & \text{if } f,g \in AS_{L_2} \\ f & \text{if } g = \# \text{ or } g \sqsubseteq_{L_1} \bot_{L_1} \text{ or } g \sqsubseteq_{L_2} \bot_{L_2} \\ g & \text{if } f = \# \text{ or } f \sqsubseteq_{L_1} \bot_{L_1} \text{ or } f \sqsubseteq_{L_2} \bot_{L_2} \\ ? & \text{otherwise} \end{cases}$$

Theorem 55 *$P'_{sum(P_1,P_2)}$ is consistent and complete in $\sqsubseteq$, $\top$, $\bot$, $\sqcap$, and complete but not consistent in $\sqcup$.*

Proof: Let f, g be formulas in $AS_{sum(L_1,L_2)}$.

(deduction) according to Definition 54, we have the following cases

- $f,g \in AS_{L_1}$: $f \sqsubseteq g \iff f \sqsubseteq_{L_1} g \iff M_{L_1}(f) \subseteq M_{L_1}(g)$ (if $\sqsubseteq_{L_1}$ is consistent ($\Rightarrow$ part) and complete ($\Leftarrow$ part)),
  $\iff M(f) \subseteq M(g)$ Definition 50
- $f,g \in AS_{L_2}$: similarly, $f \sqsubseteq g \iff M(f) \subseteq M(g)$ (if $\sqsubseteq_{L_2}$ is consistent ($\Rightarrow$ part) and complete ($\Leftarrow$ part),
- $f \sqsubseteq_{L_1} \bot_{L_1}$ or $f \sqsubseteq_{L_2} \bot_{L_2}$ or $f = \#$ or $g = ?$: $f \sqsubseteq g$ is true,
  and $M(f) \subseteq M(g)$ is also true,
- otherwise: $f \sqsubseteq g$ is false, and it can be verified that $M(f) \subseteq M(g)$ is also false.

(tautology) $\top$ is always defined, and $M(\top) = M(?) = I$ Definition 50

(contradiction) $\bot$ is always defined, and $M(\bot) = M(\#) = \emptyset$ Definition 50

(conjunction) If $f \sqcap g$ is defined, then

- either $f,g \in AS_{L_1}$ and $f \sqcap_{L_1} g$ is defined
  $\Longrightarrow M(f) \cap M(g) = M_{L_1}(f) \cap M_{L_1}(g) = M_{L_1}(f \sqcap_{L_1} g)$ $\sqcap_{L_1}$ consistent and complete
  $\Longrightarrow M(f) \cap M(g) = M(f \sqcap_{L_1} g)$,
- or $f,g \in AS_{L_2}$ and $f \sqcap_{L_2} g$ is defined: idem,
- or $f = ? \Longrightarrow M(f) \cap M(g) = I \cap M(g) = M(g)$,
- or $g = ?$: idem,
- or $f = \# \Longrightarrow M(f) \cap M(g) = \emptyset \cap M(g) = \emptyset = M(\#)$,
- or $g = \#$: idem,
- or $f \in AS_{L_1}, g \in AS_{L_2} \Longrightarrow M(f) = M_{L_1}(f) \subseteq I_{L_1}$ and $M(g) = M_{L_2}(g) \subseteq I_{L_2}$
  $\Longrightarrow M(f) \cap M(g) = \emptyset = M(\#)$ (because $I_{L_1} \cap I_{L_2} = \emptyset$),
- or $f \in AS_{L_2}, g \in AS_{L_{L_1}}$: idem.

(disjunction) If $f \sqcup g$ is defined, then

- either $f,g \in AS_{L_1}$ and $f \sqcup_{L_1} g$ is defined
  $\Longrightarrow M(f) \cup M(g) = M_{L_1}(f) \cup M_{L_1}(g) \subseteq M_{L_1}(f \sqcup_{L_1} g)$ $\sqcup_{L_1}$ complete
  $\Longrightarrow M(f) \cup M(g) \subseteq M(f \sqcup_{L_1} g)$,
- or $f,g \in AS_{L_2}$ and $f \sqcup_{L_2} g$ is defined: idem,
- or $f = \#$ or $f \sqsubseteq_{L_1} \bot_{L_1}$ or $f \sqsubseteq_{L_2} \bot_{L_2}$: $M(f) \cup M(g) = \emptyset \cup M(g) = M(g)$,
- or $g = \#$ or $g \sqsubseteq_{L_1} \bot_{L_1}$ or $g \sqsubseteq_{L_2} \bot_{L_2}$: idem,
- otherwise: $M(?) = I \supseteq M(f) \cup M(g)$. ■

The definition of P'_{sum} shows that $P_{sum(L_1,L_2)}$ is total iff implementations P_{L_1} and P_{L_2} are also total.

B.5 Product of logics

The product of two logics allows one to form descriptions or queries that are faceted. Each facet corresponds to a dimension of the domain considered in an application. For instance, in the bibliographical information system, titles, lists of authors, publication dates, etc., are facets. Each facet brings its own syntax and semantics.

We note *prod* the logic functor used for constructing the product of 2 logics. This arity of 2 is chosen for simplicity, but *prod* could easily be generalized to arbitrary arities.

Definition 56 (Syntax) *The logic functor AS_{prod} is the cartesian product of two logics.*

Definition 57 (Semantics) *S_{prod} is $(I_{L_1}, \models_{L_1}), (I_{L_2}, \models_{L_2}) \mapsto (I, \models)$ such that*
$I = I_{L_1} \times I_{L_2}$ and $(i_{L_1}, i_{L_2}) \models (f_1, f_2)$ iff $i_1 \models_1 f_1$ and $i_{L_2} \models_{L_2} f_2$.

Definition 58 *P_{prod} is $(\sqsubseteq_{L_1}, \sqcap_{L_1}, \sqcup_{L_1}, \top_{L_1}, \bot_{L_1}), (\sqsubseteq_{L_2}, \sqcap_{L_2}, \sqcup_{L_2}, \top_{L_2}, \bot_{L_2}) \mapsto (\sqsubseteq, \sqcap, \sqcup, \top, \bot)$ such that*

- *$(f_1, f_2) \sqsubseteq (g_1, g_2)$ iff $f_1 \sqsubseteq_{L_1} g_1$ and $f_2 \sqsubseteq_{L_2} g_2$, or $f_1 \sqsubseteq_{L_1} \bot_{L_1}$, or $f_2 \sqsubseteq_{L_2} \bot_{L_2}$,*
- *$(f_1, f_2) \sqcap (g_1, g_2) = (f_1 \sqcap_{L_1} g_1, f_2 \sqcap_{L_2} g_2)$ (undef if one conjunction is),*
- *$(f_1, f_2) \sqcup (g_1, g_2) = (f_1 \sqcup_{L_1} g_1, f_2 \sqcup_{L_2} g_2)$ (undef if one disjunction is),*
- *$\top = (f_1, f_2)$ such that $\top_{L_1} \sqsubseteq_{L_1} f_1$ and $\top_{L_2} \sqsubseteq_{L_2} f_2$,*
- *$\bot = (f_1, f_2)$ such that $f_1 \sqsubseteq_{L_1} \bot_{L_1}$ or $f_2 \sqsubseteq_{L_2} \bot_{L_2}$.*

Note that (f_1, f_2) is $\bot$ as soon as one of the f_i is. This is in accordance with the semantics of Definition 57, because it simply says that the semantics is a cartesian product, so that the product by an empty set is always empty.

Theorem 59 *$P_{prod(L_1,L_2)}$ is consistent and complete in operations $\sqsubseteq$, $\top$, $\bot$, $\sqcap$ and only complete in $\sqcup$ w.r.t. $S_{prod(L_1,L_2)}$ if P_{L_1} and P_{L_2} are consistent and complete w.r.t. S_{L_1} and S_{L_2}.*

Proof: Let f_1, g_1 be formulas of AS_{L_1}, and f_2, g_2 be formulas of AS_{L_2}.

(deduction) $(f_1, f_2) \sqsubseteq (g_1, g_2) \iff f_1 \sqsubseteq_{L_1} g_1$ and $f_2 \sqsubseteq_{L_2} g_2$, or $f_1 \sqsubseteq_{L_1} \bot_{L_1}$, or $f_2 \sqsubseteq_{L_2} \bot_{L_2}$ Definition 58
$\iff M_{L_1}(f_1) \subseteq M_{L_1}(g_1)$ and $M_{L_2}(f_2) \subseteq M_{L_2}(g_2)$,
or $M_{L_1}(f_1) \subseteq M_{L_1}(\bot_{L_1}) = \emptyset$, or $M_{L_2}(f_2) \subseteq M_{L_2}(\bot_{L_2}) = \emptyset$ ($\sqsubseteq_{L_1}$ and $\sqsubseteq_{L_2}$ are consistent ($\Rightarrow$ part) and complete ($\Leftarrow$ part))
$\iff M_{L_1}(f_1) \times M_{L_2}(f_2) \subseteq M_{L_1}(g_1) \times M_{L_2}(g_2) \iff M(f_1, f_2) \subseteq M(g_1, g_2)$.

(tautology) If $\top$ is defined, then both $\top_{L_1}$ and $\top_{L_2}$ are defined
$\Longrightarrow M_{L_1}(\top_{L_1}) = I_{L_1}$ and $M_{L_2}(\top_{L_2}) = I_{L_2}$ $\top_{L_1}$, $\top_{L_2}$ complete
$\Longrightarrow M((\top_{L_1}, \top_{L_2})) = I_{L_1} \times I_{L_2} = I$.

(contradiction) If $\bot$ is defined, then either $\bot_{L_1}$ or $\bot_{L_2}$ is defined. Without loss of generality, we suppose that $\bot_{L_1}$ is defined:
$\Longrightarrow M_{L_1}(\bot_{L_1}) = \emptyset$ $\bot_{L_1}$ consistent
$\Longrightarrow \forall f_2 \in AS_{L_2} : M((\bot_{L_1}, f_2)) = M_{L_1}(\bot_{L_1}) \times M_{L_2}(f_2) = \emptyset$.

(conjunction) If $(f_1, f_2) \sqcap (g_1, g_2)$ is defined, then $f_1 \sqcap_{L_1} g_1$ and $f_2 \sqcap_2 g_2$ are also defined:
$\Longrightarrow M_{L_1}(f_1 \sqcap_{L_1} g_1) = M_{L_1}(f_1) \cap M_{L_1}(g_1)$
and $M_{L_2}(f_2 \sqcap_{L_2} g_2) = M_{L_2}(f_2) \cap M_{L_2}(g_2)$ $\sqcap_{L_1}$, $\sqcap_{L_2}$ consistent and complete
$\Longrightarrow M((f_1 \sqcap_{L_1} g_1, f_2 \sqcap_{L_2} g_2)) = M_{L_1}(f_1 \sqcap_{L_1} g_1) \times M_{L_2}(f_2 \sqcap_{L_2} g_2)$
$= (M_{L_1}(f_1) \cap M_{L_1}(g_1)) \times (M_{L_2}(f_2) \cap M_{L_2}(g_2))$
$= (M_{L_1}(f_1) \times M_{L_2}(f_2)) \cap (M_{L_1}(g_1) \times M_{L_2}(g_2))$
$= M((f_1, f_2)) \cap M((g_1, g_2))$.

(disjunction) If $(f_1, f_2) \sqcup (g_1, g_2)$ is defined, then $f_1 \sqcup_{L_1} g_1$ and $f_2 \sqcup_2 g_2$ are also defined:
$\Longrightarrow M_{L_1}(f_1 \sqcup_{L_1} g_1) \supseteq M_{L_1}(f_1) \cup M_{L_1}(g_1)$
and $M_{L_2}(f_2 \sqcup_2 g_2) \supseteq M_{L_2}(f_2) \cup M_{L_2}(g_2)$ $\sqcup_{L_1}$, $\sqcup_{L_2}$ complete
$\Longrightarrow M_{L_1}(f_1 \sqcup_{L_1} g_1) \times M_{L_2}(f_2 \sqcup_{L_2} g_2) \supseteq (M_{L_1}(f_1) \times M_{L_2}(f_2)) \cup (M_{L_1}(g_1) \times M_{L_2}(g_2))$
$\Longrightarrow M((f_1 \sqcup_{L_1} g_1, f_2 \sqcup_{L_2} g_2)) \supseteq M((f_1, f_2)) \cup M((g_1, g_2))$. ■

The definition of P_{prod} shows that $P_{prod(L_1,L_2)}$ is total iff P_{L_1} and P_{L_2} are also total. It is bounded in $\top$ if P_{L_1} and P_{L_2} are; it is bounded in $\bot$ if P_{L_1} or P_{L_2} is (indeed, $M(f_1, f_2) = M_{L_1}(f_1) \times M_{L_2}(f_2)$). Theorem 59 shows that if P_{L_1} and P_{L_2} are consistent and complete, then the conjunction of the product logic is consistent and complete, but the disjunction is not consistent in general. So, one must be cautious when combining customized logics.

Example 60 *Let L_1 be a logic of sets of letters interpreted as themselves with inclusion as deduction relation. Let L_2 be a logic of sets of digits under the same conditions. These logics are consistent in disjunction (note that $\sqcup_{L_1} = \sqcup_{L_2} = \cup$).*

However, $M_{prod(L_1,L_2)}((\{a\},\{1\}) \sqcup (\{b\},\{2\})) \overset{Def}{=} 58$
$M_{prod(L_1,L_2)}(\{a\} \sqcup_{L_1} \{b\}, \{1\} \sqcup_{L_2} \{2\}) = \{a,b\} \times \{1,2\}$ is not included in
$M_{prod(L_1,L_2)}((\{a\},\{1\})) \cup M_{prod(L_1,L_2)}((\{b\},\{2\})) = \{(a,1),(b,2)\}$.
This shows that $P_{prod(L_1,L_2)}$ does not satisfy Definition 6 on consistency in $\sqcup$.

B.6 Sets of models

The logic functor *set* is useful to describe for instance sets of authors or keywords. Each item is specified by a formula of the logic argument of *set*. Models of sets of subformulas are sets of models of subformulas.

Definition 61 *AS_{set} is the set of finite subsets of formulas of a logic.*

Definition 62 *S_{set} is $(I_e, \models_e) \mapsto (I, \models)$ such that*

$$I = \mathcal{P}(I_e) \text{ and } i \models f \iff \forall f_e \in f : i \cap M_e(f_e) \neq \emptyset.$$

Definition 63 *P_{set} is $(\sqsubseteq_e, \sqcap_e, \sqcup_e, \top_e, \bot_e) \to (\sqsubseteq, \sqcap, \sqcup, \top, \bot)$ such that for all $f, g \in AS_{set(e)}$*

- $f \sqsubseteq g \iff \forall g_e \in g : \exists f_e \in f : f_e \sqsubseteq_e g_e$
- $f \sqcap g = (f \cup g)$
- $f \sqcup g = \{f_e \sqcup_e g_e \mid f_e \in f, g_e \in g, f_e \sqcup_e g_e \text{ defined}\}$
- $\top = \emptyset$
- $\bot = \{\bot_e\}$, *if $\bot_e$ is defined*

Theorem 64 *The deduction $\sqsubseteq$ is consistent (resp. complete) if deduction on elements $\sqsubseteq_e$ is also consistent (resp. complete). The tautology $\top$ is always defined and complete. The contradiction $\bot$ is defined (resp. consistent) if the element contradiction $\bot_e$ is also defined (resp. consistent). The conjunction $\sqcap$ is always totally defined, consistent and complete. The disjuction $\sqcup$ is totally defined, complete if the element disjunction $\sqcup_e$ is also complete, but not consistent in general.*

Proof: Let $f, g \in AS_{set(e)}$.

(deduction)
- if $f \sqsubseteq g$ is true, then $\forall g_e \in g : \exists f_e \in f : f_e \sqsubseteq_e g_e$
  $\Longrightarrow \forall g_e \in g : \exists f_e \in f : M_e(f_e) \subseteq_e M_e(g_e)$ — $\sqsubseteq_e$ consistent
  Moreover, for all $i \in M(f)$, $\forall f_e \in f : i \cap M_e(f_e) \neq \emptyset$ — Definition 62
  This results in $\forall g_e \in g : \exists f_e \in f : i \cap M_e(g_e) \neq \emptyset$
  $\Longrightarrow \forall g_e \in g : i \cap M_e(g_e) \neq \emptyset$
  $\Longrightarrow i \in M(g)$. — Definition 62
  Hence $M(f) \subseteq M(g)$.
- if $f \sqsubseteq g$ is false, then $\exists g_e \in g : \forall f_e \in f : f_e \not\sqsubseteq_e g_e$
  $\Longrightarrow \exists g_e \in g : \forall f_e \in f : \exists i_e \in M_e(f_e) : i_e \notin M_e(g_e)$ — $\sqsubseteq_e$ complete
  Then, it exists $i \in I$ such that $\forall f_e \in f : i \cap M_e(f_e) \neq \emptyset$ and $\exists g_e \in g : i \cap M_e(g_e) = \emptyset$
  $\Longrightarrow \exists i \in M(f) : i \notin M(g)$ — Definition 62
  $\Longrightarrow M(f) \not\subseteq M(g)$.

(tautology) $M(\top) = M(\emptyset) = \{i \in I \mid \forall f_e \in f : i \cap M_e(f_e) \neq \emptyset\} = I$.

(contradiction) If $\perp_e$ is defined, then

$M(\perp) = M(\{\perp_e\}) = \{i \in I \mid i \cap M_e(\perp_e) \neq \emptyset\}$ Definition 62
$= \{i \in I \mid i \cap \emptyset \neq \emptyset\}$ $\perp_e$ consistent
$= \emptyset$.

(conjunction) $\sqcap$ is totally defined, and for all $i \in I$

$i \in M(f \sqcap g) \iff i \in M(f \cup g)$
$\iff \forall e \in f \cup g : i \cap M_e(e) \neq \emptyset$ Definition 62
$\iff \forall e \in f : i \cap M_e(e) \neq \emptyset$ and $\forall e \in g : i \cap M_e(e) \neq \emptyset$
$\iff i \in M(f)$ and $i \in M(g) \iff i \in M(f) \cap M(g)$.

(disjunction) for all $i \in I$

$i \in M(f) \cup M(g) \iff i \in M(f)$ or $i \in M(g)$
$\Longrightarrow \forall f_e \in f : i \cap M_e(f_e) \neq \emptyset$ or $\forall g_e \in g : i \cap M_e(g_e) \neq \emptyset$ Definition 62
$\Longrightarrow \forall f_e \in f, g_e \in g : i \cap (M_e(f_e) \cup M_e(g_e)) \neq \emptyset$
$\Longrightarrow \forall f_e \in f, g_e \in g : f_e \sqcup_e g_e$ defined $\Rightarrow i \cap M_e(f_e \sqcup_e g_e) \neq \emptyset$ $\sqcup_e$ complete
$\Longrightarrow i \in M(\{f_e \sqcup_e g_e \mid f_e \in f, g_e \in g, f_e \sqcup_e g_e \text{ defined}\})$
$\Longrightarrow i \in M(f \sqcup g)$. ■

Contents

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399