A framework for developing embeddable customized logics

Sébastien Ferré** and Olivier Ridoux

IRISA, Campus Universitaire de Beaulieu, 35042 RENNES cedex, {ferre,ridoux}@irisa.fr

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. Their implementation 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 logics, one must also define composition laws, and prove their properties. We present the principles of *logic functors* and their compositions for constructing customized logics. An important issue is how the operations of different sublogics inter-operate. We propose a formalization of the logic functors, their semantics, implementations, and their composition.

1 Introduction

We present a framework for building embeddable automatic theorem provers for *customized logics*. The framework defines *logic functors* as logic components; for instance, one component may be the propositional logic, another component may be the interval logic, also called intervals. Logic functors can be composed to form new logics, for instance, propositional logic on intervals.

Each logic functor has its own proof-theory, which can be implemented as a theorem prover. We desire that the proof-theory and the theorem prover of the composition of logic functors should result from the composition of the proof-theories and the theorem provers of the component logic functors.

All logic functors and their compositions implement a common *interface*. This makes it possible to construct 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 comply with this interface.

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, such as compilers, operating systems, file-systems, and information systems.

This article is organized as follows. Section 2 presents our motivations, and Section 3 introduces the notions of logics and logic functors, and several logic

** This author is supported by a scholarship from CNRS and Région Bretagne 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. Section 5 answers these questions by introducing a new property, called *reducedness*. In Section 6, we compare this work with the literature. Appendix A presents some nullary logic functors, and Appendix B presents some 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 upon 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, if one leaves the logic totally undefined, then one puts too much responsibility on the end-users or on a knowledge-base administrator. It is unlikely they can design such a logical component themselves. By using the framework developed in this article, one can design a toolbox of logical components, and the user has only the responsibility of composing those components. 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 logic-based components 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 components model an information processing domain in logic, and also they bring to the front solutions in which logic is the main engine. This can be illustrated by the difference between using a logic of programs and programming in logic.

The logic in use in these system is often not defined by a single pure deduction system, but rather it combines several logics together. The designer of an application has to make an *ad hoc* proof of consistency and an *ad hoc* implementation (i.e., a theorem prover) every time he designs 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, it is important that the application that is based on a logic engine, and the logic engine itself, be designed separately. This implies that the interface of the logic engine should not depend on the logic itself. This is what we call *embeddability* of the logical component.

If we need to separately design the application and its logical components, then who should develop the embedded logic components?

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 Actor may gather 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.

The first Actor is the Theorist; he invents an abstract framework, like, for instance, relational algebra, lattice theory, or logic.

If the abstract framework has applications, then a second Actor, the System Programmer, implements (part of) the theory in a *generic* system for these 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 objective 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.

It is the relation between the System Programmer and the Application Designer that interests us.

2.3 Genericity and instantiation

Genericity is often achieved by designing a language: e.g., a data-base schema language, a lattice operation language, and a programming language. Correspondingly, 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, on one hand the System Programmer is competent for building a logic subsystem, but he does not know the application; he only knows the range of applications. On the other hand 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.

We explore how to systematically build 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 logic functors we describe in this article have also a concrete implementation as an actual program. We have also implemented a *logic composer* that takes the description of a customized logic and builds a concrete logic component.

2.4 Customized logics

The range of logic functors can be very large. In this article we consider only sums of logics, propositions (on arbitrary formulas), intervals, 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 is geared towards manipulating logics as lattices, as in abstract interpretation. 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, but they 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 also that in our framework not all logics are lattices (nor their deduction relation is a partial order), but the most interesting ones can always be completed in a lattice.

We will consider as a motivating example an application for dealing with bibliographic entries. Each entry has a description made of its author name(s), title, type of cover, publisher, and date. The User navigates through a set of entries by comparing descriptions with queries that are written in the same language. 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 the query:

```
title: contains "Jungle"
```

is:

```
hard-cover     publisher:"Century Co."     year: 1900..1950
paper-back     publisher:"Penguin"         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 specific 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 $\mathtt{e}_1$ such that `descr(`$\mathtt{e}_1$`)`$\models$`x`, and
2. there exists a possible answer $\mathtt{e}_2$ such that `descr(`$\mathtt{e}_2$`)`$\not\models$`x`.

We will not go any further in the description of this application (see [FR00b,FR01]). We simply note that:

1. descriptions, queries, and answers belong to the same logical language, which combines logical symbols and expressions such as strings, numbers, or intervals, and
2. one can design a similar application with a different logic, e.g., for manipulating software components. Thus, it is important that all different logics share a common interface for being able to separately write programs for the navigation system and for the logic subsystem it uses.

2.5 Summary

We define tools for building *automatic* theorem provers for *customized logics* for allowing Users who are not sophisticated logic actors. Note also that the User may be a program itself: e.g., a mobile agent running on a host system [IB96]. This rules out interactive theorem provers.

Validating a theorem prover built by using our tools must be as simple as possible. We want this because the Application designer, though it may be 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. Thus, the logic components can be safely embedded in applications as black-boxes.

3 Logics and logic functors

If an Application Designer has to define a customized logic by the means of composing primitive components, these components should be of a '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. Thus, we decided to define logical components that are very close to be logics themselves.

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 generate 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, called *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. All proofs are omitted. They are given in the companion research-report [FR02].

Definition 1 (Syntax) *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 (Semantics) *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 $M(f) \subseteq M(g)$.*

The entailment relation is never used formally in this paper, but we believe it provides a good intuition for the frequent usage in proofs of the inclusion of sets of models.

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

Definition 3 (Implementation) *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*,*
- *$\sqcap, \sqcup \in AS \times AS \rightarrow AS \cup \{undef\}$ 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 (Logic) *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 (Total/partial, bounded/unbounded) *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. Indeed, they do not 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 (Completeness/consistency) *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 matching between the expected answers, specified by the semantics, and actual answers, specified by the implementation. Thus, in these preferred logics deduction can be extended into a partial order. However, some logics defined on concrete domains are not complete. An important issue is how to build complete logics with components that are not complete.

We must add to the five operations of an implementation, a parser and a printer for handling the concrete syntax of formulas. Indeed, an application should input and output formulas in a readable format. However, we do not consider them further, because they do not determine any logical problem. On the contrary, the five logical operations (deduction, conjunction, disjunction, tautology, and contradiction) are at the core of the logics we consider.

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 reasons of uniformity, logics are considered as logic functors with arity 0 (a.k.a. atomic functors, or nullary logic functors).

Let $\mathbb{AS}$ be the class of all syntaxes, $\mathbb{S}$ be the class of all semantics, and $\mathbb{P}$ be the class of all implementations. The syntax of a logic functor is simply a function from the syntaxes of the logics which are its arguments, to the syntax of the resulting logic.

Definition 7 (Logic functor) *A logic functor F is a triple (AS_F, S_F, P_F) where*

- *the abstract syntax AS_F is a function of type $\mathbb{AS}^n \rightarrow \mathbb{AS}$, 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 $\mathbb{S}^n \rightarrow \mathbb{S}$, 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 $\mathbb{P}^n \rightarrow \mathbb{P}$, 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* $\rightarrow$ *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 (Syntax) *AS_{atom} is a set of atom names.*

Definition 9 (Semantics) *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 (Implementation) *P_{atom} is $(\sqsubseteq, \sqcap, \sqcup, \top, \bot)$ where for every $a, b \in AS_{atom}$*
- *$a \sqsubseteq b$ iff $a = b$*
- $a \sqcap b = a \sqcup b = \begin{cases} a & \textit{if } a = b \\ \textit{undef} & \textit{otherwise} \end{cases}$
- *$\top$ and $\bot$ are undefined.*

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

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 $\mathit{Prop}(A)$ 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*

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

Definition 14 (Implementation) *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 $\neg$, but 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

$\top$-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:	$\frac{a \sqcap_A b, \Delta \vdash \Gamma}{a, b, \Delta \vdash \Gamma}$	if $a \sqcap_A b$ is defined		
$\sqcup$-Rule:	$\frac{\neg(a \sqcup_A b), \Delta \vdash \Gamma}{\neg a, \neg b, \Delta \vdash \Gamma}$	if $a \sqcup_A b$ is defined		
$\neg\neg$-Rule:	$\frac{\Delta \vdash X, \Gamma}{\Delta \vdash \neg\neg X, \Gamma}$		literal-Rule:	$\frac{\overline{L}, \Delta \vdash \Gamma}{\Delta \vdash L, \Gamma}$
β-Rule:	$\frac{\Delta \vdash \beta_1, \beta_2, \Gamma}{\Delta \vdash \beta, \Gamma}$		α-Rule:	$\frac{\Delta \vdash \alpha_1, \Gamma \qquad \Delta \vdash \alpha_2, \Gamma}{\Delta \vdash \alpha, \Gamma}$

Table1. Sequent calculus for deduction in propositional logic.

logic functor, though the interface does not refer to 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 (Validity) *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.*

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

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.*

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 (Openness) *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.*

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 $\bigcap_{a \in A} M_A(a) \subseteq \bigcup_{b \in B} M_A(b)$.*

Definition 21 (Validity) *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 (Reducedness) *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 $f \vee \neg g$ 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. $S_{prop(A)}$.*

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 (Reducedness) *P_{atom} is reduced w.r.t. S_{atom}.*

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

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.

Reducedness formalizes the informal notion of an implementation being defined enough. Thus, it seems that it is useful to define it as a coherence relation between the semantics and the implementation, *via* a notion of maximaly defined implementation.

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. Note that 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 contrary to functors in ML which 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 functors. We could have implemented logic functors in a programming language that offers ML-like functors, but we did not so, mainly for the sake of compatibility with the rest of our application that was already written in λProlog.

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. The theory of institutions is developed using category theory and in that theory there are functors from signatures to formulas, from signatures to models, and from institutions to institutions. 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 *interv* (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.

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 includes a logic composer that reads logic specifications such as $sum(prop(atom), prop(interv))$ (sums of propositions on atoms and propositions on intervals) and automatically produces a printer, a parser, and a theorem

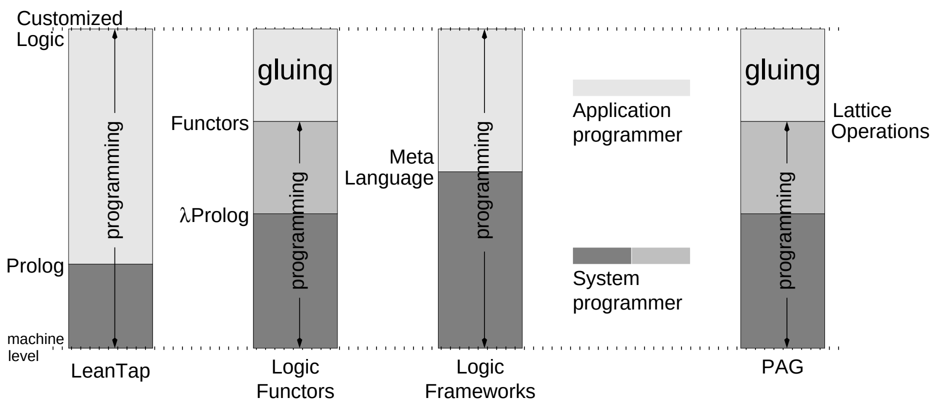

Figure1. Several related works and the respective tasks of the System Programmer and the Application Designer

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.

Our paper suggests a software architecture for logic-based systems, in which the system is generic in the logic, and the logic component can be separately defined, 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, because they are useful for making queries and answers more compact.

Another possible extension 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*.

Acknowledgements: We are pleased to acknowledge the careful reading of this article by Alberto Pettorossi. Remaining mistakes are ours.

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.

[FR02] S. Ferré and O. Ridoux. Logic functors: a framework for developing embeddable customized logics. Rapport de recherche 4457, INRIA, 2002.

[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 database 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 (Syntax) $AS_{interv} = \{[x, y] \mid x, y \in \mathbb{R} \uplus \{-, +\}\}$.

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 (Semantics) *S_{interv} 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, $M_{interv}([+, -]) = \emptyset$.

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

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

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

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

Theorem 31 (Completeness/consistency) *P_{interv} is consistent and complete in $\sqsubseteq$, $\top$, $\bot$, $\sqcap$, $\sqcup$ w.r.t. S_{interv}.*

P_{interv} 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 (Reducedness) *P_{interv} is reduced w.r.t. S_{interv}.*

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

The optional symbol ^ 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 ^*s*, and *is s* is written ^*s*\$. The symbol # denotes the empty language (matched by no string).

Definition 34 (Semantics) *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.*

Note also that only formula # has an empty model.

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

– $f \sqsubseteq g$ *iff* $f = \alpha g \beta$,

– $f \sqcap g = \begin{cases} f & \textit{if } f \sqsubseteq g \\ g & \textit{if } g \sqsubseteq f \\ \# & \textit{if } f \not\sqsubseteq g \textit{ and } g \not\sqsubseteq f \textit{ and both } f \textit{ and } g \textit{ are} \\ & \textit{either left-bounded or right-bounded, or one of them is bounded} \\ undef & \textit{otherwise} \end{cases}$

– $\sqcup$ *is undefined,*

– $\top = \epsilon$,

– $\bot = \#$.

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

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 (Reducedness) *P_{string} is reduced w.r.t. S_{string}.*

B More n-ary logic functors

We present in this appendix some more n-ary logic functors. Some of them produce reduced logics that are not necessarily total. In this case, partiality is not a problem, since it is enough to wrap them in logic functor *prop*. A few other functors produce logics that are not reduced, but that are total (if the logics to which they are applied are also total). They are useful, 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.

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

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

Definition 41 (Implementation) *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*
- $f \sqcap g = \begin{cases} f_d \sqcap_d g_d & \textit{if } f = f_d \textit{ and } g = g_d \\ f & \textit{if } f \sqsubseteq g \\ g & \textit{if } g \sqsubseteq f \\ \bot & \textit{if } f = [f_d] \not\sqsubseteq g \\ \bot & \textit{if } g = [g_d] \not\sqsubseteq f \end{cases}$
- $f \sqcup g = \textit{undef}$
- $\top = \top_d$
- $\bot = \bot_d$.

$\sqsubseteq$	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 $f_d \sqsubseteq_d \bot_d$

Table2. Definition of logical deduction in logic functor *aik*.

Theorem 42 (Completeness/consistency) *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).

Lemma 43 (Reducedness) *$P_{aik(L_d)}$ 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.*

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:*
$AS_{valattr(L)} = \{a : f \mid f \in L \wedge a \in \mathit{Attr}\}$

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 (Completeness/consistency) *$P_{valattr(V)}$ 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.*

$P_{valattr}$ 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 (Reducedness) *$P_{valattr(V)}$ is reduced w.r.t. $S_{valattr(V)}$ if P_V is reduced w.r.t. S_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 (Syntax) *AS_{sum} forms the disjoint union of two logics plus formulas ? and #.*

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

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

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 (Implementation) *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*

- $$f \sqsubseteq g = \begin{cases} f \sqsubseteq_{L_1} g & \textit{if } f, g \in AS_{L_1} \\ f \sqsubseteq_{L_2} g & \textit{if } f, g \in AS_{L_2} \\ \textit{true} & \textit{if } f \sqsubseteq_{L_1} \bot_{L_1} \textit{ or } f \sqsubseteq_{L_2} \bot_{L_2} \textit{ or } f = \# \textit{ or } g = ? \\ \textit{false} & \textit{otherwise} \end{cases}$$
- $$f \sqcap g = \begin{cases} f \sqcap_{L_1} g & \textit{if } f, g \in AS_{L_1} \\ f \sqcap_{L_2} g & \textit{if } f, g \in AS_{L_2} \\ f & \textit{if } g = ? \\ g & \textit{if } f = ? \\ \# & \textit{otherwise} \end{cases}$$
- $$f \sqcup g = \begin{cases} f \sqcup_{L_1} g & \textit{if } f, g \in AS_{L_1} \\ f \sqcup_{L_2} g & \textit{if } f, g \in AS_{L_2} \\ g & \textit{if } f = \# \textit{ or } f \sqsubseteq_{L_1} \bot_{L_1} \textit{ or } f \sqsubseteq_{L_2} \bot_{L_2} \\ f & \textit{if } g = \# \textit{ or } g \sqsubseteq_{L_1} \bot_{L_1} \textit{ or } g \sqsubseteq_{L_2} \bot_{L_2} \\ ? & \textit{if } f = ? \textit{ or } g = ? \\ ? & \textit{if } f \in AS_{L_1}, g \in AS_{L_2} \textit{ and } \top_{L_1} \sqsubseteq_{L_1} f \textit{ and } \top_{L_2} \sqsubseteq_{L_2} g \\ ? & \textit{if } f \in AS_{L_2}, g \in AS_{L_1} \textit{ and } \top_{L_1} \sqsubseteq_{L_1} g \textit{ and } \top_{L_2} \sqsubseteq_{L_2} f \\ \textit{undef} & \textit{otherwise} \end{cases}$$

– $\top$ = ? $\qquad\qquad$ $\bot$ = #

Theorem 52 (Completeness/consistency) *$P_{sum(L_1,L_2)}$ 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}.*

Lemma 53 (Reducedness) *$P_{sum(L_1,L_2)}$ 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}.*

B.4 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 54 (Syntax) *AS_{set} is the set of finite subsets of formulas of a logic.*

Definition 55 (Semantics) *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 56 (Implementation) *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 57 (Completeness/consistency) *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.*

The logic functor *set* is not reduced but it is still useful as the outermost functor of a composition.