The Principles of a Toolbox for the Implementation of 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 a mix of several logics to form structures whose status is unclear. Implementing them requires to build a parser, a printer, and a theorem-prover which are often themselves a mix of the corresponding operations for the various logic components. We present the principles of the composition of logic functors for constructing logics that are *ad-hoc*, but sound. An important issue is how the connectives of different sub-logics inter-operate. We propose a formalization of the logic functors, their semantics, implementations, proof-theoretic properties, and their composition. In particular, we show under which conditions a logic with partially defined connectives can be composed to form a logic with totally defined connectives.

1 Motivation

Several Information Processing domains have a logic-based branch in which logic plays the main role: logic-based information retrieval [SM83,vRCL98,FR01], logic-based diagnosis [Poo88], logic-based programming [Llo87,MS98], logic-based program analysis [SFRW98,AMSS98,CSS99], etc. The logic in use is often not defined by a single pure deduction system, but rather is a mix of several logics, or a mix of 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 (s)he imagines a new *ad hoc* logic.

In this article, we explore the systematic construction of logics using basic components that we call *logic functors*. By "construction of a logic" we mean the definition of its syntax, its semantics via deduction rules, and its implementation as a theorem prover. So, we define a process that goes from the description of a logic in terms of logic functors to an implementation of the logic. The process in itself is neither new nor difficult, but it is not our main focus. We insist more on the logical validity of the process.

The range of logic functors can be very large. We consider in this article only products, sums, the boolean algebra, and intervals. However, other logic functors have been integrated in the toolbox, namely sets (of models), valued attributes (abstracted w.r.t. values), strings (e.g., "begin with", "contains"), and a variant

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

of All I Know (a modal epistemic logic [Lev90]). The whole development is geared towards manipulating logics as lattices, as in abstract interpretation.

All proofs are omitted in this extended abstract.

2 Logic and Logic Functor

In this section, we define a logic as a set of formulas L, to which can be associated a semantics and an implementation. Formulas can be seen as terms that we want to construct using basic components called "logic functors". More formally, if $\mathbb{L}$ is the domain of logics, then logic functors are functions of type $\mathbb{L}^n \to \mathbb{L}$. Logic functors with arity 0 are also called logics.

In the following subsections, we define notions of *semantics* and *implementation* for logics and logic functors. We also define notions of *consistency* and *completeness* between semantics and implementations.

2.1 Semantics of Logic and Logic Functor

Def. 1 *A* semantics S_L *of a logic* L *is defined as a couple* $(I, \models)$*, where*
– I *is the domain of* interpretations*,*
– $\models: I \times L \to \{0,1\}$ *is a satisfaction relation between interpretations and formulas;* $i \models f$ *means that* i *is a* model *of* f*.*

The domain of semantics is denoted $\mathbb{S}$ and, for every formula $f \in L$, $M_L(f) := \{i \in I \mid i \models f\}$ denotes the set of all models of formula f. For every semantics S_L, a *deduction relation* is defined for every formulas $f, g \in L$: f entails g iff $M_L(f) \subseteq M_L(g)$.

Def. 2 *The semantics* S_F *of a logic functor* F *is a function of type* $\mathbb{S}^n \to \mathbb{S}$*, such that* $S_{F(L_1,..,L_n)} = S_F(S_{L_1},..,S_{L_n})$*.*

2.2 Implementation of Logic and Logic Functor

Def. 3 *An* implementation P_L *of a logic* L *is a 5-tuple* $(\sqsubseteq, \sqcap, \sqcup, \top, \bot)$*, where*
– $\sqsubseteq \in L \times L \to \{0,1\}$ *is the deduction relation,*
– $\sqcap, \sqcup \in L \times L \to L \uplus \{undef\}$ *are the conjunction and the disjunction,*
– $\top, \bot \in L \uplus \{undef\}$ *are the top and the bottom.*

The domain of implementations is denoted $\mathbb{P}$. The above definition shows that operations $\sqcap$, $\sqcup$ can be partially defined, and that operations $\top$, $\bot$ can be undefined. In the opposite case, a logic is respectively called *total* and *bounded.* Total logics are usually preferred and even necessary, but Section 3.2 shows that the propositional functor applied to a partial logic always constructs a total logic.

To the five above operations, we must add at least a parser and a printer for handling the concrete syntax of formulas. But we do not consider them further as they do not cause any logical problem. On the contrary, the five logical operations (deduction, conjunction, disjunction, top, and bottom) are at the core of a constructed logic. They make sense according to consistency and completeness with regard to the chosen semantics for L. These notions are defined respectively for each operation of an implementation.

Def. 4 *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 $f, g \in L$*
– *$op = \sqsubseteq$: $f \sqsubseteq g \Longrightarrow M_L(f) \subseteq M_L(g)$ (resp. $M_L(f) \subseteq M_L(g) \Longrightarrow f \sqsubseteq g$),*
– *$op = \top$: $\top$ is defined $\Longrightarrow M_L(\top) = I$ (resp. always complete),*
– *$op = \bot$: $\bot$ is defined $\Longrightarrow M_L(\bot) = \emptyset$ (resp. always complete),*
– *$op = \sqcap$: $f \sqcap g$ is defined $\Longrightarrow M_L(f \sqcap g) \supseteq M_L(f) \cap M_L(g)$ (resp. $f \sqcap g$ is defined $\Longrightarrow M_L(f \sqcap g) \subseteq M_L(f) \cap M_L(g)$),*
– *$op = \sqcup$: $f \sqcup g$ 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.

Def. 5 *The implementation P_F of a logic functor F is a function of type $\mathbb{P}^n \to \mathbb{P}$, such that $P_{F(L_1,..,L_n)} = P_F(P_{L_1}, .., P_{L_n})$.*

The aim of Section 3 is to show for each logic functor under which conditions its implementation is total, consistent, and complete.

3 Examples of Logic Functors

This section illustrates the notions introduced in Section 2 with some examples of logic functors: *product* and *sum* of logics, and the well-known *propositional logic*, but abstracted over atoms. In each case, we present the syntax, the semantics, the implementation and results about consistency and completeness.

3.1 Product and Sum of Logics

We use `prod` to denote 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.

Def. 6 (Syntax) *The logic functor `prod` is defined for every logics $L_1, L_2 \in \mathbb{L}$ by $\mathtt{prod}(L_1, L_2) = L_1 \times L_2$ (cartesian product).*

Def. 7 (Semantics) *The semantics of `prod` is defined by the semantic functor $S_{prod} : (I_1, \models_1), (I_2, \models_2) \mapsto (I, \models)$ such that $I = I_1 \times I_2$ and $(i_1, i_2) \models (f_1, f_2)$ iff $i_1 \models_1 f_1$ and $i_2 \models_2 f_2$.*

Def. 8 *The implementation of `prod` is defined by the implementation functor $P_{prod} : (\sqsubseteq_1, \sqcap_1, \sqcup_1, \top_1, \bot_1), (\sqsubseteq_2, \sqcap_2, \sqcup_2, \top_2, \bot_2) \mapsto (\sqsubseteq, \sqcap, \sqcup, \top, \bot)$ such that*
– *$(f_1, f_2) \sqsubseteq (g_1, g_2)$ iff $f_1 \sqsubseteq_1 g_1$ and $f_2 \sqsubseteq_2 g_2$, or $f_1 \sqsubseteq_1 \bot_1$, or $f_2 \sqsubseteq_2 \bot_2$,*
– *$(f_1, f_2) \sqcap (g_1, g_2) = (f_1 \sqcap_1 g_1, f_2 \sqcap_2 g_2)$ (or $undef$ if one conjunction is),*

– $(f_1, f_2) \sqcup (g_1, g_2) = (f_1 \sqcup_1 g_1, f_2 \sqcup_2 g_2)$ *(or* $undef$ *if one disjunction is),*
– $\top = (f_1, f_2)$ *such that* $\top_1 \sqsubseteq_1 f_1$ *and* $\top_2 \sqsubseteq_2 f_2$*,*
– $\bot = (f_1, f_2)$ *such that* $f_1 \sqsubseteq_1 \bot_1$ *or* $f_2 \sqsubseteq_2 \bot_2$*.*

Theorem 1 *Implementation* $P_{prod}(P_1, P_2)$ *is consistent and complete in operations* $\sqsubseteq, \top, \bot, \sqcap$ *and only complete in* $\sqcup$ *w.r.t. semantics* $S_{prod}(S_1, S_2)$ *if* P_1 *and* P_2 *are so w.r.t.* S_1 *and* S_2*.*

The definition of P_{prod} shows that the implementation $P_{prod}(P_1, P_2)$ is total iff implementations P_1 and P_2 are also total. It is bounded in $\top$ if P_1 and P_2 are; it is bounded in $\bot$ if P_1 or P_2 is (indeed, $M(f_1, f_2) = M_1(f_1) \times M_2(f_2)$). Theorem 1 shows that if P_1 and P_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 1 *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* $\sqcup_1 = \sqcup_2 = \cup$*).*

However, $M_{prod(L_1,L_2)}((\{a\},\{1\}) \sqcup (\{b\},\{2\})) \stackrel{Def\ 8}{=}$
$M_{prod(L_1,L_2)}(\{a\} \sqcup_1 \{b\}, \{1\} \sqcup_2 \{2\}) = \{a, b\} \times \{1, 2\}$ *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 4 on consistency in* $\sqcup$*.*

A logic functor used for constructing the sum of 2 logics can be defined along the same principles. It is total iff implementations P_1 and P_2 are also total. As for product, its implementation $P_{sum}(P_1, P_2)$ is consistent and complete w.r.t. semantics $S_{sum}(S_1, S_2)$ if P_1 and P_2 are w.r.t. S_1 and S_2, except for operation $\sqcup$ that is only complete in general.

3.2 Propositional Logic Abstracted over Atoms

Let us assume that we use a logic in an application. It is almost always the case that we want to express conjunction. Similarly, disjunction and negation could be used, especially to express incomplete information. So, the choice of propositional logic is natural. Propositional logic is defined by taking a set of *atoms* A to form a set of propositional formulas $Prop(A)$ by the closure of A for the three boolean connectives.

$Prop(A)$ is usually considered as a free boolean algebra, since there is no *a priori* relations between the atoms. I.e., they are all pairwise incomparable for the deduction order. However, in concrete applications atoms are often comparable. For instance, boolean queries based on string matching uses 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* `"abc"` implies *ends with* `"bc"`, which implies *contains* `"b"`.

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.

Def. 9 (Syntax) *The logic functor* `prop` *is defined for every logic of atoms* L_A *as the closure of* L_A *by the boolean connectives* $\wedge$, $\vee$, *and* $\neg$.

Def. 10 (Semantics) *The semantics of* `prop` *is defined by the semantic functor* $S_{prop} : (I_A, \models_A) \mapsto (I, \models)$. *The satisfaction relation is defined by structural induction on every formula* f *of* `prop`(L_A).

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

Def. 11 *The implementation of* `prop` *is defined by the implementation functor* $P_{prop} : (\sqsubseteq_A, \sqcap_A, \sqcup_A, \top_A, \bot_A) \mapsto (\sqsubseteq, \sqcap, \sqcup, \top, \bot)$ *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 [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,* $\overline{f}$ *denotes the negation of* f ($\overline{f} := \neg f$ *and* $\overline{\neg f'} := f'$), β *is either* $\beta_1 \vee \beta_2$ *or* $\neg(\overline{\beta_1} \wedge \overline{\beta_2})$, α *is either* $\alpha_1 \wedge \alpha_2$ *or* $\neg(\overline{\alpha_1} \vee \overline{\alpha_2})$.
– $f \sqcap g = f \wedge g$,
– $f \sqcup g = f \vee g$,
– $\top = a \vee \neg a$, *for any* $a \in L_A$,
– $\bot = a \wedge \neg a$, *for any* $a \in L_A$.

$\top$-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:	$\dfrac{a \sqcap_A b, \Delta \vdash \Gamma}{a, b, \Delta \vdash \Gamma}$	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.

Theorem 2 *Implementation* $P_{prop}(P_A)$ *is consistent in* $\sqsubseteq, \top, \bot, \sqcap, \sqcup$ *w.r.t. semantics* $S_{prop}(S_A)$ *if* P_A *is consistent w.r.t.* S_A.

Rules $\top$-Axiom, $\perp$-Axiom, $\sqsubseteq$-Axiom, $\sqcap$-Rule, and $\sqcup$-Rule play the role of an 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. However, 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.

Def. 12 *A sequent is* open *iff it is neither the conclusion of a rule nor an axiom.*

We define a notion of reduced set of atoms, and reduced semantics, that is a necessary and sufficient condition for completeness.

Def. 13 *An atomic implementation P_A is* reduced *w.r.t. S_A iff every* open *sequent is non-*valid*, i.e.:*

$$\bigcap_{a \in \Delta} M_A(a) \not\subseteq \bigcup_{\neg b \in \Delta} M_A(b).$$

We prove that if an implementation P_A is reduced, then it is complete in $\sqsubseteq_A$. In case it is not complete, there exist two atoms a, b such that $M_A(a) \subseteq M_A(b)$ and $a \not\sqsubseteq_A b$. Then, sequent $a, \neg b \vdash$ is both open and valid, and P_A is not reduced.

The following theorem shows that to be reduced is a sufficient condition for implementations of logics of atoms to make P_{prop} complete.

Theorem 3 *Implementation $P_{prop}(P_A)$ is complete in $\top, \perp, \sqcap, \sqcup$ w.r.t. $S_{prop}(S_A)$. It is also complete in $\sqsubseteq$ if P_A is complete in $\sqcup$, $\sqcap$ and reduced w.r.t. S_A.*

4 Examples of Logics of Atoms

Atom logics are often designed for representing concrete observations on a domain. They serve as a language to write atomic formulas. In order to serve as arguments to the logic functor `prop` (or other similar boolean functors if available), they must be equipped with a "natural" conjunction and "natural" disjunction, i.e., they must be consistent and complete in $\sqcap, \sqcup$ (cf. Definition 4). 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, the logic functor `prop` takes such an atom logic, and build a logic that is always total and bounded (Definition 11). It also provides a consistent and complete implementation if the atom logic also has a consistent, complete, and reduced implementation (Theorems 2 and 3). So, for every atom logic presented in this section, we prove that its implementation is consistent, complete, and reduced.

First, a logic of atoms with a flat structure (comparable iff identical) can be defined such that `prop(atom)` corresponds to standard propositional logic. It is made of an implementation P_{atom} that is consistent and complete w.r.t. a semantics S_{atom}. Implementation P_{atom} is not bounded and is partial in both conjunction and disjunction, but it is consistent, complete, and reduced w.r.t. S_{atom}. So, `atom` can serve as an argument of the logic functor `prop`.

Intervals are a good example of non-standard atoms. They are often used to express incomplete knowledge: e.g., incomplete knowledge of the value of a variable. We define a logic of intervals by the set $L_{interval} = \{[x, y] \mid x, y \in \mathbb{R}\}$.

Def. 14 $S_{interval} = (I, \models)$ *where* $I = \mathbb{R}$ *and* $i \models [x, y] \iff x \leq i \leq y$.

Def. 15 $P_{interval} = (\sqsubseteq, \sqcap, \sqcup, \top, \bot)$ *where forall* $[x_1, y_1], [x_2, y_2] \in L_{interval}$
- $[x_1, y_1] \sqsubseteq [x_2, y_2]$ *iff* $x_2 \leq x_1$ *and* $y_1 \leq y_2$,
- $[x_1, y_1] \sqcap [x_2, y_2] = [max(x_1, x_2), min(y_1, y_2)]$,
- $[x_1, y_1] \sqcup [x_2, y_2] = \begin{cases} [min(x_1, x_2), max(y_1, y_2)] & \textit{if } x_2 \leq y_1 \textit{ and } x_1 \leq y_2 \\ undef & \textit{otherwise} \end{cases}$,
- $\top = undef$,
- $\bot = [x, y]$ *for any* $x, y \in \mathbb{R}$ *such that* $y < x$.

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

Lemma 1 $P_{interval}$ *is consistent and complete w.r.t.* $S_{interval}$.

Implementation $P_{interval}$ is not bounded in top and is partial in disjunction, but it is consistent and complete. Furthermore, the following lemma shows that it is reduced, and so, can serve as an argument of the logic functor `prop`.

Lemma 2 *The implementation* $P_{interval}$ *is reduced w.r.t. semantics* $S_{interval}$.

5 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 processed to produce a theorem-prover. We have listed a number of logic functors, but many others can be built.

One of our main goals 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 functors behave: total/partial, consistent/complete, and reduced/unreduced.

This concern is complemantary 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]. These frameworks allow users to implement a customized logic, but do not help users in proving the completeness of the resulting theorem prover.

In essence, this work is more similar to works on static program analysis toolbox (e.g., PAG [AM95]), where authors assemble known results to combine domain operators like product, sets of, lists in order to build abstract domains and derive automatically a solver for these domains. The fact that we consider deduction relations as (partial) lattices is another connection with these works.

The principle of composing logic functors has been implemented in a prototype. It reads logic specifications such as `prod(prop(atom),prop(interv))`. This example means that logic formulas are products of a proposition on atoms and a proposition on intervals. The prototype reads such a constructed logic and builds a theorem-prover for it by instantiating the theorem-prover associated to each logic functor every time the functor is used. The prototype, each 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 [FR00].

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.

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

[Fit98] M. Fitting. leanT^AP revisited. *J. Logic and Computation*, 8(1):33–47, 1998.

[FR00] 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.

[FR01] Sébastien Ferré and Olivier Ridoux. Searching for objects and properties with logical concept analysis. In Harry S. Delugach and Gerd Stumme, editors, *International Conference on Conceptual Structures*, LNCS 2120, pages 187–201. Springer, 2001.

[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.

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

[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.

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

[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.

[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.