

# Logic Functors for Types as Search Keys

Soazig Bars<sup>1</sup>, Sébastien Ferré<sup>2</sup> and Olivier Ridoux<sup>3</sup>  
IRISA<sup>123</sup>/CNRS<sup>2</sup>/IFSIC<sup>3</sup>

25th September 2002

## Abstract

Since the works of Runciman, Toyn, Rittri, and Di Cosmo, types have been recognized as possible keys for searching software components in a repository. This idea introduces new problems like (1) defining a partial ordering on types that permits a relevant ranking of queries and answers, (2) reflecting the various type constructions on a search metaphor, and (3) controlling the number of answers. We propose a methodology based on logic components that can be combined to form a specialized logic for modeling type entailment. These components can be also combined with other non type-oriented logic components to extend the type as search key paradigm with other kinds of descriptions.

## 1 Introduction

Software reuse is recognized as a good practice because it factorizes costs, and because an improvement in a reused software may benefit to all softwares that use it. A corollary of software reuse is software retrieval. Assuming a repository of software components, software retrieval helps in finding software components using a (partial) description of what one needs. We assume in the following that software components are functions or methods.

A crucial question in information retrieval is the choice of the search keys. The basic idea is that a retrieval query defines a search key, and that an answer to a query is built from the set of “things” that “satisfy” the query. In our case, “things” are software components, and “satisfaction” is a relation between software components and search keys. For instance, Wing and Rollins have proposed to use specifications as search keys for software retrieval [RW91]. The only drawback of this method is that specifications are often expressed in undecidable fragment of predicate logic. So, we follow on other works in which search keys are types.

Using types as search keys does not make the entailment relation trivial either. For instance, if the key is  $list(char) \rightarrow char$  a function  $f$  with type  $list(char) \rightarrow int \rightarrow char$  must be considered as a possible answer, as well as a function  $g$  with type  $list(\alpha) \rightarrow \alpha$ . Runciman and Toyn [RT91] have proposed several partial orders on types for supporting the entailment relation. One such

partial order supports the addition of *extra-arguments* like in the case of function  $f$ ; a function with more parameters than required satisfies a type. Another partial order supports the generalization to universal polymorphic functions like in the case of function  $g$ ; a function whose type generalizes the search key satisfies it. However, the combination of these two partial orders leads to paradoxes:

$$(\alpha \rightarrow \alpha) \stackrel{\text{polymorphism}}{=} (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha) \stackrel{\text{extra-argument}}{=} (\alpha \rightarrow \alpha)$$

Rittri proposed to formalize what types that satisfy the same query have in common via the notion of *type isomorphism* [Rit91]. He defined it using equivalence axioms like  $A \times B \equiv B \times A$  or  $(A \times B) \rightarrow C \equiv A \rightarrow B \rightarrow C$ . Then Di Cosmo [Di 92, Di 95] introduced a semantic definition of type isomorphism. So doing, he could prove correct and complete several axiom systems for various type systems (functions, functions and products, functions, products and polymorphism, etc). However, type isomorphism is an equivalence relation; it cannot handle the dissymetry that exists between queries and answers. An answer satisfies a query, but not the converse.

We present in this article a new formalization of type entailment using logic functors and slightly different hypothesis. Logic functors define software components that can be composed to form programs that implement different facets of a logic-based tool: e.g., parsing and printing formulas, and proving entailment. They are presented in more details in Section 2.2. Logic functors can be used everywhere a software system has a logic component. For instance, a logic-based information retrieval system can be designed using an abstract entailment relation, and then can be instantiated by plugging in a logic component built with logic functors. In Sections 2.3 and 3, we will design logic functors for modeling the entailment relation between types, and then we will plug them in a generic information system.

The result is a sound formalization of type entailment, a correct implementation of it which is produced automatically, and a retrieval system for software components that is obtained by plugging the logic functors in an existing information retrieval system. The advantage is a robust design process, and a flexible retrieval system since the new logic functors for types can be combined with already existing logic functors that can model other facets of software components. So, we obtain at a rather low cost a multi-faceted software components retrieval system.

## 2 Type entailment

### 2.1 Basic principles

The previous works on type entailment and type isomorphism were based on functional language *à la* ML, and on typed  $\lambda$ -calculi in the more formal works. So, universal polymorphism and higher-order took a great importance in these works.

Our main change of hypothesis is in basing our work on the family of object-oriented language. So, it is inheritance polymorphism that takes the advantage

now. Similarly, we stick to first-order methods. Other type constructors are kept as in previous works, function type ( $\rightarrow$ ), and product type ( $\times$ ).

We also depart from the usual notations for function type and product. The usual notations for function type  $A \rightarrow B$ , and products  $A \times B$  prevent from talking of  $A$  without mentioning  $B$ . However, this is something one wants to do when using types as search keys; e.g., “give me a function that returns a file descriptor”. So, instead of notations  $A \rightarrow B$  and  $A \times B$  we will write  $\{?A, !B\}$  and  $\{*A, *B\}$  where  $*$  is a  $?$  if  $A \times B$  is a parameter type, and  $*$  is a  $!$  if  $A \times B$  is a result type. To model the difference between  $A \times A$  and  $A$  the  $\{\dots\}$  will be interpreted as multi-sets.

More formally, the change in notation is defined as follows, where the  $\mathcal{T}$ ,  $\mathcal{RT}$  and  $\mathcal{PT}$  are the non-terminal symbols that describe the usual concrete syntax, and the subscripted  $t$ ,  $pt$  and  $rt$  are fragments of our notation.

$$\begin{aligned} \mathcal{T}_{\{pt,rt\}} &::= \mathcal{PT}_{pt} \rightarrow \mathcal{RT}_{rt} \\ \mathcal{T}_{!t} &::= \text{base types}_t \\ \mathcal{PT}_{t,t'} &::= \mathcal{PT}_t \times \mathcal{PT}_{t'} \\ \mathcal{PT}_{?t} &::= \text{base types}_t \\ \mathcal{RT}_{!t} &::= \text{base types}_t \end{aligned}$$

So,  $!$  and  $?$  denotes polarities, and our notation is simply a clausal form for first-order types.

Finally, we assume a set of base types partially ordered by an inheritance relation.

## 2.2 Logic functors

Logic functors were developed for designing ad-hoc logics and have their implementation (e.g., parsing, printing, theorem-proving) automatically derived [FR01a]. We first define *logics* and then *logic functors* that operate on logics.

**Definition 1 (logic)** *A logic  $L$  is a triple  $(AS_L, S_L, P_L)$  where  $AS_L$  defines the abstract syntax of formulas of  $L$ ,  $S_L$  defines their semantics, and  $P_L$  defines their interface.*

*$S_L$  is a pair  $(I_L, \models_L)$  where*

- $I_L$  is the interpretation domain of the formulas of  $L$ , and
- $\models_L \in \mathcal{P}(I_L \times AS_L)$  is the satisfaction relation between interpretations and formulas;  $i \models_L f$  means that  $i$  is a model of  $f$ . We write  $M_L(f) = \{i \in I_L \mid i \models_L f\}$  the set of all models of a formula  $f$ .

*$P_L$  is a 5-uple  $(\sqsubseteq_L, \sqcup_L, \sqcap_L, \top_L, \perp_L)$  where*

- $\sqsubseteq_L \in \mathcal{P}(AS_L \times AS_L)$  is the entailment relation,
- $\sqcup_L, \sqcap_L \in AS_L \times AS_L \rightarrow (AS_L \cup \{\text{undef}\})$  are conjunction and disjunction, and
- $\top_L, \perp_L \in AS_L \cup \{\text{undef}\}$  are the tautology and contradiction for  $L$ .

All logics built with logic functors present the same interface  $(\sqsubseteq_L, \sqcup_L, \sqcap_L, \top_L, \perp_L)$ , plus other operations like parsing and printing. So, they implement the same abstract data-type. The 5 logic operations considered here correspond to a minimal requirement about logics; the ability to test entailment, to build new formulas using conjunction and disjunction, and to determine if a formula is a tautology or a contradiction. A logic may have more connectives, but they will appear in its abstract syntax, not in its interface.

In order to simplify the presentation, we will only consider operations  $\sqsubseteq_L$ ,  $\sqcap_L$ , and  $\top_L$  in the sequel.

The definition of  $\sqcap_L$  needs not be total. Similarly,  $\top_L$  needs not be defined. Moreover, there is no *a priori* relation between  $S_L$  and  $P_L$ . So, we need a notion of completeness and consistency that takes into account partial definitions.

**Definition 2 (completeness and consistency)** *Let  $L$  be a logic, the operations of its interface  $P_L$ , i.e.,  $\sqsubseteq_L$ ,  $\sqcap_L$ , and  $\top_L$ , are consistent (resp. complete) w.r.t. a semantics  $S_L$ , iff for all formulas  $f, g \in AS_L$  we have*

- $f \sqsubseteq_L g \implies M_L(f) \subseteq M_L(g)$  (resp.  $M_L(f) \subseteq M_L(g) \implies f \sqsubseteq_L g$ ),
- $\top_L$  is always consistent (resp.  $\top_L \neq \text{undef} \implies M_L(\top_L) = I_L$ ),
- $f \sqcap_L g \neq \text{undef} \implies M_L(f \sqcap_L g) \subseteq M_L(f) \cap M_L(g)$   
(resp.  $f \sqcap_L g \neq \text{undef} \implies M_L(f \sqcap_L g) \supseteq M_L(f) \cap M_L(g)$ ),

*An interface  $P_L$  is consistent (resp. complete) w.r.t. a semantics  $S_L$  iff each of its operation is consistent (resp. complete).*

By this definition, a completely *undefined* interface is trivially complete and consistent, but it is useless too. So, the game of designing a new logic is to make it defined enough to be useful, but still complete and consistent.

**Definition 3 (logic functor)** *Assuming  $\mathbb{L}$ ,  $\mathbb{AS}$ ,  $\mathbb{S}$ , and  $\mathbb{P}$  the collections of all logics, abstract syntax, semantics, and interface, a logic functor  $F : \mathbb{L}^n \rightarrow \mathbb{L}$  is a triple  $(AS_F, S_F, P_F)$  defined as follows:*

- $AS_F : \mathbb{AS}^n \rightarrow \mathbb{AS}$  such that  $AS_{F(L_1, \dots, L_n)} = AS_F(AS_{L_1}, \dots, AS_{L_n})$ ,
- $S_F : \mathbb{S}^n \rightarrow \mathbb{S}$  such that  $S_{F(L_1, \dots, L_n)} = S_F(S_{L_1}, \dots, S_{L_n})$ ,
- $P_F : \mathbb{P}^n \rightarrow \mathbb{P}$  such that  $P_{F(L_1, \dots, L_n)} = P_F(P_{L_1}, \dots, P_{L_n})$ ,

By convention, a logic will be considered as a logic functor of arity 0.

Logic functors are used as follows.  $S_L$  describes the semantics; it acts as a specification.  $P_L$  implements an interface; its description must be constructive enough, so that it leads directly to a program. A part of the interface describes how the concrete syntax is parsed and printed (remember that  $AS_L$  is only the abstract syntax). The other part offers logic operations. The user composes a logic by applying logic functors to logics, say  $F(L_1, \dots, L_n)$ , and a *logic composer* takes such an expression and produces automatically the concrete implementation of the logic by gluing together the concrete implementations of  $F$ ,  $L_1, \dots$ , and  $L_n$ . This results in a software component, with a formally specified interface, that can be plugged in any software system that assumes the same interface.

This methodology leads to designing three logic functors for modeling type entailment.

1. A logic functor for modeling the inheritance relation on base types. Let us call it  $inh \in \mathbb{L}$ .
2. A logic functor for modeling the input/output polarity (the ! and ?). Let us call it  $pol \in \mathbb{L} \rightarrow \mathbb{L}$ . It takes a logic as a parameter, wraps its formulas with ! and ?, preserves the entailment relation between ! formulas, and reverses it for ? formulas (i.e.,  $?a$  entails  $?b$  iff  $b$  entails  $a$ ). The latter point models contravariance.
3. A logic functor for modeling multi-sets of formulas of some logic. Let us call it  $mset \in \mathbb{L} \rightarrow \mathbb{L}$ . Using this logic functor we can prove that  $\{a, a\}$  entails  $\{a\}$ ,  $\{a, b\}$  is equivalent to  $\{b, a\}$ , and that  $\{a\}$  entails  $\{b\}$  iff  $a$  entails  $b$ .

Although each of these logic functors has its own interest, it is the combination  $mset(pol(inh))$  that will model our notion of type entailment.

### 2.3 Logic functors for types as search keys

Given an inheritance relation, the logic functor  $inh$  models basic types as logic formulas.

**Definition 4 (logic functor  $inh$ )**

- $AS_{inh}$  is the set of all basic types,
- $S_{inh}$  is  $(I_{inh}, \models_{inh})$  such that  
 $I_{inh} = \mathcal{P}(AS_{inh})$  and  $i \models_{inh} f$  iff  $\exists g \in i [f \text{ inherits from } g]$ ,
- $P_{inh}$  is defined as follows:

- $f \sqsubseteq_{inh} g$  iff  $f$  inherits from  $g$ ,
- $f \sqcap_{inh} g = \begin{cases} f & \text{if } f \sqsubseteq_{inh} g \\ g & \text{if } g \sqsubseteq_{inh} f \\ \text{undef} & \text{otherwise} \end{cases}$
- If there is a type from which every other type inherits, then  $\top_{inh}$  is this type, otherwise it is undef.

**Example 1 (java.awt)** Given the Java package *AWT*, plus a few primitive Java types, we get

- $AS_{inh} = \{Object, Component, TextComponent, Button, \dots, int\}$
- $Button \sqsubseteq_{inh} Component$   
 $Frame \sqsubseteq_{inh} Window$   
 $Container \sqcap_{inh} Component = Container$   
 (cont'd on Example 2)

**Theorem 1**  $P_{inh}$  is consistent and complete w.r.t.  $S_{inh}$ .

The logic functor  $pol$  models the polarity of parameters, and contravariance.

**Definition 5 (logic functor  $pol$ )** Given a logic  $L$

- $AS_{pol(L)}$  is  $!AS_L \cup ?AS_L$ ,
- $S_{pol}$  is  $(I_L, \models_L) \rightarrow (I_{pol(L)}, models_{pol(L)})$  such that  
 $I_{pol(L)} = I_L$ , and  $i \models_{pol(L)} !f$  iff  $i \models_L f$  and  $i \models_{pol(L)} ?f$  iff  $i \not\models_L f$ ,
- $P_{pol}$  is  $(\sqsubseteq_L, \sqcap_L, \sqsupset_L) \rightarrow (\sqsubseteq_{pol(L)}, \sqcap_{pol(L)}, \sqsupset_{pol(L)})$  such that

- $!f \sqsubseteq_{pol(L)} !g$  iff  $f \sqsubseteq_L g$
- $?f \sqsubseteq_{pol(L)} ?g$  iff  $g \sqsubseteq_L f$
- $!f \sqcap_{pol(L)} !g = !(f \sqcap_L g)$
- $?f \sqcap_{pol(L)} ?g = ?(f \sqcup_L g)$
- $\sqsupset_{pol(L)} = undef$

**Example 2 (java.awt cont'd)** In logic  $pol(inh)$  the following holds

- ?Object  $\sqsubseteq$ ?String
- !Container  $\sqsubseteq$ !Component
- ?String  $\sqcap$ !String = undef

(cont'd on Example 3)

**Theorem 2**  $P_{pol(L)}$  is consistent and complete w.r.t.  $S_{pol(L)}$  if  $P_L$  is consistent and complete w.r.t.  $S_L$ .

The proof proceeds by case analysis. E.g.,  $?f \sqsubseteq_{pol(L)} ?g \stackrel{Def. 5}{\equiv} g \sqsubseteq_L f \stackrel{hypothesis}{\equiv} M_L(g) \subseteq M_L(f) \equiv (i \in M_L(g) \Rightarrow i \in M_L(f)) \equiv (i \notin M_L(f) \Rightarrow i \notin M_L(g)) \stackrel{Def. 5}{\equiv} (i \in M_{pol(L)}(?f) \Rightarrow i \in M_{pol(L)}(?g)) \equiv M_{pol(L)}(?f) \subseteq M_{pol(L)}(?g)$ .

The logic functor  $mset$  models the multiplicity of parameters. It requires the following multi-set operations.

- *sum*:  $M + N$ . The number of occurrences of every  $x$  in  $M + N$  is the sum of its numbers of occurrences in  $M$  and  $N$ .
- *union*:  $M \cup N$ . The number of occurrences of every  $x$  in  $M \cup N$  is the maximum of its numbers of occurrences in  $M$  and  $N$ .
- *intersection*:  $M \cap N$ . The number of occurrences of every  $x$  in  $M \cap N$  is the minimum of its numbers of occurrences in  $M$  and  $N$ .
- *difference*:  $M \setminus N$ . The number of occurrences of every  $x$  in  $M \setminus N$  is its number of occurrences in  $M$  minus its number of occurrence in  $N$  (0 if  $x$  has more occurrences in  $N$  than in  $M$ ).
- $\mathcal{MS}(S)$  is the collection of all multi-sets built with elements of  $S$  having only a finite number of occurrences,.

**Definition 6 (logic functor  $mset$ )** Given a logic  $L$

- $AS_{mset(L)} = \mathcal{MS}(AS_L)$ ,
- $S_{mset}$  is  $(I_L, \models_L) \rightarrow (I_{mset(L)}, \models_{mset(L)})$  such that  
 $I_{mset(L)} = \mathcal{MS}(I_L)$ , and  $i \models_{mset(L)} f$  iff  $\forall f_L \in f [i \cap M_L(f_L) \neq \emptyset]$ ,
- $P_{mset}$  is  $(\sqsubseteq_L, \sqcap_L, \sqsupset_L) \rightarrow (\sqsubseteq_{mset(L)}, \sqcap_{mset(L)}, \sqsupset_{mset(L)})$  such that

- $f \sqsubseteq_{mset(L)} g$   
iff  $g = \emptyset$

$$\begin{aligned}
& \text{or } \exists g_L \in g \exists f_L \in f [f_L \sqsubseteq_L g_L \wedge f \setminus \{f_L\} \sqsubseteq_{mset(L)} g \setminus \{g_L\}], \\
- f \sqcap_{mset(L)} g &= \begin{cases} f & \text{if } g = \emptyset \\ g & \text{if } f = \emptyset \\ f \cup g & \text{if } \forall g_L \in g \forall f_L \in f [f_L \sqcap_L g_L = \text{undef}] \\ \{f_L \sqcap_L g_L\} + f \setminus \{f_L\} \sqcap_{mset(L)} g \setminus \{g_L\} & \\ \text{if } f_L \in f \text{ and } g_L \in g \text{ and } f_L \sqcap_L g_L \neq \text{undef} & \\ \text{and } \forall g'_L \in g \forall f'_L \in f \left[ \begin{array}{l} f'_L \sqcap_L g'_L \neq \text{undef} \\ \Rightarrow (g'_L \sqsubseteq_L g_L \wedge f'_L \sqsubseteq_L f_L) \end{array} \right] & \end{cases} \\
- \top_{mset(L)} &= \emptyset
\end{aligned}$$

**Example 3 (java.awt (cont'd))** In logic  $mset(pol(inh))$  the following holds

$$\begin{aligned}
& \{?Component, ?String, ?Component, !Component\} \\
& \sqsubseteq \{?Container, ?String, ?Component, !Component\} \\
& \quad (\text{i.e., type of method } Container.add) \\
& \{?Component, ?String, ?Component, !Component\} \\
& \sqsubseteq \{?String, ?Component, !Component\} \\
& \{?Container, ?String, ?Component, !Component\} \\
& \sqsubseteq \{?Container, ?String, ?Container, !Object\} \\
& \sqsubseteq \{?String, ?Container, !Object\}
\end{aligned}$$

**Theorem 3** If  $P_L$  is consistent and complete w.r.t.  $S_L$ , then  $P_{mset(L)}$  is complete w.r.t.  $S_{mset(L)}$ , and  $P_{mset(L)}$  is consistent w.r.t.  $S_{mset(L)}$ , except in  $\sqcup_{mset(L)}$ .

Operations  $\top_{mset(L)}$ ,  $\sqcap_{mset(L)}$ , and  $\sqcup_{mset(L)}$  are totally defined.

For binary operations, the proof proceeds by recurrence on the size of the smallest parameter. We prove a slightly stronger result than needed; e.g., for entailment, the recurrence hypothesis is

$$\forall h \in AS_{mset(L)} \forall g_L \in g [h \sqsubseteq g \setminus \{g_L\} \Rightarrow M(h) \subseteq M(g \setminus \{g_L\})].$$

### 3 Experiments

We have used the logic  $mset(pol(inh))$  in the context of a logic-based information system [FR00], of which we will only describe the navigation behavior [FR01b].

#### 3.1 Logic information systems

In a logic information system, every piece of information  $o \in \mathcal{O}$  is labelled with a formula  $d(o)$ . As is usual in logic-based information retrieval (e.g., see [vRCL98]), the logic information system considers every object  $o$  such that  $d(o)$  entails the query  $q$  as a possible answer. However, in a logic information system as defined in [FR01b], the set of all possible answers is not returned directly. Instead, the logic information system computes and returns *increment* formulas  $q'$  with the following property:  $\emptyset \subsetneq \{o \mid d(o) \text{ entails } q' \wedge q\} \subsetneq \{o \mid d(o) \text{ entails } q\}$ . All that such a system requires from a logic component is an implementation of the entailment relation,  $\sqsubseteq_L$ , and of the conjunction,  $\sqcap_L$ . There may be several increments  $q'$  in a

returned answer, and the user is supposed to examine them and choose one of them to build a new query  $q \wedge q'$ .

The advantage of doing this is to avoid long listing of answers. This is recognized as a problem in web browsers, and previous works in using types as search keys have also encountered the problem. The method used in logic information systems usually returns less answers than the classical one. It is not based on approximations, nor on statistics. So, it gives the same visibility to all possible answers.

All this has been implemented in a logical shell. This is a classical shell in which the PWD is replaced by the *current query*, directories are replaced by queries, and path construction,  $a/b$ , is replaced by conjunction,  $a \wedge b$ . Command *ls* returns increments  $q'$  that behave as virtual directories which are created when needed. The logic  $mset(pol(inh))$  has been plugged in this logical shell to perform experiments described in the next section.

### 3.2 Java methods retrieval

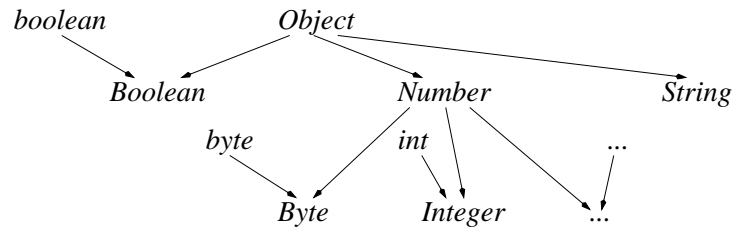
We apply the logical shell equipped with the logic  $mset(pol(inh))$  to the retrieval of Java methods. The following experiment has been done with the methods and constructors of classes *Object*, *String*, *Boolean*, *Number*, *Integer*, *Double*, *Float*, and *Byte* of package *java.lang*. For every method and constructor, an object is created whose path is the type of the method or constructor in the multi-set notation. For non-static methods, the class is added as a parameter type since these methods are applied to instances of the class.

**Example 4 (java.lang)** *An object compareTo is created at path  $\{?Float, ?Float, !int\}$  because method  $Float.compareTo : Float \rightarrow int$  is not static.*

*An object compare is created at path  $\{?float, ?float, !int\}$  because method  $Float.compare : float \times float \rightarrow int$  is static.*

*(cont'd on Example 5)*

Package *java.lang* encapsulates primitive types, like *int* and *float*, in classes, like *Integer* and *Float*. We could ignore this fact, but since one can always find an *int* as a field of an *Integer*, one may consider that *Integer* satisfies *int*. In summary, the entailment relation for classes and primitive types is as follows:



**Example 5 (java.lang)** *We consider a fragment of java.lang of about 200 methods and constructors.*

Starting from the root, i.e.,  $PWD = /$ , command `ls` returns a collection of increments  $q'$  that covers all the objects. Interestingly, there are only 19 increments, though there are about 200 objects. Moreover, `ls` also returns association rules like  $\{?Locale\} \Rightarrow \{?String\}$ , which says that every method that satisfies  $\{?Locale\}$  also satisfies  $\{?String\}$ .

Now, we search for a method that takes two *String*'s and returns a *String*. This is done via command “`cd {?String,?String,!String}`”. The answer amounts to 3 methods:

`concat` :  $\{?String, ?String, !String\}$ ,  
`replaceFirst` :  $\{?String, ?String, ?String, !String\}$ ,  
`and valueOf` :  $\{?String, ?Object, !String\}$ .

Lastly, we wish to convert a *Float* into an *int*, so we do “`cd /{?Float, !int}`”. As in a Unix shell, the `'/'` is the mark of an absolute query; omitting it would result in combining this query with the previous one, which would result in an absurd query ( $\perp$ ) with no answer. The answer amounts to 12 variants of the following methods:

`hashCode` :  $\{?Object, !int\}$ ,  
`compareTo` :  $\{?SomeClass, ?Object, !int\}$ ,  
`intValue` :  $\{?Number, !int\}$ ,  
`and floatToIntBits` :  $\{?float, !int\}$ .

## 4 Conclusions and perspectives

We have proposed a new logical model for types as search keys that is based on the development of logic functors. This has several advantages.

First, it provides an implementation of the paradigm of types as search keys that is formally grounded.

Second, it can benefit from the development of other logic functors. For instance, we have already developed a propositional functor *prop* [FR01a] for a logic information system. It happens that *pol* is simply the fragment of *prop* reduced to its negation connector. We preferred to present in this article a simpler and more focused logic functor like *pol* instead of starting with the more general *prop*. However, it is now time to generalize. Assuming the inheritance relation is a partial order, it can be completed in a lattice by adding new elements. This is exactly what *prop(inh)* will do. This will permit writing  $t \sqcup t'$  for expressing multiple inheritance, and  $t \sqcap t'$  for expressing common super-class.

Lastly, by using again the range of logic functors that have already been developed, one can combine several very different kinds of search keys. Beside types, one can use keywords, names of package, dates of release, version numbers, bits of specification, comments, access attributes (e.g., *static*, *private*, *public*), thrown exceptions, etc, in the same framework.

An immediate perspective is to develop more logic functors to cover more type constructions like arrays and pointers. These constructions ask new questions to the paradigm of types as search keys. For instance, a user asking for a

function that returns a  $t$  might be happy with a function that returns a  $t^*$  or a  $t[]$ . Similarly, a  $t[]$  may satisfy a query for a  $t^*$ , but not the opposite.

A more distant perspective is to use logic functors for modeling program abstractions to be used in static analysis. Indeed, if an interface  $P_L$  is consistent and complete w.r.t. a semantics  $S_L$ , its operations are fragments of  $\subseteq$ ,  $\cap$ , and  $\cup$ . Hence, they have enough good properties to fit in a fix-point analysis.

## References

- [Di 92] R. Di Cosmo. Type isomorphisms in a type assignment framework. In *19th ACM Symp. Principles of Programming Languages*, pages 200–210. ACM, 1992.
- [Di 95] R. Di Cosmo. *Isomorphisms of Types: from  $\lambda$ -calculus to information retrieval and language design*. Progress in theoretical computer science. Birkhäuser, 1995.
- [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.
- [FR01a] S. Ferré and O. Ridoux. A framework for developing embeddable customized logics. In *LOPSTR*, LNCS 2372, pages 191–215. Springer, 2001.
- [FR01b] S. Ferré and O. Ridoux. Searching for objects and properties with logical concept analysis. In *Int. Conf. Conceptual Structures*, LNCS 2120. Springer, 2001.
- [Rit91] Mikael Rittri. Using types as search keys in function libraries. *J. Functional Programming*, 1(1):71–89, 1991.
- [RT91] C. Runciman and I. Toyn. Retrieving reusable software components by polymorphic type. *J. Functional Programming*, 1(2):191–211, 1991.
- [RW91] E.J. Rollins and J.M. Wing. Specifications as search keys for software libraries. In K. Furukawa, editor, *8th Int. Conf. Logic Programming*, pages 173–187. MIT Press, 1991.
- [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, 1998.