INFORMATIQUE THÉORIQUE ET APPLICATIONS

M. RITTRI

Retrieving library functions by unifying types modulo linear isomorphism

Informatique théorique et applications, tome 27, n° 6 (1993), p. 523-540

<http://www.numdam.org/item?id=ITA_1993__27_6_523_0>

Informatique théorique et Applications/Theoretical Informatics and Applications
(vol. 27, n° 6, 1993, p. 523 à 540)

RETRIEVING LIBRARY FUNCTIONS BY UNIFYING TYPES MODULO LINEAR ISOMORPHISM (*)

by M. RITTRI (1)

Communicated by G. LONGO

Abstract. – *An improved method to retrieve a library function via its Hindley/Milner type is described. Previous retrieval systems have identified types that are isomorphic in any Cartesian closed category (CCC), and have retrieved library functions of types that are either isomorphic to the query, or have instances that are. Sometimes it is useful to instantiate the query too, which requires unification modulo isomorphism. Although unifiability modulo CCC-isomorphism is undecidable, it it decidable modulo linear isomorphism, that is, isomorphism in any symmetric monoidal closed (SMC) category.*

We argue that the linear isomorphism should retrieve library functions almost as well as CCC-isomorphism, and we report experiments with such retrieval from the Lazy ML library. When unification is used, the system retrieves too many functions, but sorting by the sizes of the unifiers tends to place the most relevant functions first.

Résumé. – *Cet article présente une nouvelle méthode pour la recherche d'une fonction dans une bibliothèque de programmes à partir de son type (au sens de Hindley/Milner). Les méthodes utilisées jusqu'ici identifient les types qui sont isomorphiques dans n'importe quelle catégorie cartésienne fermée (CCF), et le type résultat est soit isomorphe au type demandé, soit en est une généralisation. Il est quelquefois utile d'instancier le type demandé, ce qui nécessite de résoudre un problème d'unification modulo isomorphismes. Bien que l'unification modulo CCF-isomorphismes soit indécidable, ce problème est décidable modulo isomorphismes linéaires, c'est-à-dire isomorphismes dans une catégorie monoïdale fermée symétrique.*

Notre thèse est que la recherche d'une fonction modulo isomorphismes linéaires doit être aussi utile que modulo CCF-isomorphismes. Nous présentons quelques résultats expérimentaux, qui ont été effectués dans la librairie de fonctions de Lazy ML. En présence d'unification, le système trouve trop de fonctions, mais ce problème peut être résolu en classant les substitutions par leur taille.

(*) Received March 1992, accepted May 1993.
(1) Department of Computing Science, Chalmers University of Technology and University of Göteborg, S-412 96 Göteborg, Sweden, rittri@cs.chalmers.se.

Informatique théorique et Applications/Theoretical Informatics and Applications
0988-3754/93/06/$4.00/

1. BACKGROUND

There are many general-purpose methods for automated retrieval of documents from a database. For software libraries, one can use the special structure of software to improve the retrieval, as surveyed by Frakes [7].

In functional languages, polymorphic types work well as queries [13, 17, 18, 20]. For instance, the function that reverses lists has the type $\forall\alpha.[\alpha]\to[\alpha]$, and there are few common functions of this type, since they cannot examine the list elements. Some retrieval systems allow the query to be a type augmented with a formal specification [3, 15, 19].

I have developed a retrieval system based purely on types [17, 18]; it has become popular in the Lazy ML community at Chalmers. This article describes how the system was improved by using unification modulo type isomorphism.

1.1 Isomorphic types

In my previous papers [17, 18], I wanted to abstract from details like the currying and argument order of functions, so I needed a notion of type isomorphism that expressed the abstraction. A library function should then be retrieved if its type was isomorphic to the query, since a bijection (like *curry*) could convert the function into the query type. It turned out that the so-called *CCC-isomorphism* in category theory was suitable.

Categories are mathematical structures that possess types (or objects), functions (or arrows) between types, and a notion of type isomorphism. Some categories can be seen as models for various versions of λ-calculus; the most well-known are the *Cartesian closed categories*, or CCCs. We do not have to define the CCC-isomorphism in a categorical way; we can use a result by Lambek [10] instead: two types A and B are isomorphic in all Cartesian closed categories if, and only if, there are λ-expressions $f:A\to B$ and $g:B\to A$ such that the equalities $g\circ f=\mathrm{id}_A$ and $f\circ g=\mathrm{id}_B$ hold in simply typed λβη-calculus with surjective pairing. I write this as $A\rightleftarrows B$ or

$$A \underset{g}{\overset{f}{\rightleftarrows}} B.$$

A statement of the form "$A\rightleftarrows B$" will be called an *isomorphism*. The functions f and g are usually also called isomorphisms, but I will call them *bijections*

TABLE 1

Equational axioms (with associated bijections) for isomorphism in all Cartesian closed categories.

$$A \times B \underset{exch}{\overset{exch \equiv \lambda\,(x,\,y)\,.\,(y,\,x)}{\rightleftarrows}} B \times A \qquad \text{(Com-2)}$$

$$(A \times B) \times C \underset{assl \equiv \lambda\,(x,\,(y,\,z))\,.\,((x,\,y),\,z)}{\overset{assr \equiv \lambda\,((x,\,y),\,z)\,.\,(x,\,(y,\,z))}{\rightleftarrows}} A \times (B \times C) \qquad \text{(Ass-2)}$$

$$\mathbf{1} \times A \underset{insl \equiv \lambda\,x\,.\,((),\,x)}{\overset{dell \equiv \lambda\,((),\,x)\,.\,x}{\rightleftarrows}} A \qquad \text{(Ass-0)}$$

$$(A \times B) \to C \underset{uncurry \equiv \lambda\,g\,.\,\lambda\,(x,\,y)\,.\,gxy}{\overset{curry \equiv \lambda\,f\,x\,y\,.\,f\,(x,\,y)}{\rightleftarrows}} A \to (B \to C) \qquad \text{(Cur-2)}$$

$$\mathbf{1} \to C \underset{absunit \equiv \lambda\,x\,.\,\lambda\,()\,.\,x}{\overset{appunit \equiv \lambda f\,.\,f\,()}{\rightleftarrows}} C \qquad \text{(Cur-0)}$$

non-linear

$$A \to (B \times C) \underset{collect \equiv \lambda\,(g,\,h)\,.\,\lambda\,x\,.\,(gx,\,hx)}{\overset{distrib \equiv \lambda\,f\,.\,(f\,st \circ f,\,snd \circ f)}{\rightleftarrows}} (A \to B) \times (A \to C) \qquad \text{(Dist-2)}$$

$$A \to \mathbf{1} \underset{arr \equiv \lambda\,()\,.\,\lambda\,x\,.\,()}{\overset{unarr \equiv \lambda\,f\,.\,()}{\rightleftarrows}} \mathbf{1} \qquad \text{(Dist-0)}$$

to avoid confusion. Table 1 shows equational axioms (with associated bijections) that are sound and complete for isomorphism in all CCCs [4, 14, 22]. (The empty Cartesian product is written **1**; its single element is written ().) If $A \to B$ is written B^A instead, the axioms will look more familiar, since they also describe all identities in the algebra of natural numbers with multiplication, exponentiation and the constant 1.

Remark 1. – I use the axioms also on Hindley/Milner types, which may contain variables that may be bound at the top-level, simply by allowing renaming of bound variables. When used in this way, the axioms are not quite complete for Hindley/Milner types. Some additional axioms, like

$$\forall \alpha\,.\,A \times B \underset{\lambda\,p\,.\,p}{\overset{\lambda\,p\,.\,(\,fst\,p,\,snd\,p)}{\rightleftarrows}} \forall \alpha\beta\,.\,A \times (B[\beta/\alpha]),$$

would make them complete, but these extra axioms can instead be used directly by the type-deriver [6], in which case the retrieval system does not need them.

Remark 2. – The axioms are not valid in all λ-calculi or functional languages, but they hold in an approximate sense, which should be enough for software retrieval.

1.2. Matching and unification

Independently, Runciman and Toyn suggested retrieving library functions whose types are unifiable with the query, as well as functions with extra arguments [20]. They did not use any equivalence relation, though.

When I tried to unite the best parts of Runciman and Toyn's work and my own, my first intention was to implement matching and unification modulo CCC-isomorphism (that is to seek substitutions that can makes types isomorphic). Matching, or one-sided unification, allows us to retrieve library functions of types more general than the query, modulo isomorphism. I will assume henceforth that we wish to do so, since it is useful when we overlook a possible generalization. I gave motivating examples and an algorithm for such matching in [18]. Morgan uses a similar algorithm [15].

However, unifiability modulo CCC-isomorphism is undecidable, as was shown by Narendran, Pfenning and Statman [16]. But by removing the distributivity axiom (Dist-2), they were able to construct a unification algorithm, which I have implemented and used in a software retrieval system for the functional language Lazy ML.

This article has two main parts. In section 2, I argue that the removal of the Dist axioms usually does not harm the retrieval, and in section 3, I present some experiences with the retrieval system.

2. LINEAR ISOMORPHISM FOR LIBRARY SEARCH

If we want unifiability modulo equivalence to be decidable, we are forced to remove the (Dist-2) axiom, and since the (Dist-2) and the (Dist-0) axioms are two instances of a similar n-tuple axiom (Dist-n), it seems most consistent to remove the (Dist-0) axiom as well. The removal means that some queries will retrieve fewer functions, which is bad if the omitted functions are useful, but good if they are irrelevant. What can we expect?

Let us study the individual axioms of table 1. The axioms (Com-2), (Ass-2), and (Cur-2) are crucial for function retrieval, as they abstract from argument order and currying. (Cur-0) is useful in a strict language, if lazily evaluated expressions of type C are simulated by functions of type $\mathbf{1} \to C$. If we have (Cur-0,2), then (Ass-0) holds "to the left of an arrow" [since $(\mathbf{1} \times A) \to B \rightleftarrows_{\text{(Cur-2)}} \mathbf{1} \to (A \to B) \rightleftarrows_{\text{(Cur-0)}} A \to B$], so we may as well include (Ass-0) in general. Finally, the main motive for the (Dist-0,2) axioms has been to get a nice semantics of the equivalence relation. Isomorphism in all CCCs seemed appropriate for functional programming, since when two types

are isomorphic, one can easily convert back and forth, so the choice between them seems arbitrary and unguessable.

But using ideas from linear logic [8, 9], we see that all bijections in table 1 are linear, except those for the (Dist) axioms. (A closed λ-expression is linear if every variable is bound once and used once [9, section 7]. *distrib* and *collect* use variables twice, while *unarr* and *arr* bind variables they do not use.) Non-linear bijections will change the amount of sharing, and a library function has often a natural amount of sharing, which a user can guess. In these cases, the non-linear bijections are not needed for library search. This is illustrated best by examples.

No user should miss the (Dist-0) axiom, which says that $(A \to \mathbf{1}) \rightleftarrows \mathbf{1}$. In lazy languages, hardly any functions have the result type **1**, so we have few opportunities to apply the axiom. In strict languages, such functions are common but have side-effects, for instance $cd : [Char] \to \mathbf{1}$, which changes the working directory. The (Dist-0) axiom identifies for instance $[Char] \to \mathbf{1}$, $Bool \to \mathbf{1}$, and $Int \to \mathbf{1}$, which seems bad in the presence of side-effects.

The (Dist-2) axiom says that a function that returns a pair can be translated to two functions that return the components. But it is quite unlikely that a pair of two functions is named as a library item, so the (Dist-2) axiom will have little effect at the top-level of a type. (Of course, a query that is a Cartesian product could make the system look for possible components, but I have not implemented this. I think that if the retrieval system tries to combine different library items, too many possibilities will arise.) The (Dist-2) axiom can be applied to parts of a type, but since the *distrib* and *collect* bijections change the sharing, the user can often guess which variant occurs in a library function. Roughly, if a function returns a B-value and a C-value in a single computation for every A-value, its most natural type is $A \to (B \times C)$, but if it computes only B-values for some A-values and only C-values for others, it is more natural to split it into a function-pair of the distributed type $(A \to B) \times (A \to C)$.

Example 1. — The *choplist* function, predefined in Lazy ML [0], takes a function f and a list xs. f can take a list of the same type as xs and chop off a prefix, to return a pair of the chopped part and the rest of the list. *choplist* applies f to xs repeatedly to get a list of chopped parts, e. g., if *takeword* chops off the first lexeme of a string, then *choplist takeword* will return a list of the lexemes in a string. *choplist* can be defined by

$$choplist : \forall \alpha . ([\alpha] \to [\alpha] \times [\alpha]) \to [\alpha] \to [[\alpha]]$$

$$choplist\, f\, [\ ] = [\ ]$$

$$choplist\, f\, xs = \textbf{let}\, (ys, zs) = f\, xs\, \textbf{in}\, ys :: choplist\, f\, zs$$

where "::" is infix *cons*. By using (Dist-2), we find an alternative definition with a CCC-isomorphic type:

$$choplist' : \forall \alpha . ([\alpha] \to [\alpha]) \times ([\alpha] \to [\alpha]) \to [\alpha] \to [[\alpha]]$$

$$choplist'\, (g, h)\, [\ ] = [\ ]$$

$$choplist'\, (g, h)\, xs = g\, xs :: choplist'\, (g, h)\, (h\, xs)$$

so that $choplist'\, (g, h) = choplist\, f$ if $(g, h) = distrib\, (f)$. Now, the normal situation is that g and h do similar work. (In the lexeme example above, g should find the first lexeme of a string and keep it, whereas h should find the first lexeme and discard it.) If g and h are not encapsulated into an f function, their common work will not be shared. In this case, a library programmer can be expected to write the original version of *choplist* and the user will guess the non-distributed type. The (Dist-2) axiom is not necessary. □

Example 2. – *maplast* is like *map*, but applies a different function to the last element of the list (useful for formatting with separators and terminators). Thus,

$$maplast : \forall \alpha\beta . (\alpha \to \beta) \to (\alpha \to \beta) \to [\alpha] \to [\beta]$$

$$maplast\, g\, h\, [\ ] = [\ ]$$

$$maplast\, g\, h\, [x] = [h\ x]$$

$$maplast\, g\, h\, (x1 :: x2 :: xs) = g\ x1 :: maplast\, g\, h\, (x2 :: xs)$$

By using the axioms (Cur-2) and (Dist-2), we find an alternative definition with a CCC-isomorphic type

$$maplast' : \forall \alpha\beta . (\alpha \to \beta \times \beta) \to [\alpha] \to [\beta]$$

$$maplast'\, f\, [\ ] = [\ ]$$

$$maplast'\, f\, [x] = [snd\, (f x)]$$

$$maplast'\, f\, (x1 :: x2 :: xs) = fst\, (f\ x1) :: maplast'\, f\, (x2 :: xs)$$

so that $maplast'\, f = maplast\ g\ h$ if $distrib\, (f) = (g, h)$. The *maplast'* version computes both $g(x)$ and $h(x)$ for every element in the list, only to throw away one of them. In a strict language, this might be much more work; in a lazy language, the unwanted value need not be fully computed, but there is still unnecessary overhead in building the pair and the representation of $g(x)$

or $h(x)$. So both a library programmer and a user should feel that the distributed version of the type is more natural for *maplast*. □

These examples are representative for those I have come across: usually there is a natural amount of sharing. So the removal of the (Dist) axioms should not harm the retrieval too much.

2.1. Theoretical aspects

The exact definition of linear λ-expressions can be found in [9, section 7], except that we do not need the **if-then-else**. Our **1** corresponds to the tensor unit, our × corresponds to the tensor product ⊗, and our → corresponds to the linear implication —∘. Two types are linearly isomorphic if there are linear bijections between them; this is a stronger requirement than the linear logical equivalence ∘—∘. It is far from obvious that the five linear axioms in table 1 form a complete equational axiomatization of linear isomorphism, but I had the good luck to be able to contact Sergei Soloviev, who found a proof [23]. Instead of generating the isomorphisms that hold in any Cartesian closed category, the five axioms generate those that hold in all *symmetric monoidal closed* (SMC) categories, sometimes just called closed categories [12, section VII.7].

The equational axioms in table 1 are decorated with bijections, and Soloviev's proof implies that the decorations survive (in modified forms) during equational reasoning. This gives an inductive way of generating the linear bijections – we just extend Birkhoff's rules for equational reasoning [2] to handle decorations (table 2).

Rule 2(v) may need some examples. What happens F is *List*? Then the rule says that from

$$A \underset{f^{-1}}{\overset{f}{\rightleftarrows}} B,$$

we can infer that

$$List(A) \underset{map_{List}(f^{-1},\, f)}{\overset{map_{List}(f,\, f^{-1})}{\rightleftarrows}} List(B),$$

but what is map_{List}? It is simply the ordinary *map* function over lists, except that it has an extra argument f^{-1}, which is ignored:

$$map_{List}(f, f^{-1})[x_1, \ldots, x_n] = [f(x_1), \ldots, f(x_n)].$$

TABLE 2

How Birkhoff's laws for equational reasoning modify the bijections.

$$\frac{}{A \underset{id}{\rightleftarrows} A}\ id$$

(i) reflexivity

$$\frac{A \underset{f}{\rightleftarrows} B}{B \underset{f^{-1}}{\rightleftarrows} A}\ f^{-1}$$

(ii) symmetry

$$\frac{A \underset{f}{\rightleftarrows} B \quad B \underset{g}{\rightleftarrows} C}{A \underset{g \circ f}{\rightleftarrows} C}\ g^{-1}\ f^{-1}$$

(iii) transitivity

$$\frac{A \underset{f}{\rightleftarrows} B}{\sigma(A) \underset{f^{-1}}{\rightleftarrows} \sigma(B)}\ f^{-1}$$

(iv) stability

$$\frac{A_1 \underset{f_1}{\rightleftarrows} B_1 \quad \ldots \quad A_n \underset{f_n}{\rightleftarrows} B_n}{F(A_1, \ldots, A_n) \underset{map_F(f_1^{-1}, f_1, \ldots, f_n^{-1}, f_n)}{\rightleftarrows} F(B_1, \ldots, B_n)}\ map_F(f_1, f_1^{-1}, \ldots, f_n, f_n^{-1})\ f_1^{-1}\ f_n^{-1}$$

(v) compatibility

Similarly, $map_\times$ is defined by

$$map_\times (f_1, f_1^{-1}, f_2, f_2^{-1})(a_1, a_2) = (f_1(a_1), f_2(a_2)),$$

and also ignores its f_1^{-1} and f_2^{-1} arguments.

The general rule 2(v) must allow the *map* function to use every f_i^{-1} as well as every f_i, because it needs them when F is contravariant in an argument (roughly: an argument of F occurs to the left of an arrow). For instance, when F itself is the arrow, we have to define $map_\to$ by

$$map_\to (f_1, f_1^{-1}, f_2, f_2^{-1})(g) = f_2 \circ g \circ f_1^{-1}.$$

As another example, let

$$\textbf{type}\ F(\alpha) = C1(\alpha) + C2(\alpha \to Int),$$

then

$$map_F(f, f^{-1})(C1(a)) = C1(f(a))$$
$$\text{map}_F(f, f^{-1})(C2(g)) = C2(g \circ f^{-1}).$$

The decorated rules in table 2 give an inductive way to construct the set of linear bijections, which we can call **Linb**. Starting from the five linear axioms in table 1, we get that *exch*, *assr*, *assl*, *dell*, *insl*, *curry*, *uncurry*, *appunit*, and *absunit* belong to **Linb**. Rule 2 (i) says that id is in **Linb**. Rule 2 (iii) says that if g and f are in **Linb**, so is $g \circ f$. Finally, rule 2 (v) says that if

F is an n-ary type operator, and $f_1, \ldots, f_n$ are in **Linb**, then so is $map_F(f_1, f_1^{-1}, \ldots, f_n, f_n^{-1})$.

A consequence of this inductive definition is that whenever a function is in **Linb**, then so is its inverse.

The set **Linb** depends on which type operators exist in our language. If the only type operators are $\times$ and $\rightarrow$, then Soloviev's proof of equational completeness [23] says that **Linb** contains exactly the bijections that exist between types that are isomorphic in every SMC category. But when we search functional libraries, we must allow all type operators that occur in the library.

3. EXPERIMENTS WITH EQUATIONAL UNIFICATION

When we retrieve library functions via types, it is reasonable to allow instantiation of library types, since a polymorphic library function can be used in a less general context. Some retrieval systems also allow unrestricted instantiation of the query [13, 19, 20], but this can be slow and too permissive. We can give the user some control over query instantiation if the queries are formulated explicitly by him (rather than derived from examples, say). A query variable can then express either polymorphism or an unknown subtype, and only in the latter case should it be instantiated. Since polymorphic variables are bound at top-level, we can use free variables to stand for unknown types. For example, when we seek the *reverse* function on lists, we know that its type $\forall \alpha . [\alpha] \rightarrow [\alpha]$ is polymorphic, so we do not want to retrieve functions of type $[Float] \rightarrow [Float]$. On the other hand, the query $\alpha \rightarrow Float$ (without quantifier) would retrieve all functions that return floats.

We still need an algorithm for unification modulo isomorphism. There is no general method to unify in a given equational theory, and there are theories in which unifiability is undecidable. Usually, one has to resort to *ad hoc* algorithms. Siekmann has made a comprehensive survey [21]. Narendran, Pfenning and Statman [16] proved that unifiability modulo CCC-isomorphism in undecidable, but gave an algorithm for unification modulo linear isomorphism, which I have implemented on top of a Standard ML program for associative-commutative unification [11]. I have added the restriction that variables in library types must not be instantiated to **1**, as this seems to retrieve only rubbish.

The retrieval system is still often too liberal; for instance, if the user searches for a function of type Q, and allows extra arguments by submitting

the query $\varepsilon \to Q$, then every function of a type $\forall \alpha . A \to \alpha$ will be retrieved via the substitution $\{\alpha := Q, \varepsilon := A[Q/\alpha]\}$. Although the query and the answer are unifiable in this case, they need not be similar in any other way. This mechanism alone can retrieve a lot of rubbish, since many library functions have types that end with "$\ldots \to \alpha$".

Fortunately, the useful library types can be unified with the query by fairly simple unifiers, while the rubbish tend to require more complex ones (in the example above, Q is probably a medium-sized type, while $A[Q/\alpha]$ can be large). This observation can be explained as follows: the more general type a function has, the less it can do, since it cannot examine the internal structure of its polymorphic arguments; therefore, the more instantiation needed to fit a library type to a query, the less likely it is that the associated function is useful.

Therefore, my system ranks the retrieved items by the sizes of the unifiers. (When several most general unifiers exist, the smallest one is used for the ranking). The effect is that library functions whose types need only be instantiated a little (or not at all) are placed first.

3.1. Defining the size of substitutions

The size of a substitution can be defined in various ways. There are two primitive ways to specialize a type: either you make two variables equal, or you replace a variable by a constant or an operator applied to new, distinct variables. Therefore, it is reasonable to measure a substitution $\{\alpha_1 := t_1, \ldots, \alpha_n := t_n\}$ by assigning one weight to each repeated variable, and another weight to each occurrence of a constant or operator in the types $t_1, \ldots, t_n$. In my first tests, both weights were 1, but I got slightly better results by increasing the weight of operators and constants to 2. This fits intuition: it is a bigger step to introduce a constant or operator out of the blue, than to identify two variables that already exist.

Example 3. – A variable renaming, like

$$\{\alpha := \beta, \beta := \gamma, \gamma := \alpha\},$$

has size 0, since no variable is repeated among the right hand sides, and no operators or constants occur. □

Example 4. – The substitution

$$\{\alpha := \delta, \beta := \delta, \gamma := \delta\},$$

has the size 2, since δ is repeated twice. [A variable that occurs n times is repeated $(n-1)$ times.] □

Example 5. – The substitution

$$\{\alpha := Foo(Foo(\beta))\}$$

has the size 4, since each occurrence of the operator *Foo* carries the weight 2. □

A complication is that the unifiers can affect both polymorphic library variables and free query variables. At first, I treated these the same when I measured size. But the argument above, that more polymorphic functions can do less, only concern the polymorphic library variables – it says nothing about the free query variables. Therefore, I now split each unifier in two parts: one that acts on library variables and one that acts on free query variables. Both are measured separately, but the former is most significant and the latter is only used to break ties. This gave better results than equal significance.

In summary, the definition of substitution size is a heuristic intended to place the most relevant functions first. It works well, but can probably be improved after more experiments.

3.2. Examples of retrieval

We will look at some examples of use. The size of the smallest substitution is given as the pair of sizes of the library-variable part and the free query-variable part. The times are averages of ten trials and are CPU seconds on a SPARC Server 10, model 41, with 32 Mbyte; the system was compiled by Standard ML of NJ, version 0.75.

Example 6. – We wish to print a floating point number. If we query the Lazy ML library with the type $Float \to [Char]$, we retrieve only

ftos : $Float \to [Char]$ (0, 0)

Time : 0.12 s.

which indeed prints numbers in a standard way. If we now suspect that there is a more flexible print routine, which allows the user to choose the format, we should query with a type $\varepsilon \times Float \to [Char]$. Since ε is a free type variable, it can be instantiated to any type, which is fortunate since we do not know

the type of the extra formatting information. This query retrieves

ftos	$: \mathit{Float} \to [\mathit{Char}]$	(0, 2)
fmtf	$: [\mathit{Char}] \to \mathit{Float} \to [\mathit{Char}]$	(0, 4)
ftosf	$: \mathit{Int} \to \mathit{Int} \to \mathit{Float} \to [\mathit{Char}]$	(0, 6)
show_pair	$: \forall\, \alpha\beta.(\alpha \to [\mathit{Char}]) \times (\beta \to [\mathit{Char}]) \to \alpha \times \beta \to [\mathit{Char}]$	(2, 17)
... thirty functions omitted ...		⋮
while	$: \forall\, \beta.(\beta \to \mathit{Bool}) \to (\beta \to \beta) \to \beta \to \beta$	(8, 40)

Time: 1.13 s.

The standard formatter *ftos* is retrieved again, via the substitution $\{\varepsilon := \mathbf{1}\}$. But we also find two more flexible formatters *fmtf* and *ftosf*, via the substitutions $\{\varepsilon := [\mathit{Char}]\}$ and $\{\varepsilon := \mathit{Int} \times \mathit{Int}\}$. The first function takes a formatting string in the style of the *printf* of *C*, the other takes a minimum field width and a number of significant digits. We also retrieve thirty-two useless functions, but this does not matter much, as the useful ones were placed first. But the possibility of instantiating library types gave nothing useful in this example. □

Example 7. – Let us look for a function to check membership in a list. First we submit the query $\forall\, \alpha.\alpha \times [\alpha] \to \mathit{Bool}$, which retrieves

mem: $\forall\, \beta.\beta \to [\beta] \to \mathit{Bool}$ (0, 0)

Time: 0.75 s.

via the substitution $\{\beta := \alpha\}$.

To try Runciman and Toyn's strategy [20] to allow extra arguments to library functions, we can query with $\forall\, \alpha.\varepsilon \times \alpha \times [\alpha] \to \mathit{Bool}$. Since ε is a free variable, unlike α, it can be instantiated to the unknown type of the extra argument(s). This query retrieves

mem	$: \forall\, \beta.\beta \to [\beta] \to \mathit{Bool}$	(0, 2)
member	$: \forall\, \beta\gamma.(\beta \to \gamma \to \mathit{Bool}) \to \beta \to [\gamma] \to \mathit{Bool}$	(1, 7)
(=)	$: \forall\, \beta.\beta \to \beta \to \mathit{Bool}$	(5, 5)
... thirty-four functions omitted ...		⋮
while	$: \forall\, \beta.(\beta \to \mathit{Bool}) \to (\beta \to \beta) \to \beta \to \beta$	(9, 47)

Time: 10.1 s.

Now we also retrieve a function *member* that can take an equivalence test as an argument, but since nothing forces this test to take arguments of the same

type, the type of *member* is more general than the query. The required substitution is

$\{\beta := \alpha, \gamma := \alpha, \varepsilon := (\alpha \to \alpha \to Bool)\}$. □

Example 8. – This query was suggested by Dan Synek. He had defined *norm*:

$norm : \forall \alpha\beta . (\alpha \to [\beta] \to [\beta]) \to [\alpha] \to [\beta]$

$norm\, f\, [\ ] = [\ ]$

$norm\, f\, (x :: xs) = f\, x\, (norm\, f\, xs)$

and wondered if it was already defined in the standard library. If we use the type of *norm* as a query, nothing is retrieved. But if we again allow extra arguments in library types, by querying with

$\forall \alpha\beta . \varepsilon \to (\alpha \to [\beta] \to [\beta]) \to [\alpha] \to [\beta]$

we retrieve

itlist	$: \forall \gamma\delta . (\gamma \to \delta \to \delta) \to [\gamma] \to \delta \to \delta$	(2, 2)
revitlist	$: \forall \gamma\delta . (\gamma \to \delta \to \delta) \to [\gamma] \to \delta \to \delta$	(2, 2)
reduce	$: \forall \gamma\delta . (\gamma \to \delta \to \delta) \to \delta \to [\gamma] \to \delta$	(2, 2)
(∘)	$: \forall \alpha\beta\gamma . (\alpha \to \beta) \to (\gamma \to \alpha) \to (\gamma \to \beta)$	(10, 9)
. . . twenty-six functions omitted . . .		⋮
while	$: \forall \beta . (\beta \to Bool) \to (\beta \to \beta) \to \beta \to \beta$	(19, 90)

Time: 31.8 s.

If you are familiar with the *itlist/revitlist* functions, also known as *foldr/foldl*, you will realize that *norm* could have been defined by letting $norm\, f\, l = itlist\, f\, l\, [\ ]$, so it is just a special case of *itlist*, where the "start-state" of *itlist* has been frozen to the empty list. This also instantiates the type of *itlist*, since *norm* can return lists only. Therefore, to retrieve *itlist*, it was necessary both to instantiate ε to the extra argument, and to instantiate the library variable δ to [β]. The substitution becomes $\{\gamma := \alpha, \delta := [\beta], \varepsilon := [\beta]\}$. □

The test library contains 294 identifiers, whose types can be divided into 148 linear-isomorphism classes. The implementation of the retrieval is rather naïve; it just tests the classes one by one against the query. To get faster retrieval, it should be possible to organize the classes by their result types. It is also likely that the unification algorithm would be more efficient if it were

based directly on associative-commutative-unit unification, which gives fewer unifiers than associative-commutative unification.

3.3. Comparisons between equivalence, matching, and unification

Equational unification is normally more complex than matching and plain equivalence tests. The CCC-isomorphism test for Hindley/Milner types is graph-isomorphism-complete [1]; such problems are believed to be between polynomial and NP-complete. CCC-matchability is NP-complete, and CCC-unifiability is undecidable, although the restriction to linear CCC-isomorphism makes the unifiability NP-complete and thus decidable [16].

My implementation can usually test CCC-isomorphism of a query against 148 library types in less than a second. To find the library types that are more general than the query (modulo CCC-isomorphism), the time can be several seconds. And to test unifiability (modulo linear CCC-isomorphism), the time can be half a minute or more for the queries I have tried, but is usually less.

TABLE 3

User times in seconds, and number of retrieved items.

query	CCC-iso.		CCC-match.		lin-unif.	
Float → [*Char*]	0.22	(1)	0.27	(1)	0.12	(1)
ε × *Float* → [*Char*]	0.21	(0)	0.25	(0)	1.13	(35)
∀ α . α × [α] → *Bool*	0.21	(1)	0.24	(1)	0.75	(1)
∀ α . ε × α × [α] → *Bool*	0.20	(0)	0.27	(0)	10.1	(38)
∀ αβ . (α → [β] → [β]) → [α] → [β]	0.24	(0)	0.46	(0)	1.28	(0)
∀ αβ . ε → (α → [β] → [β]) → [α] → [β]	0.25	(0)	0.61	(0)	31.8	(31)

Table 3 gives some times for various queries. The tests of isomorphism and matching treated the free variables in the queries as if they were bound, that is, they were not instantiated, but possibly renamed. The figures in parentheses are the number of items retrieved from the library.

The queries were taken from examples 6-8. For other queries, and still using the Lazy ML library of 294 items, the number of retrieved items can be around half a dozen for isomorphism, and around a dozen for matching (that is, for checking if library types are more general than the query).

3.4. A user interface with windows

Thomas Hallgren has made a window-based user interface to the retrieval system. The answers to a query appear in one window (*fig.* 1). If the user

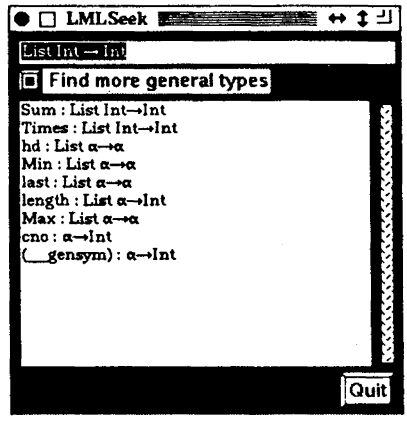

Figure 1. – Library items are retrieved.

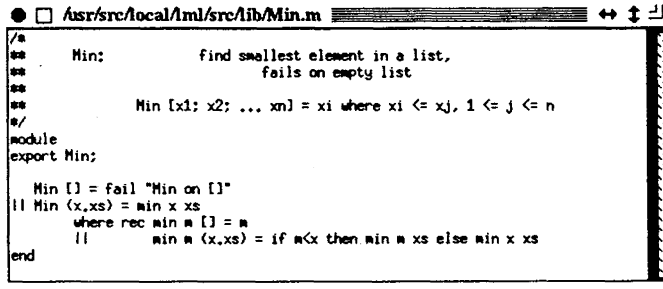

Figure 2. – When *Min* is clicked, its source file pops up.

clicks at one of the retrieved functions, its source file will appear in another window (*fig.* 2). This makes it easy to check if the functions satisfies the user's needs. The interface was made using the FUDGET toolkit [5].

Staffan Truvé made a similar interface based on the Interviews toolkit.

4. SUGGESTIONS FOR FUTURE WORK

4.1. Conjunctive queries

The free type variables in queries make conjunctive queries possible. For instance, the query

$$A \to \beta, \qquad \beta \to C$$

(where β is free) should return all pairs of functions f and g such that f has type $A \to B$ and g has type $B \to C$, for some type B. But it is not clear how to implement this efficiently.

4.2. Unifiers of bounded size

It may be possible to find a threshold size for unifiers, such that larger ones hardly ever retrieve useful functions. It would then suffice to check if substitutions smaller than the threshold are unifiers, and that could save time. What is more, there will be a finite number of substitutions to check, so the procedure will terminate even if the full CCC-isomorphism is used. This is an alternative if one wants to keep the distributivity axioms.

4.3. Retrieving proved lemmas

In semi-automated theorem proving or program verification, it would be useful to have easy access to a library of previously proved lemmas. Since lemmas can be seen as a kind of types which are more expressive than Hindley/Milner types (using the Curry/Howard correspondence), it may be possible to extend the retrieval method of this report to such types. To avoid undecidable problems, such a retrieval system must necessarily give only approximative results, but even a simple system could be useful in practice. The two basic questions are: what equivalence relation on types should be used, and to what extent should instantiation be allowed.

ACKNOWLEDGEMENTS

I thank Paliath Narendran, Frank Pfenning and Richard Statman for inventing the unification algorithm, Erik Lindström for implementing associative-commutative unification, and Sergei Soloviev for proving equational completeness. This work would not be possible without assistance from them. My retrieval system is much nicer to use since Thomas Hallgren and Staffan Truvé made a window-based user interface. I am also grateful to Magnus Carlsson, Thierry Coquand, Roberto Di Cosmo, Peter Dybjer, Yves Lafont, Pierre Lescanne, Giuseppe Longo, G. E. Mints, and Antti Valmari for help and advice. And my local user group, especially Annika Aasa, has provided valuable feedback.

REFERENCES

0. L. Augustsson and T. Johnsson, The Chalmers Lazy-ML Compiler, *Computer Journal*, 1989, *32*, pp. 127-141.

1. D. A. Basin, Equality of Terms Containing Associative-Commutative Functions and Commutative Binding Operators is Isomorphism Complete, In M. E. Stickel

Ed., 10*th Int. Conf. on Automated Deduction*, Kaiserslautern, Germany, Volume 449 of *Lecture Notes in Artificial Intelligence*, 1990, pp. 251-260, Springer-Verlag.

2. G. BIRKHOFF, On the Structure of Abstract Algebras, *Proc. Cambridge Philos. Soc.*, 1935, *31*, pp. 433-454.
3. N. BOUDRIGA, A. MILI and R. MITTERMEIR, Semantic-based Software Retrieval to Support Rapid Prototyping, *Structured Programming*, 1992, *13*, pp. 109-127.
4. K. B. BRUCE, R. Di COSMO and G. LONGO, Provable Isomorphisms of Types, *Mathematical Structures in Computer Science*, 1992, *2*, (2), pp. 231-247.
5. M. CARLSSON and T. HALLGREN, FUDGETS: A Graphical User Interface in a Lazy Functional Language, In *Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark, 1993.
6. R. Di COSMO, Type Isomorphisms in a Type-assignment Framework: From Library Searches using Types to the Completion of the ML Type Checker, In 19*th ACM Symp. on Principles of Programming Languages*, pp. 200-210, ACM Press, 1992, (Revised version to appear in *J. Functional Programming*).
7. W. B. FRAKES (chair), Panel Session: Information Retrieval and Software Reuse, In *Proc.* 12*th Ann. Int. ACMSIGIR Conf. on Research and Development in Information Retrieval*, pp. 251-256, Cambridge, Mass., USA, 1989, (Special issue of SIGIR Forum, ACM Press).
8. J.-Y. GIRARD, Linear logic, *Theoretical Computer Science*, 1987, *50*, pp. 1-102.
9. Y. LAFONT, The Linear Abstract Machine, *Theoretical Computer Science*, 1988, *59*, pp. 157-180, with corrigenda, *62*, pp. 327-328.
10. J. LAMBEK, From Lambda Calculus to Cartesian Closed Categories, In J. P. SELDIN and J. HINDLEY Eds., *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, 1980, pp. 376-402.
11. E. LINDSTRÖM, Unification in μUTRL. Report UMINF-92.10, Dept Comput. Sci., Univ. of Umeå, S-901 87 Umeå, Sweden (term@cs.umu.se), 1992.
12. S. MAC LANE, *Categories for the Working Mathematician, volume 5 of Graduate Texts in Mathematics*, Springer-Verlag, 1971.
13. B. MATTHEWS, *Reusing Functional Code using Type Classes for Library Search*, Presented at the ERCIM Workshop on Software Reuse, Heraklion, Crete, 1992, Author's address: Dept. Comput. Sci., Glasgow Univ., U.K. (brian@dcs.glasgow.ac.uk).
14. L. MEERTENS and A. SIEBES, *Universal Type Isomorphisms in Cartesian Closed Categories – Preliminary Version*, Centrum voor Wiskunde en Informatica, Amsterdam, the Netherlands (lambert@cwi.nl and arno@cwi.nl), 1990.
15. R. MORGAN, *Component Library Retrieval using Property Models*, PhD thesis, SEAS, Univ. of Durham, England, DH1 3LE (rick@easby.dur.ac.uk), 1991.
16. P. NARENDRAN, F. PFENNING and R. STATMAN, On the Unification Problem for Cartesian Closed Categories, In *Eighth Annual IEEE Symp. on Logic in Computer Science*, 1993.
17. M. RITTRI, Using Types as Search Keys in Function Libraries, *J. Functional Programming*, 1991, *1*, (1), pp. 71-89, (Earlier version in *Func. Prog. Lang. and Comput. Arch.*, ACM Press, 1989).
18. M. RITTRI, Retrieving Library Identifiers via Equational Matching of Types, In M. E. STICKEL Ed., 10*th Int. Conf. on Automated Deduction*, Kaiserslautern, Germany. Volume 449 of *Lecture Notes in Artificial Intelligence*, 1990, pp. 603-617, Springer-Verlag.

19. E. J. ROLLINS and J. M. WING, Specifications as Search Keys for Software Libraries, In K. FURUKAWA Ed., *Eighth Int. Conf. on Logic Programming*, MIT Press, 1991, pp. 173-187.

20. C. RUNCIMAN and I. TOYN, Retrieving Reusable Software Components by Polymorphic Type, *J. Functional Programming*, 1991, *1*, (2), pp. 191-211, (Earlier version in *Func. Prog. Lang. and Comput. Arch.*, ACM Press, 1989).

21. J. H. SIEKMANN, Unification Theory, *J. Symbolic Computation*, 1989, *7*, pp. 207-274.

22. S. V. SOLOVIEV, The Category of Finite Sets and Cartesian Closed Categories, *J. Soviet Math.*, 1983, *22*, (3), pp. 1387-1400, (First published in Russian in *Zapiski Nauchnykh Seminarov Leningradskogo Otdelenya Matematicheskogo Instituta*, 1981, *105*, pp. 174-194).

23. S. V. SOLOVIEV, The Ordinary Identities Form a Complete Axiom System for Isomorphism of Types in Closed Categories, 1991, The Institute for Informatics and Automation of the Academy of Sciences, 199178, St. Petersburg, Russia, e-mail via S. Baranoff, sergei@iias.spb.su. (The author is currently at Aarhus University, Denmark, e-mail: soliviev@daimi.aau.dk).