Journal of Functional Programming 1 (1): 71–89, January 1991

Using types as search keys in function libraries

MIKAEL RITTRI

Department of Computer Sciences, Chalmers University of Technology and University of Göteborg, Sweden

Abstract

A method is proposed to search for an identifier in a functional program library by using its Hindley–Milner type as a key. This can be seen as an approximation of using the specification as a key.

Functions that only differ in their argument order or currying are essentially the same, which is expressed by a congruence relation on types. During a library search, congruent types are identified. If a programmer is not satisfied with the type of a found value, he can use a conversion function (like *curry*), which must exist between congruent types, to convert the value into the type of his choice.

Types are congruent if they are isomorphic in all cartesian closed categories. To put it more simply, types are congruent if they are equal under an arithmetical interpretation, with cartesian product as multiplication and function space as exponentiation. This congruence relation is characterized by seven equational axioms. There is a simple term-rewriting algorithm to decide congruence, using which a search system has been implemented for the functional language Lazy ML, with good performance.

The congruence relation can also be used as a basis for other search strategies, e.g. searching for identifiers of a more general type, modulo congruence or allowing free type variables in queries.

Capsule review

Many functions in a polymorphic language such as ML, Miranda or Haskell may be identified by their type. For example, there are very few common functions with the type $(t \to s) \to (list\ t \to list\ s)$; most readers familiar with polymorphic typing will immediately think of the *map* function when they see this type expression. However, the *map* function has alternate names, such as *apply-to-all*. Therefore it seems very natural to search a program library by examining function types instead of function names.

One obvious problem is that functions of several arguments might be written with the parameters listed in any order. Therefore, when searching a library, we do not simply want to see all functions of a given type, but all functions that could have this type if the order of parameters is changed. Moreover, functions may be 'curried' or 'uncurried'. This difficulty is treated by identifying a natural equivalence relation on types that comes from cartesian closed categories, and examining the relevant algorithmic problems of equivalence, unification and expression matching.

1 Introduction

As a functional programmer, I often want to know whether some function I need exists in a program library. Usually, the only way to find out is to read the library, which may be alphabetical or coarsely sorted by subject matter. In either case, it is tedious to search. There are other systems where one can give a regular expression and get a list of all command names that it matches. This could be useful in a functional programming system, too, but names are hard to guess. Ideally, one would like to search by specification, but even if library functions were formally specified, it would be undecidable whether two specifications were equivalent. Still, a type can be seen as a skeleton of a specification. Types are easy to compare, and libraries normally contain the types of their identifiers.

In some cases, types work well as search keys. For instance, the function that concatenates a list of lists is known under as various names as *concat*, *flat*, and *link*, but in the usual Hindley–Milner type system it can only have the type $\forall\alpha.\ List(List(\alpha)) \Rightarrow List(\alpha)$. [About notation: I will use '$\Rightarrow$' to denote function space, '$\times$' to denote binary cartesian product, and '$\mathbf{1}$' to denote the empty cartesian product. Syntactically, $\Rightarrow$ associates to the right and binds less tightly than $\times$. Greek letters α, β, γ, ... are type variables; capital letters A, B, C, ... stand for types. Bound type variables will be quantified explicitly. (The pronoun 'he' will stand for 'he or she' when it refers to a hypothetic person.)]

Suppose the sought function takes more than one argument; for instance, I might be looking for a function to print a real number x with n significant digits. I would then want to search among functions that take a real number and an integer and return a character list, but since I do not know the argument order, nor whether the function is curried, there are four possible types:

$$Real \Rightarrow (Int \Rightarrow List(Char))$$
$$Int \Rightarrow (Real \Rightarrow List(Char))$$
$$(Real \times Int) \Rightarrow List(Char)$$
$$(Int \times Real) \Rightarrow List(Char)$$

The number of possibilities increases rapidly if the function has more than two arguments, or if it has arguments that are functions.

What I want, then, is an equivalence relation on types, which abstracts from argument order, currying, etc. I could then query a library by giving a type, and get information about all identifiers that have this *or an equivalent* type. I would still have to check whether anything found satisfies my specification, but the number of possibilities will be reduced to a handful rather than a whole library. If I find an identifier with the required semantics, whose type is only equivalent, not identical, to my query, I can either change my program to accept the library type, or else convert the library identifier to the query type by means of a higher order function.

If the type system is polymorphic, I need to decide what to do when a query and a library type differ in generality, but this decision is orthogonal to the choice of equivalence relation. In the main part of this article, I will use the simplest way to handle polymorphism, which is to insist that a retrieved identifier has a type

https://doi.org/10.1017/S095679680000006X Published online by Cambridge University Press

equivalent to the query (allowing renaming of bound type variables, of course). Other possibilities have been presented in Runciman and Toyn (1989) and Rittri (1990*a*), which are reviewed in section 6.

It may also be possible to use types as search keys in program libraries that are not functional, but I have not investigated that, as I think the idea is simpler and more useful in a functional setting.

In section 2 of this article, I define the equivalence relation. Some examples of usage are given in section 3. Section 4 discusses algorithms, and an efficient term-rewriting algorithm is given. A search system using this algorithm has been implemented in and for Lazy ML; the performance is good, as is described in section 5. Related work is reviewed in section 6. Possibilities for further research are discussed in section 7.

2 The equivalence relation

To be able to motivate a particular equivalence relation, we must find good semantics. If we look at the example in the introduction, it is natural to say that two types should be equivalent if it is easy to convert back and forth between them. For instance, the reason why the types $A \Rightarrow (B \Rightarrow C)$ and $(A \times B) \Rightarrow C$ should be equivalent, is that the functions *curry* and *uncurry* can translate values of one type into values of the other. The two functions are total and each other's inverses, which makes the types isomorphic. This isomorphism means that a library programmer has to make an arbitrary choice, and a user cannot know which one was made, which is why his query must be treated as denoting an isomorphism class.

We should now try to find an exact definition of isomorphism. It is possible to make a definition based on a concrete interpretation of types. For instance, we can interpret types as sets, and say that sets are isomorphic if and only if there are bijective functions (in the sense of set theory) between them. But we would then get that *Int* is isomorphic with $Int \times Int$, since both are countably infinite. Although it is true that two integers can be encoded as one, and vice versa, this is hardly a fact we wish to use for our search system. To avoid such unwanted isomorphisms, we will stipulate that we only use isomorphisms that come from the properties of pairs and functions.

From the example in the introduction, we can see that would at least want to use the axioms

$$\begin{aligned} A \times B &\cong B \times A \\ A \times (B \times C) &\cong (A \times B) \times C \\ A \Rightarrow (B \Rightarrow C) &\cong (A \times B) \Rightarrow C \end{aligned}$$

which allow us to ignore argument order and currying. The question is whether these are all axioms of their kind. In fact, since a function that returns a pair can be translated to two functions that return the components, there is a similar axiom

$$A \Rightarrow (B \times C) \quad \cong \quad (A \Rightarrow B) \times (A \Rightarrow C)$$

that cannot be derived from the others.

To state some sort of completeness result, we must be more explicit about what the equivalence relation means. The basic idea of bijective functions can be expressed simply:

https://doi.org/10.1017/S095679680000006X Published online by Cambridge University Press

Definition 1
If A and B are types, and there is a function f of type $A \Rightarrow B$ and a function g of type $B \Rightarrow A$, such that $\lambda x_A . g(fx) = \lambda x_A . x$ and $\lambda y_B . f(gy) = \lambda y_B . y$, then we say that A and B are *isomorphic*, and write $A \cong B$ or sometimes $f: A \cong B$ and $g = f^{-1}$.

To interpret what it means, we must specify which functional language, or λ-calculus, we work in. There are several degrees of freedom here, e.g. strict or lazy evaluation, lifted or non-lifted pairing, monomorphic or polymorphic typing, etc. We will choose typed λβη-calculus with surjective pairing, that is, a monomorphic λ-calculus where the equality of λ-expressions consist of β- and η-convertibility together with some laws for pairing and the empty tuple:

$$\begin{gathered} fst(x_A, y_B) = x_A \\ snd(x_A, y_B) = y_B \\ (fst\ x_{A\times B}, snd\ x_{A\times B}) = x_{A\times B} \\ x_1 = ()_1 \end{gathered}$$

This choice will allow us to borrow a soundness-and-completeness result from category theory. It turns out that the four axioms we have listed earlier, together with three simple axioms for the empty product, indeed characterize the isomorphic types, if we disregard the need to make arbitrary n-tuples isomorphic to nested pairs.

The particular choice of λ-calculus means that the axioms need not hold exactly for other λ-calculi or functional languages, but they do hold in an approximate sense, which should be enough for the purpose of searching a function library.

2.1 Cartesian closed categories

We will in this section briefly describe cartesian closed categories, and give both a semantic and an axiomatic characterization of the isomorphisms we are interested in. For a more detailed introduction to category theory, the reader is referred to Goldblatt (1979).

A *category* consists of a collection of *objects* and *arrows*. Each arrow must have a *domain* and a *codomain*, which are objects. There must also be an associative composition operator on arrows, and for each object there must be an arrow which has that object as both domain and codomain, and which is an identity under arrow composition. Two objects in a category are *isomorphic* if there are arrows between them that compose to identity arrows. An example of a category is the one with sets as objects and total functions between sets as arrows, and ordinary function composition as the associative operator. In this category, the isomorphism criterion is that there are bijective functions between two sets, so sets in this category are isomorphic if they have the same cardinality.

Some categories have binary operators '$\Rightarrow$' and '$\times$' on objects that behave as function space and binary cartesian product, and an object '**1**' that behaves as an empty cartesian product. These are known as *cartesian closed categories*, or *CCC*s, and can be seen as models for typed λβη-calculus with surjective pairing.

It is possible to translate arrows in a CCC into closed λ-expressions in a typed λ-

https://doi.org/10.1017/S095679680000006X Published online by Cambridge University Press

calculus, and vice versa, and also to show that the CCC-definition of isomorphism is equivalent to the definition in terms of the $\lambda\beta\eta$-calculus described earlier (definition 1). The relation between CCCs and typed λ-calculus is described in Lambek (1980), Solov'ev (1983) and Huet (1985).

A particular CCC may have isomorphisms that we do not wish to exploit for our search system, such as the isomorphism between *Int* and $Int \times Int$ in the CCC of sets. We will therefore use the *canonical* isomorphisms only, that is, those that hold in all CCCs. To express this formally, we let object expressions be generated by the grammar

$$e ::= \mathbf{1} \mid e \times e \mid e \Rightarrow e \mid \textit{variable}.$$

Given a cartesian closed category $\mathscr{C}$, a *$\mathscr{C}$-assignment* ψ is a mapping from variables to $\mathscr{C}$-objects. It is extended to a mapping from object expressions to $\mathscr{C}$-objects in the obvious way. If for every cartesian closed category $\mathscr{C}$, and every $\mathscr{C}$-assignment ψ, we have that $\psi(e_1) \cong \psi(e_2)$, we write $\mathrm{CCC} \models e_1 \cong e_2$.

$$\begin{aligned}
A \times B &\cong B \times A \\
A \times (B \times C) &\cong (A \times B) \times C \\
A \Rightarrow (B \Rightarrow C) &\cong (A \times B) \Rightarrow C \\
A \Rightarrow (B \times C) &\cong (A \Rightarrow B) \times (A \Rightarrow C) \\
\mathbf{1} \times A &\cong A \\
\mathbf{1} \Rightarrow A &\cong A \\
A \Rightarrow \mathbf{1} &\cong \mathbf{1}
\end{aligned}$$

Fig. 1. The axiom set Γ.

S. V. Solov'ev (1983) has shown that $\mathrm{CCC} \models e_1 \cong e_2$ if and only if $\Gamma \vdash e_1 \cong e_2$, where '$\vdash$' denotes equational reasoning, and Γ is the axiom set in fig. 1. The soundness of Γ is well known; the completeness less so. For his completeness proof, Solov'ev uses the fact that the natural numbers are objects in a CCC, with $\times$ as multiplication, $\mathbf{1}$ as the number 1, and $A \Rightarrow B$ as $B \uparrow A = B^A$. The arrows in this category are the total set theoretic functions between the numbers when viewed as the finite ordinals: $0 = \varnothing, 1 = \{\varnothing\}, 2 = \{\varnothing, \{\varnothing\}\}, \ldots$, and numbers are isomorphic if and only if they are equal. This leads him to consider the equational theory of natural numbers under 1, $\times$ and $\uparrow$, which can be noted $(\mathbb{N}, 1, \times, \uparrow)$. Solov'ev shows the equational completeness of Γ for this equational theory, which implies its completeness for isomorphisms that hold in all CCCs. Actually, C. F. Martin (1973) had already shown that the first four axioms of Γ are equationally sound and complete for $(\mathbb{N}, \times, \uparrow)$, from which it is easy to derive the corresponding result for Γ and $(\mathbb{N}, 1, \times, \uparrow)$. Solov'ev does not refer to Martin, but provides his own proof, which seems shorter than Martin's; I think this is because Martin gets his result as a corollary of another theorem, which also concerns the operator $+$. A different proof of Solov'ev's result, based on work in Dezani-Ciancaglini (1976), was announced recently in Longo, Asperti and Di Cosmo (1989).

It is easy and instructive to verify the soundness of Γ. We will use the notation f: $A \cong B$ introduced in definition 1; we will also use pattern matching on pairs, with the understanding that it can be translated to applications of *fst* and *snd*, and we will

https://doi.org/10.1017/S095679680000006X Published online by Cambridge University Press

omit type restrictions in λ-expressions, since they can be inferred from context. Using these conventions, we give in fig. 2 λ-expressions that show that the axioms of Γ are valid in any CCC. To verify equational reasoning, we should verify that $\cong$ is a stable congruence relation. Its stability, i.e. the fact that isomorphisms remain true when type expressions are consistently substituted for variables, is clear from the semantics of CCC $\models e_1 \cong e_2$. To see that $\cong$ is an equivalence relation, it is enough to verify the inferences in fig. 3. To see that it is a congruence relation over $\times$ and $\Rightarrow$, it is enough to verify the inferences in fig. 4.

$$\begin{array}{rrcl}
\lambda(x,y).\,(y,x): & A \times B & \cong & B \times A \\
\lambda(x,(y,z)).\,((x,y),z): & A \times (B \times C) & \cong & (A \times B) \times C \\
\lambda f.\,\lambda(x,y).\,fxy: & A \Rightarrow (B \Rightarrow C) & \cong & (A \times B) \Rightarrow C \\
\lambda f.\,(\lambda x.\,fst(fx), \lambda x.\,snd(fx)): & A \Rightarrow (B \times C) & \cong & (A \Rightarrow B) \times (A \Rightarrow C) \\
snd: & 1 \times A & \cong & A \\
\lambda f.f(): & 1 \Rightarrow A & \cong & A \\
\lambda f.\,(): & A \Rightarrow 1 & \cong & 1
\end{array}$$

Fig. 2. The axioms of Γ hold in a CCC.

$$\frac{}{\lambda x.x : A \cong A} \qquad \frac{f : A \cong B}{f^{-1} : B \cong A} \qquad \frac{f : A \cong B \quad g : B \cong C}{\lambda x.g(fx) : A \cong C}$$

Fig. 3. Isomorphism is an equivalence relation.

$$\frac{f_1 : A_1 \cong B_1 \qquad f_2 : A_2 \cong B_2}{\lambda(x_1,x_2).(f_1x_1, f_2x_2) : A_1 \times A_2 \cong B_1 \times B_2}$$

$$\frac{f_1 : B_1 \cong A_1 \qquad f_2 : A_2 \cong B_2}{\lambda g.\lambda x.f_2(g(f_1x)) : A_1 \Rightarrow A_2 \cong B_1 \Rightarrow B_2}$$

Fig. 4. Isomorphism is a congruence relation over $\times$ and $\Rightarrow$.

It is not hard to use the axioms of Γ as a tool to decide isomorphism. We will show how to do this in section 4. [A finite axiomatization is in fact not needed for decidability. Neither of the theories $(\mathbb{N}, +, \times, \uparrow)$ or $(\mathbb{N}, 1, +, \times, \uparrow)$ has a finite axiomatization (Martin, 1973; Gurevič, 1989), but both are decidable (see Macintyre, 1981).]

2.2 Other type operators

The grammar we have used for type expressions is restrictive, since it only allows the type operators **1**, $\times$ and $\Rightarrow$. If we have a modern functional language, in which the programmer can define his own types, it is clear that we must allow type operators of any arity, and the possibility, for example, to derive $List(A) \cong List(B)$ from $A \cong B$.

In equational theories, it is common to add an arbitrary number of operators to the language, and simply say that they can be used to build terms, but that there are no particular axioms concerning them. To ensure that this is sound in our case, we must

https://doi.org/10.1017/S095679680000006X Published online by Cambridge University Press

convince ourselves that all type operators preserve isomorphism. We showed explicitly in fig. 4 that $\times$ and $\Rightarrow$ do so, but we would like a general argument for all type operators. To be precise, if F is an n-ary type operator, we must know that the following inference is valid:

$$\frac{A_1 \cong B_1 \quad \ldots \quad A_n \cong B_n}{F(A_1, \ldots, A_n) \cong F(B_1, \ldots, B_n)}$$

Now, any categorical semantics for a programming language defines things only up to isomorphism; this is the whole spirit of category theory. Since we expect most languages to have categorical semantics, we assume that all type operators preserve isomorphism. It is outside the scope of this article actually to give such semantics, but one treatment, using categories of complete partial orders, can be found in Plotkin (1980).

Some intuition may be gained by looking at a simple example. If $f: A \cong B$, it is easy to verify that $map(f): List(A) \cong List(B)$. So the *map* function on lists, and similar functions on other types, must be used to construct the bijective functions between isomorphic types that contain arbitrary type operators. (A category theorist would say that *List* and *map* form a *functor*, with *List* as the object part and *map* as the arrow part.)

2.3 Removing products

It is worth noting that the arithmetical theory of exponentiation alone, $(\mathbb{N}, \uparrow)$, is decidable and characterized by a single axiom (Martin, 1973). Translated to type syntax, the axiom becomes

$$A \Rightarrow (B \Rightarrow C) \quad \cong \quad B \Rightarrow (A \Rightarrow C)$$

sometimes known as left commutativity, C_L. The corresponding result was proved for typed $\lambda\beta\eta$-calculus in Bruce and Longo (1985), but from results in Dezani-Ciancaglini (1976) rather than Martin (1973).

If we base a search system on the equational theory presented by this axiom, we ignore the argument order of curried functions but nothing else, which may be sufficient if all library functions are as curried as possible. (We would lose the distributivity axiom, but that axiom is probably not crucial in practice.) An advantage of the smaller theory is that it is known how to unify modulo C_L, but not modulo Γ. This will be discussed further in section 7.1.

3 Usage and examples

As mentioned in the introduction, the basic application is to search for identifiers whose types are partially known. The user queries with a type and gets a list of all identifiers of an equivalent type. He will also want the types of these identifiers, and access to the source files where they are defined.

Let us look at an example. There are two higher order functions on lists, that are always provided in functional libraries. Choosing an arbitrary argument order, we can define them informally as

$$\lambda f.\lambda b.\lambda[a_1, \ldots, a_n].f\ a_1\ (\ldots\ (f\ a_{n-1}\ (f a_n\ b))\ldots)$$

https://doi.org/10.1017/S095679680000006X Published online by Cambridge University Press

and

$$\lambda f.\lambda b.\lambda[a_1, \ldots, a_n].f\ a_n\ (\ldots\ (f\ a_2\ (f\ a_1\ b))\ldots)$$

Fig. 5 shows what you would find if you could query some functional programming systems with their type. The figure shows how little standardization there is in the functional community, both of names and types. Five systems use five types (and ten names). Even without introducing **1**s, the two functions can be given 144 different but isomorphic types. If you assume that functions are always as curried as possible (which is not true), there are still 12 possible types. This indicates how crucial it is to take isomorphism into account when searching by type.

ML of Edinburgh LCF[a]	*itlist*	$\forall\alpha\beta.(\alpha \Rightarrow \beta \Rightarrow \beta) \Rightarrow \mathrm{List}(\alpha) \Rightarrow \beta \Rightarrow \beta$
	revitlist	$\forall\alpha\beta.(\alpha \Rightarrow \beta \Rightarrow \beta) \Rightarrow \mathrm{List}(\alpha) \Rightarrow \beta \Rightarrow \beta$
CAML[b]	*list_it*	$\forall\alpha\beta.(\alpha \Rightarrow \beta \Rightarrow \beta) \Rightarrow \mathrm{List}(\alpha) \Rightarrow \beta \Rightarrow \beta$
	it_list	$\forall\alpha\beta.(\alpha \Rightarrow \beta \Rightarrow \alpha) \Rightarrow \alpha \Rightarrow \mathrm{List}(\beta) \Rightarrow \alpha$
Haskell[c]	*foldl*	$\forall\alpha\beta.(\alpha \Rightarrow \beta \Rightarrow \alpha) \Rightarrow \alpha \Rightarrow \mathrm{List}(\beta) \Rightarrow \alpha$
	foldr	$\forall\alpha\beta.(\alpha \Rightarrow \beta \Rightarrow \beta) \Rightarrow \beta \Rightarrow \mathrm{List}(\alpha) \Rightarrow \beta$
SML of New Jersey[d]	*fold*	$\forall\alpha\beta.(\alpha \times \beta \Rightarrow \beta) \Rightarrow \mathrm{List}(\alpha) \Rightarrow \beta \Rightarrow \beta$
	revfold	$\forall\alpha\beta.(\alpha \times \beta \Rightarrow \beta) \Rightarrow \mathrm{List}(\alpha) \Rightarrow \beta \Rightarrow \beta$
The Edinburgh SML library (draft)[e]	*fold_left*	$\forall\alpha\beta.(\alpha \times \beta \Rightarrow \beta) \Rightarrow \beta \Rightarrow \mathrm{List}(\alpha) \Rightarrow \beta$
	fold_right	$\forall\alpha\beta.(\alpha \times \beta \Rightarrow \beta) \Rightarrow \beta \Rightarrow \mathrm{List}(\alpha) \Rightarrow \beta$

[a] Gordon, Milner and Wadsworth, 1979. *Edinburgh LCF*, LNCS **78**.
[b] Cousineau and Huet, 1989. *The CAML Primer*, version 2.6. Projet Formel, INRIA-ENS, France.
[c] Hudak, Wadler (eds.) *et al.*, 1988. *Report on the Functional Prog. Language Haskell* (18 Dec.).
[d] Appel and MacQueen, 1987. A Standard ML compiler, in: Kahn (ed.), *FPCA*, LNCS **274**.
[e] Dave Berry, 1989. LFCS, U. of Edinburgh (21 Mar.). E-mail: db@lfcs.edinburgh.ac.uk

Fig. 5. Search for common list operations.

The example used only the first three axioms of Γ. The other four are not as obviously useful, but note that the common way to simulate lazy evaluation of A-values in a strict language is to represent unevaluated closures of type A by functions of type $\mathbf{1} \Rightarrow A$.

Sometimes, a user may find a value with the desired semantics in a library, and still not be satisfied with its type. For instance, he may want to use a partial application of a curried function, and therefore prefer a particular argument order. It is possible to let the search system provide a conversion function (like *curry*, or the C combinator), which can convert the library value into the type of the user's query. The conversion functions are those that prove that the query type is isomorphic to the library type, as in fig. 2. Most algorithms that decide isomorphism can be adapted to give a conversion function as well, though there are probabilistic algorithms that cannot; see section 4. Even if a search system does not provide conversion functions, the most common case will simply be that a library function takes its arguments in a different order than the user thought, and it will then be easy to figure out a conversion function.

https://doi.org/10.1017/S095679680000006X Published online by Cambridge University Press

More sophisticated ways to search by isomorphism are discussed in sections 6 and 7.

4 Algorithms to decide CCC-isomorphism

At first, it is not obvious that Γ-equality is decidable. But we can use the arithmetical interpretation of the operators and reason like this:

- If two arithmetical expressions *are* equal, the proof can be found by exhaustive search through all proofs, since the axiom system is complete.
- If they are *not* equal, they will have instances that are unequal. These can be found by exhaustive search through all natural numbers.

These two searches can be conducted in parallel, and one of them must terminate.

Equality tests for expressions often use one of the two approaches. Algorithms that rewrite expressions to normal forms (see e.g. Peterson and Stickel, 1981) correspond to a proof search. There are also probabilistic algorithms that replace variables with large random numbers and then check equality, since it is improbable to get equal results by chance (Gonnet, 1984; Schwartz, 1980; Martin, 1971). This is a counter-example search.

When you regard the expressions as types, it is possible that you are not satisfied with just a yes/no answer. You may also want a bijective function whenever the types are isomorphic. The probabilistic methods cannot be used for this version of the problem.

As it turns out, there is a simple term-rewriting algorithm to decide Γ-equality. Before we describe it, we note that the probabilistic algorithms of Gonnet (1984) can also be used. The probability of error of the algorithms can be made arbitrarily small, and their complexity is random polynomial. If a program library is very large, and seldom changes, one can preprocess it and build a hash table, and it is likely that probabilistic algorithms provide a good means to compute hash values. Their advantage is that they give a wide range of values. If a smaller hash table size is enough, it is easier to hash on some other properties which are invariant under isomorphism, such as the number of arguments to a function, or the number of distinct type variables. Furthermore, linear search with the term-rewriting algorithm is often fast enough.

4.1 A term-rewriting algorithm

A term-rewriting algorithm to decide Γ-equality has been given in Solov'ev (1983); a similar one has been sketched in Henson and Rubel (1984, p. 26). We can describe the algorithms in the style of Peterson and Stickel (1981) by singling out the associative and commutative laws into a subtheory AC, and directing the other axioms into a rewrite system R.

$$\mathsf{AC}\begin{cases} A \times B = B \times A \\ (A \times B) \times C = A \times (B \times C) \end{cases}$$

https://doi.org/10.1017/S095679680000006X Published online by Cambridge University Press

$$\mathsf{R}\begin{cases} A \Rightarrow (B \Rightarrow C) \to (A \times B) \Rightarrow C \\ A \Rightarrow (B \times C) \to (A \Rightarrow B) \times (A \Rightarrow C) \\ \mathbf{1} \times A \to A \\ A \times \mathbf{1} \to A \\ \mathbf{1} \Rightarrow A \to A \\ A \Rightarrow \mathbf{1} \to \mathbf{1} \end{cases}$$

If terms are simplified as far as possible by the rules of R, the normal forms will be unique up to AC-equality. Thus, the products in the normal forms are seen as bags (multisets) of factors, and some bag equality algorithm is used (e.g. by sorting by some arbitrary ordering).

The normal forms can be exponentially large, as is always the case when a distributive law is used to distribute. The worst case is expressions of the form

$$(\cdots((A \Rightarrow B_1 \times C_1) \Rightarrow B_2 \times C_2)\cdots) \Rightarrow B_n \times C_n.$$

That is, a pair-returning function is an argument to a pair-returning function, which is an argument to a pair-returning function, etc. Such nesting is not likely to be very deep among commonly occurring types. Furthermore, the duplication of subexpressions can be implemented by shared components in a graph, so the isomorphism test can be done in polynomial space. I do not know if it can be done in polynomial time. The algorithm I have used takes exponential time in the worst case, but performs well in practice.

I give a Standard ML (Harper and Mitchell, 1986) program to decide isomorphism in fig. 6. For simplicity, I have assumed that the only type operators are **1**, $\times$ and $\Rightarrow$, but it would not be hard to handle free operators like *List*. Note that I have not explained how to rename bound type variables; this is discussed in section 5.1.

Most of the ML program should be clear, but it is necessary to show that the function *red'* returns an irreducible expression whenever all proper subexpressions of its input are irreducible. For the first four clauses, this is obvious. For the fifth clause,

$$red'(r \Rightarrow (s \Rightarrow t)) \;=\; (r \times s) \Rightarrow t$$

note that by assumption, r, s, t and $s \Rightarrow t$ are irreducible, and that the clause is overlapped by a previous one when $r = Unit$. This means that when the fifth clause is used, r cannot be *Unit*, s cannot be *Unit*, and t cannot have *Unit*, $\times$ or $\Rightarrow$ as its main operator. These constraints rule out all possibilities of reducing $(r \times s) \Rightarrow t$. For the sixth clause,

$$red'(r \Rightarrow (s \times t)) \;=\; red'(r \Rightarrow s) \times red'(r \Rightarrow t)$$

a similar reasoning shows that none of r, s and t can be *Unit*. This implies that neither of the expressions $(r \Rightarrow s)$ or $(r \Rightarrow t)$ can be reduced to *Unit*. If we use an induction hypothesis and assume that $red'(r \Rightarrow s)$ and $red'(r \Rightarrow t)$ are irreducible, the fact that neither is *Unit* means that their product is irreducible. (The induction is not structural, but can be over the number of symbols in the expressions.) Finally, the default last clause of *red'* is safe to use, since we assume that proper subexpressions are irreducible, and since all possibilities to reduce at top level are captured by the other clauses.

https://doi.org/10.1017/S095679680000006X Published online by Cambridge University Press

```
(* We assume a signature for bags (multisets), with the obvious semantics. (bag_equal eq b1 b2
tells whether b1 and b2 are equal bags, using eq to determine equality on elements.) *)

type 'a bag
empty_bag:'a bag
singleton_bag:'a→'a bag
bag_union:'a bag→'a bag→'a bag
bag_equal:('a→bool)→'a bag→'a bag→bool

type variable = string

(* Abstract syntax for arbitrary type expressions *)
infix × infixr ⇒
datatype texp = Unit|op × of texp * texp|op ⇒ of texp * texp| V of variable
(* Abstract syntax for irreducible type expressions (normal forms) *)
infixr ⇏
datatype factor = op ⇏ of product * variable
withtype product = factor bag
type normalform = product

(* red: texp→texp. Reduce an expression to normal form. *)
fun red (Unit)       = Unit
|   red (V var)      = V var
|   red (s × t)      = red'(red s × red t)
|   red (s ⇒ t)      = red'(red s ⇒ red t)

(* red': texp→texp. Reduce an expression to normal form, assuming that all of its
proper subexpressions are irreducible. *)
and red'(Unit × t)   = t
|   red'(t × Unit)   = t
|   red'(t ⇒ Unit)   = Unit
|   red'(Unit ⇒ t)   = t
|   red'(r ⇒ s ⇒ t)) = (r × s) ⇒ t
|   red'(r ⇒ (s × t)) = red'(r ⇒ s) × red'(r ⇒ t)
|   red'(t)          = t

(* chtype:texp→normalform. Change the type of a reduced 'texp' to
'normalform'. *)
exception Not_normal
fun chtype (V var)   = singleton_bag (empty_bag ⇏ var)
|   chtype (t ⇒ V var) = singleton_bag (chtype t ⇏ var)
|   chtype (t ⇒ _)   = raise Not_normal
|   chtype (Unit)    = empty_bag
|   chtype (s × t)   = bag_union (chtype s) (chtype t)

(* eq_p: product→product→bool
   eq_f: factor →factor→bool *)
fun eq_p p1 p2 = bag_equal eq_f p1 p2
and eq_f (p1 ⇏ var1) (p2 ⇏ var2) = (var1 = var2) andalso (eq_p p1 p2)

(* equivalent: texp→texp→bool *)
fun equivalent s t = eq_p (chtype (red s)) (chtype (red t))
```

Fig. 6. A Standard ML program for deciding isomorphism. The keyword **withtype** should be **using** in some systems. See also section 5.1 on variable renaming.

https://doi.org/10.1017/S095679680000006X Published online by Cambridge University Press

5 Implementation and performance

To check the practicality of the term-rewriting algorithm, I have implemented a search system for, and in, Lazy ML (Augustsson and Johnsson, 1988, 1989). In short, my conclusion is that it is fast enough. (The search system is distributed with version 0·95 of the Lazy ML compiler; a user manual is included, which can also be obtained as Rittri, 1989.)

The primary reason for choosing Lazy ML as object language is that no knowledge about the compiler is needed. Lazy ML is designed for separate compilation, and for each source file that contains a module, the compiler writes a 'type-file', which contains the identifiers that are exported, together with their types. Thus, the search system need only read the type-files, and we can search any program, except that we cannot find identifiers that are local to a module. For an interactive functional system, the search program would either have to use the system's internal type environment or do its own type-derivation from the source files.

The search system parses type-files, and searches linearly for identifiers with a type equivalent to a query. For the type-files of the Lazy ML standard library, with 194 identifiers, a search takes typically 3–6 cpu seconds, 1·5 of which are due to the parsing. The measurements were made on a Sequent Symmetry computer (its cpu is about as fast as that of a Sun-3). The parsing time indicates that the program cannot be speeded up very much while keeping the linear search. Experience from the Lazy ML compiler (Augustsson and Johnsson, 1989) and the parser generator (Uddeborg, 1988) suggests that not much more speed-up than a factor of 2–4 can be expected by switching to an imperative implementation language. On the other hand, it should be simple to preprocess the library and search in a hash table, which ought to improve performance significantly.

5.1 Implementation details

The equivalence test for types is as described in section 4.1, though it can handle free type operators like *List*. Bag equality is decided by a simple quadratic algorithm.

To handle equivalence of type-schemes, where renamings of variables are allowed, the program first checks whether the two type-schemes have the same number of distinct variables. If they do not, they are not equivalent, but if they do, all possible renamings are generated and tested in turn. Thus, if both type-schemes have n variables, $n!$ tests may be performed. This method is clearly dangerous, but type-schemes have hardly ever more than three distinct variables, so it has worked well in practice. It would be better if the equivalence test only generated renamings that might be useful; I think this can be done with a complexity better than factorial.

The parsers for queries and type-files were made with the functional parser generator FPG (Uddeborg, 1988), which generates efficient LR(1) parsers and is easy to use.

6 Related work

I know of two other papers that deal with types as search keys.

https://doi.org/10.1017/S095679680000006X Published online by Cambridge University Press

6.1 Runciman's and Toyn's work

Independently, Colin Runciman and Ian Toyn at the University of York have had the idea of using types as search keys (Runciman and Toyn, 1989), and they support their approach with a statistical investigation of some typical functional libraries. They propose that the search system should find all identifiers whose types are *unifiable* with the query. This is to handle a method where the queries are not necessarily formulated by hand, but perhaps derived from programs with free variables, or from specification laws, etc. The authors do not abstract away from argument order or currying.

Runciman and Toyn also handle the situation when a library function has an extra argument, which corresponds to a value that is constant in the user's application.

We can say that whereas my search method is based on the user's ignorance of argument order and currying, Runciman's and Toyn's is based on the user's ignorance of extra arguments.

They define a generality order on types, which is a combination of the usual generalization (co-instance) and the idea that a type $A \Rightarrow B$ is more general than B. The most natural combination turns out to be a preorder, and to get a partial order, i.e. anti-symmetry, the authors forbid the instantiation of type variables to function types. As an example, they point out that the type of the function *map2*,

$$\forall\alpha\beta\gamma.\ (\alpha \Rightarrow \beta \Rightarrow \gamma) \Rightarrow List(\alpha) \Rightarrow List(\beta) \Rightarrow List(\gamma)$$

becomes more general than the type of *map*,

$$\forall\beta\gamma.\ (\beta \Rightarrow \gamma) \Rightarrow List(\beta) \Rightarrow List(\gamma)$$

Thus, their search system will sometimes find interesting things that mine would not (and vice versa).

In the same vein, I think it possible to regard a product type as more general than its components. Then we could show, for instance, that the type of '*mapstate*',

$$\forall\alpha\beta\gamma.\ (\alpha \Rightarrow \beta \Rightarrow \alpha \times \gamma) \Rightarrow \alpha \Rightarrow List(\beta) \Rightarrow \alpha \times List(\gamma)$$

is more general than both the type of *map*,

$$\forall\beta\gamma.\ (\beta \Rightarrow \gamma) \Rightarrow List(\beta) \Rightarrow List(\gamma)$$

and the type of *foldl*,

$$\forall\alpha\beta.\ (\alpha \Rightarrow \beta \Rightarrow \alpha) \Rightarrow \alpha \Rightarrow List(\beta) \Rightarrow \alpha$$

which confirms the description (in Augustsson and Johnsson, 1988) of *mapstate* as a hybrid between *map* and *foldl*. So it appears to be useful to allow extra arguments and extra results from library functions.

6.2 Matching modulo isomorphism

Since Runciman and Toyn retrieve unifiable types but do not take isomorphism into account, we may wonder if we could not unify modulo isomorphism; such questions are studied in unification theory. It is as yet an open question whether unifiability modulo isomorphism is decidable (see section 7).

https://doi.org/10.1017/S095679680000006X Published online by Cambridge University Press

In Rittri (1990*a*), I have presented an algorithm for matching (or one-way unification) modulo isomorphism. To be precise, a type A is said to match a type B modulo isomorphism if and only if A has a generic instance B' (see Damas and Milner, 1982) which is isomorphic to B. I use the algorithm to retrieve library identifiers of types that are at least as general as the query, modulo isomorphism. My algorithm is based on one by Bernard Lang (1978) for a similar problem. Matchability modulo isomorphism was recently shown to be NP-complete (Narendran, Pfenning and Statman, 1989), but the prototype implementation described in Rittri (1990*a*) gives access times that are usually around 2 CPU seconds when searching the Lazy ML library.

Runciman's and Toyn's reason to retrieve unifiable types is that they want to derive queries automatically from specification laws, context of use, etc. In Rittri (1990*a*), I have still explicitly formulated queries in mind, but I have another reason to allow library identifiers that are more general than the query, namely that it can be hard to figure out the most general type of a function. If we have the code, we can derive the most general Hindley–Milner type automatically, but a user of a search system will have to figure out a type just from the specification he has in mind. See for instance fig. 6. There we requested a bag equality test of type

$$\forall\alpha.\ (\alpha \Rightarrow \alpha \Rightarrow \mathit{Bool}) \Rightarrow \mathit{Bag}(\alpha) \Rightarrow \mathit{Bag}(\alpha) \Rightarrow \mathit{Bool}$$

but many implementations will have the more general type

$$\forall\alpha\beta.\ (\alpha \Rightarrow \beta \Rightarrow \mathit{Bool}) \Rightarrow \mathit{Bag}(\alpha) \Rightarrow \mathit{Bag}(\beta) \Rightarrow \mathit{Bool}$$

David Turner (1985, p. 5) has reported a similar experience with the function *foldr*; it is not obvious that the function argument to *foldr* can take arguments of different type.

7 Future work

7.1 Unification modulo isomorphism

Two types are (generically) unifiable modulo isomorphism is they have generic instances that are isomorphic. In Narendran *et al.* (1989), it is reported as an open question whether such unifiability is decidable, but it is shown that by removing the distributivity axiom from Γ, one gets an equational theory in which unification is NP-complete, and therefore decidable. This result is stated for the case where there are no free type operators (like *List*) in types to be unified, except constants (like *Int*). It does not follow immediately that the algorithm can be extended to arbitrary free operators, but possibly the methods in Schmidt-Schauss (1989) apply.

If we remove the axioms on $\mathbf{1}$, too, we get an even smaller equational theory presented by just the commutativity and associativity of $\times$ and the axiom about currying. The algorithm in Narendran *et al.* (1989) works for this subtheory, too, and the methods in Schmidt-Schauss (1989) can be used to extend it to handle free operators. We can also note that unification modulo the left commutativity of $\Rightarrow$ (see section 2.3) is known to be decidable, even in the presence of free operators (Kirchner, 1984, 1985; Jeanrond, 1980).

https://doi.org/10.1017/S095679680000006X Published online by Cambridge University Press

So if we want to match, we can do it modulo Γ, but if we want to unify, we must currently give up at least the distributivity axiom. Someone might be able to find an algorithm for Γ-unification, but the problem could also be undecidable. And even though we get decidable unification problems by removing the distributivity axiom, it is not clear that the algorithms can be made efficient enough.

A survey of unification under equational theories and an extensive bibliography on unification theory can be found in Siekmann (1989).

7.2 Queries with free type variables

There are search problems that cannot be handled by the method as described so far. The search system can only find a type, if there is an *identifier* that has the type. If you search for a tuple type, the search system will not tell you that the components exist, if there is no identifier bound to the whole tuple. If you search for a function: $A \Rightarrow C$, the search system will not tell you that you could compose a function: $A \Rightarrow B$ with a function: $B \Rightarrow C$. And if you search for a type B, the system will not tell you whether there is a function: $A \Rightarrow B$, that you could apply to an A-value to get a B-value (see section 6.1).

Each of these cases could be handled separately, but it would be better to have one general method for them all. One idea is worth mentioning. In the case above, where we cannot find a function of type $A \Rightarrow C$, and suspect that there are two functions that go via an unknown intermediate type B, it is tempting to pose a Prolog-like conjunctive query

$$A \Rightarrow \beta, \quad \beta \Rightarrow C$$

which the system can answer by finding a type B to substitute for β, and two identifiers of types $A \Rightarrow B$ and $B \Rightarrow C$. But what exactly is β here? It looks like a type variable, but type variables are normally bound locally in a Hindley–Milner type. Since we want the same β at both occurrences, we conclude that β is a free type variable. Damas and Milner (1982) use both free and bound type variables to explain the Hindley–Milner type system, and even if the derived type of an expression always is closed in a closed type environment, there is no reason to demand that queries are closed. The following definition seems natural:

Definition 2

The answer to a query containing free type variables is the union of the answers to all closed instances of the query.

An instance here means to instantiate the free type variables, rather than a generic instance, which means to instantiate the bound ones (Damas and Milner, 1982).

If we want to retrieve identifiers of types more general than an open query, we may have to instantiate both the bound variables in the library types and the free variables in the query. It is not hard to show that we can solve this problem if and only if we can unify, which was discussed in the previous section. But note that the distinction between free and bound variables makes the query language more expressive. In many cases, an experienced functional programmer *knows* that a function is polymorphic, and he can then use a bound type variable in his query. In other cases,

https://doi.org/10.1017/S095679680000006X Published online by Cambridge University Press

he wants a place-holder for a type that is simply unknown to him, such as B in the example above, and he can then use a free variable.

As another example of how free type variables can be used, not that they can be used to imitate Runciman's and Toyn's search strategy (see section 6.1). If we search for identifiers of a type $A \Rightarrow B$ and want to express that we accept extra arguments in a library type, we can query with a type $(\alpha \times A) \Rightarrow B$, where α is a free type variable that can be instantiated to a tuple type (several extra arguments), or to a single type (one extra argument), or to **1** (no extra arguments). To allow extra arguments at any level, such as in the function argument to *map2* in section 6.1, we can traverse the query and insert new free variables where necessary.

7.3 Searching for types

A variation of the theme is when we do not search for an identifier of a certain type, but for a type of a certain structure. For instance, we may want to know whether there is a standard way to add an extra element to a type. In the literature, I have seen various names:

datatype $Option(\alpha) = Some\ \alpha \mid None$
datatype $Isit(\alpha) = Is\ \alpha \mid Isn't$
datatype $Try(\alpha) = Succeed\ \alpha \mid Fail$

I am not sure whether questions of this kind are common enough to warrant a special treatment. Still, programs would certainly be easier to read if programmers used the same names for the same constructs, and they will not do so unless there is a fast way to find the standard names. That goes for type names as well as for function names.

8 Summary and conclusions

I have proposed two commands that a search system could execute, given a query type A:

1. Retrieve each identifier of a type CCC-isomorphic with A.
2. Retrieve each identifier of a type more general than A, modulo CCC-isomorphism.

If queries may contain free type variables, conjunctive queries in the Prolog style would also be useful.

I have shown in this article that when the query is closed, the first command is simple to implement and yields efficient search.

To implement the first command for queries with free type variables, or the second command for closed queries, it is enough to implement matching modulo CCC-isomorphism, and this problem is decidable but NP-complete (Narendran *et al.*, 1989). The algorithm in Rittri (1990*a*) gives acceptable access times for most queries.

To implement the second command for queries with free type variables, we need to unify modulo CCC-isomorphism, but it is an open question whether this is decidable. However, unification becomes decidable if we drop the distributivity axiom (Narendran *et al.*, 1989).

https://doi.org/10.1017/S095679680000006X Published online by Cambridge University Press

I conjecture that unification algorithms can be made efficient enough for practical use. It should also be possible to preprocess large libraries to get improved many-to-one unification.

Acknowledgements

My colleague Dan Synek has spent a lot of time discussing these ideas with me. It was he who first recognized that the equivalence relation I was trying to define should be the same as arithmetic equality. Peter Dybjer, my advisor, turned the connection between equivalent types and equal arithmetic expressions from handwaving into category theory. He has also read the article carefully and given many valuable comments. I have got further suggestions for improvement from an anonymous referee. I owe thanks also to the Programming Methodology Group at Chalmers.

The following people have helped me to find relevant literature: Stefan Arnborg, Bengt Aspvall, Tom Blenko, Juliusz Brzezinski, Jürgen Buntrock, Hans-Jürgen Bürckert, John Case, Keith Clarke, H. Comon, Roberto Di Cosmo, Conal Elliott, Torkel Franzén, Stephen J. Garland, Joseph Goguen, Tony Hearn, Gérard Huet, Michael Johnson, Claude Kirchner, Torbjörn Lager, Lambert Meertens, Albert R. Meyer, Bernard Lang, Daniel Lazard, Pierre Lescanne, Giuseppe Longo, G. E. Mints, Paliath Narendran, Kim H. Pedersen, Frank Pfenning, Andrew Pitts, Colin Runciman, Erik Tidén, Ian Toyn, Matthew P. Wiener, and A. J. Wilkie.

Sequent Symmetry is a trademark of Sequent Computer Systems, Inc. Sun-3 is a trademark of Sun Microsystems, Inc. Miranda is a trademark of Research Software Ltd.

This article is a revised version of a paper that was presented at the Fourth International Conference on Functional Programming Languages and Computer Architecture, London, UK, Sept. 11–13, 1989; published in the *Proceedings of the 1989 Conference on Functional Programming Languages and Computer Architecture*.

References

Augustsson, L. and Johnsson, T. 1988. Lazy ML User's Manual. Dept. of Comp. Sci., Chalmers Univ. of Tech., S-412 96 Göteborg, Sweden.

Augustsson, L. and Johnsson, T. 1989. The Chalmers Lazy-ML compiler. *The Computer Journal*, 32(2).

Bruce, K. B. and Longo, G. 1985. Provable isomorphisms and domain equations in models of typed languages (preliminary version). In *17th ACM Symp. on the Theory of Computing*, Providence.

Damas, L. and Milner, R. 1982. Principal type-schemes for functional programs. In *9th ACM Symp. on Principles of Programming Languages*.

Dezani-Ciancaglini, M. 1976. Characterization of normal forms possessing an inverse in the λβη-calculus. *Theoretical Comp. Sci.*, 2:323–37.

Goldblatt, R. 1979. *Topoi: The Categorial Analysis of Logic*. Volume 98 of *Studies in Logic and the Foundation of Mathematics*. North-Holland, Amsterdam.

Gonnet, G. H. 1984. Determining the equivalence of expressions in random polynomial time. In *16th ACM Symp. on the Theory of Computing*, Washington DC.

Gurevič, R. 1989. Equational theory of positive numbers with exponentiation is not finitely axiomatizable. To appear in *Ann. of Pure and Appl. Logic.*

Harper, R. and Mitchell, K. 1986. Introduction to Standard ML. Technical report, LFCS, Dept. of Comp. Sci., Univ. of Edinburgh.

Henson, C. W. and Rubel, L. A. 1984. Some applications of Nevanlinna theory to mathematical logic: Identities of exponential functions. *Trans. of Am. Math. Soc.*, 282(1): 1–32 (Mar.).

Huet, G. 1985. Cartesian closed categories and lambda calculus. In G. Cousineau, P.-L. Curien and B. Robinet (editors), *Combinators and Functional Programming Languages.* Volume 242 of *Lecture Notes in Computer Science*, Springer-Verlag.

Jeanrond, H. J. 1980. Deciding unique termination of permutative rewriting systems: Choose your term algebra carefully. In W. Bibel and R. Kowalski (editors), *5th Conf. on Automated Deduction*, Les Arcs, France. Volume 87 of *Lecture Notes in Computer Science*, Springer-Verlag.

Kirchner, C. 1984. Standardization of equational unification: Abstract and examples. Internal Report 84-R-074, Centre de Recherche en Informatique, Nancy, France, E-mail: ckirchner@crin.crin.fr.

Kirchner, C. 1985. *Méthodes et outils de conception systématique d'algorithmes d'unification dans les théories équationelles.* PhD thesis, Université de Nancy I, France. E-mail: ckirchner@crin.crin.fr.

Lambek, J. 1980. From lambda calculus to cartesian closed categories. In J. P. Seldin and J. Hindley (editors), *To H. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pp. 376–402. Academic Press.

Lang, B. 1978. Matching with multiplication and exponentiation (May). Author's current address: INRIA, Domaine de Voluceau, Rocquencourt, B.P. 105, F-78153 Le Chesnay Cedex, France. E-mail: lang@inria.fr.

Longo, G., Asperti, A. and Di Cosmo, R. 1989. Coherence and valid isomorphism in closed categories. In D. H. Pitt *et al.* (editors), *Category Theory and Computer Science*, Manchester, UK. Volume 389 of *Lecture Notes in Computer Science*, Springer-Verlag.

Macintyre, A. 1981. The laws of exponentiation. In C. Berline *et al.* (editors), *Model Theory and Arithmetic.* Volume 890 of *Lecture Notes in Mathematics*, pp. 185–97. Springer-Verlag.

Martin, C. F. 1972. Axiomatic bases for equational theories of natural numbers. *Notices of Am. Math. Soc.*, 19(7): A-778–779, Abstract 698-E1.

Martin, C. F. 1973. *Equational Theories of Natural Numbers and Transfinite Ordinals.* PhD thesis, Univ. of California, Berkeley, CA 94720, USA. The relevant results appear also in Martin (1972).

Martin, W. A. 1971. Determining the equivalence of algebraic expressions by hash coding. *JACM*, 18(4): 549–58 (Oct.).

Narendran, P., Pfenning, F. and Statman, R. 1989. On the unification problem for cartesian closed categories. Addresses: P. Narendran, State Univ. of New York at Albany, USA. F. Pfenning and R. Statman, Carnegie Mellon Univ., Pittsburgh, USA. E-mail: dran@cssun.albany.edu, fp@cs.cmu.edu, statman@c.cs.cmu.edu.

Peterson, G. E. and Stickel, M. E. 1981. Complete sets of reductions for some equational theories. *JACM*, 28(2): 233–64 (Apr.).

Plotkin, G. D. 1980. Domains. Lecture notes, Univ. of Edinburgh, UK.

Rittri, M. 1989. lmlseek – a program to search for identifiers in LML programs by type (July). PMG Memo 69, Dept. of Comp. Sci., Chalmers Univ. of Tech., S-412 96 Göteborg, Sweden. Email: rittri@cs.chalmers.se.

Rittri, M. 1990*a*. Retrieving library identifiers via equational matching of types. In M. E. Stickel (editor), *10th Int. Conf. on Automated Deduction*, Kaiserslautern, W. Germany. Volume 449 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag. This is a condensed version of Rittri (1990*b*), Part C.

Rittri, M. 1990*b*. *Searching Program Libraries by Type and Proving Compiler Correctness by*

Bisimulation. PhD thesis, Dept. of Comp. Sci., Chalmers Univ. of Tech. and Univ. of Göteborg, S-412 96 Göteborg, Sweden. E-mail: rittri@cs.chalmers.se.
Runciman, C. and Toyn, I. 1989. Retrieving re-usable software components by polymorphic type. In *4th Int. Conf. on Functional Programming Languages and Computer Architecture*, London, UK., ACM Press, Addison-Wesley.
Schmidt-Schauss, M. 1989. Unification in a combination of arbitrary disjoint equational theories. *J. of Symbolic Computation*, 8: 51–99.
Schwartz, J. T. 1980. Fast probabilistic algorithms for verification of polynomial identities. *JACM*, 27(4): 701–17 (Oct.).
Siekmann, J. H. 1989. Unification theory. *J. of Symbolic Computation*, **7**, 207–74.
Solov'ev, S. V. 1983. The category of finite sets and cartesian closed categories. *J. of Soviet Mathematics*, 22(3): 1387–1400. This is an American Mathematical Society translation of the Russian original in *Zapiski Nauchnykh Seminarov Leningradskogo Otdeleniya Matematicheskogo Instituta im. V. A. Steklova AN SSSR*, Vol. 105, 1981, pp. 174–94.
Turner, D. A. 1985. Miranda: A non-strict functional language with polymorphic types. In J.-P. Jouannaud (editor), *Functional Programming Languages and Computer Architecture*, Nancy, France. Volume 201 of *Lecture Notes in Computer Science*, pp. 1–16, Springer-Verlag.
Uddeborg, G. O. 1988. A functional parser generator. Licentiate thesis, Dept. of Comp. Sci., Chalmers Univ. of Tech., S-412 96 Göteborg, Sweden. Also as PMG Report 43 (without source code).

Mikael Rittri, Programming Methodology Group, Department of Computer Sciences, Chalmers University of Technology and University of Göteborg, S-412 96 Göteborg, Sweden (E-mail: rittri@cs.chalmers.se).