Retrieving re-usable software components by polymorphic type

Colin Runciman
Ian Toyn
Department of Computer Science
University of York, YORK YO1 5DD, UK.

ABSTRACT

Polymorphic types are labels classifying both (a) defined components in a library and (b) contexts of free variables in partially written programs. We propose to help programmers make better use of software libraries by providing a system that, given (b), identifies candidates from (a) with matching types. Assuming at first that matching means unifying (*ie* having a common instance) we discuss efficient ways of implementing such a retrieval system, and also indicate its likely effectiveness based on a quantitative study of currently available libraries. Later we introduce the *applicative instance* relation between types, which captures some intuitions about generalisation/specialisation, and discuss its use as the basis of a more flexible system.

1. INTRODUCTION

Much programming effort can be saved by making good use of libraries of pre-defined components. In a functional language the majority of such components are functions and many of them are *higher order* functions, extensive use of which can dramatically reduce the size of programs. But for components to be re-used they must be easy to find and recognise as useful; they must also be easy to combine safely with other components.

The type system adopted for a programming system has a significant impact on ways components can be combined. At one extreme, completely typeless programming systems offer no resistance at all to the combination of arbitrarily chosen components; but they also provide no indication or assurance that such combination is sensible. Conversely, there are strongly typed systems that aim to guarantee as nearly as possible that no conflict can arise from the combination of components, by insisting on identically typed facets; but they also forbid many combinations which are in fact quite reasonable from another point of view. Polymorphic type systems, originated by Milner,[1] combine some of the advantages of each extreme, offering flexibility of combination despite a considerable degree of security. Such type systems have particularly found favour with the designers of functional programming languages[2,3,4] in which it is desirable to impose some discipline on the use of higher order functions without unduly restricting the expressive power that such functions provide to the programmer.

A polymorphic type system can also play a significant role in the discovery of suitable components prior to their combination with others in a program. The requirement for safe combination with respect to polymorphic types can be used to constrain a search for suitable components in a library.

After a summary of polymorphic type essentials (§2) we briefly describe a representative library of polymorphically typed components (§3). Throughout the paper, statistics about this library are given to illustrate the practical consequences of what is being discussed.

In §4 and §5 we examine two complementary ways of applying polymorphic type information to help programmers retrieve library components that are good candidates for re-use. The first approach (§4) uses polymorphic type information, either explicitly given or implicit in the intended context of use, **as a *key*** with which to retrieve components of matching type. A problem with key-matched access is that the number of components it yields is unpredictable. If there are very few (perhaps even zero), it may be desirable to relax the precise matching requirements in favour of something more flexible; if there are very many, the organised presentation of components becomes extremely important. These requirements lead to a complementary approach (§5) defining structures over collections of polymorphic types so that collections of components can be explored in a disciplined manner.

In §6 we consider related work on a retrieval system using a different relation between types. We also discuss possible developments such as a system to handle combinations of components, and the use of component descriptions other than polymorphic types. Finally, §7 summarises and concludes.

2. POLYMORPHIC TYPES

We assume some knowledge of polymorphic type systems, and only provide a brief summary of the essentials, with our notational conventions. Those unfamiliar with polymorphic type systems are referred to the tutorial paper by Cardelli,[5] or to the relevant chapters in Peyton Jones' book.[6]

Let there be three primitive types of value – `num` (numbers) `char` (characters) and `bool` (truth values) – and three forms of construction for more complex types.

`t -> u`	functions with argument type `t` and result type `u`
`t, u`	pairs with left and right components of types `t` and `u`
`[t]`	lists of zero or more items each of type `t`

Since the type `u` may itself be functional or paired, this allows functions of more than one argument (as curried functions) and heterogeneous structures with more than two components (as binary trees). By convention the two infix type constructors are right associative, and pair construction binds more tightly than function construction. Brackets may be used to indicate other associations. Single letters are used to denote type variables, for which other types may be substituted, whereas names longer than one letter denote specific types such as the primitives.

Examples The function `and` yields the logical conjunction of two truth valued arguments; `length` computes the length of its list argument. Their types are:

```
and       bool -> bool -> bool
length    [a] -> num
```

The higher order function `fold` applies its first argument (a binary function) as an operator between the items of its second argument (a list) yielding a single value of the item type.

```
fold op [a; b; ... ; n] =
     a op (b op (... op n))
```

The higher order function map applies its first argument (a function) to each item in its second argument (a list) yielding another list.

```
map f [a; b; ... ] = [f a; f b; ... ]
```

The polymorphic types of `fold` and map are:

```
fold     (a -> a -> a) -> [a] -> a
map      (a -> b) -> [a] -> [b]
```

Definitions *(polymorphic, monomorphic)* If a type formula contains one or more type variables, the formulated type is said to be *polymorphic*; otherwise it is *monomorphic*. □

Two formulae differing only by consistent renaming of variables denote equal types. All subsequent uses of the term "unique" applied to types must be understood to mean "unique up to renaming of type variables". By convention, we usually assign the variables of a formula the names a, b, c ... in order of their first occurrence.

Definitions *(type instance, type unification)* One type t is an *instance* of another u, and we write $t \leq u$, if the formula for t can be obtained from that for u by performing some consistent substitution of type formulae for type variables. Two types t and u are *unifiable*, and we write $t \approx u$, if they have a common instance. Whenever $t \approx u$, among all their common instances there is a unique maximal type with respect to $\leq$; we shall term this type the *unification* of t and u, denoted $t \downarrow u$. □

The relation $\leq$ is reflexive, transitive and antisymmetric – a partial order. The relation $\approx$, although reflexive and symmetric, is *not* transitive and therefore not an equivalence. The operator $\downarrow$ is both commutative and associative.

In most polymorphically typed programming systems, types are inferred automatically from programmed definitions without the need for any explicit type declarations on the programmer's part. There may, however, be the option for the programmer to declare types, and any such declarations are checked against the results of type inference. Unification provides the basic mechanism for type-inference and type-checking. Components may be combined if the types presented at the interface between them have a common instance. Type inference and checking for a complete program amounts to the formation and solution of a collection of simultaneous type equations. For details of the basic algorithms by which this task can be performed see Cardelli's paper.[5] For programs of any size subject to frequent modification (for example, during their initial development) an incremental algorithm is desirable.[7]

3. A REPRESENTATIVE LIBRARY OF COMPONENTS

For experimental purposes, we have assembled a library containing 203 components of 119 different types, 88 of them polymorphic types. The distribution of components per type is shown in Figure I.

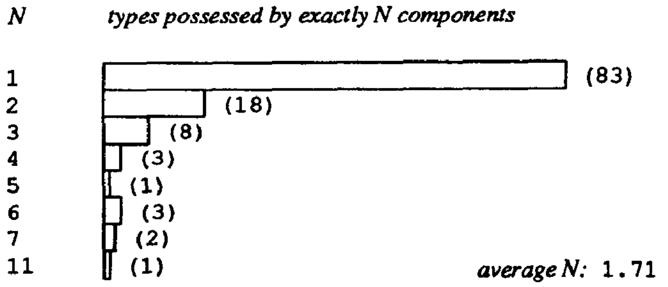


Figure I: components per type in library

This library is representative of the components typically offered for re-use in the context of functional programming systems. The components it contains are drawn from the standard libraries and preludes of four different functional programming systems: Glide,[8] LML,[9] Miranda[3] and OL.[10]

Determining what proportion of other types unify with any one type gives some idea of the balance struck between flexibility and constraint. A completely typeless system can be likened to one in which every type is a simple variable, so the answer would be 100%. In a conventional (monomorphic) strongly typed system the answer would be 0%. For our library of definitions under a polymorphic type discipline the answer is under 2% on average. Figure II shows the distribution.

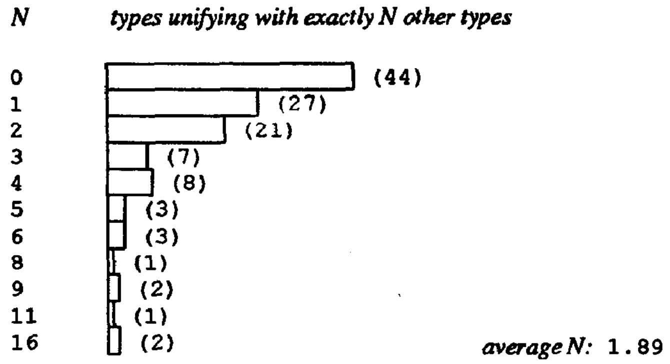


Figure II: number of other types unifiable per type in library

4. RETRIEVAL BY KEY TYPE

Assume that the number of definitions per type and the proportion of other types unifiable are independent. Assume also that the types of components in the library are a representative sample of component types more generally. Then a search for a component in our representative library that unifies with some given key type can be expected to retrieve about 5 definitions from just over 200 available. This rough estimate of the degree of selectivity that would be obtained by using types as library keys seems sufficiently attractive to warrant further investigation of the idea.

4.1. Sources of Key Types

Key types from declarations

Programmers could be required to supply a fully explicit type formula, to be used as the search key, as part of every request for information about suitable components. If the programming system is one that encourages or even requires explicit type declarations anyway, this makes no additional demand on the programmer. But one of the attractions of many systems is precisely that they do not require the programmer to formulate types. So although any library access mechanism might *allow* fully explicit formulation of key types, ideally it should not *require* it.

Key types from examples

Asking for specifications of key types is a little less demanding if the programmer can describe them in terms of the types of components they already know. This would be especially straightforward, for example, in the special case where a programmer already has a definition for the very component required, but would prefer to use a library definition if there is one. More generally, functions could be described by hybrid formulae in which expressions are permitted as the operands of ->. So, for example,

```
f x -> x
```

would describe the type of an inverse for the (known) function `f`.

Key types from laws

Increasingly, programmers formulate laws specifying required functions in the early stages of software development. Assuming that such laws adhere to the polymorphic type discipline, types inferred from them provide convenient search keys for existing definitions of specified components.

Example A law specifying a required function `positions` might relate it to two other functions, `member` and `item`.

```
member p (positions e x) <=> item p x = e
```

If `member` and `item` are known functions with types

```
member    a -> [a] -> bool
item      num -> [a] -> a
```

then the following type for `positions` may be inferred from the above law.

```
positions  a -> [a] -> [num]
```

Although some components can be characterised by a single specifying law, there are often several different laws about the same component. Since a key type can be inferred from each law, there may be several different keys.

Key types from contexts of use

Programs are frequently developed "top down", by so-called *stepwise refinement*.[11] This means, in particular, that the definition of a component typically precedes the definition (or retrieval from a library) of its auxiliary sub-components. Type information about an as yet undefined component can therefore be inferred automatically from the contexts in which it is used.

Example Suppose the `positions` function has been used as an auxiliary in the following definition.

```
pos1 x xs = head (positions x xs)
```

Given that the primitive `head` has type

```
head     [a] -> a
```

from the context of this one application of `positions` alone we may infer the following type information.

```
positions  a -> b -> [c]
```

As with the use of specifying laws, multiple uses of a name may give rise to multiple key types.

So the programmer need not give any *explicit* declarations or specifications of components to provide source of type keys. There may be sufficient information in the *implicit* description of a component in the contexts of its use during the ordinary course of top-down program development. An incremental polymorphic type-checker, such as the one we described in an earlier paper,[7] can be used to infer suitable keys from such contexts.

4.2. Matching Components against Key Types

A set of key types, obtained from sources such as those described in the previous section, defines a set of components to be retrieved from a library. Successful unification of key and component types guarantees valid application of the component in the contexts from which the keys were derived (at least so far as a polymorphic typing system is concerned). However, the library may contain a very large number of components representing a large number of types. Also, there may be not just one key type, but several, derived from various sources. If `L` is the set of library component types and `K` is the set of key types, how can we avoid $\#L \times \#K$ unification tests? We first discuss ways to reduce $\#K$, and then ways to reduce $\#L$.

Key type reduction

An efficient matching procedure should consider key types in combination. The basic observation to be used is that if one type `t` is an instance of another `u`, then whatever can be unified with `t` can also be unified with `u`. Conversely, whatever *cannot* be unified with `u` *cannot* be unified with `t` either.

Theorem *(instance unification)*

$$(t \le u \wedge t \approx v) \Rightarrow u \approx v$$

Proof

From the definitions of $\le$ and $\approx$, for some substitutions $\sigma_1, \sigma_2, \sigma_3$:

$$t = \sigma_1 u \quad \& \quad \sigma_2 t = \sigma_3 v \ .$$
$$\therefore\ \sigma_2(\sigma_1 u) = \sigma_3 v$$
$$\therefore\ (\sigma_2 \circ \sigma_1) u = \sigma_3 v$$
$$\therefore\ u \approx v$$

□

The set of key types can therefore be reduced by discarding all but the minimal elements with respect to $\le$. More importantly, there is a technique often used to improve algorithms cast in the *generate and test* form. Dijkstra[12] puts it in the form of a design rule *Search for the Small Superset.* Darlington[13] views it as a transformation he calls *filter promotion.* Although there are such variations of perspective, the same idea is in view: determine the strongest simple condition that is implied by the *test* part of the algorithm; this condition can be applied as a cost-cutting preliminary test or even built into the *generate* part. Our problem here is to devise a simple pre-test that each component type must pass before we test whether it unifies with every minimal key type. The idea is to compute from the key types a single type `x` representing as much of their common requirements as possible: the pre-test is whether a component type unifies with `x`. The instance unification theorem tells us that this pre-test is valid if `x` is a co-instance of every key type. The less general `x` is, the tougher the pre-test, so we are led to the following definition.

Definition *(type co-instance, type co-unification)* If t≤u then u may be termed a *co-instance* of t. For any types t and u their *co-unification*, denoted t↑u, is their minimal common co-instance. □

The co-unification of a set of types has also been termed by others[14] their *common part*. The ↑ operator is both commutative and associative. So from any set of minimal key types, a co-unified key type t can be computed. Any component whose type does not unify with t can be filtered out from the key-matching process before the costly multiple-key unification stage is reached.

Example Consider the following pair of key types. Neither one is an instance of the other.

```
a -> [char] -> [[char]]
num -> [a] -> [[a]]
```

Co-unified Key: `a -> [b] -> [[b]]` □

So pre-testing components against a co-unified key can reduce the computational effort of matching against several minimal key types.

A co-unified key incorporates only the type structure that is ***required by all*** the keys. It is tempting to think that when a set of keys can be unified, their unification could be made the basis of a sharper test: it incorporates all type structure ***required by any*** key.

Example Consider again the key types of the previous example.

```
a -> [char] -> [[char]]
num -> [a] -> [[a]]
```

Unified Key: `num -> [char] -> [[char]]` □

But this would be a mistake. Keys may not be unifiable even though there are component types that unify with all of them. More importantly, a component type may unify with each of a set of unifiable keys, yet not with the unified key. To illustrate, consider

Component Type: `a -> [a] -> [[a]]`

in connection with the key types in the above examples.

Component type reduction

We have seen that exploiting relations between *key* types can reduce the cost of matching. To reduce cost further, consider relations between *component* types. There are several important ways in which a set of library component types L typically differs from a set of key types K:

(1) L is much *larger* than K;

(2) L is much *less frequently changed* than K;

(3) L is much *more varied in content* than K.

(1) and (2) indicate the increased value and attraction in principle of some form of pre-processing of L. But (3) indicates that the applications of the instance unification theorem to K are unlikely to be much use for L. Indeed, in our representative library, the structure induced by the ≤ ordering over component types is *extremely* shallow. The great majority of component types are minimal, so reducing library types to a minimal subset would gain little. The co-unification of all the library component types is completely polymorphic – a single variable – affording no power of discrimination whatever against keys with no matching component. Search for components *via* the ≤ diagram, artificially completed by a top element, is unsatisfactory: there is branching of massive degree at the root, and little other structure.

This leads to a consideration of weaker relations between types that correlate in some way with unification. A very similar problem arises in the implementation of the programming language Prolog. Instead of a key type there is a current goal to be satisfied, and instead of the collection of library component types there is a collection of available heads of program clauses. Goals match clause heads exactly if the two can be unified. The collection of clauses may be large and is (almost) static. Although the problem is not quite the same as ours (for instance matching clauses *must* be considered in the order they are found in the Prolog program) it is very close. A technique used by some Prolog implementations[15,16] is *clause indexing*: clauses are indexed by a hash value computed from the outermost structure of one or more argument patterns. A similar idea can be used here. In view of the treatment of multi-argument functions by currying, it will be worth comparing indexes based on *initial* arguments of type constructors with indexes based on *final* arguments.

Definition *(initial/final index of a type)* The *initial/final index* of a type formula is a sequence of zero or more type constructor symbols, possibly followed by a primitive type name. The index of a type variable is empty. A primitive type name is its own index. The index of a constructed type is the constructor followed by the index of its initial/final argument. □

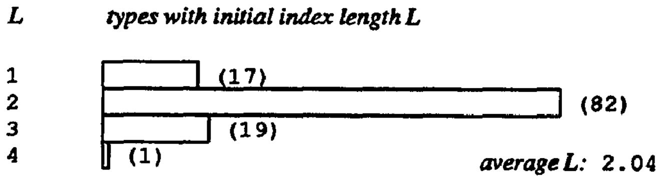
Examples

type	`num -> [a] -> [[a]]`
initial index	`-> num`
final index	`-> -> [] []`
type	`[a,b] -> [a] , [b]`
initial index	`-> [] ,`
final index	`-> , []`
type	`(a -> a -> a) -> [a] -> a`
initial index	`-> ->`
final index	`-> ->`

□

Notice how the final index of a functional type represents its arity and the index of its result type, but ignores argument types. The reverse is the case for the initial index.

In our representative library, average final index length is about 50% greater than average initial index length. For details of the distribution see Figure III.



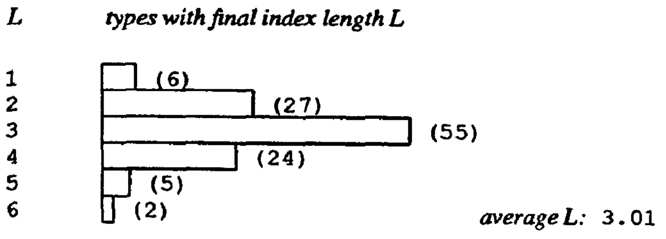


Figure III: length distributions for initial and final indexes

The average size of index-equivalent type classes for final indexes is only 2.76, which is little more than half the corresponding figure of 5.17 for initial indexes.

Theorem *(index unification)* If two types unify, then the initial (final) index of one must be a prefix of the initial (final) index of the other, allowing the case where the two indexes are equal.

Proof The index of t is a prefix of the index of σt for any substitution σ. So, since t$\downarrow$u = σ_1t = σ_2u for some σ_1 and σ_2, the index of t$\downarrow$u must include among its prefixes the indexes of both t and u. So one of these two indexes must be a prefix of the other. □

Therefore arrange the library of components as a variadic tree of type indexes, with the successors of each node corresponding to its minimal extensions. For each node in this tree there is a collection of types, and for each type a collection of components. To access the tree using a key type:

(1) compute the index of the key type;

(2) determine the path from the library root through increasing prefixes of the key index to the maximal occurring prefix MP (the full key index if it occurs); all components with type indexes on this path are candidates;

(3) determine which, if any, successors of MP are extensions of the key index; all components with type indexes in subtrees rooted by one of these successors are candidates.

By computing indexes for *both* the reduced key type *and* each actual key in the minimal set, parts of the candidates subtrees in (3) can be pruned away.

The effectiveness of this method can be gauged from the proportion of library types against which a key is matched. For our representative library, using library types themselves as sample keys, a key lookup requires an average of 17.66 attempted unifications; so a little under 15% of the 119 types are involved.

An ordering based on index prefixes provides a simpler and smaller structure than ≤: it is a tree rather than a DAG, and has a far more limited branching factor. For the final index tree of our representative library, the average number of successors of a non-leaf index is 2.9, with a maximum of 6 and a minimum of 1. By inserting a *small number* of componentless types into the tree each index extension can be made of unit length: of 41 final index extensions only 7 are non-unit. If all extensions are of unit length, there are fast methods of successor selection when searching the tree, and the tree representation may assume a branching factor bounded by the sum of the number of primitive types and the number of type constructions.

4.3. Assessment of the Key Type Approach

There are very plausible scenarios in which such a system, based on retrieval against inferred key types, "wins" by delivering a small set of components including the very one required. However, there are also circumstances in which it "loses" by failing to provide what's wanted. On the one hand, when the amount of information from which to infer key types is limited, the ability to reject candidate components is correspondingly limited. Instead of a small number of likely components for inspection, the programmer may be faced with an unmanageably large batch. Other than tedious re-submission of library requests in a revised context, there is nothing to assist the programmer to "home in" on a component by successive refinement. On the other hand, where there *is* fairly specific type information describing an ideal component, only components precisely matching this ideal are retrieved. The programmer may therefore see nothing, or only inappropriate components, even if there is a component that *could* meet the requirements – for example, a function that includes an additional argument corresponding to a value that is constant in the present application. In this case, what is required is the ability to "pan out" from a given set of components in some disciplined way that is more convenient and more general than revising the key type to some co-instance of the present one.

5. RETRIEVAL BY DISCIPLINED EXPLORATION

The observations at the close of the previous section lead to an inversion of our thinking about the mechanisms to be provided for access to a library of components. Up to this point the main task has been regarded as that of retrieving components matching given keys. Presenting collections of components to the programmer has been an implicit auxiliary task, hardly mentioned. From now on our main concern will be how to provide for the orderly presentation of a collection of components that a programmer wishes to explore. This complements the consideration in the earlier part of the paper of how to help the programmer to apply appropriate constraints determining which components are presented.

Requirements for exploration structures

Any system supporting the exploration of a large collection of items must convey to its users, in addition to information about items currently being examined, a sense of *location* relating to these items. It must also convey a sense of *direction* and *extent* relating to other parts of the collection and to the available routes by which these can be reached. Successful systems typically involve *orderings* defined over the collection.

5.1. An Exploration Ordering for Typed Components

Is there any ordering over polymorphic types that can be applied to our problem? Earlier, two orderings over types were defined: the instance relation ≤ and the prefix relation between type indexes. Indexing is fine as a device to speed up a machine process, but not as the guiding structure for the programmer's exploration: it ignores too much. At first glance, ≤ seems a far more suitable candidate – the concept of an instance should already be clear to a programmer working in a polymorphically typed language. But, as previously observed, the diagram of ≤ for those types occurring in the representative library is extremely broad and shallow. As an exploration structure it would present far too much choice in one or two places, and far too little everywhere else.

What is it about ≤ that fails to capture a richer idea of "no more general than"? Intuitively, one component X is no more general than another Y if – so far as type information reveals – Y provides all that X does. Certainly this is true in a sense if X ≤ Y but that is a rather special case. A more satisfactory way of confirming the intuition, and a very practical one from the programmer's point of view, is to observe the possibility – again, so far as type information reveals – that X *could be defined* as Y applied in a particular way. (Any *actual* definition of X may not be in terms of Y at all, of course.) If Y is a *function* (likely, since only 7 components in our representative library have *non*-functional types) then the obvious way in which it might "provide" X is as its result. Equivalently, the most common way in which a function Y might be used to define X involves applying Y to some argument.

We therefore arrive at the principle of *functional generalisation*: a functional type should be regarded as more general than its result type. This principle is intuitively consistent with the idea that specialisation involves commitment to a particular instance, since applying a function commits an expression found in its definition by substituting actual arguments for argument names.

A pre-order?

But functional generalisation *cannot* simply be added alongside the principle of *co-instance generalisation* in the definition of a new ordering, because in some cases the two principles conflict. We would obtain only a *pre-ordering*.

Example Consider the following definitions and their types.

```
apply f x = f x
Type: (a -> b) -> a -> b

id x = x
Type: a -> a
```

The type of `id` is an instance of the result type of `apply`. So, by the principle of functional generalisation, `apply` would be more general than `id`. But the whole type of `apply` is an instance of `id`'s type. So, by the principle of co-instance generalisation, `id` would be more general than `apply`. □

Exploration based on a pre-ordered collection of components is quite feasible, but is complicated by the possibility of such cycles limiting the sense of location and direction. This assertion is supported by recent developments in so-called *hypertext* systems[17] for which it has been found necessary to provide additional aids to navigation.

A partial order

We should therefore prefer to add the principle of functional generalisation in such a way that the result is a partial order. The problem is (as we have seen) that a -> symbol can be introduced *either* by instance specialisation *or* by functional generalisation. This problem can be solved by a very simple device: directly exclude substitution of functional types.

Definition *(applicative type instance/co-instance)* One type t is an *applicative instance* of another u (with respect to a substitution σ), and we write $t \ll u$, under the following conditions.

1. If t and u are the same primitive type, σ is the identity.
2. If u is a variable, t must not be a functional type, and σ is the identity for all but u, which it maps to t.
3. If u is a functional type with result type v and $t \ll v$ with respect to σ, then $t \ll u$ with respect to σ also.
4. If t and u are formed by applications of the same type constructor (possibly the functional constructor ->), and for every corresponding pair of arguments $t_i \ll u_i$ with respect to σ_i, and the non-identity part of σ is the union of such parts in every σ_i, then the relation holds.

Also, if $t \ll u$ then u is an *applicative co-instance* of t. □

Theorem *($\ll$ ordering)* $\ll$ is reflexive, transitive and anti-symmetric – a partial order.

Proof By induction, for example on the combined sizes of related terms. (Details are lengthy but straightforward.) □

It is immediate from the principle of functional generalisation that the introduction of an argument (to form a functional type from a result type) should constitute a rise in the $\ll$ ordering. One encouraging consequence of the $\ll$ definition is that an additional argument *in any position* also constitutes a generalisation.

Example Recall the `fold` function, first introduced in § 2.1,

```
fold op [a; b; ... ; n] =
    a op (b op (... op n))
```

and its polymorphic type.

```
fold      (a -> a -> a) -> [a] -> a
```

More general forms of `fold` are well-known as standard components in functional programming. For example, the `foldr` function accepts as its first argument a possibly *asymmetric* binary operator op – one whose left and right operand types may differ. The result type of op must be that of its right operand. The second argument of `foldr` is a valid ***right*** operand for op, and its third a list of items having the same type as op's *left* operand. The result of an application of `foldr` to three such arguments is expressed in the schematic equation

```
foldr op z [a; b; ... ; n] =
    a op (b op (... op (n op z)))
```

and `foldr`'s type is as follows.

```
foldr     (a -> b -> b) -> b -> [a] -> b
```

This generalisation of `fold` to `foldr` is captured by the $\ll$ ordering. It is straightforward to verify from the definition preceding this example that `foldr`'s type is an applicative co-instance of `fold`'s type. □

Example Similarly, recall the map function.

```
map f [a; b;...] = [f a; f b;...]
```

and its polymorphic type.

```
map       (a -> b) -> [a] -> [b]
```

This version of map applies `f` to ***single*** arguments drawn from a ***single*** list. There are frequent applications for a variation map2 that applies a ***binary*** operator to the corresponding items from *two* list arguments to obtain a resulting list.

```
map2 op [a1; b1;...] [a2; b2;...] =
    [a1 op b1; a2 op b2;...]
```

The polymorphic type of this function is as follows.

```
map2      (a -> b -> c) -> [a] -> [b] -> [c]
```

Intuitively, map2 is more general than map since op could be defined to apply a function `f` to one operand, ignoring the other. This generalisation is also captured by the $\ll$ ordering. □

Quantitative assessment

Comparison of the structures induced by $\leq$ and by $\ll$, reveals that $\ll$ is considerably richer. This can be quantified in measures such as the number of maximal (minimal) (co-)instances of a type. For our representative library, the average figure using $\ll$ is 3.16, whereas for $\leq$ it is only 0.34. To appreciate the practical significance of these numbers, imagine you are a programmer searching for a suitable component and currently examining some particular type-equivalent class. You wish to move on to look at similar but more general (or more specialised) components. The numbers indicate how many alternative lines of exploration are open to you, on average, for each of the two structures. It seems that the $\ll$ ordering is far more suitable as the basis for an exploration structure.

Although the average figure for $\ll$ is attractive, we must put alongside it some worst-case information. A very few types have an undesirably large number of maximal applicative instances (just 3 out of 119 have over 20). This problem is largely overcome by systematic introduction of a few intermediate types with null component sets. (In fact, we have found that awareness of these componentless types can inspire the introduction of new components, so they may not remain componentless for long.) The determination of suitable intermediate types must be based on both instances *and co-instances* because the dual generalisation principle behind $\ll$ prevents any definition analogous to the co-unification of $\leq$.

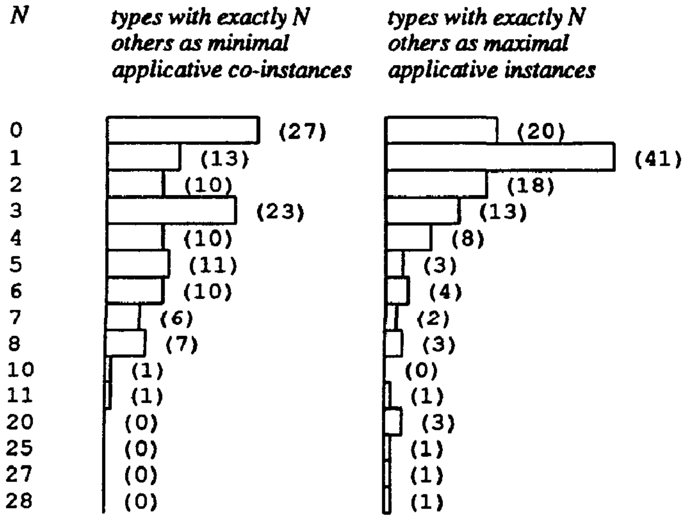


Figure IV : maximal/minimal applicative instances/co-instances per type

6. RELATED AND FUTURE WORK

We do not know of much previous work on the use of polymorphic types as keys for component retrieval. The idea was discussed in the second author's thesis[18] but without resolving details.

6.1. Rittri's Work

The main comparison to make is with the work of Mikael Rittri[19] at Chalmers. He has independently investigated the use of types as keys, but in a rather different way.

Rittri assumes a single, explicitly formulated, type key. The components that his system retrieves are those whose types are *equivalent* to the key, under the relation ≡ with the following axioms.

```
           t, u ≡ u, t
       (t, u), v ≡ t, (u, v)
  t -> (u -> v) ≡ (t, u) -> v
    t -> (u, v) ≡ (t -> u), (t -> v)
```

The intuitive justification for these axioms is that argument order should be ignored, and so should the distinction between a curried function and one that takes a tuple as argument: Rittri proposes that any such difference between key and component type can be bridged by adding a *converter* function. The axioms also have a formal justification based in category theory.

The equivalence between key and component must be exact – the system does *not* accept components whose type only *unifies* with an equivalent key. The justification here is twofold. First, it is doubtful whether unification modulo equivalence is decidable. Second, the programmer has clearly indicated the degree of polymorphism required by the choice of key type, and this should be respected.

The difference of view regarding identical *vs* unifying types is accounted for by the different sources of key type used: explicit formulae on Rittri's part, and implicit contextual information on ours. His ≡ relation and our $\ll$ ordering are both means to the same end – a relaxation of the matching condition for functional types. Our view of currying, for example, is that the curried form of a function *should* be regarded as *more general*, although this is not uniformly the case under our present definition of $\ll$. We do have the chain

$$a \rightarrow b \rightarrow c \gg b \rightarrow c \gg a, b \rightarrow c$$

but this depends on b being a type variable. As to the re-ordering, insertion and deletion of arguments it seems to us that the two approaches are complementary.

6.2. Combinations of Components

One obvious generalisation of a system to retrieve individual components from a library, would be a system that considered also suitable *combinations* of components. In fact, the possibility of component combination is already implicit in the definition of an applicative instance: a component Y is more general than another X under the principle of functional generalisation precisely because (so far as type information reveals) Y can be applied – *combined* with an argument – to obtain X. But our present system draws the programmer's attention only to Y, not to any other component in the library that might form a suitable argument for combination with it.

Re-ordering arguments can be regarded as a special case of combination. The function `flip`

```
flip f x y = f y x
```

which has the type

```
flip     (a -> b -> c) -> b -> c -> a
```

can be applied to any function f of at least two arguments, to yield a function like f but with the first two argument positions exchanged. (In Rittri's terminology, we can use `flip` as a converter function – though in general his converters may need to be complex and recursive.)

6.3. Stronger Tests than Type-matching

A component may have the desired type yet have a functionality quite unlike that required. Polymorphic types applied as keys fulfill the role of a filter, enabling *candidate* components to be identified.

But if the key types have been obtained from specifying laws there is the information available to make the *test* part of the *generate and test* procedure much more rigorous. A fully automatic test proving or disproving each law for each candidate component is not possible. But substantial support could be given to the programmer in the task of checking whether laws hold. For example, a simplifier starting from a standard inductive proof scheme might reduce some case to a falsehood, and therefore reject a component before it is ever presented to the programmer. Another possibility (requiring interaction with the programmer to avoid termination problems) would be to establish specific test values to be substituted for the free variables of a law and try each candidate component in this context.

6.4. Other Type Systems

Although most programming languages with polymorphic types follow Milner's system[1] (as we have done), other polymorphic type systems have been proposed. For example, Fairbairn's language Ponder incorporates a more expressive form of type polymorphism, allowing locally quantified type variables.[20] Type-based library access in this context might exploit the *relation of generality* defined between Ponder types. Object-oriented systems have another component-ordering that aids re-use, the *class hierarchy*. There have been various attempts to integrate the concept of subclass into a polymorphic type system.[21]

6.5. Other Relations Between Components

Component libraries could be combined with a *Literate Programming*[22] approach, emphasising integral documentation and *uses*/*used by* cross-references. Such links between component *definitions* illustrate the more general possibility of combining an ordering over polymorphic types with some other access structure. Though a programmer re-using a library component may wish to abstract away from the details of its definition, the option of employing a definition-based relation such as *uses*/*used by* could be a valuable adjunct to type-based access. For example, this would allow the function computing the product of a list of values to be described as having the *type* of sum but *using* `times` (instead of `plus`). However, the introduction of multiple access structures should be restrained to avoid presenting the programmer with an unhelpfully wide choice or an unduly complex specification task. The beauty of an access mechanism based on polymorphic types is that they represent a single concept already present in the programming system, and their specification can so often be left implicit.

6.6. Full Implementation

The techniques proposed in this paper have been tested in a prototype implementation using Prolog. We have plans to implement them to a higher standard as an enhancement to Glide, our UNIX-based exploratory functional programming environment.

7. SUMMARY & CONCLUSION

We began by observing the need for some way of finding re-usable software components and recognising their potential application. Polymorphic types can be used to provide a source of access keys corresponding to a class of components that are prime candidates for re-use in a particular context. They also provide the basis for a structure of exploration when the selection offered by key access is too narrow or too wide: there is an ordering over polymorphic types that corresponds well with usual intuitions about generalisation and specialisation of components. We have indicated the practicality of these ideas by giving figures for a representative library of components. We have also built a prototype system.

Acknowledgements

We have benefitted from discussions with Mikael Rittri of Chalmers University and with our colleagues in the functional programming group at York. Our thanks are also due to anonymous referees for helpful criticisms and suggestions. Our work is partially funded by the Science and Engineering Research Council.

References

1. R. Milner, "A theory of type polymorphism in programming", *Journal of Computer and System Sciences* 17(3) (1978).
2. R. Burstall, D. MacQueen and D. Sannella, "HOPE: an experimental applicative language", pp. 136-143 in *Record of the LISP Conference*, Association of Computing Machinery (1980).
3. D. A. Turner, "Miranda: A Non-Strict Functional Language with Polymorphic Types", pp. 1-16 in *Functional Programming Languages and Computer Architecture*, ed. Jean-Pierre Jouannaud, Springer-Verlag LNCS 201 (September 1985).
4. R. Milner, "A proposal for Standard ML", pp. 184-197 in *ACM Symposium on LISP and Functional Programming*, Austin, Texas (1984).
5. L. Cardelli, "Basic Polymorphic Typechecking", *Polymorphism* 2(1) (January 1985).
6. S. L. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice-Hall (1987).
7. I. Toyn, A. Dix and C. Runciman, "Performance Polymorphism", pp. 325-346 in *Functional Programming Languages and Computer Architecture*, Springer-Verlag LNCS 274 (1987).
8. C. Runciman and I. Toyn, *Notes for Glide users (4th edition)*, Department of Computer Science, University of York (January 1989).
9. L. Augustsson and T. Johnsson, "LML Users' Manual", PMG Report, Department of Computer Science, Chalmers University of Technology, Goteborg (1987).
10. P. Wadler, Q. Miller and M. Raskovsky, *The OL Manual*, Oxford University Computing Laboratory (1986).
11. N. Wirth, "Program Development by Stepwise Refinement (Reprint)", *Communications of the ACM* 26(1), pp. 70-74 (January 1983).
12. E. W. Dijkstra, *A discipline of programming*, Prentice-Hall (1976).
13. J. Darlington, "A synthesis of several sorting algorithms", *Acta Informatica* **11**, pp. 1-30 (1978).
14. A. Martelli and U. Montanari, "An Efficient Unification Algorithm", *ACM Transactions on Programming Languages and Systems* 4(2), pp. 258-282 (April 1982).
15. D. H. D. Warren, "Implementing Prolog – Compiling Predicate Logic Programs", Research Reports 39 & 40, Department of Artificial Intelligence, University of Edinburgh (1977).
16. H. J. Komorowski, "QLOG – the programming environment for Prolog in LISP", in *Logic Programming*, ed. K. L. Clark and S. -A. Tarnlund, Academic Press (1982).
17. J. Conklin, "Hypertext: an Introduction and Survey", *IEEE Computer* 20(9) (1987).
18. I. Toyn, "Exploratory Environments for Functional Programming", DPhil Thesis YCST 87/02, Department of Computer Science, University of York (April 1987).
19. M. Rittri, "Using Types as Search Keys in Function Libraries", in *Proc. Conf. Functional Programming Languages and Computer Architecture*, ACM Press, London (September 1989).
20. J. Fairbairn, "A new type-checker for a functional language", *Science of Computer Programming* **6**, pp. 273-290 (1986).
21. L. A. Jategaonkar and J. C. Mitchell, "ML with Extended Pattern Matching and Subtypes", pp. 198-211 in *ACM Conference on LISP and Functional Programming*, Snowbird, Utah (July 1988).
22. D. E. Knuth, "Literate Programming", *BCS Computer Journal* 27(2), pp. 97-111 (May 1984).