Flexibility *and* efficiency via operations' particularization

Report ISSI-91-005

Adolfo Guzmán
International Software Systems Inc.

ABSTRACT. A hierarchy of abstractions and particularizations is presented, together with a method that keeps them flexible (general) *and* efficient (fast).

I Abstractions

1•1 Introduction and goals

Often, we have several operations (such as "add two numbers together" or "get from this object its age") that could be implemented in several manners, all of them computing the same result or value, but every one taking different time (or, in general, using different resources), due to restrictions on the arguments. Thus, from "general add" we go to "floating add", to "integer add", to "complex number add". Another example is when we offer the user a general (over-loaded, or polymorphic) operation (perhaps depicted in graphical form), but we want to write down (to execute, to compile) the most specific form of such operation. Thus, the user writes "add a b" but we (or the compiler) translates that into "integer_add a b", because it knows that a and b are both integers.

The goal of this paper is to give a procedure [likely to be part of a compiler or translator] which replaces an expression by another "more specific" expression (in a sense well defined below), which is "better" in some way. Moreover, that procedure reverts (puts back) the original expression, if the assumptions allowing the replacement no longer hold.

It is clear that with such a compiling procedure we will achieve (1) efficiency, since the "more specific" expressions are more efficient in some well defined sense; and (2) flexibility, since the "more specific" expression gets erased and replaced by the original expression, if for some reason the more specific expression can no longer be used. It is possible to think that the compiler produces a kind of reconfigurable target code, which can move from being flexible and general (but a little bit slower) to being rigid and efficient and viceversa, automatically (driven by data changes).

1•2 Operations and instances

This paper deals with operations† (or transformations) performed on objects; in particular, with an operation t performed on an object o (or on a few objects o_1, o_2, ...). For instance, adding

†. For conciseness, we talk about hierarchies of operations. The approach, as we shall see, can be generalized to hierarchies of any objects.

file: /home/guzman/publications/flexibility+efficiency-via-particularization.fm
Version 1 (Apr 91); last change May 20, 91; *printed Jul 17, 91.* Pages 1-8

two numbers: +(a, b); getting the value of the age of Guzman: get(Guzman, age); finding the location of an object with a given string name: return_object("Guzman"), etc.

1•2•1 Operation

An operation t is a function defined on a domain D = $class_1 \times class_2 \times$ ... For instance, the domain of + is number × number.

Def. of t: t (x, y, ..)= λ(x, y, ..) where t has domain D = $c_1 \times c2 \times$ ... and range r (1)

Example: t(x, y) ≡ λ(x, y):($x^2 + y^2$) is the function that adds the squares of its two arguments. The domain (inputs) is real × real, the range (output, result) is non_negative real.

1•2•2 Instance

Let us denote by |e| stands for the value of the expression e.

An instance (or a use) of t is a particular "call" to t, for example t(7, w-3), where the formal parameters (variables) x, y, ... of (1) have been replaced by actual parameters such as 7 and w-3.

Ex. of an instance of t: t(z, k+2), where $|z| \in c_1$, $|k+2| \in c_2$, ...and $|t| \in r$. (2)

(2) requires that each argument of the instance (such as k+2) be of the corresponding class (c_2 in our example).

1•2•3 Particularization

An operation t'(x, y, ...) is a particularization of another operation t (x, y, ...) iff the domain D' of t' is a subset of the domain D of t, and the range r' of t' is a subset of the range r of t and t(x, y, ...) = t'(x, y, ..) for every set of arguments in D': t and t' produce the same value wherever t' is defined.

Def. t'(x, y, ...) is a particularization of t' if

$c_1 \subseteq c_1'$, $c_2 \subseteq c_2'$, ..., and r' ⊆ r and t'(x, y, ...) = t (x, y, ..) for every (x, y, ...) in D' (3)

where D' = $c_1' \times c_2' \times \ldots$

1•3 Replacing an instance by its particularization

A given instance t(a, b, ...) can be replaced (in the sense that this replacement keeps the value of the operation invariant) by a more particular operation t'(a,b, ...), if we know that the objects (a, b, ...) on which t works are always (or at least, for a relatively long period of time) restricted to the domain of t'. For instance, given + with domain number × number, fix_add with domain integer × integer, and float_add with domain real × real, we can specialize (+ a b) to (float_add a b), if we know for some reason that a and b will always be real numbers. In general, thus, a given instance of an operation t(a, b,...), where t is an operation with domain $c_1 \times c_2 \times \ldots$, can be replaced by a particularization t' of it, with domain $c_1' \times c_2' \times$ ..., where each $c_i' \subseteq c_i$, if we know that a $\in c_1'$, b $\in c_2'$, ...

1•3•1 Conditions for replacing t by t'

Under what conditions is safe to replace t(a, b, ..) by t'(a, b, ...)? Whenever

$$a \in c_1\text{'} \text{ and } b \in c_2\text{'} \text{ and } \ldots, \text{ and } t\text{' is a particularization of } t \qquad (4)$$

t' presumably has some advantage over t, when conditions (4) hold: it is more efficienct, or faster, or uses shorter memory, or ... . The user indicated t to begin with, but, if we could automatically detect that (4) holds, it will be advantageous to use t', so as to produce a better computation [if these transformations t' are being executed by an interpreter], code [if these transformations t' are being produced by a compiler], or execution [if these transformations t' are being executed by a cpu].

1•4 Unreplacement

And, what happens if the assuption (4) is no longer valid? t' is no longer the correct replacement for the old t, and we have to fix the "error."

1•4•1 Rigid systems: recompile

Most (compiling) systems, having replaced t by t' (having *compiled* t into t'), have no idea which expressions to change back. They are rigid. If (4) is no longer valid, you have to recompile. Even more, if you are using an "old" system (where (4) holds) to work with a "new" set of data (where (4) does not hold), you will get strange run time errors.

1•4•2 Cautious systems: error is detected

Other systems are more conservative. They replace t not by t', but by t'', defined as

$$t''(x, y, \ldots) \equiv \lambda(x, y, \ldots) : \text{if (type? (x, c1) and type? (y, c2) and ...) then } t'(x, y, \ldots) \text{ else error} \qquad (5)$$

where c1 and c2 are two types. That is, t'' tests for types of arguments at running time. This is fine, but t'' of (5) runs a bit slower than t' of Subsection 1•2•3. These systems can detect (but not correct) whether (4) still holds or no longer holds.

1•4•3 Flexible systems: replace back t instead of t'

Ideally, t should be put back, replacing t', whenever (4) no longer holds. Having the original t back in place again, the (compiling) system is ready to optimize (particularize) t again, this time to t'', a perhaps different particularization than t'.

Section II deals with mechanisms for implementing these flexible systems.

II Flexible Systems

In order to implement a flexible system (as defined in 1•4•3), we need:

(a) A sequence (or a tree) of operations t, t', t'', ..., ordered by the relation $\subseteq$ in the domains and range, so that those "at the right" (further away from the root) are particularizations of those to the left.

(b) A procedure to identify the actual types (classes) of a given instance t(a, b, ...), and compare them with the potential (or declared) types of t [given in the definition of t, probably], which will help us to select the right particularization from (a);

(c) A (compiling) procedure to go from the source instance to the most specific particularization allowed by procedure (b);

(d) A procedure to detect whenever the types of a, b, ... have changed, and no longer are those types detected by (b). In this case, uncompiling is necessary;

(e) An (uncompiling) procedure to go back from t' to the original t, whenever (d) advises to do so.

Notice that, after (e), we are back to the original starting point (a), so that indeed it is possible to specialize t again to some other particularization, and go back to being efficient.

Notice, too, that we can apply the procedure (a)-(e) *more than once*: for example, with the knowledge that a = {times, sin, log, sqrt}, we can particularize z := (first a) (u, k-2) to z := times (u, k-2), then to z := float_multiply (u, k-2).

2•1 Example: Dynamic binding of compiled code

This first example of a flexible system, which analyzes static vs. dynamic binders (or linkers), will clarify the role of parts (a) through (e) above. Suppose we have some Fortran source code(Figure 1•), organized in a main program calling several subroutines, stored in several files. When compiling a file, the compiler can not resolve where the missing subroutines will be in memory, so that an instruction such as call foo (a, b) is left unresolved. One popular way is to assemble something like jsr i α, with address α containing the string "foo(a, b)". It is the linking loader who will replace these transfer vectors by an actual memory address. The linking loader, indeed, knows where foo went in memory, so it has no trouble replacing the contents of α with hex9000, say. The code now looks jsr i α, with contents(α) = hex9000.

We have just described an static linker. The sequence (a) of operations is

call foo (a, b); jsr i α, cont(α) = "foo"; jsr i α, cont(α) = hex9000. (6)

The procedure (b) is: noticing that α contains an unresolved reference; looking for it in the linker's table. The procedure (c) is: inserting hex9000 instead of "foo".

2•1•1 Static linkers

Static linkers go this far. They don't have procedures (d) and (e), so they are unable to re-link a new definition of an old procedure; say, we redefine foo and compile it again. To use this new foo, we have to relink the whole executable into a new load module.

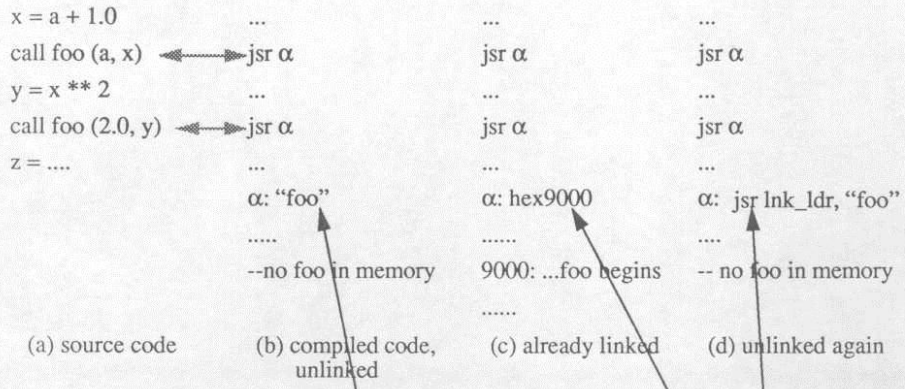
2•1•2 Dynamic linkers

On the other hand, a dynamic linking loader (the first one I am aware of was the Dynamic linking loader of the PDP-10 [circa 1970], later called the DEC 20) is able to detect that a new version of subroutine foo has been compiled. The loader keeps track in a table of all the places where the call to foo was replaced by the call to hex9000. In other words, it keeps track of all the particularizations. When a new foo is loaded, let's say, in hex90FE, the linking loader uses the table to replace back the jsr i α, cont(α) = hex9000 to jsr i α, cont(α) = "foo" (refer to (6)), that is, it is replacing the particularization by its (former) generalization. The table remembers "places that were have been jsr i α, cont(α) = "foo"; thus, it has no trouble restoring these original contents in these places.

To be more specific, the particularization jsr i α, cont(α) = hex9000 is not replaced by its generalization jsr i α, cont(α) = "foo". Instead, we replaced it by a call to the dynamic linking loader, as follows:

jsr lnk_ldr, "foo" (7)

:When a call to foo is made, the linking loader is called, instead; in its arguments it sees what to



name	loaded in	used in
foo	nowhere	α
bar	8000	7400, 75F8

(b_t) LOAD TABLE

b_t is the load table during situation (b), when the compiled code is still unlinked.

name	loaded in	used in
foo	9000	α
bar	8000	7400, 75F8

(c_t)

c_t is the load table during situation (c), where the code is already linked.

name	loaded in	used in
foo	90FE	α
bar	8000	7400, 75F8

(e_t) is the load table for situation (e), not shown, where α contains now 90FE

name	loaded in	used in
foo	nowhere	α
bar	8000	7400, 75F8

(d_t) is the load table for situation (d), where the code is again unlinked.

Figure 1• Dynamic linking and unlinking

When the source code (a) gets compiled to (b), some external subroutines, like foo, remain unlinked (unresolved). The linking loader resolves them (figures c, c_t), remembering in table c_t the places where the particularization occurred (it occurred in α). A posterior recompilation of foo produces the situation shown in d and d_t, where the original generalization "foo" is restored back in α. Later, the new foo is loaded again into memory, this time in 90FE. The load table is now e_t. Remembering in the load table where to restore our generalizations was crucial.

load; in the relocation table it knows where it is now (in hex90FE in our example) or, if not in mem-

ory, brings it from disk. Thus, the new foo is not loaded unless it is needed. The situation is depicted in Figure 1•

In the dynamic binder, the procedure (d) to detect un-applicability of the specialization is achieved by the load table of Figure 1•, and the procedure (e) to uncompile the specialization and put back the general instance, is given by that piece of code that, after consulting table c_t, decides to change the program (c) of Figure 1• to the program (d) of same figure.

Notice that the un-compiling (generalization) of contents(α)=hex9000 was to contents(α)="foo", and not all the way to call foo (a, x), the source code (a) of Figure 1•

It is possible to go through a cascade of compilations, each of which is reversible in the manner indicated. Also, notice in this example how the specialization of contents(α)="foo" is not always to the same particularization; it was in one case [(c) of Figure 1•] to contents(α)=hex9000 and in the next case [(e) of same figure] to contents(α)=hex90F0.

2•2 Example: Replacing one database by another transparently

Suppose we write an application program that uses a particular commercial database, say, Oracle. Our code will contain calls to particular Oracle functions, for instance to open the database, to retrieve a particular table, etc. It looks like

```
.....
z:= Oracle_open ("Customer", ...);                                    (8)
y := Oracle_get (z, key="John Smith", property="age", buffer2, ..)†
....
```

It is now desired to change databases,a to Sybase, say. We have to replace in (8) the calls to Oracle by the calls to Sybase, which have some other format, different number of arguments, etc.

If, in (8), we write calls to our "own" functions, as follows:

```
.....
)y := Guzman_get ("Customer", "John Smith")                           (9)
....
```

then we have to write, for each of our own functions Guzman_get, Guzman_store, and so on, an small program, defining them in terms of Oracle_open, Oracle_get, etc. For instance, Guzman_get will look like

```
define Guzman_get (obj, prop)
        { if (not Oracle_opened (obj)) then temp1 := Oracle_open (obj, ...);   (10)
          return Oracle_get(z, key=obj, property=prop)}
```

When the time comes to replace Oracle by Sybase, we have to write a new set of definitions for our own functions (10), but this time calling Sybase functions (Sybase_open, etc.) Notice that the code (9) remains unchanged. Thus, we have achieved a code that is database independent.

---

†. We could have haccessed Oracle via SQL, which is an "standard" language precisely for this purpose: to achieve database independance. We did not choose to do so because we wanted a faster access method than SQL.

Since we have the freedom to select our own set of functions (10), we shall do so by paying attention to the functions we want to do *expressed in our own manner* (for instance, tuned to the family of applications at hand)

2•3 Example: Polimorphism and overloading of functions

It is possible to have a polymorphic system (using a generic name function, such as add, as well as several particular or specific functions, such as integer_add, real_add, complex_add, matrix_add, etc.) which is both efficient and flexible. To achieve this, the compiler should try to replace all the generic function calls (which are slow and expensive, since they have to do type checking and conversion) by calls to the suitable particularization, and at the same time keep track (in a particularization table) what generalizations to insert back where, if needed.

For instance, the following program:

```
integer i, j, k;  real a, b, z;
.....
k = i + j                                   (11)
....
z = a + b
```

could be compiled initially into

```
....
(α)   store (loc(k), add(i, j))             (12)
...
(β)   store (loc(z), add(a,b))
```

where add is a generic add function. Later, the particularization will transform the above code into

```
....
(α)   store (loc(k), integer_add(i, j))     (13)
....
(β)   store (loc(z), real_add(a, b))
```

with the provision that the following information be remembered in a particularization table:

loc	previous content
α	store (loc (k), add(i, j))
β	store (loc(z), add(a, b))

It is clear that the newer code will be more efficient.

If somehow the type of i or j changes, the routine that changes the types (perhaps a part of the editor or the parser) has now the additional duty to consult the particularization table, so as to restore α back to its former content.†

†. It could do this in a "lazy" manner, for instance, restoring first α to "call fix_me_up". This function fix_me_up will later determine what to restore in α.

Why, you may ask, did the compiler have to produce the code ((10))? Why didn't it simply produce the code (11)? It certainly could, since it knew the types of i, j, a, ... at compile time. *But suppose it did not.* Or suppose this is a language (such as that of a rapid prototyping system) *where the types could change a few times* before "settling down". An ordinary compiler would require recompiling each time you change (or you don't know) the types. This compiler would "keep going", recompiling only the affected parts. Incrementally recompiling the affected parts.

Why, you may ask, don't you simplify the above procedure so that (B) now reads:

....

(α) store (loc(k), [if integer (i) then integer_add(i, j) else real_add(i, j)]) (B')

...

(β) store (loc(z), add(a,b))

Certainly, (B') does not need further optimization. It knows how to handle the general case.

Well, we have not done much improvement. (B') is as inefficient as (A), since both have to do the type testing at running time.

The efficiency comes when, at running time, you compile a generic instance into a particularization. The flexibility is retained when, in the particularization table, you remember what generic instances to restore where. One added complication is that now you, each time the type changes, have to consult the particularization table to restore all the places that depend on that type. Sort of a truth-maintennance system for the particularization table.

2•4 Example: Mixing compiled and interpreted code

Suppose a function f(x, y, ...) calls another function foo (u, v, ...). We wish to run foo sometimes interpreted, sometimes compiled. For instance, we run foo interpreted, and after enough testing we decide to compile it. Later, we detect still some errors in the compiled version, so we go to the interpreted version again for a while. We wish to do this in a transparent manner.

Let b be the body (the definition) of foo. A way to achieve this, similar to that proposed at the beginning of this chapter, is to replace the call foo (u, v+4) by either of these two calls:

call eval_expr (b, alist = ((x, lul) , (y, lv+4l)) (14)

call foo_compiled (lul, lv+4l) (15)

The first of them, (12), just calls the interpreter with the bindings of variables x and y to the values of u and v+4, respectively. The second one, (13), produces a call to the compiled function. Clearly, (13) is an specialization of "foo (lul, lv+4l)" and we should register in the load table (Refer to Figure 1•) where foo is used, in case an "uncompilation" (reversion to interpreted code) is necessary. Then we simply revert (13) back to (12).

2•5 Acknowledgments

The techniques described here were first seen in the DEC PDP-10 (later called Dec 20) around 1970, in their dynamic linking loader. Many Lisp compilers also use this technique to do dynamic loading and incremental linking or binding.