UNIVERSITÉ DE GENÈVE

Archive ouverte UNIGE

https://archive-ouverte.unige.ch

Chapitre de livre 1991

Published version Open Access

This is the published version of the publication, made available in accordance with the publisher's policy.

More Functional Reusability in C / C++ / Objective-c with Curried Functions: working paper

Dami, Laurent

How to cite

DAMI, Laurent. More Functional Reusability in C / C++ / Objective-c with Curried Functions: working paper. In: Object composition = Composition d'objets. Tsichritzis, Dionysios (Ed.). Genève : Centre universitaire d'informatique, 1991. p. 85–98.

This publication URL: https://archive-ouverte.unige.ch/unige:157921

© This document is protected by copyright. Please refer to copyright holder(s) for terms of use.

More Functional Reusability in C / C++ / Objective-c with Curried Functions

WORKING PAPER

Laurent Dami

Abstract

Several mechanisms commonly used in functional programming languages can be beneficial in terms of conciseness and reuse potential in more traditional programming areas, like applications programming or even systems programming. An implementation of functional operators for the C, C++ and Objective-C languages, based on the principle of *curried functions*, is proposed. Its implications in terms of improved power and additional cost are examined. Examples of parameterized function generators, function compositions and closures are given. A particular section shows how closures of C++/Objective-C objects with their member functions can be done with the currying operator.

1. Introduction

Functional programming is still limited mainly to academic circles; applications programmers or systems programmers generally use imperative languages like C, PASCAL, FORTRAN, COBOL or ADA, or, more recently, object-oriented languages like C++, Objective-C or Smalltalk. In these languages, functional operators are rather limited, if present at all. The interest of such operators has probably been outweighed by concerns for simplicity or efficiency during the language design process.

In the light of the growing concern for reusability of code, this position must be reconsidered. Several mechanisms commonly used in functional languages seem very useful for combining software components together, and seem to lead to more concise programs with a higher reusability potential. Recent approaches to building applications, in which the essential activity is no longer programming, but rather configuring and gluing together predefined components (*scripting* [9]), are in great need of such versatile mechanisms for making connections.

This paper discusses an implementation of some functional constructs within the framework of the C programming language. C is a traditional, imperative language which is widely used for both applications and systems programming[8]. We propose to enrich C with a single mechanism inspired from the *curried functions* used in denotational semantics. This mechanism is relatively simple to add in a language that uses a stack for parameter passing across function calls, and it is sufficient for generating a large number of the constructs commonly used in functional programming. Several examples demonstrate the usefulness of C curried functions, in particular when interface protocols between modules exchange function parameters, like in event-driven programming. We also explore the implications of this mechanism in the object-oriented derivatives of C, namely C++ and Objective-C.

2. Motivation

2.1 Functional constructs and reusability

Functions (procedures, subroutines) are probably the most important abstraction in programming languages. The ability to group a collection of instructions under a single name, and to use that name in other places in a program, is essential for managing the complexity of large systems through different levels of abstraction. Nearly every programming language has a notion of function, procedure or subroutine. However, the allowed operations on functions largely differ from one language to the other. To make this clear, we propose a brief classification of some major programming languages, according to their flexibility in the use of functions:

- in COBOL or FORTRAN, functions are only denotable (they can only be named and called)
- in PASCAL, functions or procedures can be passed as parameters to other functions
- in C, *function pointers* can be stored and can be the result of an expression
- functional languages like ML, Miranda, LISP or SCHEME support function abstraction through lambda expressions.

Functions that are only denotable are sufficient as long as the traditional software development scenario is respected, in which the software environment provides basic building blocks (language constructs and libraries of functions) and the programmer provides the global architecture (the 'main' program). However, when it comes to other development scenarios where it is the system that provides the global architecture, and where application programmers provide some of the building elements, then denotable functions are not enough. Such development situations appear in most event-driven programs, particularly in windowing applications; they also belong to the notion of reusable *application frames*[2] that programmers can customize to suit particular needs. The problem there is that programmers must bind their own functions to the global framework, and that in most traditional languages they do not have rich possibilities for creating their own functions. Even the C language, which is quite flexible in the use of function pointers, is limited by the fact that functions can only be defined statically.

Various degrees of versatility in the use of functions also have important consequences on the reusability of code because the function call is still the major mechanism associated with modularity. Even if techniques like genericity or inheritance now offer richer binding possibilities the main glue between modules is primarily the function call. Therefore an environment that supports functional operators can create more flexible binding configurations between modules, and provides more freedom for reusing modules in different contexts.

Now the obvious question is: if functional operators appear to be very useful for some programming tasks, why are they not more widely used? A possible answer is that most imperative programming languages are implemented in a way which is not convenient for supporting functional operators. On the other hand, functional programming languages do not always fit the requirements for application programming, notably because of their increased needs for computing resources. Here we propose a compromise, where an application programming language is

extended with some functional programming features, at a cost which we consider acceptable when compared to the additional power it provides.

2.2 Why C?

The choice of C as a target was very natural. C is widely used on many architectures and has important libraries that we want to be able to use. Furthermore, C already supports function pointers, which is a good basis for adding functional capabilities. Finally, C has two object-oriented derivatives (C++ and Objective-C) in which we can experiment with the interference between objects and functions.

2.3 Designing a C extension

Readers of electronic news on the Internet network regularly see proposals for extending C or C++ with new features. Actually, several recent postings denoted a large interest for concepts like function composition, closures or continuations. Nevertheless, most of these proposals are quickly rejected by the C/C++ programmers community, which is not ready to pay the additional cost that these extensions would imply. In order to be successful, then, an extension proposal should be very careful about the repercussions on existing code. We designed our extension with the following points in mind:

- would the normal C function call protocol be affected by a new functional mechanism?
- to what extent would a new mechanism be compatible with existing C code (source-level/binary level)?
- would functional mechanisms interfere with objects in C++ or Objective-C?
- what is the additional cost (cpu time / memory usage) of the new mechanism?

We came up with a solution that implements *currying*, or partial function application, for C. This gives an interesting degree of dynamic function creation, while keeping full binary compatibility with existing C code. Additionally, this mechanism costs nothing when it is not used; in other words, programmers can consciously decide in each case whether they are ready to pay additional cpu time and memory consumption for additional power.

In order to support dynamic function creation without modifying the basic C function call protocol, the implementation uses run-time code creation. This approach was chosen because of its simple and clean integration with existing C code, without need for compiler modification. The drawback, however, is that a part of the implementation is machine-dependent and needs to be rewritten when ported to new architectures. Currently we have implementations of curried functions for Motorola 68020/30/40 processors and for the Sun SPARC architecture.

3. Curried functions

3.1 What are curried functions?

Curried functions are named after the mathematician H. B. Curry. They are frequently used in some fields of theoretical computer science (Denotational Semantics), where among other advantages they are very convenient for reducing the plethora of parentheses in functional expres-

sions. Practical use of curried functions for programming is limited to a few functional languages like ML or Miranda [3].

Curried functions are functions of more than one arguments that take "one argument at a time", instead of taking all arguments together. In general, for any function **f** of **n** arguments from domains D_1 to D_n, that returns a result in domain D_r:

$$f: D_1 \times D_2 \times \ldots \times D_n \rightarrow D_r$$

there is a curried equivalent:

$$\text{curry } f: D_1 \rightarrow (D_2 \rightarrow \ldots \rightarrow (D_n \rightarrow D_r))$$

which is defined as (using lambda notation)

$$\text{curry } f = \lambda a_1.\lambda a_2. \ldots .\lambda a_n.f(a_1, a_2, \ldots, a_n) \qquad \text{where } a_i \in D_i$$

In other words, **curry f** is a function that takes a first argument a_1 and returns a function, which in turn takes an argument and returns another function, which continues the same process until all arguments are present, at which point the original function **f** is evaluated to yield a result. A standard example, borrowed from Gordon[7], is the function:

```
plusc :Int -> Int -> Int
plusc = λx.λy.x+y
```

that yields the sum of two curried arguments (**plusc** stands for 'plus curried'). From that function, we can build new functions:

```
succ, pred: Int -> Int
succ = plusc 1
pred = plusc -1
```

and then evaluate these functions to get final results.

```
succ 7 = 8
pred 0 = -1
```

In a theoretical context, a curried function partially applied to some of its first arguments is an expression that can be simplified by β-reduction, i.e. if all arguments are supplied, this yields a closed expression with no bound variables. However, in the context of C programming, where functions are compiled and can no longer be simplified, the behavior is slightly different. Currying a function with some of its arguments yields an intermediate structure that just stores the function address together with its partial environment. This is sometimes referred to as a *partial closure*.It can then be applied to the rest of the arguments, or be curried further. When all arguments have been supplied, the function can finally execute.

3.2 C interface

The details of our C implementation may not be of general interest, and therefore are presented later in section 6. Here we only give the interface, in order to make the examples of the next section comprehensible. The code examples use ANSI C syntax, where function pointers can be directly applied to arguments without explicit dereferencing. Curried functions are created with the call:

```
curry(function, nargs, arg1, arg2, ...)
```

where **nargs** is the number of curried arguments. For example a C function like:

```
int f(int, char, float)
```

can be curried in the following way:

```
g = curry(f, 2, 19, 'X');
```

then g can be used as any C function pointer; so the expression:

```
g (4.5)
```

in fact calls

```
f(19, 'X', 4.5)
```

Since curry expressions return function pointers that differ in no way from ordinary function pointers, which is an essential feature of our implementation, it is possible to do multiple currying:

```
g1 = curry(f, 1, 19);
g2 = curry(g1, 1, 'X);
g2 (4.5)
```

This code yields the same result as the previous expressions.

Our current implementation uses existing C constructs to add the currying mechanism. This was much easier to do than modifying the compiler, and is sufficient to prove the interest of currying. This approach has two drawbacks, however. First, curried functions cannot be type-checked, and therefore require careful use in order to avoid errors. Second, the **curry** function cannot know the size of its arguments, and counts them as if they were all of the size of an integer. In consequence, arguments that are larger that an integer, like for example floating-point values, must be counted as several arguments in the **nargs** parameter. Again, this is very likely to yield errors. It should be well understood, however, that these problems are bound to the current implementation and would not be present if currying were supported directly in the C compiler.

4. Examples

The usefulness of a currying mechanism is essentially to provide a means for dynamic function creation. Three examples demonstrate the power of dynamically defined functions in various situations. The first example uses a generic, parameterized event handler to dynamically create and register different handlers for different signals. The second example builds complex selection and comparison functions out of simple elementary functions, which can then be used in collaboration with generic sort and selection algorithms. The final example uses currying to implement other functional operators. Programmers accustomed to work in functional languages may find these examples rather trivial; nevertheless, C programmers will hopefully realize how curried functions can open a vast domain of new possibilities in their customary language.

4.1 Event handling

During its computation, a program may need to be informed about events occurring in its environment, like a mouse click on a window, a timer alert, or an exception signal. The traditional

way to do this is to program *event handlers* that are called by the operating system or the window manager whenever such events occur. Usually it is necessary to write as many different handlers as there are different kinds of events to handle. This is illustrated with the **signal** library function,which is used in the Unix environment for associating handlers with particular events (the **signal** example was chosen because of its simplicity; similar techniques are applied in other event-handling situations, like *callbacks* in a windowing system). Here is how a normal C program would use **signal** to register handlers for three different kinds of events:

```
void sigint_handler()
{
    fprintf(stderr, "interrupt (SIGINT)");
}

void sigalrm_handler()
{
    fprintf(stderr, "alarm clock (SIGALRM)");
}

void sigwinch_handler()
{
    fprintf(stdout, "window size changed (SIGWINCH)");
}

main()
{
    signal(SIGINT, sigint_handler);
    signal(SIGALRM, sigalrm_handler);
    signal(SIGWINCH, sigwinch_handler);

    sleep(360);        /* wait for signals */
}
```

Although the three handlers are very similar, they cannot be replaced by one single, parameterized handler, because the **signal** library function has been defined in such a way that the handlers take no parameters. So if we want different behaviors for different signals - which is the case here because we print different messages on different output streams - then we really must write one handler for each signal we want to catch. Now here is the same program with currying:

```
void handler(char *sig, FILE *f)
{
    fprintf(f, "got signal %s\n", sig);
}

main()
{
    signal(SIGINT, curry(handler, 2, "interrupt (SIGINT)", stderr));
    signal(SIGALRM, curry(handler, 2, "alarm clock(SIGALRM)", stderr));
    signal(SIGWINCH, curry(handler, 2, "window size changed (SIGWINCH)", stdout));

    sleep(360);        /* wait for signals */
}
```

Each **curry** expression supplies different arguments to the generic handler, and the resulting closures are registered to handle different signals. Not only is the curried solution shorter and

more elegant; it also has the possibility of dynamically adding new handlers during program execution:

```
int sig;
char signame[50];
printf("Enter a signal number you want to catch, and its name:");
scanf("%d %s", &sig, signame);
signal(sig, curry(handler, signame, stderr));
```

This would be impossible without the **curry** operator, because standard C does not have the possibility of generating a new handler at run-time.

A simple variation on this example will demonstrate another use of currying, namely as a glue between library functions that otherwise would be incompatible. Suppose that, instead of printing a message, we want to put a window on the screen whenever a signal is catched. On the NeXT workstation, a window for alerting the user is created with the following call:

```
int NXRunAlertPanel(const char *title, const char *msg, const char *defaultButton,
                    const char *alternateButton, const char *otherButton, ...)
```

Obviously this function is not appropriate for becoming a signal handler because it takes many parameters. However, currying can hide these parameters and provide a glue between **signal** and **NXRunAlertPanel**:

```
#define sigAlert(message)       curry(NXRunAlertPanel, 5, "YOU GOT A SIGNAL",        \
                                            message, "OK", NULL, NULL)

main()
{
        signal(SIGINT, sigAlert("interrupt (SIGINT)"));
        signal(SIGALRM, sigAlert("alarm clock(SIGALRM)"));
        signal(SIGWINCH, sigAlert("window size changed (SIGWINCH)"));

        sleep(360);     /* wait for signals */
}
```

The fact that library functions can be combined in unusual configurations, that were never foreseen by the original designers of these functions, clearly demonstrates how currying can improve the reusability potential of existing code. The next examples provide more insight in some configurations of functions.

4.2 Generic sort and selection

Suppose we want to write a program that works with collections of words and needs to perform several sort and selection operations. The standard C library has a generic function **qsort** that sorts an array according to a particular *comparison function* that must be supplied by the caller. Similarly, we can easily write a generic selection function that takes out of a list the words that match a particular *predicate* supplied by the caller. Now we shall see that the **curry** operator opens a large range of possibilities for generating interesting predicates or comparison functions out of a small number of basic elementary functions. First let us define some useful function types:

```
typedef int (*intFunc) (char*);
typedef int (*cmpFunc) (char*, char*);
```

```
typedef bool (*selFunc) (char*);
```

Now here are some elementary functions (our "building blocks"):

```
int countCharacter(char c, char* word);     /* number of occurences of c in word
                                                (obvious implementation omitted) */
extern cmpFunc strcmp;                      /* comparison on lexical order (C library function) */

int strcmpIndirect(intFunc f, char* word1, char* word2)    /* apply f to word1 and word2, and*/
{    return f (word1) - f(word2); }                        /* compare the results */

bool inIntBounds(intFunc f, int min, int max, char* word)  /* apply f to word, check if the result*/
{    int x = f(word); return (min <= x) && (x <= max); }   /* is in the bounds [min .. max] */

bool inWordBounds(cmpFunc cmp,                             /* check if word is in the bounds */
         char* minWord, char* maxWord, char *word)          /* [minWord .. maxWord], according */
{                                                          /* to comparison function cmp */
    return (cmp(word, minWord) >= 0) && (cmp(maxWord, word) >= 0);
}
```

From these basic functions we can generate a number of different comparison and selection functions:

```
cmpFunc cmpLength = curry(strcmpIndirect, 1, strlen);

intFunc countSpaces = curry(countCharacter, 1, ' ');
intFunc countDots = curry(countCharacter, 1, '.');
cmpFunc cmpSpaces = curry(strcmpIndirect, 1, countSpaces));

selFunc containsSpaces = curry(inIntBounds, 3, countSpaces, 1, INT_MAX);
selFunc containsNoSpaces = curry(inIntBounds, 3, countSpaces, 0, 0);

selFunc shortWord = curry(inIntBounts, 3, strlen, 1, 5);
selFunc longWord = curry(inIntBounds, 3, strlen, 15, INT_MAX);
```

Programmers can extend this list at will, and they can write code that generates new predicates or comparison functions at run-time. Used in conjunction with generic sort and selection algorithms, this allows one to sort or select words according to an unlimited number of criteria, like lexical order, word length, regular expression matching, or the number of occurrences of a particular character. In order to do the same thing without **curry**, it would be necessary to explicitly write all function combinations (**cmpWordsOnLength, cmpWordsOnNumberOfSpaces**), and to design global data structures for storing values like word length bounds. Needless to say, this would require a lot more programming work.

4.3 Other functional operators

Currying can be used to implement other functional operators. Functional operators may be very useful for example if we want to design a mathematical package that constructs various functions and displays their graphs. Here is an example where we implement function composition and an approximation of the derivative function. Other higher-order functions could be written in a similar fashion, for example for function integration, or for a conditional function. For reasons of simplicity, the example only shows functions from float to float.

```
typedef float (*floatFunc) (float);
```

```
float FuncCompose(floatFunc f, floatFunc g, float arg)      /* composes two functions */
{       return f(g(arg)); }

float FuncDerive(floatFunc f, float epsilon, float x)       /* approximates the derivative if
                                                               epsilon is small enough */
{       return (f(x+epsilon) - f(x)) / epsilon; }


#define COMP(f, g)          curry(FuncCompose, 2, f, g)
#define DERIVATIVE(f)       curry(FuncDerive, 3, f, 0.0001)  /* nargs=3 because the float value
                                                                counts for two args */

float Inverse(float x) {return 1/x;}

/* now build a composite function */
floatFunc sinCos = COMP(sin, cos);                          /* sin and cos from the library */
floatFunc invSinCos = COMP(Inverse, sinCos);

/* take the derivative of it */
floatFunc df = DERIVATIVE(invSinCos);

/* apply it to some value */
result =df (3.14);

/* or pass it to another module */
drawGraph(df, <drawing bounds>);
```

Interactive applications that do some interpretation can benefit from such functional operators. In a typical interpreter architecture, there is a table that associates basic commands to basic functionalities of the application. With functional composition, conditional functions and other functional operators, it becomes possible to extend this table at run-time, so that users can easily extend the application with their own functions.

5. Use in Object-Oriented Languages

The C language was used as a basis for the development of two object-oriented programming languages: C++[10] and Objective-C[4]. Since they are compatible with C, the **curry** operator is applicable in those languages as well. Merging curried functions with object-oriented concepts yields some interesting results.

5.1 C++ implementation of curried functions: easier, but less powerful

Before we go to more interesting considerations, it should be mentioned that a partial implementation of curried functions can be done relatively easily in C++. It suffices to create a class that overloads the function call operator and stores the curried arguments. However this solution is not fully satisfactory, because it creates curried *objects* instead of curried functions. This does not make much difference as far as C++ source code is concerned, but such objects cannot be passed to other modules that expect C functions, like **qsort** or **signal** in our previous examples. Furthermore, the fact that function pointers and curried object pointers are not uniform complicates the programming task: both kinds of pointers cannot be freely exchanged, for example as nodes in an expression tree, because they have different calling protocols.

5.2 Objects can be seen as functions

Currying can be applied to C++ or Objective-C objects, and yields ordinary, non-member functions out of object member functions. In many cases a uniform view of objects and functions can simplify the programming task. Besides, object-oriented features like inheritance, delegation or archiving support can be used to implement families of functions, or even 'persistent functions'. But the most important result is that objects can then be used with other modules that do not know about object-oriented programming. For example currying offers a clean solution for merging X toolkit windowing software with C++ code and letting *widgets* use C++ objects as *callbacks*. By contrast, the solution currently used by C++/Motif programmers [5], very similar to what we describe in 5.1, is much more cumbersome.

Let us consider an example of currying in C++. Suppose there is a C++ class for windows:

```
class Window{
      <instance variables>
public:
      void open();
      void close();
      void move(int x, int y);
      ...
};
```

and suppose our application wants to bind some keys on the keyboard to some actions on windows. The code could look like this

```
Window w1, w2;

bindKey(aKey0, exit);                              /* key 0 exits the application */
bindKey(aKey1, curry(Window::open, 1, &w1));       /* key 1 opens window 1 */
bindKey(aKey2, curry(Window::close, 1, &w1));      /* key 2 closes window 1 */
bindKey(aKey3, curry(Window::move, 3, &w2, 1, 0)); /* key 3 moves window 2 to the right */
bindKey(aKey4, curry(Window::move, 3, &w2, 0, -1)); /* key 4 moves window 2 down */
```

Note that with one single protocol the keys can be bound to ordinary functions (**exit**) or to object member functions. Besides, adding new bindings at run-time for new windows is trivial.

5.3 Dynamic class modification Objective-C

In Objective-C, object closures can be achieved by currying the message dispatch function:

```
f = curry(objc_msgSend, 2, anObject, aMessage)
```

Here **f** is a function pointer that in fact applies **aMessage** to **anObject**. Since Objective-C keeps more information than C++ in its run-time environment, we can use dynamic function generation to get some interesting configurations. For example a class could dynamically decide to modify its method table, delegating some functionality to other objects via curried messages. The dynamic delegation would be invisible to the clients. Similarly, the interface of a class could be dynamically augmented by using functional operators to create composite methods that are added in the method table.

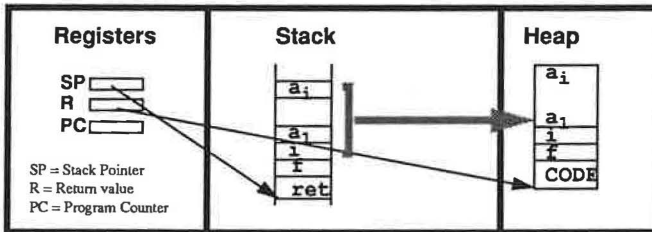
6. Implementation details

The general procedure for implementing curried functions in C is relatively straightforward. When currying occurs, memory must be allocated for storing the curried function and its arguments. When a curried function is called, the first arguments that were curried need to be merged with the last arguments supplied by the caller, so as to build a complete environment for the function to execute. There is one complication, however, in that we want curried functions to look exactly like normal function pointers. This means that we cannot afford to store the function and its partial environment in two different pointers: everything must be seen as one single unit by the caller. We circumvent this problem with run-time code generation. Memory areas are allocated in which some machine instructions are stored together with curried environments. In this way, the caller of a curried function can literally branch into the data. The program counter address can be used to retrieve the curried environment, and then control is transferred to the curried function.

The various steps of a curried function call will be more comprehensible with some pictures. The illustrations in this section reflect the behaviour of the 68030 implementation on Sun and NeXT workstations; our SPARC implementation is based on the same principles but is slighly more complex than what the pictures display. Notice how code implementing the currying mechanism is intercalated in such a way that *neither the caller nor the curried function* need to be aware that a currying mechanism is employed.

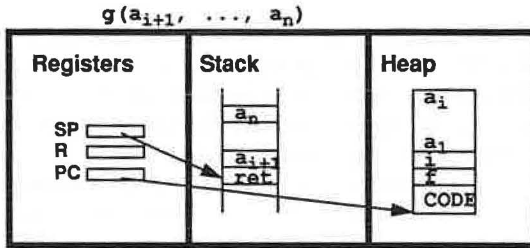
6.1 Creating a curried structure

```
g = curry(f, i, a1, ..., ai);
```

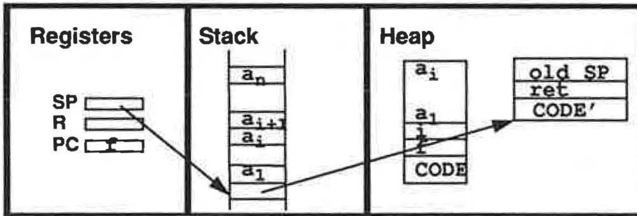


Here the **curry** function has been called with a function pointer and some arguments to be curried. All these parameters have been copied into a new structure allocated on the heap. Some code has also been copied at the beginning of that structure, so that the program can branch to that address. The **R** register, which is used to return function results, contains the address of the new structure.

6.2 Invoking a curried function

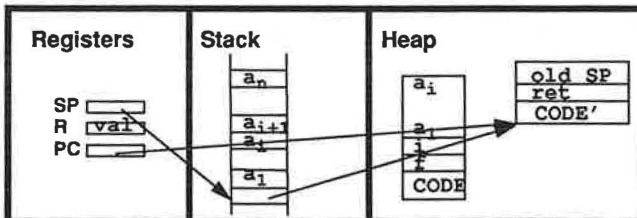


The curried function g is called with some parameters. The above picture shows the situation just before branching to **g**. As for any normal function call, the parameters have been pushed on the stack. The program counter points to the structure on the heap, so the next instruction to be executed will be CODE.

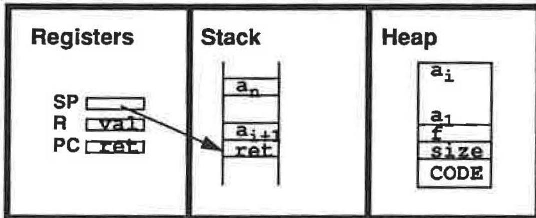


Now CODE was executed and the situation looks quite different. The stack has grown and the curried arguments were copied to it, so now the stack contains a complete environment for calling the function f. A new structure has been allocated on the heap, that saves the previous value of the stack pointer and the return address to the original caller. The return address that resides on the stack *points to that new return structure*, which means that whenever **f** returns it will not go directly to the original caller, but rather to CODE' in the return structure, so that some cleanup operations can be performed. The program counter holds the address of the next instruction, the beginning of function **f**.

6.3 Returning from a curried function



The function **f** is going to return, so the program counter contains the return address that was on the stack. As we saw before, this is the address of some cleanup code in a return structure. The **R** register contains the return value from function **f**.



What the cleanup code did was to bring the stack back to its original state and to delete the return structure. The **D0** register contains the return value from function **f**, and the program counter will branch back to the original caller.

6.4 Comments on implementation choices

Clearly the **curry** mechanism has some associated costs. At each creation of a curried function some memory must be allocated on the heap. Calling a curried function involves allocating a return structure, and copying arguments from the heap to the stack. Even returning from a curried function implies additional cleanup work. Although we did not run extensive tests, we can estimate a curried function call to be approximately 60 times slower than a normal function call. This figure is quite high, but it should be weighed against two factors. First, our current implementation can be significantly improved by optimizing memory allocation and getting support from the compiler (if currying is fully integrated into the language). Second, currying is not likely to be used heavily in an application, but rather marginally, only in the situations where it is convenient. Therefore the overall performance of the whole application will probably not be affected too much.

Another aspect of this implementation is that it has no automatic garbage collection. Creating a curried functions is like allocating memory for a new structure or a new object: the programmer should know when a curried function is no longer used and then explicitly free the memory. Instructions to do this were omitted in the programming examples just to keep the code as short as possible.

Finally, it should be mentioned that currying has the disadvantage of being dependent on the order of arguments. Not all C functions are good candidates for currying: for example the library function **recmp(char* regexp, char *string)**, that checks if a string matches a particular regular expression, can be curried easily:

```
f = curry(recmp, 1, "aRegularExpression")
```

f is a boolean function that compares its argument with "aRegularExpression", which may be very convenient; however, currying would be useless if the **recmp** function had been defined with the arguments in reverse order. This is not a hopeless problem, because currying could also be used to implement combinators that would permute, duplicate or cancel function arguments. However, performance degradation might then become too high to be acceptable.

7. Final considerations

An extension to C/C++/Objective-C has been proposed for extending functional programming capabilities. Its usefulness was shown in several examples, and an implementation was described that should be acceptable for the C programmers community. Some drawbacks have been educed, but should in our view be outweighed by the gain in expressiveness and power.

Further work should progress on two fronts. On the one hand we need to develop a more efficient implementation, and to port it on several machines in order to check whether the currying concept is as portable as the C language itself. An important addition would be to fully integrate currying within a C compiler, so that curried functions can be type-checked and can use compile-time information to generate better code. On the other hand, we are interested in exploring in more depth the possibilities and limitations of currying, especially when objects are concerned, and to compare currying with other related concepts like closures, environments or continuations.

References

[1] Abelson, H., and Sussman, G.J, *Structure and Interpretation of Computer Programs*, MIT Press, 1985.

[2] M. Ader, O.M. Nierstrasz, S. McMahon, G. Müller and A-K. Pröfrock, *The ITHACA Technology: A Landscape for Object-Oriented Application Development*, Proceedings, Esprit 1990 Conference, pp. 31-51, Kluwer Academic Publishers, Dordrecht, NL, 1990.

[3] Bird, Richard, and Wadler, Philip, *Introduction to Functional Programming*, Prentice Hall, 1988.

[4] Cox, Brad, *Object-Oriented Programming, An Evolutionary Approach*, Addison-Wesley, 1987

[5] Fekete, Jean-Daniel, *WWL, a Widget Wrapper Library for C++*, Laboratoire de Recherche en Informatique, Orsay, 1991.

[6] Field, Anthony J., and Harrison, Peter G., *Functional Programming*, Addison-Wesley, 1988.

[7] Gordon, Michael J.C., *The Denotational Description of Programming Languages*, Springer Verlag, 1979.

[8] Kernighan, Brian W., and Ritchie, Dennis M., *The C Programming Language*, Second Edition, Prentice-Hall, 1988.

[9] Nierstrasz, O., Tsichritzis, D., de Mey, V. and Stadelmann, M., *Objects + Scripts = Applications*, in *Object Composition*, ed D. Tsichritzis, Université de Genève, 1991.

[10] Stroustrup, Bjarne, *The C++ Programming Language*, Addison-Wesley, 1986.