3-1995

A Physical Example for Teaching Curried Functions

Gary T. Leavens
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports

Part of the Science and Mathematics Education Commons, and the Theory and Algorithms Commons

Recommended Citation
http://lib.dr.iastate.edu/cs_techreports/119

This Article is brought to you for free and open access by the Computer Science at Digital Repository @ Iowa State University. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Digital Repository @ Iowa State University. For more information, please contact hinefuku@iastate.edu.
A PHYSICAL EXAMPLE
FOR TEACHING CURRIED FUNCTIONS
Gary T. Leavens

TR #95-05
March 1995

Keywords: functional programming, physics, currying, teaching, examples.

1994 CR Categories: D.1.1 [Programming Techniques] Applicative (Functional) Pro-
gramming; J.2 [Physical Sciences and Engineering] Physics; K.3.2 [Computers and
Education] Computer and Information Science Education — Computer science educa-
tion.

Submitted for publication.

© Gary T. Leavens, 1995. All rights reserved.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040 USA
A PHYSICAL EXAMPLE FOR
TEACHING CURRIED FUNCTIONS

ABSTRACT

Curried functions are an important topic in Computing courses that teach functional programming, including courses that study programming languages. Good motivating examples for teaching curried functions and their utility can be taken from Physics.

BACKGROUND

Curried functions and currying are an important topic in computer science courses that teach functional programming [14, section 7.3]. Such courses include undergraduate courses in programming paradigms (unit PL11 in the ACM’s Computing Curricula 1991 [15]), with titles such as “Principles of Programming Languages” [11, p. 388] [10, p. 100]. Also included are undergraduate and graduate courses in programming language semantics (unit PL10 in [15]), with titles such as “Essentials of Programming Languages” [7, p.27]. Curried functions are also directly supported by some modern functional programming languages, such as Haskell [4].

Curry [2,3] and others [13] [5, pages 153-156] studying the concept of a function asked the question: “does a programming language need to provide functions with an arbitrary number of arguments?” Of course, there are many examples of useful functions that take several arguments, such as addition. There are at least to ways that one can code such functions in a language in which functions were only permitted to have one argument. The first is to pass a record, or some other similar data structure to the function. However, if one is interested, as Curry was, in finding a minimal core of ideas needed to express logic or programs, substituting data structures for multiple-argument functions merely rephrases the question as “does a programming language need to provide several built-in datatypes?” The second way to code functions with multiple arguments is to translate function calls such as into \((g(x))(y)\), for a suitable function \(g\). The notation \((g(x))(y)\) means that \(g(x)\) returns a function, which can then be applied to \(y\). The function \(g\) is called a curried function. It is a curried version of \(f\) if it satisfies the following equation.

\[ (g(x))(y) = f(x,y) \]

In general, this process, called currying, converts an \(n\)-argument function into a nest of \(n\) functions of one argument each. Although a data structure, called a closure (described below), must be built-in to a language that supports currying, it is well-known that no other data structures are needed for a programming language to be universal [6]. Furthermore, this data structure is all that is needed to implement functions, so this way of resolving the question seems to be simpler.

The main point of this paper is to give an example that can be used to explain the utility of curried functions. Before doing that, the rest of this introduction describes currying and closures in more detail. The way this background material is presented is also useful for teaching curried functions, but is itself novel only to the extent that it uses C to explain the need for closures.
C, BASIC, and Pascal programmers will wonder why the idea of curried functions cannot be easily programmed in their favorite language.\(^1\) The difficulty is not absolute. However, because these languages do not support function closures, curried functions are difficult to write.

A *closure* is a data structure that holds a function together with the environment (a programming language’s mapping from names to values) in which it was created. To illustrate the idea of function closures, and why they are needed if one is to be able to curry functions, consider the traditional first example of a curried function, a curried version of addition [14, p.210] [7, p. 27] [10, p. 100]. Figure 1 shows an attempt to write a curried addition function, called *cadd*, in ANSI C [8]. (A language with functions, or function pointers, as values is needed for this illustration.) The goal of this attempt is that one should be able to write \((\text{cadd}(2))(3)\) and have the result be 5.

```c
#include <stdio.h>
typedef int (*func)(int); /* functions that take and return integers */

int takes_y(int y) { return x + y; }

func cadd(int x) { return &takes_y; }

int main() { printf("%i\n", (cadd(2))(3)); }
```

Figure 1: Incorrect attempt to code curried addition in ANSI C. Note that \(x\) is undefined in the function *takes_y*.

The problem with the coding of *cadd* in Figure 1 is that \(x\) is undefined within the function *takes_y*. That is, when the address of *takes_y* is returned from *cadd*, nothing is done to remember the value of \(x\); so if the program actually compiled, *cadd*(2) would be the same as *cadd*(4). This is the problem that closures solve. That is, to program *cadd* correctly, one has to return some data structure that contains both the value of the actual parameter, \(x\), and the pointer to the function *takes_y*. A data structure that is sufficient for this example is shown in Figure 2. Note that in Figure 2, *cadd* returns a pointer to a newly allocated closure record. This record stores the relevant information from the environment (the value of \(x\)), as well as the address of the procedure to execute. Since a closure record is just a data structure in C, the helping function *invoke_closure* is needed to call the function in the closure with an additional argument.

---

\(^1\) If one looks at the semantics of object-oriented programming languages (see pages 140-142 of [1] for the idea), one can see that the objects in an object-oriented programming language are very similar to closures. Thus in a language like C++ or Smalltalk, there is a different way to simulate closures: one makes a closure object that contains the relevant part of the environment and which has as its only method (function member) the function. Details of this simulation are beyond the scope of this paper.
A PHYSICAL EXAMPLE FOR TEACHING CURRIED FUNCTIONS

#include <stdio.h>

typedef int (*func)(int, int);                      /* binary functions on integers */
typedef struct { func f; int x; } closure;                      /* closure records */
typedef closure *closurePtr;                     /* pointers to closure records */

int add(int x, int y) { return x+y; }

closurePtr cadd(int x) {
closurePtr c;
c = (closurePtr) malloc(sizeof(closure));             /* no check for NULL */
c->f = add;
c->x = x;
return c;
}

int invoke_closure(closurePtr c, int arg) { return (c->f)(c->x, arg); }

int main() { printf("%i\n", invoke_closure(cadd(2), 3)); }

Figure 2: Corrected curried addition written in ANSI C.

Figure 2 demonstrates the idea of closures, and it is quite helpful in teaching programmers what a closure is, and how they can be used to implement curried functions. However, to demonstrate the utility of curried functions, one should use a language, such as Scheme [1, 6, 14] that supports the automatic creation and invocation of closures. Because Scheme automatically creates closures, and can call them easily, students are not burdened with extraneous details and can concentrate on how to use the concept. For this reason, and also because the language is used in several texts that teach curried functions in the context of introductory programming [1, 6, 14] and programming languages [10], Scheme will be used in the rest of this paper.

As a simple example of the power of Scheme for this kind of example, consider Figure 3. Scheme is a dialect of LISP, and as such uses a fully parenthesized syntax, and prefix notation. In Scheme, the form (define n e) binds the name n to the value of the expression e. In Figure 3 the define binds the name cadd to the function created by the lambda-expression (lambda (x) ...). The form (lambda (n) e) makes a closure, which remembers its environment of creation and a function with formal parameter n and body e. In Figure 3 the body of (lambda (x) ...) is another lambda-expression, so when the closure created by (lambda (x) ...) is invoked, it returns the closure created by its body, which is the lambda-expression (lambda (y) (+ x y)). The body of this closure, when invoked, adds together the values of x and y; it can find a value for x in the environment remembered by the closure when it was created.

(define cadd
  (lambda (x)
    (lambda (y)
      (+ x y))))

; doing the following is similar to running the C program in Figure 2
(display ((cadd 2) 3))
(newline)

Figure 3: Curried addition written in Scheme.
A PHYSICAL EXAMPLE FOR TEACHING CURRIED FUNCTIONS

In Scheme, comments begin with a semicolon (;) and continue to the end of the line. Scheme has built-in procedures for I/O, including display and newline. In Scheme, the call of a procedure (i.e., a closure), \( f \), with arguments \( x \) and \( y \) is written as follows: \( (f \ x \ y) \). All procedures, including built-in ones such as + are invoked with the same syntax. Hence \( (+ \ 2 \ 3) \) is how one adds 2 and 3. Since, \( \text{cadd} \) is a curried version of +, \( (\text{cadd} \ 2) \) is also a closure; when invoked it adds 2 to its argument. Hence \((\text{cadd} \ 2) \ 3\) first produces the closure for \( (\text{cadd} \ 2) \) and then applies that to the argument 3. Since \text{newline} takes no arguments, it is invoked by writing \((\text{newline})\).

THE GRAVITATIONAL FORCE EXAMPLE

Besides the curried addition example, typical examples used in textbooks are curried mapping and reduction functions on lists. Such examples connect only with students’ experience in programming. Thus many students think that curried functions have little to do with their everyday experience, and so have trouble grasping the concept. A novel\footnote{Gerald J. Sussman apparently discussed something like this in his invited lecture at the 1982 ACM Symposium on LISP and Functional Programming. Sussman says that he never wrote down his lecture. Although his lecture was titled “Teaching the Control of Complexity,” the idea does not seem to have made it into the practice of teaching functional programming, and it does not appear in [1].} approach is to draw more compelling examples from Physics. Imagine telling students that the universe itself is held together by forces that can be modeled using curried functions.

A simple physical example that can be used to teach curried functions is the Newtonian gravitational force law. Figure 4 codes the gravitational force law as the function \( \text{grav-force} \) in the programming language Scheme\[1,14\]. Readers not familiar with Scheme may wish to consult the Appendix for additional explanation of this code.

\[
\begin{align*}
(\text{define} \ G \ 6.670\text{e-11}) & \quad ; \text{N} \times \text{m}^2 / \text{kg}^2 \\
(\text{define} \ \text{grav-force} & \quad \lambda (m1 \ r \ m2) \quad ; \text{kg} \ \text{m} \ \text{kg} \\
 & \quad (\text{if} \ (\text{zero?} \ r) \\
 & \quad \quad 0.0 \\
 & \quad \quad (- \ (/ \ (* \ G \ (* \ m1 \ m2))) \\
 & \quad \quad (\text{square} \ r)))))) \\
(\text{define} \ \text{square} & \quad \lambda (\text{num}) \\
 & \quad (* \ \text{num} \ \text{num}))
\end{align*}
\]

Figure 4: Definitions of the universal gravitational constant, \( G \), and the function \( \text{grav-force} \) in Scheme. The auxiliary Scheme function \( \text{square} \) returns the square of its actual parameter.

Currying the function \( \text{grav-force} \) gives an excellent teaching example, because at each stage of partial application, a useful function is obtained. The curried version of \( \text{grav-force} \), called \( \text{grav-force-c} \), is given in Figure 5. As a start to explaining this example, note that the Scheme expression

\[
((\text{grav-force-c} \ 5.96\text{e24}) \ 6.37\text{e6}) \ 68.0)
\]

\footnote{Gerald J. Sussman apparently discussed something like this in his invited lecture at the 1982 ACM Symposium on LISP and Functional Programming. Sussman says that he never wrote down his lecture. Although his lecture was titled “Teaching the Control of Complexity,” the idea does not seem to have made it into the practice of teaching functional programming, and it does not appear in [1].}
also has as its approximate value 666.2 Newtons. In general, the closure named \texttt{grav-force-c} is such that for all \( m_1, r, \) and \( m_2 \), the following equation between Scheme expressions holds.

\[
(((\text{grav-force-c} \ m_1) \ r) \ m_2) = (\text{grav-force} \ m_1 \ r \ m_2)
\]

In the Scheme expression (((\text{grav-force-c} \ m_1) \ r) \ m_2), the closure \texttt{grav-force-c} is applied to \( m_1 \), and this returns another closure. The closure returned is the one defined by the expression (\texttt{lambda} (r) ...) in Figure 5. Note that when this closure is created, there is already a value for \( m_1 \), which this function will ultimately use. That closure is applied to \( r \), and that application returns another closure. This closure is the one defined by the expression (\texttt{lambda} (m_2) ...) in Figure 5. This closure is then applied to \( m_2 \), which returns a number.

\[
\begin{align*}
\text{(define grav-force-c} \\
\text{(lambda} (m1) ; kg \\
\text{(lambda} (r) ; m \\
\text{(lambda} (m2) ; kg \\
\text{(if} (\text{zero?} \ r) \\
\quad 0.0 \\
\quad (- (/ (* G (* m1 m2)) \text{(square} \ r))))))))
\end{align*}
\]

Figure 5: The curried function \texttt{grav-force-c}. The universal gravitational constant, \texttt{G}, and the Scheme function \texttt{square} are defined above.

A helpful adjunct to a preliminary explanation of curried functions is a discussion of their types. To explain the notation for function types, consider first a function that is not curried. For example, the type of \texttt{grav-force} is as follows.

\[
[(\text{kg x m x kg}) \rightarrow \text{N}]
\]

In the above notation, “\texttt{kg}” stands for a set of numbers that are thought of as kilograms, “\texttt{m}” stands for a set of numbers that are thought of as meters, “\texttt{N}” stands for a set of numbers that are thought of as Newtons, and the notation \([S \rightarrow T]\) means the set of all functions with domain \( S \) and range \( T \). That is, \texttt{grav-force} has a type that is the set of functions whose domain is triples of kilograms, meters, and kilograms, and whose range is Newtons. The type of the curried function \texttt{grav-force-c} is as follows.

\[
[	ext{kg} \rightarrow [	ext{m} \rightarrow [	ext{kg} \rightarrow \text{N}]])
\]

That is, \texttt{grav-force-c} has a type that is the set of functions whose domain is kilograms, and whose range is the set of functions \([\text{m} \rightarrow \text{[kg} \rightarrow \text{N}]])\}. Thus the range type of \texttt{grav-force-c} is the set of functions whose domain is meters, and whose range is the set of functions \([\text{kg} \rightarrow \text{N}]\). Using the notation, “\( x : T \)”, to mean that \( x \) has type \( T \), the table in Figure 6 illustrates the type of \texttt{grav-force-c} using examples.
A PHYSICAL EXAMPLE FOR TEACHING CURRIED FUNCTIONS

<table>
<thead>
<tr>
<th>Scheme Expression</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>grav-force-c : [kg → [m → [kg → N]]]</td>
<td></td>
</tr>
<tr>
<td>5.96e24 : kg</td>
<td></td>
</tr>
<tr>
<td>(grav-force-c 5.96e24) : [m → [kg → N]]</td>
<td></td>
</tr>
<tr>
<td>6.37e6 : m</td>
<td></td>
</tr>
<tr>
<td>(((grav-force-c 5.96e24) 6.37e6) : [kg → N]</td>
<td></td>
</tr>
<tr>
<td>68.0 : kg</td>
<td></td>
</tr>
<tr>
<td>(((grav-force-c 5.96e24) 6.37e6) 68.0) : N</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6: Scheme expressions and their types, illustrating the type of the curried function grav-force-c.

Once this example is understood, it can be used to explain to students why curried functions are useful. Curried functions are useful as planned tool-makers, since a curried function is a function that produces other functions, which can be used to do useful computations (including making yet more functions). One way to illustrate this is by a series of applications, and discussions about the utility of each step.

For example, to work with the gravitational force exerted by the earth, one passes to grav-force-c the mass of the earth. This is done in the definition of the function earths-force-fun below, which assigns to each radius from the earth’s center a function that assigns to each mass placed at that distance a force.

```
(define mass-of-earth 5.96e24) ; kg
(define earths-force-fun ; type: [m → [kg → N]]
 (grav-force-c mass-of-earth))
```

Passing any other mass, such as the mass of the galaxy or a student to grav-force-c gives the analogous function for that mass.

To work with the gravitational force of the earth at the earth’s surface, one can pass the earth’s radius to earths-force-fun. This is done below in the definition of the Scheme function earths-force-at-surface. This function assigns to each mass at the earth’s surface the magnitude of the force exerted by the earth’s gravity on that mass.

```
(define radius-of-earth 6.37e6) ; m
(define earths-force-at-surface ; type: [kg → N]
 (earths-field radius-of-earth))
```

Passing any other radius from the earth’s center to earths-force-fun, such as the distance from the earth’s center to the orbit of the space shuttle or to the sun or moon, gives an analogous function for that radius.

As an example of how to use earths-force-at-surface, the following expression computes the gravitational force exerted by the earth (at its surface) on a mass of 68 kilograms (about 150 pounds).

```
(earths-force-at-surface 68.0)
```

The value of this expression is approximately 666.2 Newtons. Using a unit mass, one can find the acceleration per unit mass at the earth’s surface, which is about 9.8 meters per second squared.

In summary, the advantage of currying grav-force is that it allowed us to partially apply that function, and at each stage a useful function is obtained. One should also note a disadvantage to curried functions: the designer of a curried function has to foresee what partial applications will be useful. For example, if one wanted to investigate the gravitational force of different masses at a distance of a meter, then the currying of grav-force-c would not be appropriate, because it was not planned so that one could
A PHYSICAL EXAMPLE FOR TEACHING CURRIED FUNCTIONS

apply it to a distance first. However, in this case the order of arguments does reflect the order implicit in the concept of the Newtonian gravitational field used in Physics. (See [9] for more details on the relationship between curried functions and fields.)

RECOMMENDATION

I recommend the use of such physical examples for demonstrating the utility of curried functions to students of functional programming. I have used such examples on homeworks with some success in an introductory class in computer programming that is taught using Scheme. I have also used such examples in lectures in a graduate course in programming languages that has been taught using Standard ML [12]. Students find such example are intuitive, and connected with their real-world experience. They are particularly excited about such examples if they have already taken a course in Physics. The aura of Physics as being “natural’ adds “naturality’ to the concept of curried functions, which might otherwise seem highly “artificial.’’

ACKNOWLEDGMENTS

Thanks Dan Friedman for comments on drafts, and to Dan, Mitch Wand, and Phil Wadler for helping me track down people who might have thought of this before. Thanks to David A. Turner, Hal Abelson, and Gerry Sussman, for responding to that search. Thanks to Al Baker, John Clem, Brian Harvey, Akhilesh Tyagi, David Fernandez-Baca, Phil Wadler, Yoonsik Cheon, K. Kishore Dhara, Marybeth Gurski, Ke Chen, Yanfang Li, and Mang-Mang Ling, Jack Lutz, and Tim Wahls for comments on drafts of other versions of this paper. Thanks to the anonymous referees for comments that helped improve the paper’s relevance for this journal.

APPENDIX: EXPLANATION OF THE SCHEME CODE

IN FIGURE 4

For those not familiar with Scheme, the following dissection of Figure 4 may help in reading the rest of the paper.

As mentioned in the introduction, define introduces a name being defined. The name is followed by an expression whose value becomes the name’s value. There are three define expressions in Figure 4. The first such define binds the name G to the universal gravitational constant in SI units. Recall that a semicolon (;) starts a comment, which continues to the end of the line; the first comment gives the units of G.

The second define in Figure 4 binds the name grav-force to a Scheme function. Recall that lambda expression makes a closure; the closure’s function has formal parameters named within the following set of parentheses, and a body which is an expression. The function grav-force thus has three formal parameters: m1, r, and m2. When a function is called, it returns as its value the result of the expression that is its body, using the actual parameter values as the values of its formal parameters. For example, the function call

(grav-force 5.96e24 6.37e6 68.0)

has as its approximate value 666.2 Newtons; this is the magnitude of the gravitational force when m1 is 5.96 x 10^24 kilograms (the earth’s mass), r is 6.37 x 10^6 meters (the
earth’s radius), and \( m_2 \) is 68.0 kilograms. An if-expression of the form \((\text{if } b \ e_2 \ e_3)\) returns the value of \( e_2 \) if the value of the test \( b \) is true, and otherwise returns the value of \( e_3 \). The built-in Scheme predicate, \texttt{zero?}, used in Figure 4, returns true just when its argument is zero.

REFERENCES


