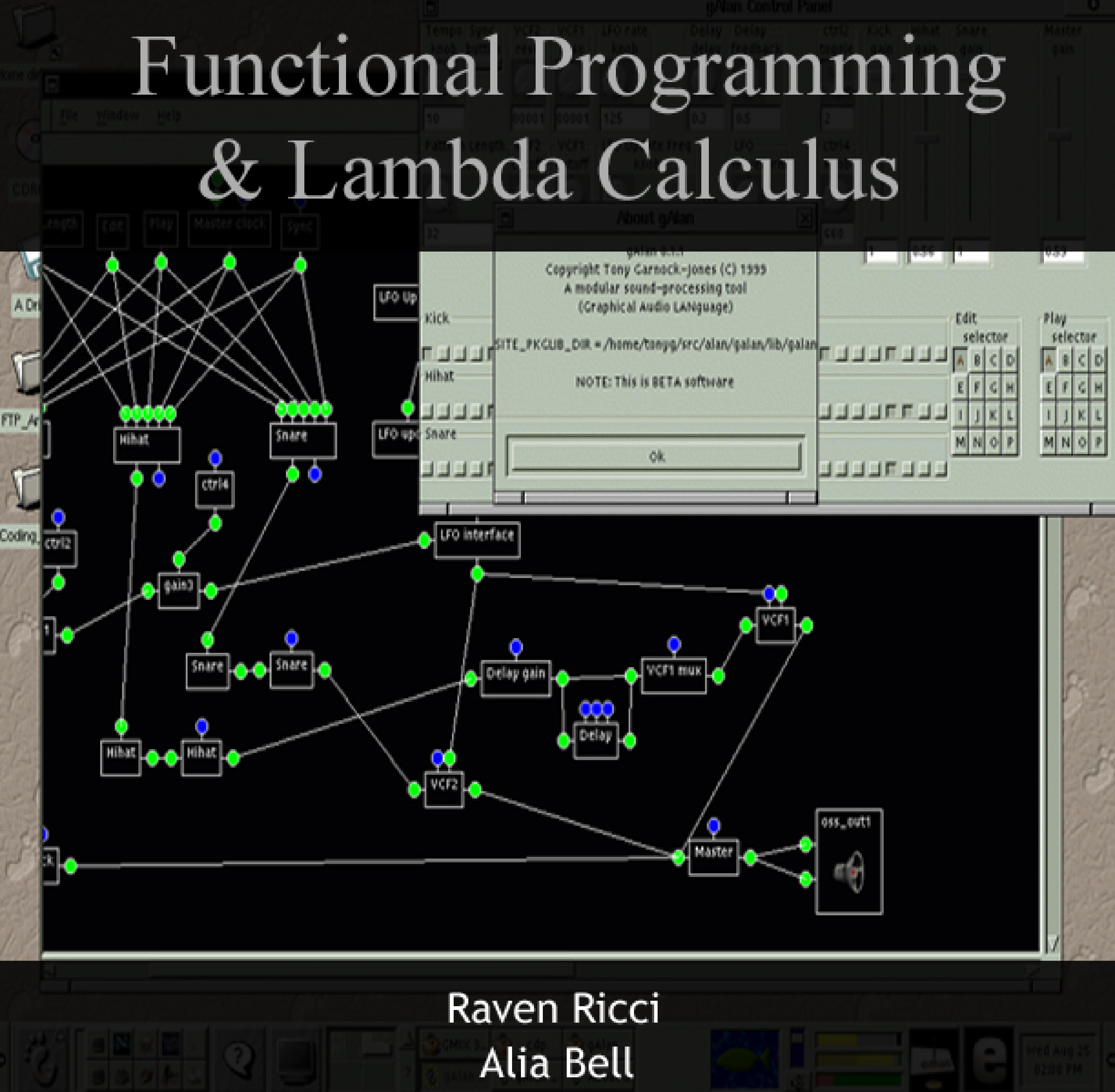
Functional Programming & Lambda Calculus



gAlan 0.1.1
Copyright Tony Garnock-Jones (C) 1999
A modular sound-processing tool
(Graphical Audio LANguage)

SITE_PKGLIB_DIR = /home/tonyg/src/alan/galan/lib/galan

NOTE: This is BETA software

Raven Ricci
Alia Bell

First Edition, 2012

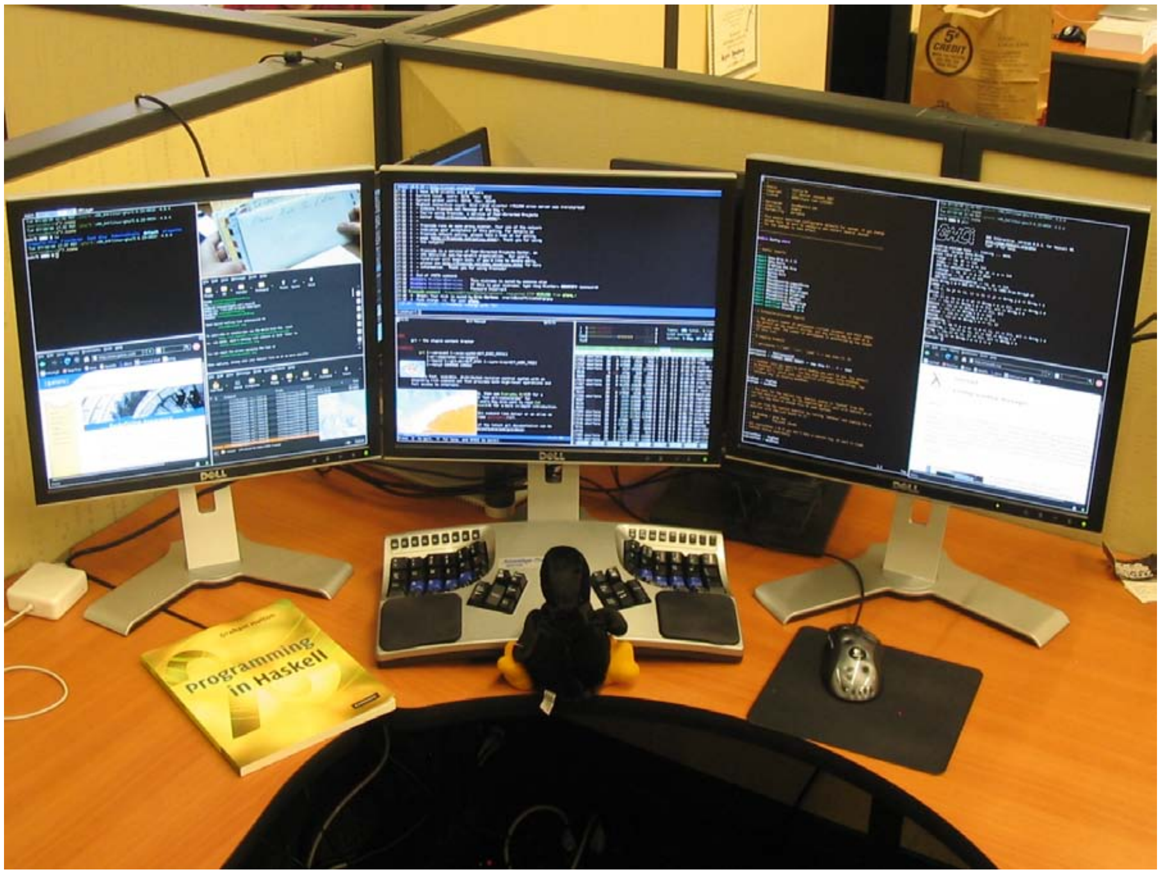
ISBN 978-81-323-1272-7

Published by:
College Publishing House
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Chapter 1

Introduction to Functional Programming



In computer science, **functional programming** is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data. It emphasizes the application of functions, in contrast to the imperative programming style, which emphasizes changes in state. Functional programming has its roots in lambda calculus, a formal system developed in the 1930s to investigate function definition, function application, and recursion. Many functional programming languages can be viewed as elaborations on the lambda calculus.

In practice, the difference between a mathematical function and the notion of a "function" used in imperative programming is that imperative functions can have side effects, changing the value of program state. Because of this they lack referential transparency, i.e. the same language expression can result in different values at different times depending on the state of the executing program. Conversely, in functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function *f* twice with the same value for an argument *x* will produce the same result *f(x)* both times. Eliminating side effects can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

Functional programming languages, especially purely functional ones, have largely been emphasized in academia rather than in commercial software development. However, prominent functional programming languages such as Scheme, Erlang, Objective Caml, and Haskell have been used in industrial and commercial applications by a wide variety of organizations. Functional programming also finds use in industry through domain-specific programming languages like R (statistics), Mathematica (symbolic math), J and K (financial analysis), F# in Microsoft .NET and XSLT (XML). Widespread declarative domain specific languages like SQL and Lex/Yacc, use some elements of functional programming, especially in eschewing mutable values. Spreadsheets can also be viewed as functional programming languages.

Programming in a functional style can also be accomplished in languages that aren't specifically designed for functional programming. For example, the imperative Perl programming language has been the subject of a book describing how to apply functional programming concepts. JavaScript, one of the most widely employed languages today, incorporates functional programming capabilities.

History

Lambda calculus provides a theoretical framework for describing functions and their evaluation. Although it is a mathematical abstraction rather than a programming language, it forms the basis of almost all functional programming languages today. An equivalent theoretical formulation, combinatory logic, is commonly perceived as more abstract than lambda calculus and preceded it in invention. It is used in some esoteric languages including Unlambda. Combinatory logic and lambda calculus were both originally developed to achieve a clearer approach to the foundations of mathematics .

An early functional flavored language was LISP, developed by John McCarthy while at MIT for the IBM 700/7000 series scientific computers in the late 1950s. LISP introduced many features now found in functional languages, though LISP is technically a multi-paradigm language. Scheme and Dylan were later attempts to simplify and improve LISP.

Information Processing Language (IPL) is sometimes cited as the first computer-based functional programming language. It is an assembly-style language for manipulating lists of symbols. It does have a notion of "generator", which amounts to a function accepting a function as an argument, and, since it is an assembly-level language, code can be used as data, so IPL can be regarded as having higher-order functions. However, it relies heavily on mutating list structure and similar imperative features.

Kenneth E. Iverson developed APL in the early 1960s, described in his 1962 book *A Programming Language* (ISBN 9780471430148). APL was the primary influence on John Backus's FP. In the early 1990s, Iverson and Roger Hui created J. In the mid 1990s, Arthur Whitney, who had previously worked with Iverson, created K, which is used commercially in financial industries.

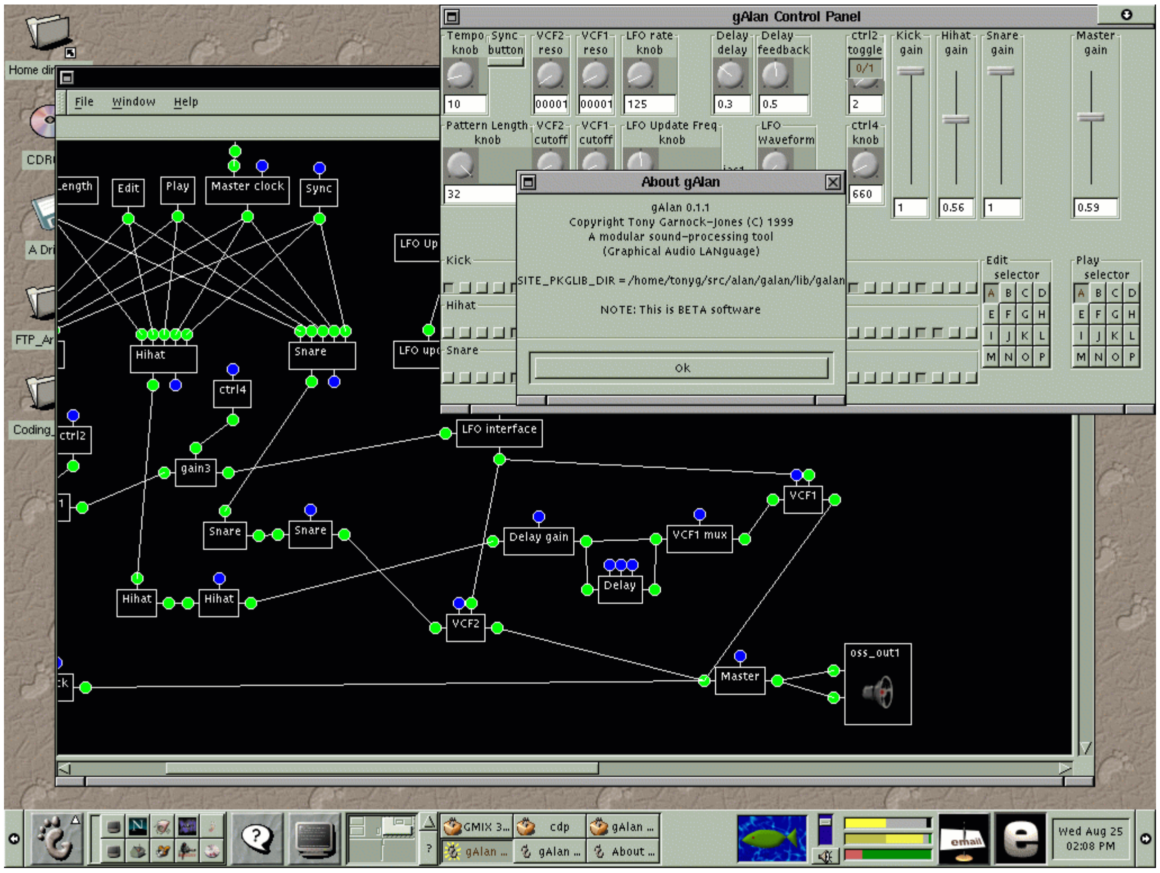
John Backus presented FP in his 1977 Turing Award lecture Can Programming Be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs. He defines functional programs as being built up in a hierarchical way by means of "combining forms" that allow an "algebra of programs"; in modern language, this means that functional programs follow the principle of compositionality. Backus's paper popularized research into functional programming, though it emphasized function-level programming rather than the lambda-calculus style which has come to be associated with functional programming.

In the 1970s ML was created by Robin Milner at the University of Edinburgh, and David Turner developed initially the language SASL at the University of St. Andrews and later the language Miranda at the University of Kent. ML eventually developed into several dialects, the most common of which are now Objective Caml and Standard ML. Also in the 1970s, the development of Scheme (a partly-functional dialect of Lisp), as described in the influential Lambda Papers and the 1985 textbook Structure and Interpretation of Computer Programs, brought awareness of the power of functional programming to the wider programming-languages community.

In the 1980s, Per Martin-Löf developed intuitionistic type theory (also called *Constructive* type theory), which associated functional programs with constructive proofs of arbitrarily complex mathematical propositions expressed as dependent types. This led to powerful new approaches to interactive theorem proving and has influenced the development of many subsequent functional programming languages.

The Haskell language began with a consensus in 1987 to form an open standard for functional programming research; implementation releases have been ongoing since 1990.

Concepts



gAlan 0.1.1
Copyright Tony Garnock-Jones (C) 1999
A modular sound-processing tool
(Graphical Audio LANguage)

SITE_PKGLIB_DIR = /home/tonyg/src/alan/galan/lib/galan

NOTE: This is BETA software

A number of concepts and paradigms are specific to functional programming, and generally foreign to imperative programming (including object oriented programming). However, programming languages are often hybrids of several programming paradigms so programmers using "mostly imperative" languages may have utilized some of these concepts.

Type systems, polymorphism, algebraic datatypes and pattern matching

Especially since the development of Hindley–Milner type inference in the 1970s, functional programming languages have tended to use typed lambda calculus, as opposed to the untyped lambda calculus used in Lisp and its variants (such as Scheme). The use of algebraic datatypes and pattern matching makes manipulation of complex data structures more convenient and expressive; the presence of strong compile-time type checking makes programs more reliable, while type inference frees the programmer from the need to manually declare types to the compiler.

Some research-oriented functional languages such as Coq, Agda, Cayenne, and Epigram are based on intuitionistic type theory, which allows types to depend on terms. Such types are called dependent types. These type systems do not have decidable type inference and are difficult to understand and program with. But dependent types can express arbitrary propositions in predicate logic. Through the Curry–Howard isomorphism, then, well-typed programs in these languages become a means of writing formal mathematical proofs from which a compiler can generate certified code. While these languages are mainly of interest in academic research (including in formalized mathematics), they have begun to be used in engineering as well. Compcert is a compiler for a subset of the C programming language that is written in Coq and formally verified.

A limited form of dependent types called generalized algebraic data types (GADT's) can be implemented in a way that provides some of the benefits of dependently-typed programming while avoiding most of its inconvenience. GADT's are available in the Glasgow Haskell Compiler and in Scala (as "case classes"), and have been proposed as additions to other languages including Java and C#.

Functional programming in non-functional languages

It is possible to employ a functional style of programming in languages that are not traditionally considered functional languages. Some non-functional languages have borrowed features such as higher-order functions, and list comprehensions from functional programming languages. This makes it easier to adopt a functional style when using these languages. Functional constructs such as higher-order functions and lazy lists can be obtained in C++ via libraries. In C, function pointers can be used to get some of the effects of higher-order functions. For example the common function map can be implemented using function pointers. In Visual Basic 9 and C# 3.0 and higher, lambda functions can be employed to write programs in a functional style. In Java, anonymous classes can sometimes be used to simulate closures, however anonymous classes are not always proper replacements to closures because they have more limited capabilities.

Many object-oriented design patterns are expressible in functional programming terms: for example, the strategy pattern simply dictates use of a higher-order function, and the visitor pattern roughly corresponds to a catamorphism, or fold.

The benefits of immutable data can be seen even in imperative programs, so programmers often strive to make some data immutable even in imperative programs.

Comparison of functional and imperative programming

Functional programming is very different from imperative programming. The most significant differences stem from the fact that functional programming avoids side effects, which are used in imperative programming to implement state and I/O. Pure functional programming disallows side effects completely. Disallowing side effects provides for referential transparency, which makes it easier to verify, optimize, and parallelize programs, and easier to write automated tools to perform those tasks.

Higher order functions are rarely used in older imperative programming. Where a traditional imperative program might use a loop to traverse a list, a functional style would often use a higher-order function, map, that takes as arguments a function and a list, applies the function to each element of the list, and returns a list of the results.

Simulating state

There are tasks (for example, maintaining a bank account balance) that often seem most naturally implemented with state. Pure functional programming performs these tasks, and I/O tasks such as accepting user input and printing to the screen, in a different way.

The pure functional programming language Haskell implements them using monads, derived from category theory. Monads are powerful and offer a way to abstract certain types of computational patterns, including (but not limited to) modeling of computations with mutable state (and other side effects such as I/O) in an imperative manner without losing purity. While existing monads may be easy to apply in a program, given appropriate templates and examples, many find them difficult to understand conceptually, e.g., when asked to define new monads (which is sometimes needed for certain types of libraries).

Alternative methods such as Hoare logic and uniqueness have been developed to track side effects in programs. Some modern research languages use effect systems to make explicit the presence of side effects.

Efficiency issues

Functional programming languages are often less efficient in their use of CPU and memory than imperative languages such as C and Pascal. However, for programs that perform intensive numerical computations, functional languages such as Objective Caml and Clean are only slightly slower than C. For programs that handle large matrices and multidimensional databases, array functional languages (such as J and K) were designed with speed optimization.

Further, immutability of data can, in many cases, lead to execution efficiency in allowing the compiler to make assumptions that are unsafe in an imperative language, thus increasing opportunities for inline expansion.

Lazy evaluation may also speed up the program, even asymptotically, whereas it may slow it down at most by a constant factor (however, it may introduce memory leaks when used improperly).

Coding styles

Imperative programs tend to emphasize the series of steps taken by a program in carrying out an action, while functional programs tend to emphasize the composition and arrangement of functions, often without specifying explicit *steps*. A simple example illustrates this with two solutions to the same programming goal (calculating Fibonacci numbers) using the same multi-paradigm language Standard ML.

```
(* Fibonacci numbers, imperative style *)
val n = 10

val first = ref 0   (* seed value fibonacci(0) *)
val second = ref 1  (* seed value fibonacci(1) *)
val fib_number = ref (!first + !second)  (* calculate fibonacci(2) *)
val position = ref 0
val () =
    while !position < n-2 (* iterate n-2 times to give Fibonacci number
n (for N > 2) *)
    do (
        first := !second; (* update the value of the two 'previous'
variables *)
        second := !fib_number;
        fib_number := !first + !second; (* update the result value to
fibonacci position *)
        position := !position + 1)

val () = print (Int.toString (!fib_number) ^ "\n")
```

A functional version has a different feel to it:

```
(* Fibonacci numbers, functional style *)
fun fibonacci 0 = 0 (* base case *)
  | fibonacci 1 = 1 (* base case *)
  | fibonacci n = fibonacci (n-1) + fibonacci (n-2) (* recursively
calculate fibnacci(n) *)

val () = print (Int.toString (fibonacci 10) ^ "\n")
```

The imperative style describes the intermediate steps involved in calculating `fibonacci(N)`, and places those steps inside a loop statement. In contrast, the functional style describes the mathematical equation that defines a `fibonacci(N)` number with respect to previous numbers in the Fibonacci sequence, where intermediate calculation steps are calculated using recursion.

Use in industry

Functional programming has a reputation for being of purely academic interest. However, several prominent functional programming languages have been used in commercial or industrial applications. For example, the Erlang programming language, which was developed by the Swedish company Ericsson in the late 1980s, was originally used to implement fault-tolerant telecommunications systems. It has since become popular for building a range of applications at companies such as T-Mobile, Nortel, and Facebook. The Scheme dialect of Lisp was used as the basis for several applications on early Apple Macintosh computers, and has more recently been applied to problems such as training simulation software and telescope control. Objective Caml, which was introduced in the mid 1990s, has seen commercial use in areas such as financial analysis, driver verification, industrial robot programming, and static analysis of embedded software. Haskell, although initially intended as a research language, has also been applied by a range of companies, in areas such as aerospace systems, hardware design, and web programming.

Other functional programming languages that have seen use in industry include Scala, F#, Lisp, Standard ML, and Clojure.

Chapter 2

First-Class Function and Purely Functional

First-class function

In computer science, a programming language is said to support **first-class functions** (also called **function literals**, **function types**) if it treats functions as first-class objects. Specifically, this means that the language supports constructing new functions during the execution of a program, storing them in data structures, passing them as arguments to other functions, and returning them as the values of other functions. This concept doesn't cover any means external to the language and program (metaprogramming), such as invoking a compiler or an eval function to create a new function.

These features are a necessity for the functional programming style, in which (for instance) the use of higher-order functions is a standard practice. A simple example of a higher-ordered function is the *map* or *mapcar* function, which takes as its arguments a function and a list, and returns the list formed by applying the function to each member of the list. For a language to support *map*, it must support passing a function as an argument.

There are certain implementation difficulties in passing functions as arguments and returning them as results. Historically, these were termed the funarg problems, the name coming from "function argument".

In type theory, the type of functions accepting values of type A and returning values of type B may be written as $A \to B$ or B^A. In the Curry-Howard correspondence, function types are related to logical implication; lambda abstraction corresponds to discharging hypothetical assumptions and function application corresponds to the modus ponens inference rule. Besides the usual case of programming functions, type theory also uses first-class functions to model associative arrays and similar data structures.

In category-theoretical accounts of programming, the availability of first-class functions corresponds to the closed category assumption. For instance, the simply-typed lambda calculus corresponds to the internal language of cartesian closed categories.

Availability

Languages which are strongly associated with functional programming, such as Lisp, Scheme, ML, F#, and Haskell, all support first-class functions. Other languages which also support them include Perl, Python, PHP, Lua, Tcl/Tk, ECMAScript (JavaScript, ActionScript), Ruby, Io, Scala, and Nemerle.

Most modern, natively compiled programming languages (e.g. C, C++, and Pascal) support function pointers, which can be stored in data structures and passed as arguments to other functions. Nevertheless, they are not considered to support first-class functions, since anonymous functions are not generally available, and functions defined by function pointers cannot dynamically encapsulate state by closing over their free variables. C# has type-safe delegates; a delegate can encapsulate either a reference to a static method or a method reference plus an object to call it on. The C++ programming language similarly supports function objects through the '()' user-defined operator, which allows first-class objects to be treated as functions. However, neither of these support true first-class functions, since the objects which hold the function's dynamic state must be constructed manually. Additionally, real anonymous functions ('lambdas') have no language support in the latest C++ standard, although they are included in the working draft of C++0x. Recent versions of C# support lambdas in many uses, including anonymous methods. C# Version 3 adds a lambda expression lexer and parser which run at compile-time and then a lambda code-generation portion of the compiler which executes at run-time.

Comparison

In this section we compare how some concepts are handled in a functional language with first-class functions (Haskell) compared to an imperative language where functions are second-class citizens (C).

Higher-order functions

In languages where functions are first-class citizens, function can be passed as arguments to other functions in the same way other types can (a function taking another function as argument is called a higher-order function):

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

Languages where functions are not first-class often still allow one to write higher-order functions through the use of concepts such as function pointers or delegates.

```
void map(int (*f)(int), int x[], size_t n) {
    for (int i = 0; i < n; i++)
        x[i] = (*f)(x[i]);
}
```

When comparing the two samples one should note that there are a number of differences between the two approaches that are *not* directly related to the support of first-class functions. The Haskell sample operates on lists, while the C sample operates on arrays. Both are the most natural compound data structures in the respective languages and making the C sample operate on linked lists would have made it unnecessarily complex. This also accounts for the fact that the C functions needs an additional parameter (giving the size of the array) and that the C function updating the array in-place, returning no value, whereas in Haskell data structures are persistent (a new list is returned while the old is left intact.) The Haskell sample uses recursion to traverse the list, while the C sample uses iteration. Again, this is the most natural way to express this function in both languages, but the Haskell sample could easily have been expressed in terms of a fold and the C sample in terms of recursion. Finally, the Haskell function has a polymorphic type, as this is not supported by C we have fixed all type variables to the type constant `int`.

Examples

D

```
alias real delegate(real x) Delegate;

Delegate makeDerivative(Delegate f, real deltaX)
{
    return delegate real (real x)
    {
       return (f(x + deltaX) - f(x)) / deltaX;
    };
}

void main()
{
    real mySin(real s)
    {
        return std.math.sin(s);
    }

    Delegate cos = makeDerivative(&mySin, 0.000001);
    writefln(cos(0));
}
```

Erlang

```
-module(calculus).
-export([derivate/2]).

% F: function that takes a number and returns a number
% DeltaX: small positive number
% Returns a function that is an approximate derivative of F

derivative(F, DeltaX) ->
  fun(X) ->
    (F(X + DeltaX) - F(X)) / DeltaX
  end.
```

Scheme

```
(define square (lambda (x) (* x x))) ;assigning a function literal to a
variable
(define fns (list + sin square)) ;functions as elements of a list
(define (compose f g) (lambda (x) (f (g x)))) ;a function that takes
two functions as arguments and
        ;returns a composite function
((compose sqrt square) -3) ;function as return value of another
function, applied to the argument -3
;the result is: 3
(define fns-squared (map (lambda (fn) (compose square fn)) fns))
;compose the list of function with
        ;the square function -> list of functions
(map (lambda(fn) (fn 3)) fns-squared) ;apply each function to the
argument 3
;the result is the list: (9 0.01991485667481699 81)
```

Lisp

```
(lambda (a b) (* a b)) ; function literal
((lambda (a b) (* a b)) 4 5) ; function is passed 4 and 5
```

Lua

In this example a first-class function is being used to specify the sort strategy for a table of IP addresses.

```
     network = {
       {name = "grauna",  IP = "210.26.30.34"},
       {name = "arraial", IP = "210.26.30.23"},
       {name = "lua",     IP = "210.26.23.12"},
       {name = "derain",  IP = "210.26.23.20"},
     }
```

To sort the table in reverse alphabetical order by the host name, we just write

```
    table.sort(network, function (a,b)
      return (a.name > b.name)
    end)
```

ALGOL 68

```
PROC newton = (REAL x, error, PROC (REAL) REAL f, f prime) REAL:
# Newton's method #
  IF f(x) <= error
  THEN x
  ELSE newton (x - f(x) / f prime (x), error, f, f prime)
  FI;
```

```
print((
  newton(0.5, 0.001, (REAL x) REAL: x**3 - 2*x**2 + 6, (REAL x) REAL:
3*x**2 - 4**x),
  newline
));
```

ECMAScript

```
function(message) { print(message); } // function literal
SomeObject.SomeCallBack=function(message) { print(message); } //
function literal assigned to callback
```

Note that the *callee* keyword in ECMAScript makes it possible to write recursive functions without naming them.

Perl

```
my $divisor = 10;
my $checker = sub {
                my $val = shift;
                if ($val % $divisor ) {
                      return 0;
                } else {
                      return 1;
                }
         };
```

This illustrates not only first-class-object functions in Perl but also closures, an important language feature that first-class-object functions enable. It should also be noted that an interesting feature of Object Oriented Perl is that functions-as-objects do not have to be anonymous and it is entirely possible to bless a function into a named class.

Python

Python creates functions by the *def* statement or the *lambda* expression. Although *lambda* expressions cannot create functions that use statements in their definition as they are confined to be just expressions, they are flexible enough to show that they create first class functions as well as if the more powerful *def* statement were used:

```
>>> #some built in functions and their inverses
>>> from math import sin, cos, acos, asin
>>> # Add a user defined function and its inverse
>>> cube = lambda x: x * x * x
>>> croot = lambda x: x ** (1/3.0)
>>> # First class functions allow run-time creation of functions from
functions
>>> # return function compose(f,g)(x) == f(g(x))
>>> compose = lambda f1, f2: ( lambda x: f1(f2(x)) )
>>> # first class functions should be able to be members of collection
types
```

```
>>> funclist = [sin, cos, cube]
>>> funclisti = [asin, acos, croot]
>>> # Apply functions from lists as easily as integers
>>> [compose(inversef, f)(.5) for f, inversef in zip(funclist,
funclisti)]
[0.5, 0.49999999999999989, 0.5]
>>>
```

JavaScript

JavaScript supports both first class functions and lexical scope.

```
// f: function that takes a number and returns a number
// deltaX: small positive number
// returns a function that is an approximate derivative of f
function makeDerivative ( f, deltaX ) {
    return function (x) {
       return ( f(x + deltaX) - f(x) )/ deltaX;
    };
}
var cos = makeDerivative( Math.sin, 0.000001);
// cos(0)     ~> 1
// cos(Math.PI/2)  ~> 0
```

C#

C#, since version 3, supports anonymous functions and lambda expressions.

```
using System;
class Program
{
  // f: function that takes a double and returns a double
  // deltaX: small positive number
  // returns a function that is an approximate derivative of f
  static Func<double, double> MakeDerivative(Func<double, double> f,
double deltaX)
  {
    return x => (f(x + deltaX) - f(x)) / deltaX;
  }
  static void Main()
  {
    var cos = MakeDerivative(Math.Sin, 0.00000001);
    Console.WriteLine(cos(0));            // 1
    Console.WriteLine(cos(Math.PI / 2));  // 0
  }
}
```

Ruby 1.9

```
def deriv(f, dx)
  ->(x){ (f[x + dx] - f[x]) / dx }
end
```

```
sin = ->(x){ Math.sin(x) }

cos = deriv(sin, 0.00000001)

puts sin[Math::PI / 2]
puts cos[Math::PI / 2]
```

Scala

```
 import Math._

 def deriv(f:Double => Double, dx:Double) = (x:Double) => (f(x + dx) -
f(x)) / dx

 val cos = deriv(sin, 0.00000001)

 Console.println(sin(Pi / 2))
 Console.println(cos(Pi / 2))
```

Haskell

```
 derivative f delta = \x -> (f (x+delta) - f x) / delta

 main = do
        let sin' = derivative sin 0.00000001
        let x = pi / 3
        print (sin x)
        print (sin' x)
```

Purely functional

Purely functional is a term in computing used to describe algorithms, data structures or programming languages that exclude destructive modifications (updates). According to this restriction, variables are used in a mathematical sense, with identifiers referring to immutable, persistent values.

Benefits and applications

The persistence property of purely functional data structures can be advantageous in the development of many applications which deal with multiple versions of an object.

For example, consider a comprehensive web-based thesaurus service that uses a large red-black tree to store its list of synonym relationships, and that allows each user to add their own custom words to their personal thesaurus. One way to do this is to make a copy of the tree for each user, and then add their custom words to it; however, this duplication is wasteful, both of space and of time.

A better approach is to store the words in an immutable (and therefore purely functional) red-black tree. Then, one can simply take the original version and produce a new tree based on it for each set of custom words. Because these new trees share large amounts of structure with the main tree, the space overhead for each additional user is at most $2k\log_2 n$, where k is the number of custom nodes. With a mutable red-black tree, this approach would not work, since changes to the main tree would affect all users.

Besides their efficiency benefits, the inherent referential transparency of functional data structures tends to make purely functional computation more amenable to analysis and optimization, both formal and informal.

Examples of purely functional data structures

Linked lists

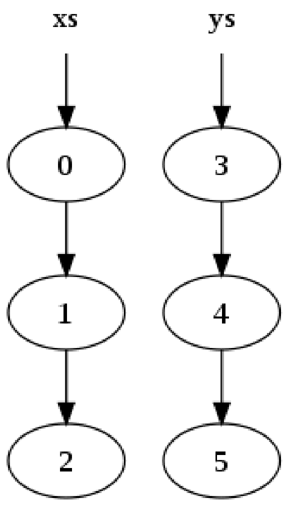
Singly linked lists are the bread and butter data structure in functional languages. In ML-derived languages and Haskell, they are purely functional because once a node in the list has been allocated, it cannot be modified, only copied or destroyed.

Note: According to the ML Programming Language page ML itself is **not** purely functional.

Consider the two lists:

```
xs = [0, 1, 2]
ys = [3, 4, 5]
```

These would be represented in memory by:

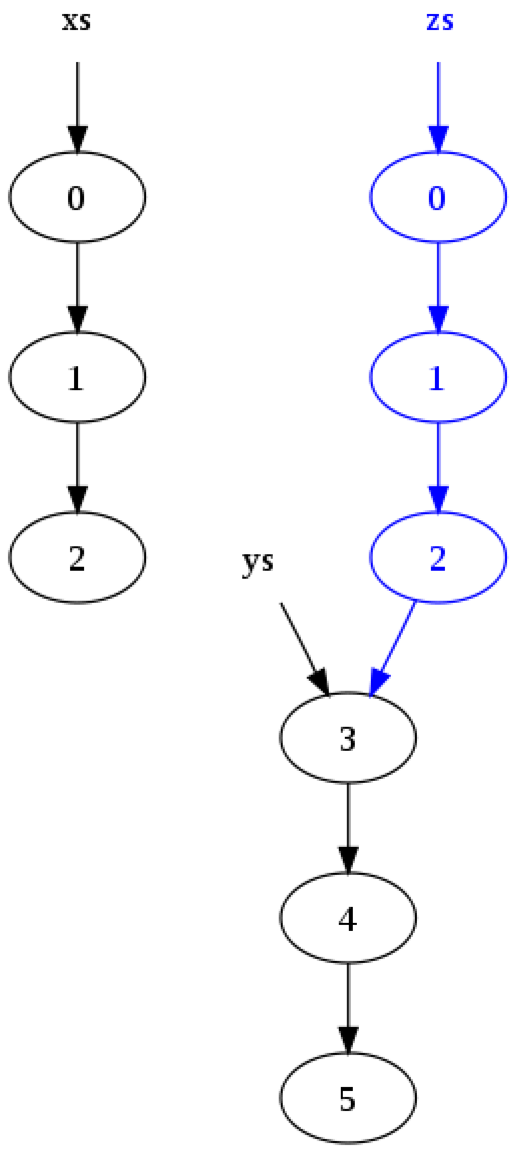


where a circle indicates a node in the list (the arrow out showing the second element of the node which is a pointer to another node).

Now concatenating the two lists:

```
zs = xs ++ ys
```

results in the following memory structure:



Notice that the nodes in list `xs` have been copied, but the nodes in `ys` are shared. As a result, the original lists (`xs` and `ys`) persist and have not been modified.

The reason for the copy is that the last node in `xs` (the node containing the original value `2`) cannot be modified to point to the start of `ys`, because that would change the value of `xs`.

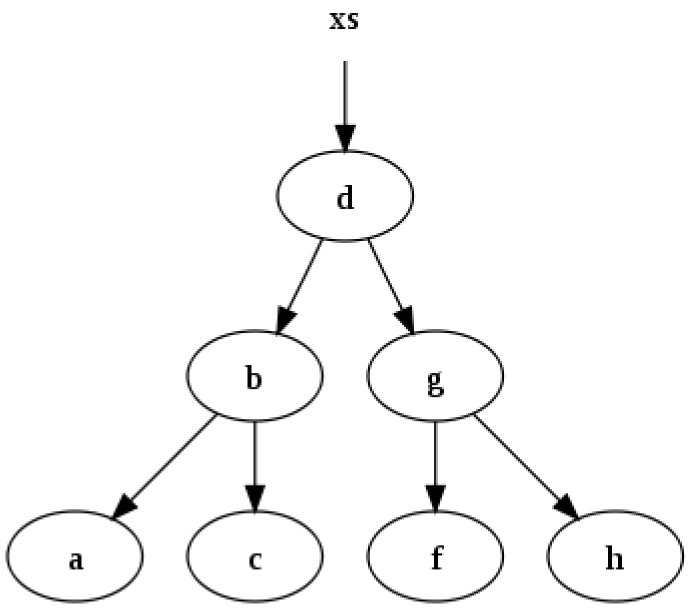
Trees

Consider a binary tree used for fast searching, where every node has the recursive invariant that subnodes on the left are less than the node, and subnodes on the right are greater than the node.

For instance, the set of data

```
xs = [a, b, c, d, f, g, h]
```

might be represented by the following binary search tree:



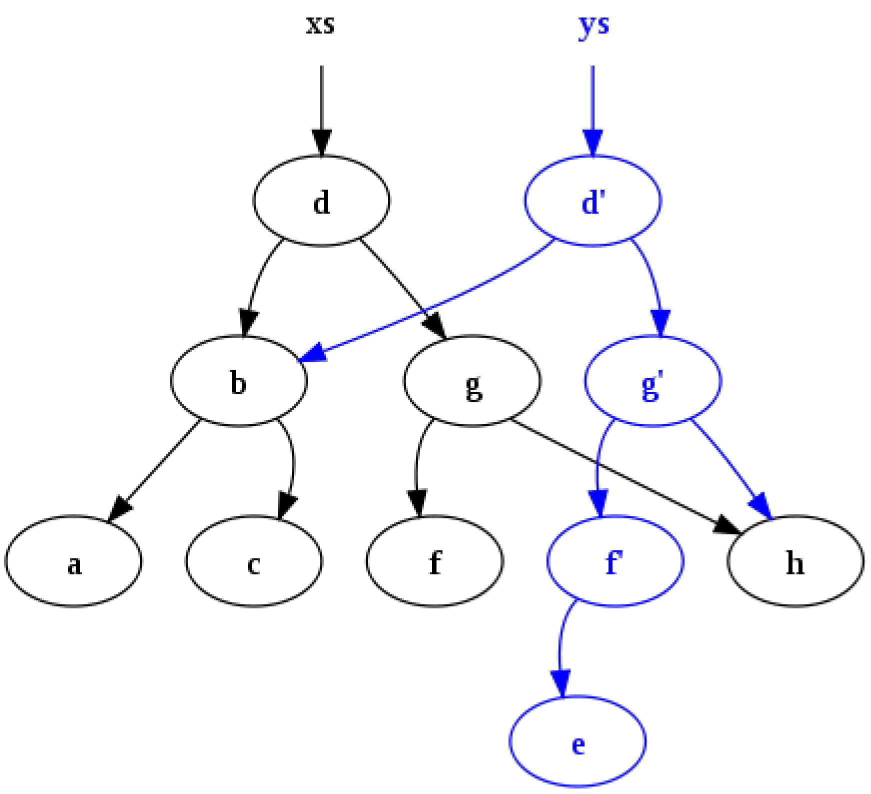
A function which inserts data into the binary tree and maintains the invariant is:

```
fun insert (x, E) = T (E, x, E)
  | insert (x, s as T (a, y, b)) =
       if x < y then T (insert (x, a), y, b)
       else if x > y then T (a, y, insert (x, b))
       else s
```

After executing

```
ys = insert ("e", xs)
```

we end up with the following:



Notice two points: Firstly the original tree (`xs`) persists. Secondly many common nodes are shared between the old tree and the new tree. Such persistence and sharing is difficult to manage without some form of garbage collection (GC) to automatically free up nodes which have no live references, and this is why GC is a feature commonly found in functional programming languages.

Reference cycles

Since every value in a purely functional computation is built up out of existing values, it would seem that it is impossible to create a cycle of references. In that case, the reference graph (the graph of the references from object to object) could only be a directed acyclic graph. However, in most functional languages, functions can be defined recursively; this capability allows recursive structures using functional suspensions. In lazy languages, such as Haskell, all data structures are represented as implicitly suspended thunks; in these languages any data structure can be recursive because a value can be defined in terms of itself. Some other languages, such as Objective Caml, allow the explicit definition of recursive values.

Chapter 3

Referential Transparency (Computer Science) and Continuation-Passing Style

Referential transparency (computer science)

Referential transparency and **referential opaqueness** are properties of parts of computer programs. An expression is said to be referentially transparent if it can be replaced with its value without changing the program (in other words, yielding a program that has the same effects and output on the same input). The opposite term is referentially opaque.

While in mathematics all function applications are referentially transparent, in programming this is not always the case. The importance of referential transparency is that it allows the programmer and the compiler to reason about program behavior. This can help in proving correctness, simplifying an algorithm, assisting in modifying code without breaking it, or optimizing code by means of memoization, common subexpression elimination or parallelization.

Referential transparency is one of the principles of functional programming; only referentially transparent functions can be memoized (transformed into equivalent functions which cache results). Some programming languages provide means to guarantee referential transparency. Some functional programming languages enforce referential transparency for all functions.

As referential transparency requires the same results for a given set of inputs at any point in time, a referentially transparent expression is therefore deterministic by definition.

Examples and counterexamples

If all functions involved in the expression are pure functions, then the expression is referentially transparent. Also, some impure functions can be included in the expression if their values are discarded and their side effects are insignificant.

Take a function that takes no parameters and returns input from the keyboard. A call to this function might be `GetInput()`. The return value of `GetInput()` depends on what the user types in, so multiple calls to `GetInput()` with identical parameters (the empty list) may return different results. Therefore, `GetInput()` is neither determined nor referentially transparent.

A more subtle example is that of a function that uses a global variable (or a dynamically scoped variable, or a lexical closure) to help it compute its results. Since this variable is not passed as a parameter but can be altered, the results of subsequent calls to the function can differ even if the parameters are identical. (In pure functional programming, destructive assignment is not allowed; thus a function that uses global (or dynamically scoped) variables is still referentially transparent, since these variables cannot change.)

Arithmetic operations are referentially transparent: `5*5` can be replaced by `25`, for instance. In fact, all functions in the mathematical sense are referentially transparent: `sin(x)` is transparent, since it will always give the same result for each particular `x`.

Assignments are not transparent. For instance, the C expression `x = x + 1` changes the value assigned to the variable `x`. Assuming `x` initially has value `10`, two consecutive evaluations of the expression yield, respectively, `11` and `12`. Clearly, replacing `x = x + 1` with either `11` or `12` gives a program with different meaning, and so the expression is not referentially transparent. However, calling a function such as `int plusone(int x) {return x+1;}` *is* transparent, as it will not implicitly change the input x and thus has no such side effects.

In most languages, `print( "Hello world" )` is not transparent, as replacing it by its value (11) changes the behavior of the program, as "Hello world" isn't printed.

`today()` is not transparent, as if you evaluate it and replace it by its value (say, "Jan 1, 2001"), you don't get the same result as you will if you run it tomorrow. This is because it depends on a state (the time).

Contrast to imperative programming

If the substitution of an expression with its value is valid only at a certain point in the execution of the program, then the expression is not referentially transparent. The definition and ordering of these sequence points are the theoretical foundation of imperative programming, and part of the semantics of an imperative programming language.

However, because a referentially transparent expression can be evaluated at any time, it is not necessary to define sequence points nor any guarantee of the order of evaluation at all. Programming done without these considerations is called purely functional programming.

The chief advantage of writing code in a referentially transparent style is that given an intelligent compiler, static code analysis is easier and better code-improving transformations are possible automatically. For example, when programming in C, there will be a performance penalty for including a call to an expensive function inside a loop, even if the function call could be moved outside of the loop without changing the results of the program. The programmer would be forced to perform manual code motion of the call, possibly at the expense of source code readability. However, if the compiler is able to determine that the function call is referentially transparent, it can perform this transformation automatically.

The primary disadvantage of languages which enforce referential transparency is that it makes the expression of operations that naturally fit a sequence-of-steps imperative programming style more awkward and less concise. Such languages often incorporate mechanisms to make these tasks easier while retaining the purely functional quality of the language, such as definite clause grammars and monads.

With referential transparency, no difference is made or recognized between a reference to a thing and the corresponding thing itself. Without referential transparency, such difference can be easily made and utilized in programs.

Another example

As an example, let's use two functions, one which is referentially opaque, and the other which is referentially transparent:

```
globalValue = 0;

integer function rq(integer x)
begin
  globalValue = globalValue + 1;
  return x + globalValue;
end

integer function rt(integer x)
begin
  return x + 1;
end
```

The function `rt` is referentially transparent, which means that `rt(x) = rt(y)` if `x = y`. For instance, `rt(6) = 6 + 1 = 7, rt(4) = 4 + 1 = 5`, and so on. However, we can't say any such thing for `rq` because it uses a global variable which it modifies.

The referential opacity of `rq` makes reasoning about programs more difficult. For example, say we wish to reason about the following statement:

```
integer p = rq(x) + rq(y) * (rq(x) - rq(x));
```

One may be tempted to simplify this statement to:

```
integer p = rq(x) + rq(y) * (0)
integer p = rq(x) + 0
integer p = rq(x);
```

However, this will not work for `rq()` because each occurrence of `rq(x)` evaluates to a different value. Remember, that the return value of `rq` is based on a global value which isn't passed in and which gets modified on each call to `rq`. This means that mathematical identities such as $x - x = 0$ no longer hold.

Such mathematical identities *will* hold for referentially-transparent functions such as `rt`.

Therefore, referential transparency allows us to reason about our code which will lead to more robust programs, the possibility of finding bugs that we couldn't hope to find by testing, and the possibility of seeing opportunities for optimization.

Continuation-passing style

In functional programming, **continuation-passing style** is a style of programming in which control is passed explicitly in the form of a continuation.

History

Gerald Jay Sussman and Guy L. Steele, Jr. coined the phrase "continuation-passing style" in AI Memo 349 (1975), defining the first version of the Scheme programming language.

Introduction

Instead of "returning" values as in the more familiar direct style, a function written in continuation-passing style (CPS) takes an explicit "continuation" argument, i.e. a function which is meant to receive the result of the computation performed within the original function. Similarly, when a subroutine is invoked within a CPS function, the calling function is required to supply a procedure to be invoked with the subroutine's "return" value. Expressing code in this form makes a number of things explicit which are implicit in direct style. These include: procedure returns, which become apparent as calls to a continuation; intermediate values, which are all given names; order of argument evaluation, which is made explicit; and tail calls, which is simply calling a procedure with the continuation that was passed to the caller.

Programs can be automatically transformed from direct style to CPS. Functional and logic compilers often use CPS as an intermediate representation where a compiler for an imperative or procedural programming language would use static single assignment form (SSA); however, SSA and CPS are equivalent [technically there are constructions in CPS that cannot be translated to SSA, but they do not occur in practice]. Functional compilers can also use Administrative Normal Form (ANF) instead of or in conjunction with CPS. CPS is used more frequently by compilers than by programmers as a local or global style.

Examples

In CPS, each procedure takes an extra argument representing what should be done with the result the function is calculating. This, along with a restrictive style prohibiting a variety of constructs usually available, is used to expose the semantics of programs, making them easier to analyze. This style also makes it easy to express unusual control structures, like catch/throw or other non-local transfers of control.

The key to CPS is to remember that (a) *every* function takes an extra argument, its continuation, and (b) every argument in a function call must be either a variable or a lambda expression (not a more complex expression). This has the effect of turning expressions "inside-out" because the innermost parts of the expression must be evaluated first. Some examples of code in direct style and the corresponding CPS style appear below. These examples are written in the Scheme programming language.

Direct style

```
(define (pyth x y)
 (sqrt (+ (* x x) (* y
y))))
```

```
(define (factorial n)
 (if (= n 0)
     1
     (* n (factorial (- n
1)))))
```

```
(define (factorial n) (f-
aux n 1))
(define (f-aux n a)
 (if (= n 0)
     a
     (f-aux (- n 1) (* n
a))))
```

Continuation passing style

```
(define (pyth x y k)
 (* x x (lambda (x2)
         (* y y (lambda (y2)
                 (+ x2 y2 (lambda (x2py2)
                           (sqrt x2py2
k))))))))
```

```
(define (factorial n k)
  (= n 0 (lambda (b)
          (if b
              (k 1)
              (- n 1 (lambda (nm1)
                      (factorial nm1
(lambda (f)
                                          (* n
f k)))))))))
```

```
(define (factorial n k)
 (f-aux n 1 k))
(define (f-aux n a k)
 (= n 0 (lambda (b)
         (if b
             (k a)
             (- n 1 (lambda (nm1)
                     (* n a (lambda (nta)
                             (f-aux nm1 nta
k)))))))))
```

Note that in the CPS versions, we are assuming that primitives like `+` and `*` are in CPS, not in direct style, so to make the above example work in a Scheme system we would need to write new CPS versions of these primitives and use them instead, for instance `cps*` instead of `*`, where `cps*` is defined by:

```
(define (cps* x y k)
 (k (* x y)))
```

To do this in general, we might write a conversion routine:

```
(define (cps-prim f)
 (lambda args
  (let ((r (reverse args)))
   ((car r) (apply f
             (reverse (cdr r)))))))
(define cps* (cps-prim *))
(define cps+ (cps-prim +))
```

In order to call a procedure written in CPS from a procedure written in direct style, it is necessary to provide a continuation. In the example above (assuming that CPS-style primitives have been provided), we might call `(factorial 10 identity)`.

There is some variety between compilers in the way primitive functions are provided in CPS. Above we have used the simplest convention, however sometimes boolean primitives take two continuations, so `(if (= a b) c d)` in direct style would be translated to `(= a b (lambda () (k c)) (lambda () (k d)))`. Similarly, sometimes `if` itself is not included in CPS, and instead a primitive function `%if` is provided which takes three arguments: a boolean condition and two continuations corresponding to the two arms of the conditional.

The translations shown above show that CPS is a global transformation; the direct-style *factorial* takes, as might be expected, a single argument. The CPS *factorial* takes two: the argument and a continuation. Any function calling a CPS-ed function must either provide a new continuation or pass its own; any calls from a CPS-ed function to a non-CPS function will use implicit continuations. Thus, to ensure the total absence of a function stack, the entire program must be in CPS.

Continuations as objects

Programming with continuations can also be useful when a caller does not want to wait until the callee completes. For example, in user-interface (UI) programming, a routine can set up dialog box fields and pass these, along with a continuation function, to the UI framework. This call returns right away, allowing the application code to continue while the user interacts with the dialog box. Once the user presses the "OK" button, the framework calls the continuation function with the updated fields. Although this style of coding uses continuations, it is not full CPS.

```
function Confirm_name()
{
    fields.name = name;
    framework.Show_dialog_box(fields, Confirm_name_continuation);
}

function Confirm_name_continuation(fields)
{
    name = fields.name;
}
```

A similar idea can be used when the function must run in a different thread or on a different processor. The framework can execute the called function in a worker thread, then call the continuation function in the original thread with the worker's results. This is in Java using the Swing UI framework:

```
void buttonHandler() {
    // This is executing in the Swing UI thread.
    // We can access UI widgets here to get query parameters.
    final int parameter = getField();

    new Thread(new Runnable() {
        public void run() {
            // Now we're in a separate thread.
            // We can do things like hit a database or access
            // a blocking resource like the network to get data.
            final int result = lookup(parameter);

            javax.swing.SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                    // Now we're back in the UI thread and can use
                    // the fetched data to fill in UI widgets.
                    setField(result);
                }
            });
        }
    }).start();
}
```

CPS and tail calls

Note that in CPS, there is no implicit continuation — every call is a tail call. There is no "magic" here, as the continuation is simply explicitly passed. Using CPS without tail call optimization (TCO) will cause not only the explicit continuation to grow during recursion, but also the function stack itself.

As CPS and TCO eliminate the concept of an implicit function return, their combined use can eliminate the need for a runtime stack. Several compilers and interpreters for functional programming languages use this ability in novel ways.

Use and implementation

Continuation passing style can be used to implement continuations and control flow operators in a functional language that does not feature first-class continuations but does have first-class functions. Without first-class functions, techniques such as trampolining of thunk closures can be used; in this case, it is possible to convert tail calls into gotos in a loop, eliminating even the need for TCO.

Writing code in CPS, while not impossible, is often error-prone. There are various translations, usually defined as one- or two-pass conversions of pure lambda calculus, which convert direct style expressions into CPS expressions. Writing in trampolined style, however, is extremely difficult; when used, it is usually the target of some sort of transformation, such as compilation.

Use in other fields

Outside of computer science, CPS is of more general interest as an alternative to the conventional method of composing simple expressions into complex expressions. For example, within linguistic semantics, Chris Barker and his collaborators have suggested that specifying the denotations of sentences using CPS might explain certain phenomena in natural language .

In mathematics, the Curry–Howard isomorphism between computer programs and mathematical proofs relates continuation-passing style translation to a variation of double-negation embeddings of classical logic into intuitionistic (constructive) logic. Unlike the regular double-negation translation, which maps atomic propositions p to $((p \to \bot) \to \bot)$, the continuation passing style replaces $\bot$ by the type of the final expression. Accordingly, the result is obtained by passing the identity function as a continuation to the CPS-style expression, as in the above example.

Classical logic itself relates to manipulating the continuation of programs directly, as in Scheme's call-with-current-continuation control operator.

Chapter 4

Anamorphism and Evaluation Strategy

Anamorphism

In functional programming, an **anamorphism** is a kind of generic function which can corecursively construct a result of a certain type and which is parameterized by functions that determine what the next single step of the construction is. The term comes from Greek ἀνά (upwards) + morphism (from Greek μορφή, or form, shape) and the concept has its grounding in category theory.

The concept of anamorphism generalizes the list-producing *unfold* functions to arbitrary algebraic data types that can be described as final coalgebras. In fact, the terms 'anamorphism' and "unfold" (as a noun) are often used interchangeably. Anamorphisms, which are co-recursive, are dual to catamorphisms, which are recursive, just as unfolds are dual to folds.

Examples

Anamorphisms on lists

An *unfold* on lists would build a (potentially infinite) list from a seed value. Typically, the unfold takes a seed value `x`, a one-place operation `unspool` that yields a pairs of such items, and a predicate `finished` which determines when to finish the list (if ever). In the action of unfold, the first application of `unspool`, to the seed `x`, would yield `unspool x = (y,z)`. The list defined by the unfold would then begin with `y` and be followed with the (potentially infinite) list that unfolds from the second term, `z`, with the same operations. So if `unspool z = (u,v)`, then the list will begin `y:u:...`, where `...` is the result of unfolding v with r, and so on.

A Haskell definition of an unfold, or anamorphism for lists, called `ana`, is as follows:

```
-- The type signature of 'ana':
ana :: (b->(a,b)) -> (b->Bool)-> b -> [a]
-- Its definition:
ana unspool finished x = if finished x
                         then []
```

```
                          else a : ana unspool finished y
                              where (a,y) = unspool x
```

(Here `finished` and `unspool` are parameters of the function, like `x`.) We can now implement quite general functions using *ana*.

Anamorphisms on other data structures

An anamorphism can be defined for any recursive type, according to a generic pattern. For example, the unfold for the tree data structure

```
data Tree a = Leaf a | Branch Tree a Tree
```

is as follows

```
ana :: (b->Either a (b,a,b)) -> b -> Tree a
ana unspool x = case unspool x of
                  Left a -> Leaf a
                  Right (l,x,r) -> Branch (ana unspool l) x (ana
unspool r)
```

History

One of the first publications to introduce the notion of an anamorphism in the context of programming was the paper *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire*, by Erik Meijer *et al.*, which was in the context of the Squiggol programming language.

Applications

Functions like `zip` and `iterate` are examples of anamorphisms. `zip` takes a pair of lists, say ['a','b','c'] and [1,2,3] and returns a list of pairs [('a',1),('b',2),('c',3)]. `Iterate` takes a thing, x, and a function, f, from such things to such things, and returns the infinite list that comes from repeated application of f, i.e. the list [x, (f x), (f (f x)), (f (f (f x))), ...].

```
zip (a:as) (b:bs) = if (as==[]) || (bs ==[])   -- || means 'or'
                     then [(a,b)]
                     else (a,b):(zip as bs)

iterate f x = x:(iterate f (fx))
```

To prove this, we can implement both using our generic unfold, `ana`, using a simple recursive routine:

```
zip2 = ana g p
   where
   p (as,bs) = (as==[]) || (bs ==[])
   g ((a:as), (b:bs)) = ((a,b),(as,bs))
iterate2 f = ana (\a->(a,f a)) (\x->False)
```

In a language like Haskell, even the abstract functions `fold`, `unfold` and `ana` are merely defined terms, as we have seen from the definitions given above.

Anamorphisms in category theory

In category theory, anamorphisms are the categorical dual of catamorphisms (and catamorphisms are the categorical dual of anamorphisms).

That means the following. Suppose (*A*, *fin*) is a final *F*-coalgebra for some endofunctor *F* of some category into itself. Thus, *fin* is a morphism from *A* to *FA*, and since it is assumed to be final we know that whenever (*X*, *f*) is another *F*-coalgebra (a morphism *f* from *X* to *FX*), there will be a unique homomorphism *h* from (*X*, *f*) to (*A*, *fin*), that is a morphism *h* from *X* to *A* such that *fin* **.** *h* = *Fh* **.** *f*. Then for each such *f* we denote by **ana *f*** that uniquely specified morphism *h*.

In other words, we have the following defining relationship, given some fixed *F*, *A*, and *fin* as above:

- $h = \mathrm{ana}\ f$
- $fin \circ h = Fh \circ f$

Notation

A notation for ana *f* found in the literature is $[\!(f)\!]$. The brackets used are known as **lens brackets**, after which anamorphisms are sometimes referred to as *lenses*.

Evaluation strategy

In computer science, an **evaluation strategy** is a set of (usually deterministic) rules for evaluating expressions in a programming language. Emphasis is typically placed on functions or operators: an evaluation strategy defines when and in what order the arguments to a function are evaluated, when they are substituted into the function, and what form that substitution takes. The lambda calculus, a formal system for the study of functions, has often been used to model evaluation strategies, where they are usually called **reduction strategies**. Evaluation strategies divide into two basic groups, strict and non-strict, based on how arguments to a function are handled. A language may combine several evaluation strategies; for example, C++ combines call-by-value with call-by-reference. Most languages that are predominantly strict use some form of non-strict evaluation for boolean expressions and if-statements.

Strict evaluation

In *strict evaluation,* the arguments to a function are always evaluated completely before the function is applied.

Under Church encoding, eager evaluation of operators maps to strict evaluation of functions; for this reason, strict evaluation is sometimes called "eager". Most existing programming languages use strict evaluation for functions.

Applicative order

Applicative order (or *leftmost innermost*) evaluation refers to an evaluation strategy in which the arguments of a function are evaluated from left to right in a post-order traversal of reducible expressions (redexes). Unlike call-by-value, applicative order evaluation reduces terms within a function body as much as possible before the function is applied.

Call by value

Call-by-value evaluation (also referred to as *pass-by-value*) is the most common evaluation strategy, used in languages as different as C and Scheme. In call-by-value, the argument expression is evaluated, and the resulting value is bound to the corresponding variable in the function (frequently by copying the value into a new memory region). If the function or procedure is able to assign values to its parameters, only its local copy is assigned — that is, anything passed into a function call is unchanged in the caller's scope when the function returns.

Call-by-value is not a single evaluation strategy, but rather the family of evaluation strategies in which a function's argument is evaluated before being passed to the function. While many programming languages (such as Eiffel and Java) that use call-by-value evaluate function arguments left-to-right, some evaluate functions and their arguments right-to-left, and others (such as Scheme, OCaml and C) leave the order unspecified (though they generally require implementations to be consistent).

In some cases, the term "call-by-value" is problematic, as the value which is passed is not the value of the variable as understood by the ordinary meaning of value, but an implementation-specific reference to the value. The description "call-by-value where the value is a reference" is common (but should not be understood as being call-by-reference); another term is call-by-sharing. Thus the behaviour of call-by-value Java or Visual Basic and call-by-value C or Pascal are significantly different: in C or Pascal, calling a function with a large structure as an argument will cause the entire structure to be copied, potentially causing serious performance degradation, and mutations to the structure are invisible to the caller. However, in Java or Visual Basic only the reference to the structure is copied, which is fast, and mutations to the structure are visible to the caller.

Call by reference

In *call-by-reference* evaluation (also referred to as *pass-by-reference*), a function receives an implicit reference to the argument, rather than a copy of its value. This typically means that the function can modify the argument- something that will be seen by its caller. Call-by-reference therefore has the advantage of greater time- and space-efficiency (since arguments do not need to be copied), as well as the potential for greater communication between a function and its caller (since the function can return information using its reference arguments), but the disadvantage that a function must often take special steps to "protect" values it wishes to pass to other functions.

Many languages support call-by-reference in some form or another, but comparatively few use it as a default; Perl and Visual Basic are two that do, though Visual Basic also offers a special syntax for call-by-value parameters. A few languages, such as C++ and REALbasic, default to call-by-value, but offer special syntax for call-by-reference parameters. C++ additionally offers call-by-reference-to-const. In purely functional languages there is typically no semantic difference between the two strategies (since their data structures are immutable, so there is no possibility for a function to modify any of its arguments), so they are typically described as call-by-value even though implementations frequently use call-by-reference internally for the efficiency benefits.

Even among languages that don't exactly support call-by-reference, many, including C and ML, support explicit references (objects that refer to other objects), such as pointers (objects representing the memory addresses of other objects), and these can be used to effect or simulate call-by-reference (but with the complication that a function's caller must explicitly generate the reference to supply as an argument).

Call by sharing

Also known as "call by object" or "call by object-sharing" is an evaluation strategy first named by Barbara Liskov et al. for the language CLU in 1974. It is used by languages such as Python, Iota, Java (for object references), Ruby, Scheme, OCaml, AppleScript, and many other languages. However, the term "call by sharing" is not in common use; the terminology is inconsistent across different sources. For example, in the Java community, they say that Java is pass-by-value, whereas in the Ruby community, they say that Ruby is pass-by-reference, even though the two languages exhibit the same semantics. Call-by-sharing implies that values in the language are based on objects rather than primitive types.

The semantics of call-by-sharing differ from call-by-reference in that assignments to function arguments within the function aren't visible to the caller (unlike by-reference semantics). However since the function has access to the same object as the caller (no copy is made), mutations to those objects within the function are visible to the caller, which differs from call-by-value semantics.

Although this term has widespread usage in the Python community, identical semantics in other languages such as Java and Visual Basic are often described as call-by-value, where the value is implied to be a reference to the object.

Call by copy-restore

Call-by-copy-restore, *call-by-value-result* or *call-by-value-return* (as termed in the Fortran community) is a special case of call-by-reference where the provided reference is unique to the caller. If a parameter to a function call is a reference that might be accessible by another thread of execution, its contents are copied to a new reference that is not; when the function call returns, the updated contents of this new reference are copied back to the original reference ("restored").

The semantics of call-by-copy-restore also differ from those of call-by-reference where two or more function arguments alias one another; that is, point to the same variable in the caller's environment. Under call-by-reference, writing to one will affect the other; call-by-copy-restore avoids this by giving the function distinct copies, but leaves the result in the caller's environment undefined (depending on which of the aliased arguments is copied back first).

When the reference is passed to the callee uninitialized, this evaluation strategy may be called *call-by-result*.

Partial evaluation

In *partial evaluation*, evaluation may continue into the body of a function that has not been applied. Any sub-expressions that do not contain unbound variables are evaluated, and function applications whose argument values are known may be reduced. In the presence of side-effects, complete partial evaluation may produce unintended results; for this reason, systems that support partial evaluation tend to do so only for "pure" expressions (expressions without side-effects) within functions.

Non-strict evaluation

In *non-strict evaluation,* arguments to a function are not evaluated unless they are actually used in the evaluation of the function body.

Under Church encoding, lazy evaluation of operators maps to non-strict evaluation of functions; for this reason, non-strict evaluation is often referred to as "lazy". Boolean expressions in many languages use lazy evaluation; in this context it is often called short circuiting. Conditional expressions also usually use lazy evaluation, albeit for different reasons.

Normal order

Normal-order (or *leftmost outermost*) evaluation is the evaluation strategy where the outermost redex is always reduced, applying functions before evaluating function arguments. It differs from call-by-name in that call-by-name does not evaluate inside the body of an unapplied function.

Call by name

In *call-by-name* evaluation, the arguments to functions are not evaluated at all — rather, function arguments are substituted directly into the function body using capture-avoiding substitution. If the argument is not used in the evaluation of the function, it is never evaluated; if the argument is used several times, it is re-evaluated each time.

Call-by-name evaluation can be preferable over call-by-value evaluation because call-by-name evaluation always yields a value when a value exists, whereas call-by-value may not terminate if the function's argument is a non-terminating computation that is not needed to evaluate the function. Opponents of call-by-name claim that it is significantly slower when the function argument is used, and that in practice this is almost always the case as a mechanism such as a thunk is needed.

Call by need

Call-by-need is a memoized version of call-by-name where, *if the function argument is evaluated,* that value is stored for subsequent uses. In a "pure" (effect-free) setting, this produces the same results as call-by-name; when the function argument is used two or more times, call-by-need is almost always faster.

Because evaluation of expressions may happen arbitrarily far into a computation, languages using call-by-need generally do not support computational effects (such as mutation) except through the use of monads and uniqueness types. This eliminates any unexpected behavior from variables whose values change prior to their delayed evaluation.

This is a kind of Lazy evaluation.

Haskell is the most well-known language that uses call-by-need evaluation.

R also uses a form of call-by-need.

Call by macro expansion

Call-by-macro-expansion is similar to call-by-name, but uses textual substitution rather than capture-avoiding substitution. With uncautious use, macro substitution may result in variable capture and lead to undesired behavior. Hygienic macros avoid this problem by checking for and replacing shadowed variables that are not parameters.

Nondeterministic strategies

Full β-reduction

Under *full β-reduction,* any function application may be reduced (substituting the function's argument into the function using capture-avoiding substitution) at any time. This may be done even within the body of an unapplied function.

Call by future

Call-by-future (or *parallel call-by-name*) is like call-by-need, except that the function's argument may be evaluated in parallel with the function body (rather than only if used). The two threads of execution synchronize when the argument is needed in the evaluation of the function body; if the argument is never used, the argument thread may be killed.

Optimistic evaluation

Optimistic evaluation is another variant of call-by-need in which the function's argument is partially evaluated for some amount of time (which may be adjusted at runtime), after which evaluation is aborted and the function is applied using call-by-need. This approach avoids some of the runtime expense of call-by-need, while still retaining the desired termination characteristics.

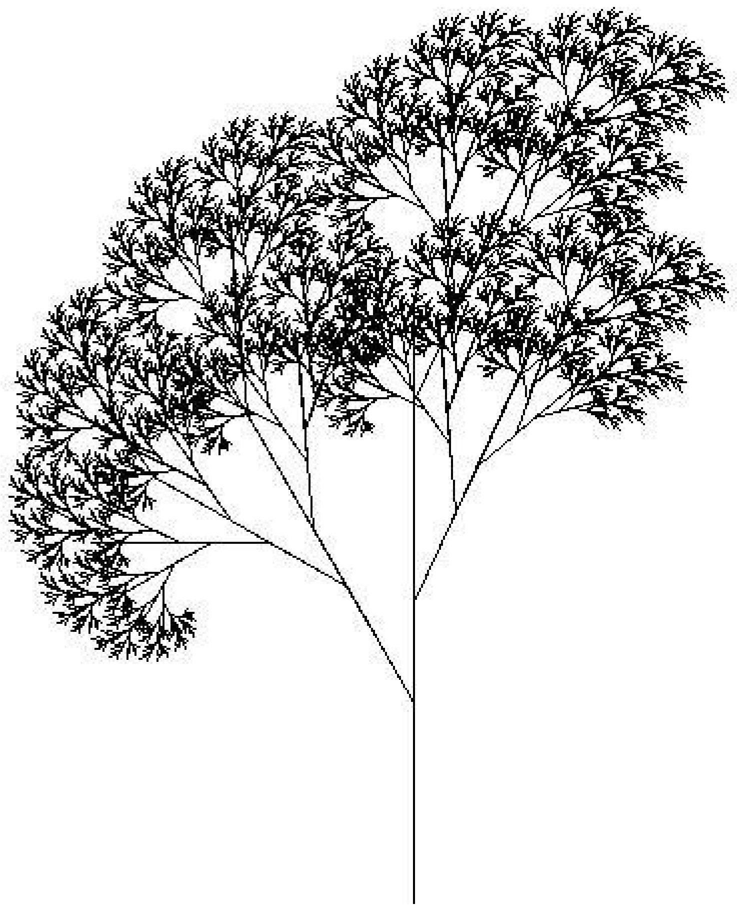
Chapter 5

Recursion (Computer Science)

Recursion in computer science is a method where the solution to a problem depends on solutions to smaller instances of the same problem. The approach can be applied to many types of problems, and is one of the central ideas of computer science.

"The power of recursion evidently lies in the possibility of defining an infinite set of objects by a finite statement. In the same manner, an infinite number of computations can be described by a finite recursive program, even if this program contains no explicit repetitions."

Most high-level computer programming languages support recursion by allowing a function to call itself within the program text. Some functional programming languages do not define any looping constructs but rely solely on recursion to repeatedly call code. Computability theory has proven that these recursive-only languages are mathematically equivalent to the imperative languages, meaning they can solve the same kinds of problems even without the typical control structures like “while” and “for”.



Tree created using the Logo programming language and relying heavily on recursion

Recursive data types

Many computer programs must process or generate an arbitrarily large quantity of data. Recursion is one technique for representing data whose exact size the programmer does not know: the programmer can specify this data with a self-referential definition. There are two flavors of self-referential definitions: inductive and coinductive definitions.

Inductively-defined data

An inductively-defined recursive data definition is one that specifies how to construct instances of the data. For example, linked lists can be defined inductively (here, using Haskell syntax):

```
data ListOfStrings = EmptyList | Cons String ListOfStrings
```

The code above specifies a list of strings to be either empty, or a structure that contains a string and a list of strings. The self-reference in the definition permits the construction of lists of any (finite) number of strings.

Another example of inductive definition is the natural numbers (or non-negative integers):

```
A natural number is either 1 or n+1, where n is a natural number.
```

Similarly recursive definitions are often used to model the structure of expressions and statements in programming languages. Language designers often express grammars in a syntax such as Backus-Naur form; here is such a grammar, for a simple language of arithmetic expressions with multiplication and addition:

```
 <expr> ::= <number>
          | (<expr> * <expr>)
          | (<expr> + <expr>)
```

This says that an expression is either a number, a product of two expressions, or a sum of two expressions. By recursively referring to expressions in the second and third lines, the grammar permits arbitrarily complex arithmetic expressions such as `(5 * ((3 * 6) + 8))`, with more than one product or sum operation in a single expression.

Coinductively-defined data and corecursion

A coinductive data definition is one that specifies the operations that may be performed on a piece of data; typically, self-referential coinductive definitions are used for data structures of infinite size.

A coinductive definition of infinite streams of strings, given informally, might look like this:

```
A stream of strings is an object s such that
 head(s) is a string, and
 tail(s) is a stream of strings.
```

This is very similar to an inductive definition of lists of strings; the difference is that this definition specifies how to access the contents of the data structure—namely, via the accessor functions `head` and `tail` -- and what those contents may be, whereas the inductive definition specifies how to create the structure and what it may be created from.

Corecursion is related to coinduction, and can be used to compute particular instances of (possibly) infinite objects. As a programming technique, it is used most often in the context of lazy programming languages, and can be preferable to recursion when the desired size or precision of a program's output is unknown. In such cases the program requires both a definition for an infinitely large (or infinitely precise) result, and a mechanism for taking a finite portion of that result. The problem of computing the first n prime numbers is one that can be solved with a corecursive program.

Recursive algorithms

A common computer programming tactic is to divide a problem into sub-problems of the same type as the original, solve those problems, and combine the results. This is often referred to as the divide-and-conquer method; when combined with a lookup table that stores the results of solving sub-problems (to avoid solving them repeatedly and incurring extra computation time), it can be referred to as dynamic programming or memoization.

A recursive function definition has one or more *base cases*, meaning input(s) for which the function produces a result trivially (without recurring), and one or more *recursive cases*, meaning input(s) for which the program recurs (calls itself). For example, the factorial function can be defined recursively by the equations $0! = 1$ and, for all $n > 0$, $n! = n(n - 1)!$. Neither equation by itself constitutes a complete definition; the first is the base case, and the second is the recursive case.

The job of the recursive cases can be seen as breaking down complex inputs into simpler ones. In a properly-designed recursive function, with each recursive call, the input problem must be simplified in such a way that eventually the base case must be reached. (Functions that are not intended to terminate under normal circumstances—for example, some system and server processes -- are an exception to this.) Neglecting to write a base case, or testing for it incorrectly, can cause an infinite loop.

For some functions (such as one that computes the series for e = 1+1/1!+1/2!+1/3!...) there is not an obvious base case implied by the input data; for these one may add a parameter (such as the number of terms to be added, in our series example) to provide a 'stopping criterion' that establishes the base case.

Structural versus generative recursion

Some authors classify recursion as either "generative" or "structural". The distinction is related to where a recursive procedure gets the data that it works on, and how it processes that data:

[Functions that consume structured data] typically decompose their arguments into their immediate structural components and then process those components. If one of the immediate components belongs to the same class of data as the input, the function is recursive. For that reason, we refer to these functions as (STRUCTURALLY) RECURSIVE FUNCTIONS.

Thus the defining characteristic of a structurally recursive function is that the argument to each recursive call is the contents of a field of the original input. Generative recursion is the alternative:

Many well-known recursive algorithms generate an entirely new piece of data from the given data and recur on it. HTDP (How To Design Programs) refers to this kind as generative recursion. Examples of generative recursion include: gcd, quicksort, binary search, mergesort, Newton's method, fractals, and adaptive integration.

This distinction is important in proving termination of a function. All structurally-recursive functions on finite (inductively-defined) data structures can easily be shown to terminate, via structural induction: intuitively, each recursive call receives a smaller piece of input data, until a base case is reached. Generatively-recursive functions, in contrast, do not necessarily feed smaller input to their recursive calls, so proof of their termination is not necessarily as simple, and avoiding infinite loops requires greater care.

Recursive programs

Recursive procedures

Factorial

A classic example of a recursive procedure is the function used to calculate the factorial of a natural number:

$$\operatorname{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \operatorname{fact}(n-1) & \text{if } n > 0 \end{cases}$$

Pseudocode (recursive):

```
function factorial is:
input: integer $n$ such that $n >= 0$
output: [$n × (n-1) × (n-2) × … × 1$]

    1. if $n$ is 0, return 1
    2. otherwise, return [ $n × \text{factorial}(n-1)$ ]

end factorial
```

The function can also be written as a recurrence relation:

$$b_n = nb_{n-1}$$
$$b_0 = 1$$

This evaluation of the recurrence relation demonstrates the computation that would be performed in evaluating the pseudocode above:

Computing the recurrence relation for n = 4:

```
$b_4$            = 4 * $b_3$
                 = 4 * 3 * $b_2$
                 = 4 * 3 * 2 * $b_1$
                 = 4 * 3 * 2 * 1 * $b_0$
```

```
                = 4 * 3 * 2 * 1 * 1
                = 4 * 3 * 2 * 1
                = 4 * 3 * 2
                = 4 * 6
                = 24
```

This factorial function can also be described without using recursion by making use of the typical looping constructs found in imperative programming languages:

Pseudocode (iterative):

```
function factorial is:
input: integer n such that n >= 0
output: [n × (n-1) × (n-2) × … × 1]

    1. create new variable called running_total with a value of 1

    2. begin loop
          1. if n is 0, exit loop
          2. set running_total to (running_total × n)
          3. decrement n
          4. repeat loop

    3. return running_total

end factorial
```

The imperative code above is equivalent to this mathematical definition using an accumulator variable *t:*

$$\begin{aligned}\operatorname{fact}(n) &= \operatorname{fact}_{\mathrm{acc}}(n,1)\\ \operatorname{fact}_{\mathrm{acc}}(n,t) &= \begin{cases} t & \text{if } n=0\\ \operatorname{fact}_{\mathrm{acc}}(n-1,nt) & \text{if } n>0\end{cases}\end{aligned}$$

The definition above translates straightforwardly to functional programming languages such as Scheme; this is an example of iteration implemented recursively.

Fibonacci

Another well known mathematical recursive function is one that computes the Fibonacci numbers:

$$\operatorname{fib}(n)=\begin{cases}0 & \text{if } n=0\\ 1 & \text{if } n=1\\ \operatorname{fib}(n-1)+\operatorname{fib}(n-2) & \text{if } n>=2\end{cases}$$

Pseudocode

```
function fib is:
input: integer n such that n >= 0
```

```
    1. if n is 0, return 0
    2. if n is 1, return 1
    3. otherwise, return [ fib(n-1) + fib(n-2) ]

end fib
```

Java language implementation:

```
    /**
     * Recursively calculate the kth Fibonacci number.
     *
     * @param k indicates which (positive) Fibonacci number to compute.
     * @return the kth Fibonacci number.
     */
    private static int fib(int k) {

        // Base Cases:
        //   If k == 0 then fib(k) = 0.
        //   If k == 1 then fib(k) = 1.
        if (k < 2) {
            return k;
        }
        // Recursive Case:
        //   If k >= 2 then fib(k) = fib(k-1) + fib(k-2).
        else {
            return fib(k-1) + fib(k-2);
        }
    }
```

Recurrence relation for Fibonacci:
$b_n = b_{n-1} + b_{n-2}$
$b_1 = 1, b_0 = 0$

Computing the recurrence relation for n = 4:

$$\begin{aligned} b_4 &= b_3 + b_2 \\ &= b_2 + b_1 + b_1 + b_0 \\ &= b_1 + b_0 + 1 + 1 + 0 \\ &= 1 + 0 + 1 + 1 + 0 \\ &= 3 \end{aligned}$$

This Fibonacci algorithm is a particularly poor example of recursion, because each time the function is executed on a number greater than one, it makes two function calls to itself, leading to an exponential number of calls (and thus exponential time complexity) in total. The following alternative approach uses two accumulator variables `TwoBack` and `OneBack` to "remember" the previous two Fibonacci numbers constructed, and so avoids the exponential time cost:

Pseudocode

```
function fib is:
input: integer Times such that Times >= 0, relative to TwoBack and
```

```
OneBack
             long TwoBack such that TwoBack = fib(x)
             long OneBack such that OneBack = fib(x)

    1. if Times is 0, return TwoBack
    2. if Times is 1, return OneBack
    3. if Times is 2, return TwoBack + OneBack
    4. otherwise, return [ fib(Times-1, OneBack, TwoBack + OneBack) ]

end fib
```

To obtain the tenth number in the Fib. sequence, one must perform Fib(10,0,1). Where 0 is considered TwoNumbers back and 1 is considered OneNumber back. As can be seen in this approach, no trees are being created, therefore the efficiency is much greater, being a linear recursion. The recursion in condition 4, shows that OneNumber back becomes TwoNumbers back, and the new OneNumber back is calculated, simply decrementing the Times on each recursion.

Implemented in the Java or the C# programming language:

```
public static long fibonacciOf(int times, long twoNumbersBack, long
oneNumberBack) {

    if (times == 0) {                            // Used only for
fibonacciOf(0, 0, 1)
        return twoNumbersBack;
    } else if (times == 1) {                     // Used only for
fibonacciOf(1, 0, 1)
        return oneNumberBack;
    } else if (times == 2) {                     // When the 0 and 1
clauses are included,
        return oneNumberBack + twoNumbersBack;   // this clause merely
stops one additional
    } else {                                     // recursion from
occurring
        return fibonacciOf(times - 1, oneNumberBack, oneNumberBack +
twoNumbersBack);
    }
}
```

Greatest common divisor

Another famous recursive function is the Euclidean algorithm, used to compute the greatest common divisor of two integers. Function definition*:*

$$\gcd(x,y) = \begin{cases} x & \text{if } y = 0 \\ \gcd(y, \text{remainder}(x,y)) & \text{if } x \geq y \text{ and } y > 0 \end{cases}$$

Pseudocode (recursive):

```
function gcd is:
input: integer x, integer y such that x >= y and y >= 0
```

```
    1. if y is 0, return x
    2. otherwise, return [ gcd( y, (remainder of x/y) ) ]

end gcd
```

Recurrence relation for greatest common divisor, where $x\%y$ expresses the remainder of x / y:

$$\gcd(x,y) = \gcd(y,x\%y)$$
$$\gcd(x,0) = x$$

Computing the recurrence relation for x = 27 and y = 9:

```
gcd(27, 9)    = gcd(9, 27 % 9)
              = gcd(9, 0)
              = 9
```

Computing the recurrence relation for x = 259 and y = 111:

```
gcd(259, 111)    = gcd(111, 259 % 111)
                 = gcd(111, 37)
                 = gcd(37, 0)
                 = 37
```

The recursive program above is tail-recursive; it is equivalent to an iterative algorithm, and the computation shown above shows the steps of evaluation that would be performed by a language that eliminates tail calls. Below is a version of the same algorithm using explicit iteration, suitable for a language that does not eliminate tail calls. By maintaining its state entirely in the variables *x* and *y* and using a looping construct, the program avoids making recursive calls and growing the call stack.

Pseudocode (iterative):

```
function gcd is:
input: integer x, integer y such that x >= y and y >= 0

    1. create new variable called remainder

    2. begin loop
          1. if y is zero, exit loop
          2. set remainder to the remainder of x/y
          3. set x to y
          4. set y to remainder
          5. repeat loop

    3. return x

end gcd
```

The iterative algorithm requires a temporary variable, and even given knowledge of the Euclidean algorithm it is more difficult to understand the process by simple inspection, although the two algorithms are very similar in their steps.

Towers of Hanoi

Simply put the problem is this: given three pegs, one with a set of N disks of increasing size, determine the minimum (optimal) number of steps it takes to move all the disks from their initial position to another peg without placing a larger disk on top of a smaller one.

Function definition:

$$\text{hanoi}(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2 \cdot \text{hanoi}(n-1) + 1 & \text{if } n > 1 \end{cases}$$

Recurrence relation for hanoi:

$h_n = 2h_{n-1} + 1$
$h_1 = 1$

Computing the recurrence relation for n = 4:

```
hanoi(4)     = 2*hanoi(3) + 1
             = 2*(2*hanoi(2) + 1) + 1
             = 2*(2*(2*hanoi(1) + 1) + 1) + 1
             = 2*(2*(2*1 + 1) + 1) + 1
             = 2*(2*(3) + 1) + 1
             = 2*(7) + 1
             = 15
```

Example Implementations:

Pseudocode (recursive):

```
function hanoi is:
input: integer n, such that n >= 1

    1. if n is 1 then return 1

    2. return [2 * [call hanoi(n-1)] + 1]

end hanoi
```

Although not all recursive functions have an explicit solution, the Tower of Hanoi sequence can be reduced to an explicit formula.

An explicit formula for Towers of Hanoi:

$h_1 = 1 \quad = 2^1 - 1$
$h_2 = 3 \quad = 2^2 - 1$
$h_3 = 7 \quad = 2^3 - 1$
$h_4 = 15 \quad = 2^4 - 1$

$h_5 = 31 = 2^5 - 1$
$h_6 = 63 = 2^6 - 1$
$h_7 = 127 = 2^7 - 1$
In general:
$h_n = 2^n - 1$, for all $n >= 1$

Binary search

The binary search algorithm is a method of searching an ordered array for a single element by cutting the array in half with each pass. The trick is to pick a midpoint near the center of the array, compare the data at that point with the data being searched and then responding to one of three possible conditions: the data is found, the data at the midpoint is greater than the data being searched for, or the data at the midpoint is less than the data being searched for.

Recursion is used in this algorithm because with each pass a new array is created by cutting the old one in half. The binary search procedure is then called recursively, this time on the new (and smaller) array. Typically the array's size is adjusted by manipulating a beginning and ending index. The algorithm exhibits a logarithmic order of growth because it essentially divides the problem domain in half with each pass.

Example Implementation of Binary Search in C:

```
/*
 Call binary_search with proper initial conditions.

 INPUT:
   data is a array of integers SORTED in ASCENDING order,
   toFind is the integer to search for,
   count is the total number of elements in the array

 OUTPUT:
   result of binary_search

*/
int search(int *data, int toFind, int count)
{
   //  Start = 0 (beginning index)
   //  End = count - 1 (top index)
   return binary_search(data, toFind, 0, count-1);
}

/*
  Binary Search Algorithm.

  INPUT:
       data is a array of integers SORTED in ASCENDING order,
       toFind is the integer to search for,
       start is the minimum array index,
       end is the maximum array index
  OUTPUT:
       position of the integer toFind within array data,
       -1 if not found
 */
 int binary_search(int *data, int toFind, int start, int end)
 {
    //Get the midpoint.
    int mid = start + (end - start)/2;   //Integer division

    //Stop condition.
    if (start > end)
       return -1;
    else if (data[mid] == toFind)        //Found?
       return mid;
    else if (data[mid] > toFind)         //Data is greater than toFind,
search lower half
       return binary_search(data, toFind, start, mid-1);
    else                                 //Data is less than toFind,
search upper half
       return binary_search(data, toFind, mid+1, end);
 }
```

Recursive data structures (structural recursion)

An important application of recursion in computer science is in defining dynamic data structures such as Lists and Trees. Recursive data structures can dynamically grow to a theoretically infinite size in response to runtime requirements; in contrast, a static array's size requirements must be set at compile time.

"Recursive algorithms are particularly appropriate when the underlying problem or the data to be treated are defined in recursive terms."

The examples in this section illustrate what is known as "structural recursion". This term refers to the fact that the recursive procedures are acting on data that is defined recursively.

As long as a programmer derives the template from a data definition, functions employ structural recursion. That is, the recursions in a function's body consume some immediate piece of a given compound value.

Linked lists

Below is a simple definition of a linked list node. Notice especially how the node is defined in terms of itself. The "next" element of struct node is a pointer to a struct node.

```
struct node
{
  int n;              // some data
  struct node *next;  // pointer to another struct node
};

// LIST is simply a synonym for struct node * (aka syntactic sugar).
typedef struct node *LIST;
```

Procedures that operate on the LIST data structure can be implemented naturally as a recursive procedure because the data structure it operates on (LIST) is defined recursively. The printList procedure defined below walks down the list until the list is empty (NULL), for each node it prints the data element (an integer). In the C implementation, the list remains unchanged by the printList procedure.

```
void printList(LIST lst)
{
    if (!isEmpty(lst))         // base case
    {
       printf("%d ", lst->n);  // print integer followed by a space
       printList(lst->next);   // recursive call
    }
}
```

Binary trees

Below is a simple definition for a binary tree node. Like the node for Linked Lists, it is defined in terms of itself (recursively). There are two self-referential pointers - left (pointing to the left sub-tree) and right (pointing to the right sub-tree).

```
struct node
{
  int n;               // some data
  struct node *left;   // pointer to the left subtree
  struct node *right;  // point to the right subtree
};

typedef struct node *TREE; // TREE is a synonym for struct node *
```

Operations on the tree can be implemented using recursion. Note that because there are two self-referencing pointers (left and right), tree operations may require two recursive calls:

```
// Test if t contains i; return 1 if so, 0 if not.
int contains(TREE t, int i) {
        if (isEmpty(t))
                return 0;  // base case
        else if (t->n == i)
                return 1;
        else
                return contains(t->left, i) || contains(t->right, i);
}
```

At most two recursive calls will be made for any given call to *contains* as defined above.

```
// Inorder traversal:
void printTree(TREE t) {
        if (!isEmpty(t)) {            // base case
                printTree(t->left);   // go left
                printf("%d ", t->n);  // print the integer followed by
a space
```

```
                printTree(t->right);  // go right
        }
}
```

The above example illustrates an in-order traversal of the binary tree. A Binary search tree is a special case of the binary tree where the data elements of each node are in order.

Filesystem traversal

Since the number of files in a filesystem may vary, recursion is the only practical way to traverse and thus enumerate its contents. Traversing a filesystem is very similar to that of tree traversal, therefore the concepts behind tree traversal are applicable to traversing a filesystem. More specifically, the code below would be an example of a preorder traversal of a filesystem.

```
import java.io.*;

public class FileSystem {

        public static void main (String [] args) {
                traverse ();
        }

        /**
         * Obtains the filesystem roots
         * Proceeds with the recurisve filesystem traversal
         */
        private static void traverse () {
                File [] fs = File.listRoots ();
                for (int i = 0; i < fs.length; i++) {
                        if (fs[i].isDirectory () && fs[i].canRead ()) {
                                rtraverse (fs[i]);
                        }
                }
        }

        /**
         * Recursively traverse a given directory
         *
         * @param fd indicates the starting point of traversal
         */
        private static void rtraverse (File fd) {
                File [] fss = fd.listFiles ();

                for (int i = 0; i < fss.length; i++) {
                        System.out.println (fss[i]);
                        if (fss[i].isDirectory () && fss[i].canRead ()) {
                                rtraverse (fss[i]);
                        }
                }
        }

}
```

This code blends the lines, at least somewhat, between recursion and iteration. It is, essentially, a recursive implementation, which is the best way to traverse a filesystem. It is also an example of direct and indirect recursion. "rtraverse" is purely a direct example; "traverse" is the indirect, which calls "rtraverse." This example needs no "base case" scenario due to the fact that there will always be some fixed number of files and/or directories in a given filesystem.

Recursion versus iteration

Expressive power

Most programming languages in use today allow the direct specification of recursive functions and procedures. When such a function is called, the program's runtime environment keeps track of the various instances of the function (often using a call stack, although other methods may be used). Every recursive function can be transformed into an iterative function by replacing recursive calls with iterative control constructs and simulating the call stack with a stack explicitly managed by the program.

Conversely, all iterative functions and procedures that can be evaluated by a computer can be expressed in terms of recursive functions; iterative control constructs such as while loops and do loops routinely are rewritten in recursive form in functional languages. However, in practice this rewriting depends on tail call elimination, which is not a feature of all languages. C, Java, and Python are notable mainstream languages in which all function calls, including tail calls, cause stack allocation that would not occur with the use of looping constructs; in these languages, a working iterative program rewritten in recursive form may overflow the call stack.

Performance issues

In languages (such as C and Java) that favor iterative looping constructs, there is usually significant time and space cost associated with recursive programs, due to the overhead required to manage the stack and the relative slowness of function calls; in functional languages, a function call (particularly a tail call) is typically a very fast operation, and the difference is usually less noticeable.

As a concrete example, the difference in performance between recursive and iterative implementations of the "factorial" example above depends highly on the language used. In languages where looping constructs are preferred, the iterative version may be as much as several orders of magnitude faster than the recursive one. In functional languages, the overall time difference of the two implementations may be negligible; in fact, the cost of multiplying the larger numbers first rather than the smaller numbers (which the iterative version given here happens to do) may overwhelm any time saved by choosing iteration.

Other considerations

In some programming languages, the stack space available to a thread is much less than the space available in the heap, and recursive algorithms tend to require more stack space than iterative algorithms. Consequently, these languages sometimes place a limit on the depth of recursion to avoid stack overflows. (Python is one such language.) Note the caveat below regarding the special case of tail recursion.

There are some types of problems whose solutions are inherently recursive, because of prior state they need to track. One example is tree traversal; others include the Ackermann function, depth-first search, and divide-and-conquer algorithms such as Quicksort. All of these algorithms can be implemented iteratively with the help of an explicit stack, but the programmer effort involved in managing the stack, and the complexity of the resulting program, arguably outweigh any advantages of the iterative solution.

Tail-recursive functions

Tail-recursive functions are functions in which all recursive calls are tail calls and hence do not build up any deferred operations. For example, the gcd function (shown again below) is tail-recursive. In contrast, the factorial function (also below) is **not** tail-recursive; because its recursive call is not in tail position, it builds up deferred multiplication operations that must be performed after the final recursive call completes. With a compiler or interpreter that treats tail-recursive calls as jumps rather than function calls, a tail-recursive function such as gcd will execute using constant space. Thus the program is essentially iterative, equivalent to using imperative language control structures like the "for" and "while" loops.

Tail recursion:

```
//INPUT: Integers x, y such that x >=
y and y > 0
int gcd(int x, int y)
{
  if (y == 0)
     return x;
  else
     return gcd(y, x % y);
}
```

Augmenting recursion:

```
//INPUT: n is an Integer such
that n >= 1
int fact(int n)
{
   if (n == 1)
      return 1;
   else
      return n * fact(n - 1);
}
```

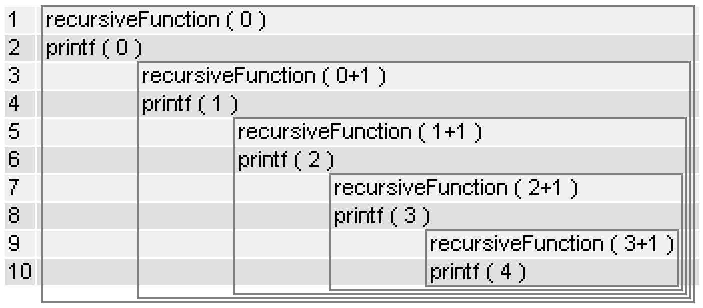
The significance of tail recursion is that when making a tail-recursive call, the caller's return position need not be saved on the call stack; when the recursive call returns, it will branch directly on the previously saved return position. Therefore, on compilers that support tail-recursion optimization, tail recursion saves both space and time.

Order of function calling

The order of calling a function may change the execution of a function, see this example in C language:

Function 1

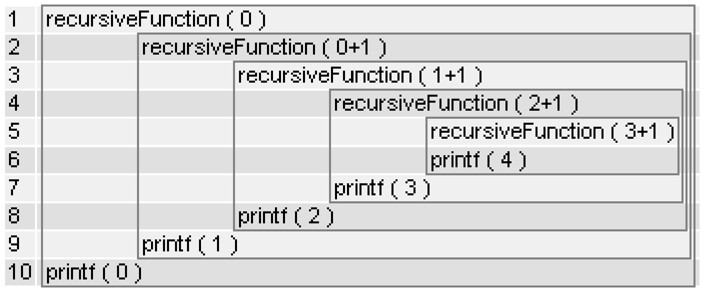
```
void recursiveFunction(int num) {
   printf("%d\n", num);
   if (num < 4)
      recursiveFunction(num + 1);
}
```



1
2
3
4
5
6
7
8
9
10
recursiveFunction (0)
printf (0)
recursiveFunction (0+1)
printf (1)
recursiveFunction (1+1)
printf (2)
recursiveFunction (2+1)
printf (3)
recursiveFunction (3+1)
printf (4)

Function 2 with swapped lines

```
void recursiveFunction(int num) {
   if (num < 4)
      recursiveFunction(num + 1);
   printf("%d\n", num);
}
```



1
2
3
4
5
6
7
8
9
10
recursiveFunction (0)
recursiveFunction (0+1)
recursiveFunction (1+1)
recursiveFunction (2+1)
recursiveFunction (3+1)
printf (4)
printf (3)
printf (2)
printf (1)
printf (0)

Direct and indirect recursion

Direct recursion is when a function calls itself. Indirect recursion is when (for example) function A calls function B, function B calls function C, and then function C calls function A.

Chapter 6

Introduction to Lambda Calculus

In mathematical logic and computer science, **lambda calculus**, also written as **λ-calculus**, is a formal system for function definition, function application and recursion. The portion of lambda calculus relevant to computation is now called the **untyped lambda calculus**. In both typed and untyped versions, ideas from lambda calculus have found application in the fields of logic, recursion theory (computability), and linguistics, and have played an important role in the development of the theory of programming languages (with untyped lambda calculus being the original inspiration for functional programming, in particular Lisp, and typed lambda calculi serving as the foundation for modern type systems).

History

Lambda calculus was introduced by Alonzo Church in the 1930s as part of an investigation into the foundations of mathematics. The original system was shown to be logically inconsistent in 1935 when Stephen Kleene and J. B. Rosser developed the Kleene–Rosser paradox.

Subsequently, in 1936 Church isolated and published just the portion relevant to computation, what is now called the untyped lambda calculus. In 1940, he also introduced a computationally weaker but logically consistent system, known as the simply typed lambda calculus.

Informal description

Motivation

Functions are a fundamental concept within computer science and mathematics. The λ-calculus provides simple semantics for computation with functions so that properties of functional computation can be studied.

Consider the following two examples. The identity function `I(x) = x` takes a single input, `x`, and immediately returns `x` (i.e. the identity does nothing with its input), whereas the function `sqadd(x, y) = x*x + y*y` takes a pair of inputs, `x` and `y` and returns the sum of their squares, `x*x + y*y`. Using these two examples, we can make some useful observations that motivate the major ideas in the lambda calculus.

The first observation is that functions need not be explicitly named. That is, the function `sqadd(x, y) = x*x + y*y` can be rewritten in *anonymous form* as `(x, y) ↦ x*x + y*y`, (read as "the pair of `x` and `y` is mapped to `x*x + y*y`"). Similarly, `I(x) = x` can be rewritten in anonymous form as `x ↦ x`, where the input is simply mapped to itself.

The second observation is that the specific choice of name for a function's arguments is largely irrelevant. That is, `x ↦ x` and `y ↦ y` express the same function: the identity. Similarly, `(x, y) ↦ x*x + y*y` and `(u, v) ↦ u*u + v*v` also express the same function.

Finally, any function that requires two inputs, for instance the before mentioned `sqadd` function, can be reworked into an equivalent function that accepts a single input, and as output returns *another* function, that in turn accepts a single input. For example, `(x, y) ↦ x*x + y*y` can be reworked into `x ↦ (y ↦ x*x + y*y)`. This transformation is called currying, and can be generalized to functions accepting an arbitrary number of arguments.

Currying may be best grasped intuitively through the use of an example. Compare the function `(x, y) ↦ x*x + y*y` with its curried form, `x ↦ (y ↦ x*x + y*y)`. Given two arguments, we have:

```
((x, y) ↦ x*x + y*y)(5, 2)
= 5*5 + 2*2 = 29.
```

However, using currying, we have:

```
((x ↦ (y ↦ x*x + y*y))(5))(2)
= (y ↦ 5*5 + y*y)(2)
= 5*5 + 2*2 = 29
```

and we see the uncurried and curried forms compute the same result. Notice that x*x has become a constant.

The lambda calculus

The lambda calculus consists of **lambda terms** and some extra operations.

Since the names of functions are largely a convenience, the lambda calculus has no means of naming a function. Since all functions expecting more than one input can be transformed into equivalent functions accepting a single input (via Currying), the lambda calculus has no means for creating a function that accepts more than one argument. Since the names of arguments are largely irrelevant, the native notion of equality on lambda terms is **alpha-equivalence** (see below), which codifies this principle.

Lambda terms

The syntax of lambda terms is particularly simple. There are three ways in which to obtain them:

- a lambda term may be a variable, `x`;
- if `t` is a lambda term, and `x` is a variable, then `λx.t` is a lambda term (called a **lambda abstraction**);
- if `t` and `s` are lambda terms, then `ts` is a lambda term (called an **application**).

Nothing else is a lambda term, though bracketing may be used to disambiguate terms.

Intuitively, a lambda abstraction `λx.t` represents an anonymous function that takes a single input, and the `λ` is said to **bind** `x` in `t`, and an application `ts` represents the application of input `s` to some function `t`. In the lambda calculus, functions are taken to be **first class values**, so functions may be used as the inputs to other functions, and functions may return functions as their outputs.

For example, `λx.x` represents the identity function, `x ↦ x`, and `(λx.x)y` represents the identity function applied to `y`. Further, `(λx.y)` represents the **constant function** `x ↦ y`, the function that always returns y, no matter the input. It should be noted that function application is left-associative, so `(λx.x)y z = ((λx.x)y)z`.

Lambda terms on their own aren't particularly interesting. What makes them interesting are the various notions of **equivalence** and **reduction** that can be defined over them.

Alpha equivalence

A basic form of equivalence, definable on lambda terms, is alpha equivalence. This states that the particular choice of bound variable, in a lambda abstraction, doesn't (usually) matter. For instance, `λx.x` and `λy.y` are alpha-equivalent lambda terms, representing the same identity function. Note that the terms `x` and `y` **aren't** alpha-equivalent, because they are not bound in a lambda abstraction. In many presentations, it is usual to identify alpha-equivalent lambda terms.

The following definitions are necessary in order to be able to define beta reduction.

Free variables

The **free variables** of a term are those variables not bound by a lambda abstraction. That is, the free variables of `x` are just `x`; the free variables of `λx.t` are the free variables of `t`, with `x` removed, and the free variables of `ts` are the union of the free variables of `t` and `s`.

For example, the lambda term representing the identity `λx.x` has no free variables, but the constant function `λx.y` has a single free variable, `y`.

Capture-avoiding substitutions

Using the definition of free variables, we may now define a capture-avoiding substitution. Suppose `t`, `s` and `r` are lambda terms and `x` and `y` are variables where `x` does not equal `y`. We write `t[x := r]` for the substitution of `r` for `x` in `t`, in a capture-avoiding manner. That is:

- `x[x := r] = r`;
- `y[x := r] = y`;
- `(ts)[x := r] = (t[x := r])(s[x := r])`;
- `(λx.t)[x := r] = λx.t`;
- `(λy.t)[x := r] = λy.(t[x := r])` if `x ≠ y` and `y` is not in the free variables of `r` (sometimes said "`y` is fresh for `r`").

For example, `(λx.x)[y := y] = λx.(x[y := y]) = λx.x`, and `((λx.y)x)[x := y] = ((λx.y)[x := y])(x[x := y]) = (λx.y)y`.

The freshness condition (requiring that `y` is not in the free variables of `r`) is crucial in order to ensure that substitution does not change the meaning of functions. For example, suppose we define another substitution action without the freshness condition. Then, `(λx.y)[y := x] = λx.(y[y := x]) = λx.x`, and the constant function `λx.y` turns into the identity `λx.x` by substitution.

If our freshness condition is not met, then we may simply alpha-rename with a suitably fresh variable. For example, switching back to our correct notion of substitution, in `(λx.y)[y := x]` the lambda abstraction can be renamed with a fresh variable `z`, to obtain `(λz.y)[y := x] = λz.(y[y := x]) = λz.x`, and the meaning of the function is preserved by substitution.

Beta reduction

Beta reduction states that an application of the form `(λx.t)s` reduces to the term `t[x := s]` (we write `(λx.t)s → t[x := s]` as a convenient shorthand for "`(λx.t)s` beta reduces to `t[x := s]`"). For example, for every `s` we have `(λx.x)s → x[x := s] = s`, demonstrating that `λx.x` really is the identity. Similarly, `(λx.y)s → y[x := s] = y`, demonstrating that `λx.y` really is a constant function.

The lambda calculus may be seen as an idealised functional programming language, like Haskell or Standard ML. Under this view, beta reduction corresponds to a computational step, and in the untyped lambda calculus, as presented here, reduction need not terminate. For instance, consider the term `(λx.xx)(λx.xx)`. Here, we have `(λx.xx)(λx.xx) → (xx)[x := λx.xx] = (x[x := λx.xx])(x[x := λx.xx]) = (λx.xx)(λx.xx)`. That is, the term reduces to itself in a single beta reduction, and therefore reduction will never terminate.

Another problem with the untyped lambda calculus is the inability to distinguish between different kinds of data. For instance, we may want to write a function that only operates on numbers. However, in the untyped lambda calculus, there's no way to prevent our function from being applied to truth values, or strings, for instance.

Formal definition

Definition

Lambda expressions are composed of

> variables $v_1, v_2, ..., v_n$
> the abstraction symbols λ and .
> parentheses ( )

The set of lambda expressions, Λ, can be defined recursively:

1. If x is a variable, then $x \in \Lambda$
2. If x is a variable and $M \in \Lambda$, then $(\lambda x.M) \in \Lambda$
3. If $M, N \in \Lambda$, then $(M\ N) \in \Lambda$

Instances of rule 2 are known as abstractions and instances of rule 3 are known as applications.

Notation

To keep the notation of lambda expressions uncluttered, the following conventions are usually applied.

- Outermost parentheses are dropped: M N instead of (M N).
- Applications are assumed to be left associative: M N P means (M N) P.
- The body of an abstraction extends as far right as possible: λx.M N means λx.(M N) and not (λx.M) N.
- A sequence of abstractions is contracted: λx.λy.λz.N is abbreviated as λxyz.N.

Free and bound variables

The abstraction operator, λ, is said to bind its variable wherever it occurs in the body of the abstraction. Variables that fall within the scope of a lambda are said to be *bound*. All other variables are called *free*. For example in the following expression y is a bound variable and x is free: `λy.x x y`. Also note that a variable binds to its "nearest" lambda. In the following expression one single occurrence of x is bound by the second lambda: `λx.y (λx.z x)`

The set of *free variables* of a lambda expression, M, is denoted as FV(M) and is defined by recursion on the structure of the terms, as follows:

1. FV(x) = {x}, where x is a variable
2. FV(λx.M) = FV(M) - {x}
3. FV(M N) = FV(M) $\cup$ FV(N)

An expression which contains no free variables is said to be *closed*. Closed lambda expressions are also known as combinators and are equivalent to terms in combinatory logic.

Reduction

The meaning of lambda expressions is defined by how expressions can be reduced.

There are three kinds of reduction:

- **α-conversion**: changing bound variables;
- **β-reduction**: applying functions to their arguments;
- **η-conversion**: which captures a notion of extensionality.

We also speak of the resulting equivalences: two expressions are *β-equivalent* if they can be β-converted into the same expression, and α/η-equivalence are defined similarly.

The term *redex*, short for *reducible expression*, refers to subterms which can be reduced by one of the reduction rules. For example, `(λx.M) N` is a beta-redex; if `x` is not free in `M`, `λx.M x` is an eta-redex. The expression to which a redex reduces is called its reduct; using the previous example, the reducts of these expressions are respectively `M[x:=N]` and `M`.

α-conversion

Alpha-conversion, sometimes known as alpha-renaming, allows bound variable names to be changed. For example, alpha-conversion of `λx.x` might yield `λy.y`. Terms that differ only by alpha-conversion are called *α-equivalent*. Frequently in uses of lambda calculus, α-equivalent terms are considered to be equivalent.

The precise rules for alpha-conversion are not completely trivial. First, when alpha-converting an abstraction, the only variable occurrences that are renamed are those that are bound to the same abstraction. For example, an alpha-conversion of `λx.λx.x` could result in `λy.λx.x`, but it could *not* result in `λy.λx.y`. The latter has a different meaning from the original.

Second, alpha-conversion is not possible if it would result in a variable getting captured by a different abstraction. For example, if we replace `x` with `y` in `λx.λy.x`, we get `λy.λy.y`, which is not at all the same.

In programming languages with static scope, alpha-conversion can be used to make name resolution simpler by ensuring that no variable name masks a name in a containing scope.

Substitution

Substitution, written `E[V := E']`, is the process of replacing all free occurrences of the variable `V` by expression `E'`. Substitution on terms of the λ-calculus is defined by recursion on the structure of terms, as follows.

```
x[x := N]         ≡ N
y[x := N]         ≡ y, if x ≠ y
(M1 M2)[x := N]  ≡ (M1[x := N]) (M2[x := N])
(λy.M)[x := N]   ≡ λy.(M[x := N]), if x ≠ y and y ∉ FV(N)
```

To substitute into a lambda abstraction, it is sometimes necessary to α-convert the expression. For example, it is not correct for `(λx.y)[y := x]` to result in `(λx.x)`, because the substituted `x` was supposed to be free but ended up being bound. The correct substitution in this case is `(λz.x)`, up-to α-equivalence. Notice that substitution is defined uniquely up-to α-equivalence.

β-reduction

Beta-reduction captures the idea of function application. Beta-reduction is defined in terms of substitution: the beta-reduction of `((λV.E) E')` is `E[V := E']`.

For example, assuming some encoding of `2, 7, *`, we have the following β-reductions: `((λn.n*2) 7)` → `7*2`.

η-conversion

Eta-conversion expresses the idea of extensionality, which in this context is that two functions are the same if and only if they give the same result for all arguments. Eta-conversion converts between `λx.f x` and `f` whenever `x` does not appear free in `f`.

This conversion is not always appropriate when lambda expressions are interpreted as programs. Evaluation of `λx.f x` can terminate even when evaluation of *f* does not.

Normal forms and confluence

For the untyped lambda calculus, β-reduction as a rewriting rule is neither strongly normalising nor weakly normalising.

However, it can be shown that β-reduction is confluent. (Of course, we are working up to α-conversion, i.e. we consider two normal forms to be equal if it is possible to α-convert one into the other.)

Therefore, both strongly normalising terms and weakly normalising terms have a unique normal form. For strongly normalising terms, any reduction strategy is guaranteed to yield the normal form, whereas for weakly normalising terms, some reduction strategies may fail to find it.

Arithmetic in lambda calculus

There are several possible ways to define the natural numbers in lambda calculus, but by far the most common are the Church numerals, which can be defined as follows:

```
0 := λfx.x
1 := λfx.f x
2 := λfx.f (f x)
3 := λfx.f (f (f x))
```

and so on. A Church numeral is a higher-order function—it takes a single-argument function *f*, and returns another single-argument function. The Church numeral *n* is a function that takes a function *f* as argument and returns the *n*-th composition of *f*, i.e. the function *f* composed with itself *n* times. This is denoted $f^{(n)}$ and is in fact the *n*-th power of *f* (considered as an operator); $f^{(0)}$ is defined to be the identity function. Such repeated compositions (of a single function *f*) obey the laws of exponents, which is why these numerals can be used for arithmetic. (In Church's original lambda calculus, the formal parameter of a lambda expression was required to occur at least once in the function body, which made the above definition of `0` impossible.)

We can define a successor function, which takes a number *n* and returns `n + 1` by adding an additional application of *f*:

```
SUCC := λnfx.f (n f x)
```

Because the *m*-th composition of *f* composed with the *n*-th composition of *f* gives the *m*+*n*-th composition of *f*, addition can be defined as follows:

```
PLUS := λmnfx.m f (n f x)
```

`PLUS` can be thought of as a function taking two natural numbers as arguments and returning a natural number; it can be verified that

`PLUS 2 3` and
`5`

are equivalent lambda expressions. Since adding *m* to a number *n* can be accomplished by adding 1 *m* times, an equivalent definition is:

```
PLUS := λmn.m SUCC n
```

Similarly, multiplication can be defined as

```
MULT := λmnf.m (n f)
```

Alternatively

```
MULT := λmn.m (PLUS n) 0
```

since multiplying m and n is the same as repeating the add n function m times and then applying it to zero. Exponentiation has a rather simple rendering in Church numerals, namely

```
POW := λbe.e b
```

The predecessor function defined by `PRED` $n = n - 1$ for a positive integer n and `PRED 0 = 0` is considerably more difficult. The formula

```
PRED := λnfx.n (λgh.h (g f)) (λu.x) (λu.u)
```

can be validated by showing inductively that if `T` denotes `(λgh.h (g f))`, then $\mathtt{T}^{(n)}(\lambda u.x) = (\lambda h.h(f^{(n-1)}(x)))$ for $n > 0$. Two other definitions of `PRED` are given below, one using conditionals and the other using pairs. With the predecessor function, subtraction is straightforward. Defining

```
SUB := λmn.n PRED m,
```

`SUB` m n yields $m - n$ when $m > n$ and `0` otherwise.

Logic and predicates

By convention, the following two definitions (known as Church booleans) are used for the boolean values `TRUE` and `FALSE`:

```
TRUE := λxy.x
FALSE := λxy.y
```

(Note that `FALSE` is equivalent to the Church numeral zero defined above)

Then, with these two λ-terms, we can define some logic operators (these are just possible formulations; other expressions are equally correct):

```
AND := λpq.p q p
OR := λpq.p p q
NOT := λpab.p b a
IFTHENELSE := λpab.p a b
```

We are now able to compute some logic functions, for example:

```
AND TRUE FALSE
≡ (λpq.p q p) TRUE FALSE →β TRUE FALSE TRUE
≡ (λxy.x) FALSE TRUE →β FALSE
```

and we see that `AND TRUE FALSE` is equivalent to `FALSE`.

A *predicate* is a function which returns a boolean value. The most fundamental predicate is `ISZERO` which returns `TRUE` if its argument is the Church numeral `0`, and `FALSE` if its argument is any other Church numeral:

```
ISZERO := λn.n (λx.FALSE) TRUE
```

The following predicate tests whether the first argument is less-than-or-equal-to the second:

```
LEQ := λmn.ISZERO (SUB m n),
```

and since `m = n` iff `LEQ m n` and `LEQ n m`, it is straightforward to build a predicate for numerical equality.

The availability of predicates and the above definition of `TRUE` and `FALSE` make it convenient to write "if-then-else" expressions in lambda calculus. For example, the predecessor function can be defined as:

```
PRED := λn.n (λgk.ISZERO (g 1) k (PLUS (g k) 1)) (λv.0) 0
```

which can be verified by showing inductively that `n (λgk.ISZERO (g 1) k (PLUS (g k) 1)) (λv.0)` is the add *n* - 1 function for *n* > 0.

Pairs

A pair (2-tuple) can be defined in terms of `TRUE` and `FALSE`, by using the Church encoding for pairs. For example, `PAIR` encapsulates the pair (*x*,*y*), `FIRST` returns the first element of the pair, and `SECOND` returns the second.

```
PAIR := λxyf.f x y
FIRST := λp.p TRUE
SECOND := λp.p FALSE
NIL := λx.TRUE
NULL := λp.p (λxy.FALSE)
```

A linked list can be defined as either NIL for the empty list, or the `PAIR` of an element and a smaller list. The predicate `NULL` tests for the value `NIL`. (Alternatively, with `NIL := FALSE`, the construct `l (λhtz.deal_with_head_h_and_tail_t) (deal_with_nil)` obviates the need for an explicit NULL test).

As an example of the use of pairs, the shift-and-increment function that maps `(m, n)` to `(n, n + 1)` can be defined as

```
Φ := λx.PAIR (SECOND x) (SUCC (SECOND x))
```

which allows us to give perhaps the most transparent version of the predecessor function:

```
PRED := λn.FIRST (n Φ (PAIR 0 0)).
```

Standard terms

Certain terms have commonly accepted names:

```
I := λx.x
K := λxy.x
S := λxyz.(x z (y z))
ω := λx.(x x)
Ω := ω ω
```

Typed lambda calculi

Computable functions and lambda calculus

A function F: **N** → **N** of natural numbers is a computable function if and only if there exists a lambda expression f such that for every pair of x, y in **N**, $F(x)=y$ if and only if f $x =_\beta y$, where x and y are the Church numerals corresponding to x and y, respectively and $=_\beta$ meaning equivalence with beta reduction.

Undecidability of equivalence

There is no algorithm which takes as input two lambda expressions and outputs `TRUE` or `FALSE` depending on whether or not the two expressions are equivalent. This was historically the first problem for which undecidability could be proven. As is common for a proof of undecidability, the proof shows that no computable function can decide the equivalence. Church's thesis is then invoked to show that no algorithm can do so.

Church's proof first reduces the problem to determining whether a given lambda expression has a *normal form*. A normal form is an equivalent expression which cannot be reduced any further under the rules imposed by the form. Then he assumes that this predicate is computable, and can hence be expressed in lambda calculus. Building on earlier work by Kleene and constructing a Gödel numbering for lambda expressions, he constructs a lambda expression e which closely follows the proof of Gödel's first incompleteness theorem. If e is applied to its own Gödel number, a contradiction results.

Lambda calculus and programming languages

As pointed out by Peter Landin's 1965 paper *A Correspondence between ALGOL 60 and Church's Lambda-notation*, sequential procedural programming languages can be understood in terms of the lambda calculus, which provides the basic mechanisms for procedural abstraction and procedure (subprogram) application.

Lambda calculus reifies "functions" and makes them first-class objects, which raises implementation complexity when implementing lambda calculus. A particular challenge is related to the support of higher-order functions, also known as the Funarg problem. Lambda calculus is usually implemented using a virtual machine approach. The first practical implementation of lambda calculus was provided in 1963 by Peter Landin, and is known as the SECD machine. Since then, several optimized abstract machines for lambda calculus were suggested, such as the G-machine and the categorical abstract machine.

The most prominent counterparts to lambda calculus in programming are functional programming languages, which essentially implement the calculus augmented with some constants and datatypes. Lisp uses a variant of lambda notation for defining functions, but only its purely functional subset ("Pure Lisp") is really equivalent to lambda calculus.

Reduction strategies

Whether a term is normalising or not, and how much work needs to be done in normalising it if it is, depends to a large extent on the reduction strategy used. The distinction between reduction strategies relates to the distinction in functional programming languages between eager evaluation and lazy evaluation.

Full beta reductions

Any redex can be reduced at any time. This means essentially the lack of any particular reduction strategy—with regard to reducibility, "all bets are off".

Applicative order

The leftmost, innermost redex is always reduced first. Intuitively this means a function's arguments are always reduced before the function itself. Applicative order always attempts to apply functions to normal forms, even when this is not possible.

Most programming languages (including Lisp, ML and imperative languages like C and Java) are described as "strict", meaning that functions applied to non-normalising arguments are non-normalising. This is done essentially using applicative order, call by value reduction (see below), but usually called "eager evaluation".

Normal order

The leftmost, outermost redex is always reduced first. That is, whenever possible the arguments are substituted into the body of an abstraction before the arguments are reduced.

Call by name

As normal order, but no reductions are performed inside abstractions. For example `λx.(λx.x)x` is in normal form according to this strategy, although it contains the redex `(λx.x)x`.

Call by value

Only the outermost redexes are reduced: a redex is reduced only when its right hand side has reduced to a value (variable or lambda abstraction).

Call by need

> As normal order, but function applications that would duplicate terms instead name the argument, which is then reduced only "when it is needed". Called in practical contexts "lazy evaluation". In implementations this "name" takes the form of a pointer, with the redex represented by a thunk.

Applicative order is not a normalising strategy. The usual counterexample is as follows: define `Ω = ωω` where `ω = λx.xx`. This entire expression contains only one redex, namely the whole expression; its reduct is again `Ω`. Since this is the only available reduction, `Ω` has no normal form (under any evaluation strategy). Using applicative order, the expression `KIΩ = (λxy.x) (λx.x)Ω` is reduced by first reducing `Ω` to normal form (since it is the leftmost redex), but since `Ω` has no normal form, applicative order fails to find a normal form for `KIΩ`.

In contrast, normal order is so called because it always finds a normalising reduction if one exists. In the above example, `KIΩ` reduces under normal order to `I`, a normal form. A drawback is that redexes in the arguments may be copied, resulting in duplicated computation (for example, `(λx.xx) ((λx.x)y)` reduces to `((λx.x)y) ((λx.x)y)` using this strategy; now there are two redexes, so full evaluation needs two more steps, but if the argument had been reduced first, there would now be none).

The positive tradeoff of using applicative order is that it does not cause unnecessary computation if all arguments are used, because it never substitutes arguments containing redexes and hence never needs to copy them (which would duplicate work). In the above example, in applicative order `(λx.xx) ((λx.x)y)` reduces first to `(λx.xx)y` and then to the normal order `yy`, taking two steps instead of three.

Most *purely* functional programming languages (notably Miranda and its descendents, including Haskell), and the proof languages of theorem provers, use *lazy evaluation*, which is essentially the same as call by need. This is like normal order reduction, but call by need manages to avoid the duplication of work inherent in normal order reduction using *sharing*. In the example given above, `(λx.xx) ((λx.x)y)` reduces to `((λx.x)y) ((λx.x)y)`, which has two redexes, but in call by need they are represented using the same object rather than copied, so when one is reduced the other is too.

A note about complexity

While the idea of beta reduction seems simple enough, it is not an atomic step, in that it must have a non-trivial cost when estimating computational complexity. To be precise, one must somehow find the location of all of the occurrences of the bound variable V in the expression E, implying a time cost, or one must keep track of these locations in some way, implying a space cost. A naïve search for the locations of V in E is $O(n)$ in the length n of E. This has led to the study of systems which use explicit substitution. Sinot's director strings offer a way of tracking the locations of free variables in expressions.

Parallelism and concurrency

The Church-Rosser property of the lambda calculus means that evaluation (β-reduction) can be carried out in *any order*, even in parallel. This means that various nondeterministic evaluation strategies are relevant. However, the lambda calculus does not offer any explicit constructs for parallelism. One can add constructs such as Futures to the lambda calculus. Other process calculi have been developed for describing communication and concurrency.

Chapter 7

Church Encoding

In mathematics, **Church encoding** is a means of embedding data and operators into the lambda calculus, the most familiar form being the **Church numerals**, a representation of the natural numbers using lambda notation. The method is named for Alonzo Church, who first encoded data in the lambda calculus this way.

Terms that are usually considered primitive in other notations (such as integers, booleans, pairs, lists, and tagged unions) are mapped to higher-order functions under Church encoding; the Church-Turing thesis asserts that any computable operator (and its operands) can be represented under Church encoding.

Many students of mathematics are familiar with Gödel numbering members of a set; Church encoding is an equivalent operation defined on lambda abstractions instead of natural numbers.

Church numerals

Church numerals are the representations of natural numbers under Church encoding. The higher-order function that represents natural number *n* is a function that maps any other function *f* to its *n*-fold composition. In simpler terms, the "value" of the numeral is equivalent to the number of times the function encapsulates its argument.

$$f^n = f \circ f \circ \cdots \circ f.$$

Definition

Church numerals **0**, **1**, **2**, ..., are defined as follows in the lambda calculus:

0 ≡ `λf.λx. x`
1 ≡ `λf.λx. f x`
2 ≡ `λf.λx. f (f x)`
3 ≡ `λf.λx. f (f (f x))`
...
n ≡ λf.λx. f^n x
...

That is, the natural number *n* is represented by the Church numeral **n**, which has the property that for any lambda-terms `F` and `X`,

n `F X` $=_\beta$ `F`^{n} `X`

Computation with Church numerals

In the lambda calculus, numeric functions are representable by corresponding functions on Church numerals. These functions can be implemented in most functional programming languages (subject to type constraints) by direct translation of lambda terms.

The addition function **plus**(*m*,*n*)=*m*+*n* uses the identity $f^{(m+n)}(x) = f^m(f^n(x))$.

plus ≡ `λm.λn.λf.λx. m f (n f x)`

The successor function **succ**(*n*)=*n*+1 is β-equivalent to (**plus 1**).

succ ≡ `λn.λf.λx. f (n f x)`

The multiplication function **mult**(*m*,*n*)=*m***n* uses the identity $f^{(m * n)} = (f^n)^m$.

mult ≡ `λm.λn.λf.λx. m (n f) x`

The exponentiation function **exp**(*m*,*n*)=*m*^*n* is straightforward using the multiplication function defined above.

exp ≡ `λm.λn. n (`**`mult`**` m) `**`1`**

The predecessor function $\operatorname{pred}(n) = \begin{cases} 0 & \text{if } n = 0, \\ n-1 & \text{otherwise} \end{cases}$ works by generating an *n*-fold composition of functions that each apply their argument `g` to `f`; the base case discards its copy of `f` and returns `x`.

pred ≡ `λn.λf.λx. n (λg.λh. h (g f)) (λu. x) (λu. u)`

The subtraction function can be written based on the predecessor function.

sub ≡ `λm.λn. (n `**`pred`**`) m`

The zero predicate can be written as:

zero? ≡ `λn. n (λx.F) T`

Now:

> **zero? 0** $=_\beta$ *T* if **n** $=_\beta$ **0**.
> **zero? n** $=_\beta$ *F* if **n** $\neq_\beta$ **0**, provided **n** is a Church numeral and where $\neq_\beta$ is the negation of $=_\beta$ restricted to reducible lambda terms.

Translation with other representations

Most real-world languages have support for machine-native integers; the *church* and *unchurch* functions (given here in Haskell) convert between nonnegative integers and their corresponding church numerals. Implementations of these conversions in other languages are similar.

```
type Church a = (a -> a) -> (a -> a)

church :: Integer -> Church Integer
church 0 = \f -> \x -> x
church n = \f -> \x -> f (church (n-1) f x)

unchurch :: Church Integer -> Integer
unchurch n = n (\x -> x + 1) 0
```

In Haskell, the `\` corresponds to the λ of Lambda calculus.

Church booleans

Church booleans are the Church encoding of the boolean values *true* and *false.* Some programming languages use these as an implementation model for boolean arithmetic; examples are Smalltalk and Pico. The boolean values are represented as functions of two values that evaluate to one or the other of their arguments.

Formal definition in lambda calculus:

> **true** ≡ `λa.λb. a`
> **false** ≡ `λa.λb. b`

Note that this definition allows predicates (i.e. functions returning logical values) to directly act as if-clauses, e.g. if **predicate** is a unary predicate,

> **predicate** x **then-clause else-clause**

evaluates to **then-clause** if **predicate** x evaluates to **true**, and to **else-clause** if **predicate** x evaluates to **false**.

Functions of boolean arithmetic can be derived for Church booleans:

and ≡ `λm.λn. m n m`
or ≡ `λm.λn. m m n`
not ≡ `λm.λa.λb. m b a`
xor ≡ `λm.λn.λa.λb. m (n b a) (n a b)`

Some examples:

and true false ≡ `(λm.λn. m n m) (λa.λb. a) (λa.λb. b) ≡ (λa.λb. a) (λa.λb. b) (λa.λb. a) ≡ (λa.λb. b) ≡` **`false`**
or true false ≡ `(λm.λn. m m n) (λa.λb. a) (λa.λb. b)` ≡ `(λa.λb. a) (λa.λb. a) (λa.λb. b)` ≡ `(λa.λb. a)` ≡ **true**
not true ≡ `(λm.λa.λb. m b a) (λa.λb. a)` ≡ `(λa.λb. (λa.λb. a) b a)` ≡ `(λa.λb. b)` ≡ **false**

Church pairs

Church pairs are the Church encoding of the pair (two-tuple) type. The pair is represented as a function that takes a function argument. When given its argument it will apply the argument to the two components of the pair.

Formal definition in lambda calculus:

pair ≡ `λx.λy.λz.z x y`
fst ≡ `λp.p (λx.λy.x)`
snd ≡ `λp.p (λx.λy.y)`

An example:

fst (**pair** a b) ≡ `λp.p (λx.λy.x) ((λx.λy.λz.z x y) a b) ≡ λp.p (λx.λy.x) (λz.z a b) ≡ (λz.z a b) (λx.λy.x) ≡ (λx.λy.x) a b ≡ a`

List encodings

An encoding of (immutable) lists of varying length must define a constructor for creating an empty list (**nil**), an operation testing whether or not a list is empty (**isnil**), an operation to prepend a given value to a (possibly empty) list (**cons**), and two operations to determine the first element and the list of the remaining elements of a nonempty list (**head** and **tail**).

Church pairs

A nonempty list can basically be encoded by a Church pair with the head of the list stored in the first component of the pair and the tail of the list in the second component. However, special care is needed to unambiguously encode the empty list. This can be achieved by encapsulating any individual list node with another pair with the second component containing the list node and the first component containing a Church boolean which is **true** for the empty list and **false** otherwise, similarly to a tagged union. Using this idea the basic list operations can be defined like this:

nil ≡ **pair true true**
isnil ≡ **fst**
cons ≡ λh.λt.**pair false** (**pair** h t)
head ≡ λz.**fst** (**snd** z)
tail ≡ λz.**snd** (**snd** z)

The second component of the pair encoding **nil** is never used provided that **head** and **tail** are only applied to nonempty lists.

Higher-order function

As an alternative to the encoding using Church pairs, a list can be encoded by identifying it with its right fold function. For example, a list of three elements x, y and z can be encoded by a higher-order function which when applied to a combinator c and a value n returns c x (c y (c z n))).

nil ≡ `λc.λn.n`
isnil ≡ `λl.l (λh.λt.`**`false`**`)` **`true`**
cons ≡ `λh.λt.λc.λn.c h (t c n)`
head ≡ `λl.l (λh.λt.h)` **`false`**
tail ≡ `λl.`**`fst`**` (l (λx.λp.`**`pair`**` (`**`snd`**` p) (`**`cons`**` x (`**`snd`**` p))) (`**`pair nil nil`**`))`

Chapter 8

Fixed Point Combinator

A **fixed point combinator** (or **fixed-point operator**) is a higher-order function that computes a fixed point of other functions. A *fixed point* of a function **f** is a value *x* such that **f**(*x*) = *x*. For example, 0 and 1 are fixed points of the function **f**(*x*) = x^2, because $0^2 = 0$ and $1^2 = 1$. Whereas a fixed-point of a first-order function (a function on "simple" values such as integers) is a first-order value, a fixed point of a higher-order function **f** is *another function* **p** such that **f**(**p**) = **p**. A fixed point combinator, then, is a function **g** which produces such a fixed point **p** for any function **f**:

$$\mathbf{p} = \mathbf{g}(\mathbf{f}),\ \mathbf{f}(\mathbf{p}) = \mathbf{p}$$

or, alternately:

$$\mathbf{f}(\mathbf{g}(\mathbf{f})) = \mathbf{g}(\mathbf{f}).$$

Because fixed-point combinators are higher-order functions, their history is intimately related to the development of lambda calculus. One well-known fixed point combinator in the untyped lambda calculus is Haskell Curry's **Y** = λf·(λx·f (x x)) (λx·f (x x)). The name of this combinator is incorrectly used sometimes to refer to any fixed point combinator. The untyped lambda calculus however, contains an infinity of fixed point combinators. Fixed point combinators do not necessarily exist in more restrictive models of computation. For instance, they do not exist in simply typed lambda calculus.

In programming languages that support function literals, fixed point combinators allow the definition and use of anonymous recursive functions, i.e. without having to bind such functions to identifiers. In this setting, the use of fixed point combinators is sometimes called **anonymous recursion**.

How it works

Ignoring for now the question whether fixed point combinators even exist (to be addressed in the next section), we illustrate how a function satisfying the fixed point combinator equation works. To easily trace the computation steps, we choose untyped lambda calculus as our programming language. The (computational) equality from the equation that defines a fixed point combinator corresponds to beta reduction in lambda calculus.

As a concrete example of a fixed point combinator applied to a function, we use the standard recursive mathematical equation that defines the factorial function

fact(*n*) = if *n*=0 then 1 else *n* * **fact**(*n*-1)

We can express a single step of this recursion in lambda calculus as the lambda abstraction

```
F = λf. λn. IFTHENELSE (ISZERO n) 1 (MULT n (f (PRED n)))
```

using the usual encoding for booleans, and Church encoding for numerals. The functions in CAPITALS above can all be defined as lambda abstractions with their intuitive meaning, see lambda calculus for their precise definitions. Intuitively, f in **F** is a place-holder argument for the factorial function itself.

We now investigate what happens when a fixed point combinator FIX is applied to **F** and the resulting abstraction, which we hope it would be the factorial function, is in turn applied to a (Church-encoded) numeral.

```
(FIX F) n = (F (FIX F)) n =  --- expanded the defining equation of FIX
          = (λx. IFTHENELSE (ISZERO x) 1 (MULT x ((FIX F) (PRED x)))) n
--- expanded the first F
          = IFTHENELSE (ISZERO n) 1 (MULT n ((FIX F) (PRED n))) ---
applied abstraction to n.
```

If we now abbreviate FIX **F** as FACT, we recognize that any application of the abstraction FACT to a Church numeral calculates its factorial

```
FACT n = IFTHENELSE (ISZERO n) 1 (MULT n (FACT (PRED n))).
```

Recall that in lambda calculus all abstractions are anonymous, and that the names we give to some of them are only syntactic sugar. Therefore, it's meaningless in this context to contrast FACT as a recursive function with F as somehow being "not recursive". What fixed point operators really buy us here is the ability to *solve recursive equations*. That is, we can ask the question in reverse: does there exist a lambda abstraction that satisfies the equation of FACT? The answer is yes, and we have a "mechanical" procedure for producing such an abstraction: simply define **F** as above, and then use any fixed point combinator FIX to obtain FACT as FIX **F**.

In the practice of programming, substituting FACT at a call site by the expression FIX **F** is sometimes called *anonymous recursion*. In the lambda abstraction **F**, FACT is represented by the bound variable f, which corresponds to an argument in a programming language, therefore **F** need not be bound to an identifier. **F** however is a higher-order function because the argument *f* is itself called as a function, so the programming language must allow passing functions as arguments. It must also allow function literals because FIX **F** is an expression rather than an identifier. In practice, there are further limitations imposed on FIX, depending on the evaluation strategy employed by the programming environment and the type checker. These are described in the implementation section.

Existence of fixed point combinators

In certain mathematical formalizations of computation, such as the untyped lambda calculus and combinatory logic, every expression can be considered a higher-order function. In these formalizations, the existence of a fixed-point combinator means that *every function has at least one fixed point;* a function may have more than one distinct fixed point.

In some other systems, for example in the simply typed lambda calculus, a well-typed fixed-point combinator cannot be written. In those systems any support for recursion must be explicitly added to the language. In still others, such as the simply-typed lambda calculus extended with recursive types, fixed-point operators can be written, but the type of a "useful" fixed-point operator (one whose application always returns) may be restricted. In polymorphic lambda calculus (System F) a polymorphic fixed point combinator has type $\forall a.(a\rightarrow a)\rightarrow a$, where *a* is a type variable.

Y combinator

One well-known (and perhaps the simplest) fixed point combinator in the untyped lambda calculus is called the **Y** combinator. It was discovered by Haskell B. Curry, and is defined as:

$$\mathbf{Y} = \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$$

We can see that this function acts as a fixed point combinator by applying it to an example function *g* and rewriting:

$\mathbf{Y}$ g	$= (\lambda f\ .\ (\lambda x\ .\ f\ (x\ x))\ (\lambda x\ .\ f\ (x\ x)))\ g$	(by definition of **Y**)
	$= (\lambda x\ .\ g\ (x\ x))\ (\lambda x\ .\ g\ (x\ x))$	(β-reduction of λf: applied main function to g)
	$= (\lambda y\ .\ g\ (y\ y))\ (\lambda x\ .\ g\ (x\ x))$	(α-conversion: renamed bound variable)
	$= g\ ((\lambda x\ .\ g\ (x\ x))\ (\lambda x\ .\ g\ (x\ x)))$	(β-reduction of λy: applied left function to right function)
	$= g\ (\mathbf{Y}\ g)$	(by second equality)

In programming practice, the **Y** combinator is useful only in those languages that provide a call-by-name evaluation strategy, since (**Y** *g*) diverges (for any *g*) in call-by-value settings.

Other fixed point combinators

In untyped lambda calculus fixed point combinators are not especially rare. In fact there are infinitely many of them. In 2005 Mayer Goldberg showed that the set of fixed point combinators of untyped lambda calculus is recursively enumerable.

A version of the **Y** combinator that can be used in call-by-value (applicative-order) evaluation is given by η-expansion of part of the ordinary **Y** combinator:

$$\mathbf{Z} = \lambda f.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))$$

Here is an example of this in Python:

```
>>> Z = lambda f: (lambda x: f(lambda *args: x(x)(*args)))(lambda x:
f(lambda *args: x(x)(*args)))
>>> fact = lambda f: lambda x: 1 if x == 0 else x * f(x-1)
>>> Z(fact)(5)
120
```

The **Y** combinator can be expressed in the SKI-calculus as

$$\mathbf{Y} = S\ (K\ (S\ I\ I))\ (S\ (S\ (K\ S)\ K)\ (K\ (S\ I\ I)))$$

The simplest fixed point combinator in the SK-calculus, found by John Tromp, is

$$\mathbf{Y'} = S\ S\ K\ (S\ (K\ (S\ S\ (S\ (S\ S\ K))))\ K)$$

which corresponds to the lambda expression

$$\mathbf{Y'} = (\lambda x.\ \lambda y.\ x\ y\ x)\ (\lambda y.\ \lambda x.\ y\ (x\ y\ x))$$

Another common fixed point combinator is the Turing fixed-point combinator (named after its discoverer, Alan Turing):

$$\mathbf{\Theta} = (\lambda x.\ \lambda y.\ (y\ (x\ x\ y)))\ (\lambda x.\ \lambda y.\ (y\ (x\ x\ y)))$$

It also has a simple call-by-value form:

$$\mathbf{\Theta_v} = (\lambda x.\ \lambda y.\ (y\ (\lambda z.\ x\ x\ y\ z)))\ (\lambda x.\ \lambda y.\ (y\ (\lambda z.\ x\ x\ y\ z)))$$

Some fixed point combinators, such as this one (constructed by Jan Willem Klop) are useful chiefly for amusement:

$$\mathbf{Y_k} = (L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L)$$

where:

> L = λabcdefghijklmnopqstuvwxyzr. (r (t h i s i s a f i x e d p o i n t c o m b i n a t o r))

Strictly non-standard fixed point combinators

In untyped lambda calculus there are terms that have the same Böhm tree as a fixed point combinator, that is they have the same infinite extension λx.x(x(x ...)). These are called **non-standard fixed point combinators**. Evidently, any fixed point combinator is also a non-standard one, but not all non-standard fixed point combinators are fixed point combinators because some of them fail to satisfy the equation that defines the "standard" ones. These strange combinators are called **strictly non-standard fixed point combinators**; an example is the following combinator N = BM(B(BM)B), where B = λxyz.x(yz) and M = λx.xx. The set of non-standard fixed point combinators is not recursively enumerable.

Implementing fixed point combinators

In a language that supports lazy evaluation, like in Haskell, it is possible to define a fixed point combinator using the defining equation of the fixed point combinator. A programming language of this kind effectively solves the fixed point combinator equation by means of its own (lazy) recursion mechanism. This implementation of a fixed point combinator in Haskell is sometimes referred to as defining the **Y** combinator in Haskell. This is incorrect because the actual **Y** combinator is rejected by the Haskell type checker (but see the following section for a small modification of **Y** using a recursive type which works). The listing below shows the implementation of a fixed point combinator in Haskell that exploits Haskell's ability to solve the fixed point combinator equation. This fixed point combinator is traditionally called `fix` in Haskell. Observe that `fix` is a polymorphic fixed point combinator (c.f. the discussion in previous section on System F); its type is only shown for clarity—it can be inferred in Haskell. The definition is followed by some usage examples.

```
fix :: (a -> a) -> a
fix f = f (fix f)

fix (const 9) -- const is a function that returns its first parameter
and ignores the second; this evaluates to 9

factabs fact 0 = 1 -- factabs is F from our lambda calculus example
factabs fact x = x * fact (x-1)

(fix factabs) 5 -- evaluates to 120
```

The application of `fix` does not loop infinitely, because of lazy evaluation; e.g. in the expansion of `fix (const 9)` as `(const 9)(fix f)` the subexpression `fix f` is not evaluated. In contrast, the definition of fix from Haskell loops forever when applied in a strict programming language, because the argument to f is expanded beforehand, yielding an infinite call sequence `f (f ... (fix f) ... ))`, which causes a stack overflow in most implementations.

In a strict language like OCaml, we can avoid the infinite recursion problem by forcing the use of a closure. The strict version of `fix` shall have the type $\forall a.\forall b.((a\to b)\to(a\to b))\to(a\to b)$. In other words, it works only on a function that itself takes and returns a function. For example, the following OCaml code implements a strict version of `fix`:

```
let rec fix f x = f (fix f) x (* note the extra x *)

let factabs fact = function (* factabs now has extra level of lambda
abstraction *)
 0 -> 1
 | x -> x * fact (x-1)

let _ = (fix factabs) 5 (* evaluates to "120" *)
```

The same idea can be used to implement a (monomorphic) fixed combinator in strict languages that support inner classes inside methods (called local inner classes in Java), which are used as 'poor man's closures' in this case. Even when such classes may be anonymous, as in the case of Java, the syntax is still cumbersome. Java code. Function objects, e.g. in C++, simplify the calling syntax, but they still have to be generated, preferably using a helper function such as boost::bind. C++ code.

Example of encoding via recursive types

In programming languages that support recursive types, it is possible to type the Y combinator by appropriately accounting for the recursion at the type level. The need to self-apply the variable x can be managed using a type (Rec a), which is defined so as to be isomorphic to (Rec a -> a).

For example, in the following Haskell code, we have In and out being the names of the two directions of the isomorphism, with types:

```
In :: (Rec a -> a) -> Rec a
out :: Rec a -> (Rec a -> a)
```

which lets us write:

```
newtype Rec a = In { out :: Rec a -> a }

y :: (a -> a) -> a
y = \f -> (\x -> f (out x x)) (In (\x -> f (out x x)))
```

Or equivalently in OCaml:

```
type 'a recc = In of ('a recc -> 'a)
```

```
let out (In x) = x

let y f = (fun x a -> f (out x x) a) (In (fun x a -> f (out x x) a))
```

Anonymous recursion by other means

Although fixed point combinators are the standard solution for allowing a function not bound to an identifier to call itself, some languages like Javascript provide a syntactical construct that allows anonymous functions to refer to themselves. For example, in Javascript one can write:, although it has been removed in the strict mode of the latest standard edition.

```
function(x) {
  return x === 0 ? 1 : x * arguments.callee(x-1);
}
```

Chapter 9

Binary Lambda Calculus

Binary lambda calculus (**BLC**) is a technique for using the lambda calculus to study Kolmogorov complexity, by working with a standard binary encoding of lambda terms, and a designated universal machine. Binary lambda calculus is a new idea introduced by John Tromp in 2008.

Background

BLC is designed to provide a very simple and elegant concrete definition of descriptional complexity (Kolmogorov complexity).

Roughly speaking, the complexity of an object is the length of its shortest description.

To make this precise, we take descriptions to be bitstrings, and identify a description method with some computational device, or machine, that transforms descriptions into objects. Objects are usually also just bitstrings, but can have additional structure as well, e.g., pairs of strings.

Originally, Turing machines were used for this purpose, being perhaps the most well known formalism for computation. But they are somewhat lacking in ease of construction and composability. Another classical computational formalism, the Lambda calculus, offers distinct advantages in ease of use. The lambda calculus doesn't incorporate any notion of I/O though, so one needs to be designed.

Binary I/O

Adding a readbit primitive function to lambda calculus, as Chaitin did for LISP, would destroy its referential transparency, unless one distinguishes between an I/O action and its result, as Haskell does with its monadic I/O. But that requires a type system, an unnecessary complication.

Instead, BLC requires translating bitstrings into lambda terms, to which the machine (itself a lambda term) can be readily applied.

Bits 0 and 1 are translated into the standard lambda booleans B_0 = True and B_1 = False:

True = $\lambda x\, \lambda y.\, x$
False = $\lambda x\, \lambda y.\, y$

which can be seen to directly implement the if-then-else operator.

Now consider the pairing function

$$\langle , \rangle = \lambda x\, \lambda y\, \lambda z.\, zxy$$

applied to two terms *M* and *N*

$$\langle M, N \rangle = \lambda z.\, zMN.$$

Applying this to a boolean yields the desired component of choice.

A string $s = b_0 b_1 \ldots b_{n-1}$ is represented by repeated pairing as

$\langle B_0, \langle B_1 \ldots \langle B_{n-1}, z \rangle \ldots \rangle \rangle$ which is denoted as $s : z$.

The *z* appearing in the above expression requires some further explanation.

Delimited versus undelimited

Descriptional complexity actually comes in two distinct flavors, depending on whether the input is considered to be delimited.

Knowing the end of your input makes it easier to describe objects. For instance, you can just copy the whole input to output. This flavor is called *plain* or *simple* complexity.

But in a sense it is additional information. A file system for instance needs to separately store the length of files. The C language uses the null character to denote the end of a string, but this comes at the cost of not having that character available within strings.

The other flavor is called *prefix* complexity, named after prefix codes, where the machine needs to figure out, from the input read so far, whether it needs to read more bits. We say that the input is self-delimiting. This works better for communication channels, since one can send multiple descriptions, one after the other, and still tell them apart.

In the I/O model of BLC, the flavor is dictated by the choice of *z*. If we keep it as a free variable, and demand that it appears as part of the output, then the machine must be working in a self-delimiting manner. If on the other hand we use a lambda term specifically designed to be easy to distinguish from any pairing, then the input becomes delimited. BLC chooses *False* for this purpose but gives it the more descriptive alternative name of *Nil*. Dealing with lists that may be Nil is straightforward: since

$$\langle x, y \rangle \; M \; N = M \; x \; y \; N$$, and
$$Nil \; M \; N = N$$

one can write function *M* and *N* to deal with the two cases, the only caveat being that *N* will be passed to *M* as its third argument.

Universality

We can find a description method *U* such that for any other description method *D*, there is a constant *c* (depending only on *D*) such that no object takes more than *c* extra bits to describe with method *U* than with method *D*. And BLC is designed to make these constants relatively small. In fact the constant will be the length of a binary encoding of a *D*-interpreter written in BLC, and *U* will be a lambda term that parses this encoding and runs this decoded interpreter on the rest of the input. *U* won't even have to know whether the description are delimited or not; it works the same either way.

Having already solved the problem of translating bitstring into lambda calculus, we now face the opposite problem: how to encode lambda terms into bitstrings?

Lambda encoding

First we need to write our lambda terms in a particular notation using what is known as De Bruijn indices. Our encoding is then defined recursively as follows

$$\widehat{\lambda M} = 00\widehat{M}$$
$$\widehat{M \; N} = 01\widehat{M}\widehat{N}$$
$$\widehat{i} = 1^i0$$

For instance, the pairing function λ*x*λ*y*λ*z*.*zxy* is written λλλ.132 in De Bruijn format, which has encoding $00\ 00\ 00\ 01\ 01\ 10\ 1110\ 110$.

A **closed** lambda term is one in which all variables are bound, i.e. without any free variables. In De Bruijn format, this means that an index *i* can only appear within at least *i* nested lambdas. The number of closed terms of size *n* bits is given by sequence A114852 of the On-Line Encyclopedia of Integer Sequences.

The shortest possible closed term is the identity function $\widehat{\lambda 1} = 0010$. In delimited mode, this machine just copies its input to its output.

So, what is this universal machine *U*?

Here it is, in De Bruijn format (all indices are single digit):

(λ11)(λ(λλλ1(λλ1(λ3(6(λ2(6(λλ3(λ123)))(7(λ7(λ31(21))))))(1(5(λ12))(λ7(λ7(λ2(14)))3)))))(11))(λ1((λ11)(λ11)))

This is in binary:

```
0101000110100001000000001100000001100001011110011111110000101
1100111111100000001111000010110110111001111111110000111111110
0001011110100111010010110011111100001101100001011111110000
1111111100001110011011110111001101000011001000110100001101010
```

A detailed analysis of machine *U* may be found in .

Complexity, concretely

In general, we can make complexity of an object conditional on several other objects that are provided as additional argument to the universal machine. Plain (or simple) complexity *KS* and prefix complexity *KP* are defined by

$$\begin{aligned} KS(x|y_1,\ldots,y_k) &= \min\{\ell(p) \mid U\ (p:Nil)\ y_1\ \ldots\ y_k = \ \ x \quad\ \} \\ KP(x|y_1,\ldots,y_k) &= \min\{\ell(p) \mid U\ (p:\ \ z\ \ )\ y_1\ \ldots\ y_k = \langle x,z\rangle\} \end{aligned}$$

Theorems, concretely

The identity program λ1 proves that

$$KS(x) \leq \ell(x) + 4$$

The program
λλ(λ11)(λ(λλλ1(λλλλ1(λ8(λ8(λ132))((λ11)(λ(λλ1(λλ2(1(λλλλ1(λλ1)(85))1)(λ1(λλ2)2)))(1 1))6)))(21))(11))(λλλ132)12 proves that

$$KP(x|\ell(x)) \leq \ell(x) + 239$$

The program
(λ11)(λλλ1(λ1(3(λλ1))(44((λ11)(λ(λλλ1(λλλλ1(λ8(λ8(λ132))((λ11)(λ(λλ1(λλ2(1(λλλλ1(λ λ1)(85))1)
(λ1(λλ2)2)))(11))6)))(21))(11))(λ4((λ11)(λλλ2(λλλ4(λλ512)(3(λ1)(66)2))(1(λλ2)2))1))))))
(λλλ132) proves that

$$KP(x) \leq \ell(\overline{x}) + 401$$

where $\overline{x}$ is a self-delimiting code for *x* defined by

$$\begin{aligned} \overline{0} \quad &= 0 \\ \overline{n+1} \quad &= 1\ \overline{\ell(n)}\ n \end{aligned}$$

in which we identify numbers and bitstrings according to lexicographic-in-reverse order. This code has the nice property that for all *k*,

$$\ell(\overline{n}) \leq \ell(n) + \ell(\ell(n)) + \cdots + \ell^{k-1}(n) + O(\ell^k(n))$$

Number	**String**	**Delimited**
0		0
1	0	10
2	1	110 0
3	00	110 1
4	10	1110 0 00
5	01	1110 0 10
6	11	1110 0 01
7	000	1110 0 11
8	100	1110 1 000
9	010	1110 1 100

Symmetry of information

The program

```
(λ(λ(λλ(λ12(λλ31(54)(λλλ1(λ164)2)))((λ11)(λ(λλλ1(λλ1(λ3(6(λ2(6(λλ3(λ123)))(7
(λ7(λ31(21))))))(1(5(λ12))
(λ7(λ7(λ2(14)))3))))))(11))(λ11)))((λ11)(λ(λλλ1(λλ1(λ(λ4(7(λλ4(8(λ4(31))(λλλ34)
)(9(λλ10(λ6(5(31)))(λλλ264)))))
(λ(λ11)(λλλ1(λ1(3(λ1(λ1)))(44(λ4(161)))))(λ8(λ4(2(λλ2)31))(λλλ14))4))(λλ15(λ1
43))))))(11))(λλλ3((λ11)(λλλ
λλ(λ11)(λ(λλλ2(λλλλ3(λ85(λ127)))(λλ523(λ1(624))(λλλ1)(λλλ1))(λ1(λλ1(λλ2))2)
)(11))(λλλ2
(λλλ210(9(λλλ111)))(12(λλ1)))(λλλλ2)22(1(λ665(λ5(λλλ3(λλλ2(λλλ2(λλλ1(λ1(λ1
)))(2011))7)))(113))))
2(λ1)(λ1(λ1))1(λλ2)1(λλ1)))))(λλλλ3(λλλ3(λλλ1(1011(λ1)(λ1(λ1)))))))(λλλ1(λ4(λ
4(λλ14(32)))))
```

proves that

$$KP(x, y) \leq KP(x) + KP(y|x^*) + 1388$$

where x^* is a shortest program for *x*.

This inequality is the easy half of an equality (up to a constant) known as **Symmetry of information**. Proving the converse

$$KP(y|x^*) \leq KP(x, y) - KP(x) + O(1)$$

involves simulating infinitely many programs in dovetailing fashion, seeing which ones halt and output the pair of *x* (for which a shortest program is given) and any *y*, and for each such program *p*, requesting a new codeword for *y* of length $\ell(p) - KP(x)$. The Kraft inequality ensures that this infinite enumeration of requests can be fulfilled, and in fact codewords for *y* can be decoded on the fly, in tandem with the above enumeration. Details of this fundamental result by Chaitin can be found in .

A quine

The term *Q* = λ(λ11)(λλ1(λλλλ14(6632)))11 concatenates two copies of its input, proving that

$$KS(xx) \leq \ell(x) + 74$$

Applying it to its own encoding gives a 148 bit quine:

$$U(\hat{Q}\hat{Q} : Nil) = \hat{Q}\hat{Q}$$

Compression

So far, we've seen surprisingly little in the way of actually compressing an object into a shorter description; in the last example, we were only breaking even. For $\ell(x) > 74$though, we do actually compress *xx* by $\ell(x) - 74$bits.

What could be the shortest program that produces a larger output? The following is a good candidate: the program (λ1111(λλ1(λλ1)2))(λλ2(21)), of size 55 bits, uses Church numerals to output exactly $2^{2^{2^2}} = 65536$ones. That beats both gzip and bzip2, compressors that need 360 and 352 bits respectively, to describe the same output (as a 8192 byte file with a single letter name).

Halting probability

The halting probability of the prefix universal machine is defined as the probability it will output any term that has a normal form (this includes all translated strings):

$$\Omega_\lambda = \sum_{U(p:z)=\langle x,z\rangle,\ x\in NF} 2^{-\ell(p)}$$

With some effort, we can determine the first 4 bits of this particular number of wisdom:

$$\Omega_\lambda = .0001\ldots_2$$

where probability $.0001_2 = 2^{-4}$ is already contributed by programs *00100* and *00101* for terms True and False.

BLC8: byte sized I/O

While bit streams are nice in theory, they fare poorly in interfacing with the real world. The language BLC8 is a more practical variation on BLC in which programs operate on a stream of bytes, where each byte is represented as a delimited list of 8 bits in little-endian order.

BLC8 requires a more complicated universal machine:

*U*8 = (λ11)(λ(λλλ1(λλλ2(λλλ(λ7(10(λ5(2(λλ3(λ123)))(11(λ3(λ31(21)))))3)
(4(1(λ15)3)(10(λ2(λ2(16)))6)))8)(λ1(λ87(λ162))))(λ1(43)))(11))(λλ2((λ11)(λ11)))(λλ1)

The parser in U8 keeps track of both remaining bytes, and remaining bits in the current byte, discarding the latter when parsing is completed. Thus the size of U8, which is 357 bits as a BLC program, is 45 bytes in BLC8.

Brainfuck

The following BLC8 program

> (λ(λ(λ11)(λ(λλλ1(λ1(λλ1(λλ1(λλ1)(λλ1)(λλ(λ3(1(7(5(λλλλ1(λ6(λ1)143))(λλ2(λλ
> 132)1))(λλ2(λ1)((λ11)
> (λλ(λλ1(λλλ4(1(λλ1)(76(λ1)3))(1(λλ2)(7(λ1)63)))1)(221))71))))(7(1(5(λλλλ2(λλ
> 6(λ1)21(λ164)))(λλλλ1(λ5(λ1)1
> (λ154)))))(5(10(λ1(λ1)))(11(λ2((λ11)(λλλ(λ11)(λλ1(λλλ3(552)(λλ2))1)1(3(λ554(
> λ4(λ3(21)))))(2(λ1))1)1))))))
> (λ11(λ11(λ3(λ3(λ3(λ3(21)))))))))))(11))(λ1(λλλ4)322))((λ11)(λλλ12(332))1))((λ
> λ2(2(21)))(λλ2(21))(λλ1(λλ2)2)(λλ1))

is an unbounded tape Brainfuck interpreter in 936 bits (or, equivalently, 117 bytes). The formatting obscures the fact that line 3 has two double digit indices (10 and 11), while line 4 starts with another 2 occurrences of 11.

This provides a nice example of the property that universal description methods differ by at most a constant in complexity. Writing a BLC8 interpreter in Brainfuck, which would provide a matching upper bound in the other direction, is left as an exercise for die-hard Brainfuck programmers.

The interpreter expects its input to consist of a Brainfuck program followed by a `]` followed by the input for the Brainfuck program. The interpreter only looks at bits 0,1,4 of each character to determine which of `,-.+<>][` it is, so any characters other than those 8 should be stripped from a Brainfuck program. Any unconsumed input is appended to the output of the Brainfuck program. Consuming more input than is available results in an error (the non-list result λ*x*.*x*).

To conform to the more standard behaviour of not changing the current cell on reaching EOF, replace the fragment 1(λ6(λ1)143) on line 1 with (λ21(1521))(λλλ8(λ1)3652) at a cost of 45 more bits.

This interpreter is a rough translation of the following version written in Haskell

```
import Text.ParserCombinators.Parsec

putc = do (      _,       _,b,       _) <- get; tell [b]
getc = do (  left,  right,_,b:input) <- get; put (  left,  right,
b,input)
prev = do (l:left,  right,b,  input) <- get; put (  left,b:right,
l,input)
next = do (  left,r:right,b,  input) <- get; put (b:left,  right,
r,input)
decr = do (  left,  right,b,  input) <- get; put (  left,  right,pred
b,input)
incr = do (  left,  right,b,  input) <- get; put (  left,  right,succ
b,input)
loop body = do (_,_,b,_) <- get; when (b /= '\0') (body >> loop body)
parseInstr = liftM loop (between (char '[') (char ']') parseInstrs)
             <|> (char '<' >> return prev)
             <|> (char '>' >> return next)
             <|> (char '-' >> return decr)
             <|> (char '+' >> return incr)
             <|> (char '.' >> return putc)
             <|> (char ',' >> return getc)
             <|> (noneOf "]" >> return (return ()))
parseInstrs = liftM sequence_ (many parseInstr)
main = do [name] <- getArgs
          source <- readFile name
          let init = (repeat '\0', repeat '\0', '\0', repeat  '\0')
          putStrLn $ either show (execWriter . (`evalStateT` init))
(parse parseInstrs name source)
```

Chapter 10

Anonymous Function

In computing, an **anonymous function** is a function (or a subroutine) defined, and possibly called, without being bound to an identifier.

Anonymous functions originate in the work of Alonzo Church in his invention of the lambda calculus in 1936 (prior to electronic computers), in which all functions are anonymous. The Y combinator can be utilised in these circumstances to provide anonymous recursion, which Church used to show that some mathematical questions are unsolvable by computation. (Note: this result was disputed at the time, and later Alan Turing - who became Church's student - provided a proof that was more generally accepted.)

Anonymous functions have been a feature of programming languages since Lisp in 1958. An increasing number of modern programming languages support anonymous functions, and some notable mainstream languages have recently added support for them, the most widespread being JavaScript; also C# and PHP support anonymous functions. Anonymous functions were added to the C++ language as of C++0x

Some object-oriented programming languages have anonymous classes, which are a similar concept, but do not support anonymous functions. Java is such a language.

Uses

Anonymous functions can be used to contain functionality that need not be named and possibly for short-term use. Some notable examples include closures and currying.

All of the code in the following sections is written in Python.

Sorting

When attempting to sort in a non-standard way it may be easier to contain the comparison logic as an anonymous function instead of creating a named function. Most languages provide a generic sort function that implements a sort algorithm that will sort arbitrary objects. This function usually accepts an arbitrary comparison function that is supplied two items and the function indicates if they are equal or if one is "greater" or "less" than the other (typically indicated by returning a negative number, zero, or a positive number).

Consider sorting items in a list by the name of their class (in Python, everything has a class):

```
a = [10, '10', 10.0]
a.sort(lambda x,y: cmp(x.__class__.__name__, y.__class__.__name__))
print a
[10.0, 10, '10']
```

Note that `10.0` has class name "`float`", `10` has class name "`int`", and `'10'` has class name "`str`". The sorted order is "`float`", "`int`", then "`str`".

The anonymous function in this example is the lambda expression:

```
lambda x,y: cmp(...)
```

The anonymous function accepts two arguments, `x` and `y`, and returns the comparison between them using the built-in function `cmp()`. Another example would be sorting a list of strings by length of the string:

```
a = ['three', 'two', 'four']
a.sort(lambda x,y: cmp(len(x), len(y)))
print a
['two', 'four', 'three']
```

which clearly has been sorted by length of the strings.

Closures

Closures are functions evaluated in an environment containing bound variables. The following example binds the variable "threshold" in an anonymous function that compares the input to the threshold.

```
def comp(threshold):
  return lambda x: x < threshold
```

This can be used as a sort of generator of comparison functions:

```
a = comp(10)
b = comp(20)

print a(9), a(10), a(20), a(21)
True False False False

print b(9), b(10), b(20), b(21)
True True False False
```

It would be very impractical to create a function for every possible comparison function and may be too inconvenient to keep the threshold around for further use. Regardless of the reason why a closure is used, the anonymous function is the entity that contains the functionality that does the comparing.

Currying

In mathematics and computer science, **currying** is the technique of transforming a function that takes multiple arguments (or an n-tuple of arguments) in such a way that it can be called as a chain of functions each with a single argument. It was invented by Moses Schönfinkel and later re-invented by Haskell Curry,

Uncurrying is the dual transformation to currying, and can be seen as a form of defunctionalization. It takes a function $f(\boldsymbol{x})$ which returns another function $g(\boldsymbol{y})$ as a result, and yields a new function $f'(\boldsymbol{x}, \boldsymbol{y})$ which takes a number of additional parameters and applies them to the function returned by *f*. The process can be iterated if necessary.

Motivation

Currying is similar to the process of calculating a function of multiple variables for some given values on paper. For example, given the function $f(x,y) = y / x$:

To evaluate $f(2,3)$, first replace *x* with 2.
Since the result is a function of *y*, this function $g(y)$ can be defined as $g(y) = f(2,y) = y / 2$.
Next, replace the *y* argument with 3, producing $g(3) = f(2,3) = 3 / 2$.

On paper, using classical notation, this is usually done all in one step. However, each argument can be replaced sequentially as well. Each replacement results in a function taking exactly one argument. This produces a chain of functions as in lambda calculus, and multi-argument functions are usually represented in curried form.

Some programming languages almost always use curried functions to achieve multiple arguments; notable examples are ML and Haskell, where in both cases all functions have exactly one argument.

This is similar in computer code: If we let f be a function $f(x,y) = y / x$, then the function $g_x(y) = (y \mapsto f(x,y))$is a curried version of f. In particular, $g_{(2)}(y) = (y \mapsto f(2,y))$is the curried equivalent of the example above. Though note that currying, while similar, is not the same operation as partial function application.

Definition

Given a function *f* of type $f\colon (X \times Y) \to Z$, **currying** it makes a function $\text{curry}(f)\colon X \to (Y \to Z)$. That is, curry(*f*) takes an argument of type *X* and returns a function of type $Y \to Z$. **Uncurrying** is the reverse transformation, and is most easily understood in terms of its right adjoint, apply.

The → operator is often considered right-associative, so the curried function type $X \to (Y \to Z)$is often written as $X \to Y \to Z$. Conversely, function application is considered to be left-associative, so that $f\ \langle x, y \rangle$is equivalent to $\mathrm{curry}(f)\ x\ y$.

Curried functions may be used in any language that supports closures; however, uncurried functions are generally preferred for efficiency reasons, since the overhead of partial application and closure creation can then be avoided for most function calls.

Mathematical view

In theoretical computer science, currying provides a way to study functions with multiple arguments in very simple theoretical models such as the lambda calculus in which functions only take a single argument.

In a set-theoretic paradigm, currying is the natural code between the set $A^{B \times C}$ of functions from $B \times C$to A, and the set $(A^B)^C$ of functions from C to the set of functions from B to A. In category theory, currying can be found in the universal property of an exponential object, which gives rise to the following adjunction in cartesian closed categories: There is a natural isomorphism between the morphisms from a binary product $f\colon (X \times Y) \to Z$and the morphisms to an exponential object $g\colon X \to Z^Y$. In other words, currying is the statement that product and Hom are adjoint functors; that is there is a natural transformation:

$$\hom(A \times B, C) \cong \hom(A, \hom(B, C)).$$

This is the key property of being a Cartesian closed category.

Under the Curry-Howard correspondence, the existence of currying and uncurrying is equivalent to the logical theorem $(A \wedge B) \to C \Leftrightarrow A \to (B \to C)$, as tuples (product type) corresponds to conjunction in logic, and function type corresponds to implication.

Curry is a continuous function in the Scott topology.

Naming

The name "currying", coined by Christopher Strachey in 1967, is a reference to logician Haskell Curry. The alternative name "Schönfinkelisation", has been proposed as a reference to Moses Schönfinkel.

Contrast with partial function application

Currying and partial function application are often conflated. The difference between the two is clearest for functions taking more than two variables.

Given a function of type $f\colon (X \times Y \times Z) \to N$, currying produces $\text{curry}(f)\colon X \to (Y \to (Z \to N))$. That is, while an evaluation of the first function might be represented as *f*(1,2,3), evaluation of the curried function would be represented as $f_{curried}(1)(2)(3)$, applying each argument in turn to a single-argument function returned by the previous invocation. Note that after calling $f_{curried}(1)$, we are left with a function that takes a single argument and returns another function, not a function that takes two arguments.

In constrast, **partial function application** refers to the process of fixing a number of arguments to a function, producing another function of smaller arity. Given the definition of *f* above, we might fix (or 'bind') the first argument, producing a function of type $\text{partial}(f)\colon (Y \times Z) \to N$. Evaluation of this function might be represented as $f_{partial}(2,3)$. Note that the result of partial function application in this case is a function that takes two arguments.

Intuitively, partial function application says "if you fix the first arguments of the function, you get a function of the remaining arguments". For example, if function *div* stands for the division operation *x* / *y*, then *div* with the parameter *x* fixed at 1 (i.e. *div* 1) is another function: the same as the function *inv* that returns the multiplicative inverse of its argument, defined by *inv*(*y*) = 1 / *y*.

The practical motivation for partial application is that very often the functions obtained by supplying some but not all of the arguments to a function are useful; for example, many languages have a function or operator similar to `plus_one`. Partial application makes it easy to define these functions, for example by creating a function that represents the addition operator with 1 bound as its first argument.

Higher-order functions

Map

The map function performs a function call on each element of an array. The following example squares every element in an array with an anonymous function.

```
a = [1, 2, 3, 4, 5, 6]
print map(lambda x: x*x, a)
[1, 4, 9, 16, 25, 36]
```

The anonymous function accepts an argument and multiplies it by itself (squares it).

Filter

The filter function returns all elements from a list that evaluate True when passed to a certain function.

```
a = [1, 2, 3, 4, 5, 6]
print filter(lambda x: x % 2 == 0, a)
[2, 4, 6]
```

List of languages

The following is a list of programming languages that fully support unnamed anonymous functions; support some variant of anonymous functions; and have no support for anonymous functions.

This table shows some general trends. First, the languages that do not support anonymous functions—C, Pascal, Object Pascal, Java—are all conventional statically-typed languages. This does not, however, mean that statically-typed languages are incapable of supporting anonymous functions. For example, the ML languages are statically typed and fundamentally include anonymous functions, and Delphi, a dialect of Object Pascal, has been extended to support anonymous functions. Second, the languages that treat functions as first-class functions—Dylan, JavaScript, Lisp, Scheme, ML, Haskell, Python, Ruby, Perl—generally have anonymous function support so that functions can be defined and passed around as easily as other data types. However, the coming C++0x standard adds them to C++, even though this is conventional, statically typed language.

Language	Support	Notes
ActionScript	✓	
C	✗	
C#	✓	
C++	✗	Planned for C++0x
Curl	✓	
D	✓	
Delphi	✓	Starting with Delphi 2009.
Dylan	✓	
Erlang	✓	
Frink	✓	
Haskell	✓	
Java	✗	Planned for Java 8 or later
JavaScript	✓	
Lisp	✓	
Logtalk	✓	
Lua	✓	
Mathematica	✓	

ML languages (Objective Caml, Standard ML, etc.)	✓	
Octave	✓	
Object Pascal	✓	Delphi, a dialect of Object Pascal, implements support for anonymous functions (formally, *anonymous methods*) natively. The Oxygene Object Pascal dialect also supports them.
Objective-C (Mac OS X 10.6+)	✓	called blocks
Pascal	✗	
Perl	✓	
PHP	✓	As of PHP 5.3.0, true anonymous functions are supported; previously only partial anonymous functions were supported, which worked much like C#'s implementation.
Python	✓	
R	✓	
Ruby	✓	Ruby's anonymous functions, inherited from Smalltalk, are called blocks.
Scala	✓	
Scheme	✓	
Smalltalk	✓	Smalltalk's anonymous functions are called blocks.
Visual Basic .NET v9	✓	
Visual Prolog v 7.2	✓	
Vala	✓	

Examples

Numerous languages support anonymous functions, or something similar.

C# lambda expressions

Support for anonymous functions in C# has deepened through the various versions of the language compiler. The C# language v3.0, released in November 2007 with the .NET Framework v3.5, has full support of anonymous functions. C# refers to them as "lambda expressions", following the original version of anonymous functions, the Lambda Calculus.

```
//the first int is the x' type
//the second int is the return type
```

```
//<see href="http://msdn.microsoft.com/en-us/library/bb549151.aspx" />
Func<int,int> foo = x => x*x;
Console.WriteLine(foo(7));
```

While the function is anonymous, it cannot be assigned to an implicitly typed variable, because the lambda syntax may be used to denote an anonymous function or an expression tree, and the choice cannot automatically be decided by the compiler. E.g., this does not work:

```
// will NOT compile!
var foo = (int x) => x*x;
```

However, a lambda expression can take part in type inference and can be used as a method argument, e.g. to use anonymous functions with the Map capability available with `System.Collections.Generic.List` (in the `ConvertAll()` method):

```
// Initialize the list:
var values = new List<int>() { 7, 13, 4, 9, 3 };
// Map the anonymous function over all elements in the list, return the
new list
var foo = values.ConvertAll(d => d*d) ;
// the result of the foo variable is of type
System.Collections.Generic.List<Int32>
```

Prior versions of C# had more limited support for anonymous functions. C# v1.0, introduced in February 2002 with the .NET Framework v1.0, provided partial anonymous function support through the use of delegates. This construct is somewhat similar to PHP delegates. In C# 1.0, Delegates are like function pointers that refer to an explicitly named method within a class. (But unlike PHP the name is not required at the time the delegate is used.) C# v2.0, released in November 2005 with the .NET Framework v2.0, introduced the concept of anonymous methods as a way to write unnamed inline statement blocks that can be executed in a delegate invocation. C# 3.0 continues to support these constructs, but also supports the lambda expression construct.

This example will compile in C# 3.0, and exhibits the three forms:

```
  public class TestDriver
  {
    delegate int SquareDelegate(int d);
    static int Square(int d)
    {
      return d*d;
    }

    static void Main(string[] args)
    {
      // C# 1.0: Original delegate syntax required
      // initialization with a named method.
      SquareDelegate A = new SquareDelegate(Square);
      System.Console.WriteLine(A(3));
      // C# 2.0: A delegate can be initialized with
      // inline code, called an "anonymous method." This
      // method takes an int as an input parameter.
      SquareDelegate B = delegate(int d) { return d*d; };
      System.Console.WriteLine(B(5));

      // C# 3.0. A delegate can be initialized with
      // a lambda expression. The lambda takes an int, and returns an
int.
      // The type of x is inferred by the compiler.
      SquareDelegate C = x => x*x;
      System.Console.WriteLine(C(7));

      // C# 3.0. A delegate that accepts a single input and
      // returns a single output can also be implicitly declared with
the Func<> type.
      System.Func<int,int> D = x => x*x;
      System.Console.WriteLine(D(9));
    }
  }
```

In the case of the C# 2.0 version, the C# compiler takes the code block of the anonymous function and creates a static private function. Internally, the function gets a generated name, of course; this generated name is based on the name of the method in which the Delegate is declared. But the name is not exposed to application code except by using reflection.

In the case of the C# 3.0 version, the same mechanism applies.

D

```
(x){return x*x;}
```

Delphi

```
program demo;

type
  TSimpleProcedure = reference to procedure;
  TSimpleFunction = reference to function(x: string): Integer;

var
  x1: TSimpleProcedure;
  y1: TSimpleFunction;

begin
  x1 := procedure
    begin
      Writeln('Hello World');
    end;
  x1;   //invoke anonymous method just defined

  y1 := function(x: string): Integer
```

```
    begin
      Result := Length(x);
    end;
  Writeln(y1('bar'));
end.
```

Erlang

Erlang uses a syntax for anonymous functions similar to that of named functions.

```
% Anonymous function bound to the Square variable
Square = fun(X) -> X * X end.

% Named function with the same functionality
square(X) -> X * X.
```

Haskell

Haskell uses a concise syntax for anonymous functions (lambda expressions).

```
\x -> x * x
```

Lambda expressions are fully integrated with the type inference engine, and support all the syntax and features of "ordinary" functions (except for the use of multiple definitions for pattern-matching, since the argument list is only specified once).

```
map (\x -> x * x) [1..5] -- returns [1, 4, 9, 16, 25]
```

The following are all equivalent:

```
f x y = x + y
f x = \y -> x + y
f = \x y -> x + y
```

JavaScript

JavaScript supports anonymous functions.

```
alert((function(x){
        return x*x;
})(10));
```

This construct is often used in Bookmarklets. For example, to change the title of the current document (visible in its window's title bar) to its URL, the following bookmarklet may seem to work.

```
javascript:document.title=location.href;
```

However, as the assignment statement returns a value (the URL itself), many browsers actually create a new page to display this value.

Instead, an anonymous function can be used so that no value is returned:

```
javascript:(function(){document.title=location.href;})();
```

The function statement in the first (outer) pair of parentheses declares an anonymous function, which is then executed when used with the last pair of parentheses. This is equivalent to the following.

```
javascript:var f = function(){document.title=location.href;}; f();
```

Lisp

Lisp and Scheme support anonymous functions using the "lambda" construct, which is a reference to lambda calculus. Clojure supports anonymous functions with the "fn" special form and #() reader macro.

```
(lambda (arg) (* arg arg))
```

Interestingly, Scheme's "named functions" is simply syntactic sugar for anonymous functions bound to names:

```
(define (somename arg)
  (do-something arg))
```

expands (and is equivalent) to

```
(define somename
  (lambda (arg)
    (do-something arg)))
```

Clojure supports anonymous functions through the "fn" special form:

```
(fn [x] (+ x 3))
```

There is also a reader macro to define a lambda:

```
# (+ % %2 %3) ; Defines an anonymous function that takes three
arguments and sums them.
```

Like Scheme, Clojure's "named functions" are simply syntactic sugar for lambdas bound to names:

```
(defn func [arg] (+ 3 arg))
```

expands to:

```
(def func (fn [arg] (+ 3 arg)))
```

Logtalk

Logtalk uses the following syntax for anonymous predicates (lambda expressions):

```
{FreeVar1, FreeVar2, ...}/[LambdaParameter1, LambdaParameter2,
...]>>Goal
```

A simple example with no free variables and using a list mapping predicate is:

```
| ?- meta::map([X,Y]>>(Y is 2*X), [1,2,3], Ys).
Ys = [2,4,6]
yes
```

Currying is also supported. The above example can be written as:

```
| ?- meta::map([X]>>([Y]>>(Y is 2*X)), [1,2,3], Ys).
Ys = [2,4,6]
yes
```

Lua

In Lua (much as in Scheme) all functions are anonymous. A "named function" in Lua is simply a variable holding a reference to a function object .

Thus, in Lua

```
function foo(x) return 2*x end
```

is just syntactical sugar for

```
foo = function(x) return 2*x end
```

An example of using anonymous functions for reverse-order sorting:

```
table.sort(network, function(a,b)
  return a.name > b.name
end)
```

ML

The various dialects of ML support anonymous functions.

Objective Caml:

```
fun arg -> arg * arg
```

Standard ML:

```
fn arg => arg * arg
```

Octave

```
 (@(x)x*x)(2)
ans =  4
```

Perl 5

Perl 5 supports anonymous functions, as follows:

```
(sub { print "I got called\n" })->();         # 1. fully anonymous,
called as created

my $squarer = sub { my $x = shift; $x * $x }; # 2. assigned to a
variable

sub curry {
    my ($sub, @args) = @_;
    return sub { $sub->(@args, @_) };         # 3. as a return value of
another function
}

# example of currying in Perl
sub sum { my $tot = 0; $tot += $_ for @_; $tot } # returns the sum of
its arguments
my $curried = curry \&sum, 5, 7, 9;
print $curried->(1,2,3), "\n";    # prints 27 ( = 5 + 7 + 9 + 1 + 2 + 3
)
```

Other constructs take "bare blocks" as arguments, which serve a function similar to lambda functions of a single parameter, but don't have the same parameter-passing convention as functions -- @_ is not set.

```
my @squares = map { $_ * $_ } 1..10;   # map and grep don't use the
'sub' keyword
my @square2 = map $_ * $_, 1..10;      # parentheses not required for a
single expression

my @bad_example = map { print for @_ } 1..10; # values not passed like
normal Perl function
```

Perl 6

In Perl 6, all blocks (even the ones associated with if, while, etc.) are anonymous functions. A block that is not used as an rvalue is executed immediately.

```
{ say "I got called" };           # 1. fully anonymous, called as
created
```

```
my $squarer1 = -> $x { $x * $x };             # 2a. assigned to a
variable, pointy block
my $squarer2 = { $^x * $^x };                 # 2b. assigned to a
variable, twigil
my $squarer3 = { my $x = shift @_; $x * $x }; # 2b. assigned to a
variable, Perl 5 style

# 3 curring
sub add ($m, $n) { $m + $n }
my $seven   = add(3, 4);
my $add_one = &add.assuming(m => 1);
my $eight   = $add_one($seven);
```

PHP

Prior to 4.0.1, PHP had no anonymous function support.

4.0.1 to 5.3

PHP 4.0.1 introduced the `create_function` which was the initial anonymous function support. This function call creates a new randomly named function and returns its name (as a string)

```
$foo = create_function('$x', 'return $x*$x;');
$bar = create_function("\$x", "return \$x*\$x;");
echo $foo(10);
```

It is important to note that the argument list and function body must be in single quotes or the dollar signs must be escaped. Otherwise PHP will assume "`$x`" means the variable `$x` and will substitute it into the string (despite possibly not existing) instead of leaving "`$x`" in the string. For functions with quotes or functions with lots of variables, it can get quite tedious to ensure the intended function body is what PHP interprets.

5.3

PHP 5.3 added a new class called `Closure` and magic method `__invoke()` that makes a class instance invocable. Lambda functions are a compiler "trick" that instantiates a new `Closure` instance that can be invoked as if the function were invokable.

```
$x = 3;
$func = function($z) { return $z *= 2; };
echo $func($x); // prints 6
```

In this example, `$func` is an instance of `Closure` and `echo $func()` is equivalent to `$func->__invoke($z)`. PHP 5.3 mimics anonymous functions but it does not support true anonymous functions because PHP functions are still not first-class functions.

PHP 5.3 does support closures but the variables must be explicitly indicated as such:

```
$x = 3;
$func = function() use(&$x) { $x *= 2; };
$func();
echo $x; // prints 6
```

The variable `$x` is bound by reference so the invocation of `$func` modifies it and the changes are visible outside of the function.

Python

Python supports simple anonymous functions through the lambda form. The executable body of the lambda must be an expression and can't be a statement, which is a restriction that limits its utility. The value returned by the lambda is the value of the contained expression. Lambda forms can be used anywhere ordinary functions can, however these restrictions make it a very limited version of a normal function. Here is an example:

```
foo = lambda x: x*x
print foo(10)
```

This example will print: 100.

In general, Python convention encourages the use of named functions defined in the same scope as one might typically use an anonymous functions in other languages. This is acceptable as locally defined functions implement the full power of closures and are almost as efficient as the use of a lambda in Python. In this example, the built-in power function can be said to have been curried:

```
def make_pow(n):
    def fixed_exponent_pow(x):
        return pow(x, n)
    return fixed_exponent_pow
sqr = make_pow(2)
print sqr(10) # Emits 100
cub = make_pow(3)
print cub(10) # Emits 1000
```

R

```
 (function(x)x*x)(2)
=> 4
```

Ruby

Ruby supports anonymous functions by using a syntactical structure called *block*. When passed to a method, a block is converted into an object of class *Proc* in some circumstances.

```
# Example 1:
# Purely anonymous functions using blocks.
```

```
ex = [16.2, 24.1, 48.3, 32.4, 8.5]
ex.sort() {|x,y| (x % x.to_i) <=> (y % y.to_i)} #sort by fractional
part, ignoring integer part.
# [24.1, 16.2, 48.3, 32.4, 8.5]

# Example 2:
# First-class functions as an explicit object of Proc -
ex = Proc.new { puts("Hello, world!") }
ex.call() # Hello, world!
```

Smalltalk

In Smalltalk anonymous functions are called blocks

Visual Basic

VB9, introduced in November 2007, supports anonymous functions through the lambda form. Combined with implicit typing, VB provides an economical syntax for anonymous functions. As with Python, in VB v9, anonymous functions must be defined on a single line; they cannot be compound statements. Further, an anonymous function in VB must truly be a VB "`Function`" - it must return a value.

```
Dim foo = Function(x) x * x
Console.WriteLine(foo(10))
```

VB10, released April 12, 2010, adds support for multiline lambda expressions and anonymous functions without a return value. For example, a function for use in a Thread.

```
Dim t As New System.Threading.Thread(Sub()
    For n as Integer = 0 to 10   'Count to 10
        Console.WriteLine(n)     'Print each number
    Next
    End Sub)
t.Start()
```

Visual Prolog

Anonymous functions (in general anonymous *predicates*) were introduced in Visual Prolog in version 7.2. Anonymous predicates can capture values from the context. If created in an object member it can also access the object state (by capturing `This`).

`mkAdder` returns an anonymous function, which has captured the argument `X` in the closure. The returned function is a function that adds `X` to its argument:

```
clauses
    mkAdder(X) = { (Y) = X+Y }.
```

Mathematica

Anonymous Functions are important in programming Mathematica. There are several ways to create them. Below are a few anonymous function that increment a number. The first is the most common. '#1' refers to the first argument and '&' makes the end of the anonymous function.

```
#1+1&
Function[x,x+1]
x \[Function] x+1
```

Additionally, Mathematica has an additional construct to for making recursive anonymous functions. The symbol '#0' refers to the entire function.

```
If[#1 == 1, 1, #1 * #0[#1-1]]&
```

Chapter 11

Apply and Director String

Apply

In mathematics and computer science, **Apply** is a function that applies functions to arguments. It is central to programming languages derived from lambda calculus, such as LISP and Scheme, and also in functional languages. In particular, it has a role in the study of the denotational semantics of computer programs, by virtue of the fact that it is a continuous function on complete partial orders.

In category theory, **Apply** is important in Cartesian closed categories, (and thus, also in Topos theory), where it is a universal morphism, right adjoint to currying.

Programming

In computer programming, **apply** applies a function to a list of arguments. *Eval* and *apply* are the two interdependent components of the *eval-apply cycle*, which is the essence of evaluating Lisp, described in SICP.

Apply function

Apply is also the name of a special function in many languages, which takes a function and a list, and uses the list as the function's own argument list, as if the function were called with the elements of the list as the arguments. This is important in languages with variadic functions, because this is the only way to call a function with an indeterminate (at compile time) number of arguments.

In Common Lisp **apply** is a function that applies a function to a list of arguments (note here that "+" is a variadic function that takes any number of arguments):

```
(apply #'+ (list 1 2))
```

Similarly in Scheme:

```
(apply + (list 1 2))
```

In C# and Java, variadic arguments are simply collected in an array; you can explicitly pass in an array in place of the variadic arguments:

```
variadicFunc(arrayOfArgs);
```

In JavaScript, function objects have an `apply` method, the first argument is the value of the `this` keyword inside the function; the second is the list of arguments:

```
func.apply(null, args);
```

In Perl, arrays, hashes and expressions are automatically "flattened" into a single list when evaluated in a list context, such as in the argument list of a function:

```
# Equivalent subroutine calls:
@args = (@some_args, @more_args);
func(@args);

func(@some_args, @more_args);
```

In PHP, `apply` is called `call_user_func_array`:

```
call_user_func_array('func_name', $args);
```

In Python and Ruby, you use the same asterisk notation used in defining variadic functions to call a function on a sequence:

```
func(*args)
```

In Lua, apply can be written as follows:

```
function apply(f,l)
  return f(unpack(l))
end
```

Universal property

Consider a function $g : (X \times Y) \to Z$, that is, $g \in [(X \times Y) \to Z]$where the bracket notation $[A \to B]$denotes the space of functions from *A* to *B*. By means of currying, there is a unique function $\text{curry}(g) : X \to [Y \to Z]$. Then **Apply** provides the universal morphism

$$\text{Apply} : ([Y \to Z] \times Y) \to Z,$$

so that

Apply(*f*,*y*) = *f*(*y*)

or, equivalently one has the commuting diagram

$$\mathrm{Apply} \circ (\mathrm{curry}(g) \times \mathrm{id}_Y) = g$$

The notation $[A \to B]$ for the space of functions from *A* to *B* occurs more commonly in computer science. In category theory, however, $[A \to B]$ is known as the exponential object, and is written as B^A. There are other common notational differences as well; for example *Apply* is often called *Eval*, even though in computer science, these are not the same thing, with eval distinguished from *Apply*, as being the evaluation of the quoted string form of a function with its arguments, rather than the application of a function to some arguments.

Also, in category theory, *curry* is commonly denoted by λ, so that λ*g* is written for *curry*(*g*). This notation is in conflict with the use of λ in lambda calculus, where lambda is used to denote free variables. With all of these notational changes accounted for, the adjointness of *Apply* and *curry* is then expressed in the commuting diagram

$$\begin{tikzcd}
X \arrow[d, dashed, "\lambda g"'] & X \times Y \arrow[d, dashed, "\lambda g \times \mathrm{id}_Y"'] \arrow[dr, "g"] & \\
Z^Y & Z^Y \times Y \arrow[r, "\mathrm{eval}"'] & Z
\end{tikzcd}$$

Topological properties

In order theory, in the category of complete partial orders endowed with the Scott topology, both *curry* and *apply* are continuous functions (that is, they are Scott continuous). This property helps establish the foundational validity of the study of the denotational semantics of computer programs.

Director string

In mathematics, in the area of lambda calculus and computation, **directors** or **director strings** are a mechanism for keeping track of the free variables in a term. Director strings were introduced by Kennaway and Sleep in 1982 and further developed by Sinot, Fernández and Mackie as a mechanism for understanding and controlling the computational complexity cost of beta reduction.

Motivation

In beta reduction, one defines the value of the expression on the left to be that on the right:

$$(\lambda x.E)y \equiv E[x := y]$$

While this is a conceptually simple operation, the computational complexity of the step can be non-trivial: a naive algorithm would scan the expression *E* for all occurrences of the free variable *x*. Such an algorithm is clearly *O*(*n*) in the length of the expression *E*. Thus, one is motivated to somehow track the occurrences of the free variables in the expression. One may attempt to track the position of *every* free variable, where-ever it may occur in the expression, but this can clearly become very costly in terms of storage; furthermore, it provides a level of detail that is not really needed. Director strings suggest that the correct model is to track free variables in a hierarchical fashion, by tracking their use in component terms.

Definition

Consider, for simplicity, a term algebra, that is, a collection of free variables, constants, and operators which may be freely combined. Assume that a term *t* takes the form

$$t ::= f(t_1, t_2, \ldots, t_n)$$

where *f* is a function, of arity *n*, with no free variables, and the t_i are terms that may or may not contain free variables. Let *V* denote the set of all free variables that may occur in the set of all terms. The director is then the map

$$\sigma_t : V \to P(\{1, 2, \ldots, n\})$$

from the free variables to the power set *P*(*X*) of the set $X = \{1, 2, \ldots, n\}$. The values taken by σ_t are simply a list of the indices of the t_i in which a given free variable occurs. Thus, for example, if a free variable $x \in V$ occurs in t_3 and t_5 but in no other terms, then one has $\sigma_t(x) = \{3,5\}$.

Thus, for every term $t \in T$ in the set of all terms *T*, one maintains a function σ_t, and instead of working only with terms *t*, one works with pairs (t,σ_t). Thus, the time complexity of finding the free variables in *t* is traded for the space complexity of maintaining a list of the terms in which a variable occurs.

General case

Although the above definition is formulated in terms of a term algebra, the general concept applies more generally, and can be defined both for combinatory algebras and for lambda calculus proper, specifically, within the framework of explicit substitution.

Chapter 12

Explicit Substitution and Lambda Lifting

Explicit substitution

In computer science, **explicit substitution** is any of several calculi based on the lambda calculus that pay special attention to the formalization of the process of substitution. The concept of explicit substitutions has become notorious (despite a large number of published calculi of explicit substitutions in the literature with quite different characteristics) because the notion often turns up (implicitly and explicitly) in formal descriptions and implementation of all the mathematical forms of substitution involving variables such as in abstract machines, predicate logic, and symbolic computation.

Basics

A simple example of a lambda calculus with explicit substitution is "λx", which adds one new form of term to the lambda calculus, namely the form $M\langle x{:=}N\rangle$, which reads "M where x will be substituted by N". (The meaning of the new term is the same as the common idiom **let** x:=N **in** M from many programming languages.) λx can be written with the following rewriting rules:

1. $(\lambda x.M)\ N \rightarrow M\langle x{:=}N\rangle$
2. $x\langle x{:=}N\rangle \rightarrow N$
3. $x\langle y{:=}N\rangle \rightarrow x\ (x{\neq}y)$
4. $(M_1M_2)\ \langle x{:=}N\rangle \rightarrow (M_1\langle x{:=}N\rangle)\ (M_2\langle x{:=}N\rangle)$
5. $(\lambda x.M)\ \langle y{:=}N\rangle \rightarrow \lambda x.(M\langle y{:=}N\rangle)\ (x{\neq}y)$

While making substitution explicit, this formulation still retains the complexity of the lambda calculus "variable convention", requiring arbitrary renaming of variables during reduction to ensure that the "(x≠y)" condition on the last rule is always satisfied before applying the rule. Therefore many calculi of explicit substitution avoid variable names altogether by using a so-called "name-free" De Bruijn index notation.

History

Explicit substitutions grew out of an ‘implementation trick’ used, for example, by AUTOMATH, and became a respectable syntactic theory in lambda calculus and rewriting theory. The idea of a specific calculus where substitutions are part of the object language, and not of the informal meta-theory, is credited to Abadi, Cardelli, Curien, and Levy. Their seminal paper on the λσ calculus explains that implementations of Lambda calculus need to be very careful when dealing with substitutions. Without sophisticated mechanisms for structure-sharing, substitutions can cause a size explosion, and therefore, in practice, substitutions are delayed and explicitly recorded. This makes the correspondence between the theory and the implementation highly non-trivial and correctness of implementations can be hard to establish. One solution is to make the substitutions part of the calculus, that is, to have a calculus of explicit substitutions.

Once substitution has been made explicit, however, the basic properties of substitution change from being semantic to syntactic properties. One most important example is the "substitution lemma", which with the notation of λx becomes

- $(M\langle x:=N\rangle)\langle y:=P\rangle = (M\langle y:=P\rangle)\langle x:=(N\langle y:=P\rangle)\rangle$ $(x\neq y)$

A surprising counterexample, due to Melliès, shows that the way this rule is encoded in the original calculus of explicit substitutions is not strongly normalizing. Following this, a multitude of calculi were described trying to offer the best compromise between syntactic properties of explicit substitution calculi.

Lambda lifting

Lambda lifting or **closure conversion** is the process of eliminating free variables from local function definitions from a computer program. The elimination of free variables allows the compiler to hoist local definitions out of their surrounding contexts into a fixed set of top-level functions with an extra parameter replacing each local variable. By eliminating the need for run-time access-links, this may reduce the run-time cost of handling implicit scope. Many functional programming language implementations use lambda lifting during compilation.

The term **lambda lifting** was first introduced by Thomas Johnsson around 1982.

Algorithm

The following algorithm is one way to lambda-lift an arbitrary program in a language which doesn't support closures as first-class objects:

1. Rename the functions so that each function has a unique name.
2. Replace each free variable with an additional argument to the enclosing function, and pass that argument to every use of the function.
3. Replace every local function definition that has no free variables with an identical global function.
4. Repeat steps 2 and 3 until all free variables and local functions are eliminated.

If the language has closures as first-class objects that can be passed as arguments or returned from other functions (closures), the closure will need to be represented by a data structure that captures the bindings of the free variables.

Example

Consider the following OCaml program that computes the sum of the integers from 1 to 100:

```
let rec sum n =
  if n = 1 then
    1
  else
    let f x =
      n + x in
    f (sum (n - 1)) in
sum 100
```

(The word `let rec` declares `sum` as a function that may call itself.) The function f, which adds sum's argument to the sum of the numbers less than the argument, is a local function. Within the definition of f, n is a free variable. Start by converting the free variable to an argument:

```
let rec sum n =
  if n = 1 then
    1
  else
    let f w x =
      w + x in
     f n (sum (n - 1)) in
sum 100
```

Next, lift f into a global function:

```
let rec f w x =
  w + x
and sum n =
  if n = 1 then
    1
  else
    f n (sum (n - 1)) in
sum 100
```

Finally, convert the functions into rewriting rules:

```
f w x → w + x
sum 1 → 1
sum n → f n (sum (n - 1)) when n ≠ 1
```

The expression "sum 100" rewrites as:

```
sum 100 → f 100 (sum 99)
 → 100 + (sum 99)
 → 100 + (f 99 (sum 98))
 → 100 + (99 + (sum 98)
 . . .
 → 100 + (99 + (98 + (... + 1 ...)))
```

The following is the same example, this time written in JavaScript:

```
// Initial version

function sum(n) {
        if(n == 1) {
                return 1;
        }
        else {
                function f(x) {
                        return n + x;
                };
                return f( sum(n - 1) );
        }
}


// After converting the free variable n to a formal parameter w

function sum(n) {
        if(n == 1) {
                return 1;
        }
        else {
                function f(w, x) {
                        return w + x;
                };
                return f( n, sum(n - 1) );
        }
}


// After lifting function f into the global scope

function f(w, x) {
        return w + x;
};

function sum(n) {
        if(n == 1) {
```

```
        return 1;
}
else {
        return f( n, sum(n - 1) );
```

Chapter 13

Normalisation by Evaluation and SKI Combinator Calculus

Normalisation by evaluation

In programming language semantics, **normalisation by evaluation (NBE)** is a style of obtaining the normal form of terms in the λ calculus by appealing to their denotational semantics. A term is first *interpreted* into a denotational model of the λ-term structure, and then a canonical (β-normal and η-long) representative is extracted by *reifying* the denotation. Such an essentially semantic approach differs from the more traditional syntactic description of normalisation as a reductions in a term rewrite system where β-reductions are allowed deep inside λ-terms.

NBE was first described for the simply typed lambda calculus. It has since been extended both to weaker type systems such as the untyped lambda calculus using a domain theoretic approach, and to richer type systems such as several variants of Martin-Löf type theory.

Outline

Consider the simply typed lambda calculus, where types τ can be basic types (α), function types (→), or products (×), given by the following BNF grammar (→ associating to the right, as usual):

(Types) $\tau ::= \alpha \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2$

These can be implemented as a datatype in the meta-language; for example, for Standard ML, we might use:

```
datatype ty = Basic of string
            | Arrow of ty * ty
            | Prod of ty * ty
```

Terms are defined at two levels. The lower *syntactic* level (sometimes called the *dynamic* level) is the representation that one intends to normalise.

(Syntax Terms) *s*,*t*,… ::= **var** *x* | **lam** (*x*, *t*) | **app** (*s*, *t*) | **pair** (*s*, *t*) | **fst** *t* | **snd** *t*

Here **lam**/**app** (resp. **pair**/**fst**,**snd**) are the intro/elim forms for → (resp. ×), and *x* are variables. These terms are intended to be implemented as a first-order in the meta-language:

```
datatype tm = var of string
            | lam of string * tm | app of tm * tm
            | pair of tm * tm | fst of tm | snd of tm
```

The denotational semantics of (closed) terms in the meta-language interprets the constructs of the syntax in terms of features of the meta-language; thus, **lam** is interpreted as abstraction, **app** as application, etc. The semantic objects constructed are as follows:

(Semantic Terms) *S*,*T*,… ::= **LAM** (λ*x*. *S x*) | **PAIR** (*S*, *T*) | **SYN** *t*

Note that there are no variables or elimination forms in the semantics; they are represented simply as syntax. These semantic objects are represented by the following datatype:

```
datatype sem = LAM of (sem -> sem)
             | PAIR of sem * sem
             | SYN of tm
```

There are a pair of type-indexed functions that move back and forth between the syntactic and semantic layer. The first function, usually written $\uparrow_\tau$, *reflects* the term syntax into the semantics, while the second *reifies* the semantics as a syntactic term (written as $\downarrow^\tau$). Their definitions are mutually recursive as follows:

$$\begin{aligned}
\uparrow_\alpha t &= \mathbf{SYN}\ t \\
\uparrow_{\tau_1 \to \tau_2} v &= \mathbf{LAM}(\lambda S.\ \uparrow_{\tau_2} (\mathbf{app}\ (v, \downarrow^{\tau_1} S))) \\
\uparrow_{\tau_1 \times \tau_2} v &= \mathbf{PAIR}(\uparrow_{\tau_1} (\mathbf{fst}\ v), \uparrow_{\tau_2} (\mathbf{snd}\ v))
\end{aligned}$$

$$\begin{aligned}
\downarrow^\alpha (\mathbf{SYN}\ t) &= t \\
\downarrow^{\tau_1 \to \tau_2} (\mathbf{LAM}\ S) &= \mathbf{lam}\ (x, \downarrow^{\tau_2} (S\ (\uparrow_{\tau_1} (\mathbf{var}\ x))))\text{ where } x \text{ is fresh} \\
\downarrow^{\tau_1 \times \tau_2} (\mathbf{PAIR}\ (S, T)) &= \mathbf{pair}\ (\downarrow^{\tau_1} S, \downarrow^{\tau_2} T)
\end{aligned}$$

These definitions are easily implemented in the meta-language:

```
(* reflect : ty -> tm -> sem *)
fun reflect (Arrow (a, b)) t =
      LAM (fn S => reflect b (app t (reify a S)))
  | reflect (Prod (a, b)) t =
      PAIR (reflect a (fst t)) (reflect b (snd t))
  | reflect (Basic _) t =
      SYN t
```

```
(* reify   : ty -> sem -> tm *)
and reify (Arrow (a, b)) (LAM S) =
      let x = fresh_var () in
        Lam (x, reify b (S (reflect a (var x))))
      end
  | reify (Prod (a, b)) (PAIR S T) =
      Pair (reify a S, reify b T)
  | reify (Basic _) (SYN t) = t
```

By induction on the structure of types, it follows that if the semantic object *S* denotes a well-typed term *s* of type τ, then reifying the object (i.e., $\downarrow^{\tau}$ S) produces the β-normal η-long form of *s*. All that remains is, therefore, to construct the initial semantic interpretation *S* from a syntactic term *s*. This operation, written $\| s \|_{\Gamma}$, where Γ is a context of bindings, proceeds by induction solely on the term structure:

$$\begin{aligned}
\|\mathbf{var}\ x\|_\Gamma &= \Gamma(x) \\
\|\mathbf{lam}\ (x,s)\|_\Gamma &= \mathbf{LAM}\ (\lambda S.\ \|s\|_{\Gamma, x\mapsto S}) \\
\|\mathbf{app}\ (s,t)\|_\Gamma &= S\ (\|t\|_\Gamma)\ \text{where}\ \|s\|_\Gamma = \mathbf{LAM}\ S \\
\|\mathbf{pair}\ (s,t)\|_\Gamma &= \mathbf{PAIR}\ (\|s\|_\Gamma, \|t\|_\Gamma) \\
\|\mathbf{fst}\ s\|_\Gamma &= S\ \text{where}\ \|s\|_\Gamma = \mathbf{PAIR}\ (S,T) \\
\|\mathbf{snd}\ t\|_\Gamma &= T\ \text{where}\ \|t\|_\Gamma = \mathbf{PAIR}\ (S,T)
\end{aligned}$$

In the implementation:

```
(* meaning : ctx -> tm -> sem *)
fun meaning G t =
      case t of
        var x => lookup G x
      | lam (x, s) => LAM (fn S => meaning (add G (x, S)) s)
      | app (s, t) => (case meaning G s of
                         LAM S => S (meaning G t))
      | pair (s, t) => PAIR (meaning G s, meaning G t)
      | fst s => (case meaning G s of
                    PAIR (S, T) => S)
      | snd t => (case meaning G t of
                    PAIR (S, T) => T)
```

Note that there are many non-exhaustive cases; however, if applied to a *closed* well-typed term, none of these missing cases are ever encountered. The NBE operation on closed terms is then:

```
(* nbe : ty -> tm -> tm *)
fun nbe a t = reify a (meaning empty t)
```

As an example of its use, consider the syntactic term `SKK` defined below:

```
val K = lam ("x", lam ("y", var "x"))
val S = lam ("x", lam ("y", lam ("z", app (app (var "x", var "z"),
app (var "y", var "z")))))
val SKK = app (app (S, K), K)
```

This is the well-known encoding of the identity function in combinatory logic. Normalising it at an identity type produces:

```
- nbe (Arrow (Basic "a", Basic "a")) SKK;
val it = lam ("v0",var "v0") : tm
```

The result is actually in η-long form, as can be easily seen by normalizing it at a different identity type:

SKI combinator calculus

SKI combinator calculus is a computational system that may be perceived as a reduced version of untyped Lambda calculus.

All operations in Lambda calculus are expressed in SKI as binary trees whose leaves are one of the three symbols **S**, **K**, and **I** (called **combinators**). In fact, the symbol **I** is added only for convenience, and just the other two suffice for all of the purposes of the SKI system.

Although the most formal representation of the objects in this system requires binary trees, they are usually represented, for typesettability, as parenthesized expressions, either with all the subtrees parenthesized, or only the right-side children subtrees parenthesized. So, the tree whose left subtree is the tree KS and whose right subtree is the tree SK is usually typed as **((KS)(SK))**, or more simply as **KS(SK)**, instead of being fully drawn as a tree (as formality and readability would require).

Informal description

Informally, and using programming language jargon, a tree **(xy)** can be thought of as a "function" x applied to an "argument" y. When "evaluated" (i.e., when the function is "applied" to the argument), the tree "returns a value". i.e., transforms into another tree. Of course, all three of the "function", the "argument" and the "value" are either combinators, or binary trees, and if they are binary trees they too may be thought of as functions whenever the need arises.

The **evaluation** operation is defined as follows:

(*x*, *y*, and *z* represent expressions made from the functions **S**, **K**, and **I**, and set values):

I returns its argument:

$\mathbf{I}x \to x$

K, when applied to any argument x, yields a one-argument constant function Kx , which, when applied to any argument, returns x:

$\mathbf{K}xy \to x$

S is a substitution operator. It takes three arguments and then returns the first argument applied to the third, which is then applied to the result of the second argument applied to the third. More clearly:

$\mathbf{S}xyz \to xz(yz)$

Example computation: SKSK evaluates to KK(SK) by the S-rule. Then if we evaluate KK(SK), we get K by the K-rule. As no further rule can be applied, the computation halts here.

Note that, for all trees x and all trees y, SKxy will always evaluate to y in two steps, Ky(xy) → y, so the ultimate result of evaluating SKxy will always equal the result of evaluating y. We say that SKx and I are "functionally equivalent" because they always yield the same result when applied to any y.

Note that from these definitions it can be shown that SKI calculus is not the minimum system that can fully perform the computations of Lambda calculus, as all occurrences of **I** in any expression can be replaced by (SKK) or (SKS) or (S K whatever) and the resulting expression will yield the same result. So the "I" is merely syntactic sugar.

In fact, it is possible to define a complete system using only one combinator. An example is Chris Barker's iota combinator, defined as follows:

$\iota x = x\mathbf{SK}$

Formal definition

The terms and derivations in this system can also be more formally defined:

Terms: The set *T* of terms is defined recursively by the following rules.

1. **S**, **K**, and **I** are terms.
2. If τ_1 and τ_2 are terms, then $(\tau_1\tau_2)$ is a term.
3. Nothing is a term if not required to be so by the first two rules.

Derivations: A derivation is a finite sequence of terms defined recursively by the following rules (where all Greek letters represent valid terms or expressions with fully balanced parentheses):

1. If Δ is a derivation ending in an expression of the form α(**I**β)ι, then Δ followed by the term αβι is a derivation.
2. If Δ is a derivation ending in an expression of the form α((**K**β)γ)ι, then Δ followed by the term αβι is a derivation.
3. If Δ is a derivation ending in an expression of the form α(((**S**β)γ)δ)ι, then Δ followed by the term α((βδ)(γδ))ι is a derivation.

Assuming a sequence is a valid derivation to begin with, it can be extended using these rules.

SKI expressions

Self-application and recursion

SII is an expression that takes an argument and applies that argument to itself:

SIIα → **I**α(**I**α) → αα

One interesting property of this is that it makes the expression **SII**(**SII**) irreducible:

SII(**SII**) → **I**(**SII**)(**I**(**SII**)) → **I**(**SII**)(**SII**) → **SII**(**SII**)

Another thing that results from this is that it allows you to write a function that applies something to the self application of something else:

(**S**(**K**α)(**SII**))β → **K**αβ(**SII**β) → α(**SII**β) → α(ββ)

This function can be used to achieve recursion. If β is the function that applies α to the self application of something else, then self-applying β performs α recursively on ββ. More clearly, if:

β = **S**(**K**α)(**SII**)

then:

SIIβ → ββ → α(ββ) → α(α(ββ)) → …

The reversal expression

S(**K**(**SI**))**K** reverses the following two terms:

S(**K**(**SI**))**K**αβ →

$\mathbf{K}(\mathbf{SI})\alpha(\mathbf{K}\alpha)\beta \rightarrow$
$\mathbf{SI}(\mathbf{K}\alpha)\beta \rightarrow$
$\mathbf{I}\beta(\mathbf{K}\alpha\beta) \rightarrow$
$\mathbf{I}\beta\alpha \rightarrow \beta\alpha$

Boolean logic

SKI combinator calculus can also implement Boolean logic in the form of an if-then-else structure. An if-then-else structure consists of a Boolean expression that is either **T** (True) or **F** (False) and two arguments, such that:

$\mathbf{T}xy \rightarrow x$

and

$\mathbf{F}xy \rightarrow y$

The key is in defining the two Boolean expressions. The first works just like one of our basic combinators:

$\mathbf{T} = \mathbf{K}$
$\mathbf{K}xy \rightarrow x$

The second is also fairly simple:

$\mathbf{F} = \mathbf{KI}$
$\mathbf{KI}\mathrm{xy} \rightarrow \boldsymbol{I}\mathrm{y} \rightarrow \mathrm{y}$

Once True and False are defined, all Boolean logic can be implemented in terms of if-then-else structures.

Boolean NOT (which returns the opposite of a given boolean) works the same as the if-then-else structure, with False and True as the second and third values, so it can be implemented as a postfix operation:

$\mathrm{NOT} = (\mathbf{F})(\mathbf{T}) = (\mathbf{KI})(\mathbf{K})$

If this is put in an if-then-else structure, it can be shown that this has the expected result

$(\mathbf{T})\mathrm{NOT}=\mathbf{T}(\mathbf{F})(\mathbf{T})=\mathbf{F}$
$(\mathbf{F})\mathrm{NOT}=\mathbf{F}(\mathbf{F})(\mathbf{T})=\mathbf{T}$

Boolean OR (which returns True if either of the two Boolean values surrounding it is True) works the same as an if-then-else structure with True as the second value, so it can be implemented as an infix operation:

OR = **T** = **K**

If this is put in an if-then-else structure, it can be shown that this has the expected result:

(**T**)OR(**T**)=**T**(**T**)(**T**)=**T**
(**T**)OR(**F**)=**T**(**T**)(**F**)=**T**
(**F**)OR(**T**)=**F**(**T**)(**T**)=**T**
(**F**)OR(**F**)=**F**(**T**)(**F**)=**F**

Boolean AND (which returns True if both of the two Boolean values surrounding it are True) works the same as an if-then-else structure with False as the third value, so it can be implemented as a postfix operation:

AND = **F** = **KI**

If this is put in an if-then-else structure, it can be shown that this has the expected result:

(**T**)(**T**)AND=**T**(**T**)(**F**)=**T**
(**T**)(**F**)AND=**T**(**F**)(**F**)=**F**
(**F**)(**T**)AND=**F**(**T**)(**F**)=**F**
(**F**)(**F**)AND=**F**(**F**)(**F**)=**F**

Because this defines True, False, NOT (as a postfix operator), OR (as an infix operator), and AND (as a postfix operator) in terms of SKI notation, this proves that the SKI system can fully express Boolean logic.

Connection to intuitionistic logic

The combinators **K** and **S** correspond to two well-known axioms of sentential logic:

AK: $A \to (B \to A)$,
AS: $(A \to (B \to C)) \to ((A \to B) \to (A \to C))$.

Function application corresponds to the rule modus ponens:

MP: from A and $A \to B$ infer B.

The axioms **AK** and **AS**, and the rule **MP** are complete for the implicational fragment of intuitionistic logic. In order for combinatory logic to have as a model:

- The implicational fragment of classical logic, would require the combinatory analog to the law of excluded middle, e.g., Peirce's law;
- Complete classical logic, would require the combinatory analog to the sentential axiom $\mathrm{F} \to A$.