First-class function

From Wikipedia, the free encyclopedia

In computer science, a programming language is said to have **first-class functions** if it treats functions as first-class citizens. Specifically, this means the language supports passing functions as arguments to other functions, returning them as the values from other functions, and assigning them to variables or storing them in data structures.[1] Some programming language theorists require support for anonymous functions (function literals) as well.[2] In languages with first-class functions, the names of functions do not have any special status; they are treated like ordinary variables with a function type.[3] The term was coined by Christopher Strachey in the context of "functions as first-class citizens" in the mid-1960s.[4]

First-class functions are a necessity for the functional programming style, in which the use of higher-order functions is a standard practice. A simple example of a higher-ordered function is the *map* 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 or returning them as results, especially in the presence of non-local variables introduced in nested and anonymous functions. Historically, these were termed the funarg problems, the name coming from "function argument".[5] In early imperative languages these problems were avoided by either not supporting functions as result types (e.g. ALGOL 60, Pascal) or omitting nested functions and thus non-local variables (e.g. C). The early functional language Lisp took the approach of dynamic scoping, where non-local variables refer to the closest definition of that variable at the point where the function is executed, instead of where it was defined. Proper support for lexically scoped first-class functions was introduced in Scheme and requires handling references to functions as closures instead of bare function pointers,[4] which in turn makes garbage collection a necessity.

Contents

Concepts

In this section we compare how particular programming idioms 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: passing functions as arguments

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

```
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 features such as function pointers or delegates. In the language C:

```
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 function needs an additional parameter (giving the size of the array.) The C function updates 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`.

Anonymous and nested functions

In languages supporting anonymous functions, we can pass such a function as an argument to a higher-order function:

```
main = map (\x -> 3 * x + 1) [1, 2, 3, 4, 5]
```

In a language which does not support anonymous functions, we have to bind it to a name instead:

```
int f(int x) {
    return 3 * x + 1;
}

int main() {
    int list[] = {1, 2, 3, 4, 5};
    map(f, list, 5);
}
```

Non-local variables and closures

Once we have anonymous or nested functions, it becomes natural for them to refer to variables outside of their body (called *non-local variables*):

```
main = let a = 3
           b = 1
        in map (\x -> a * x + b) [1, 2, 3, 4, 5]
```

If functions are represented with bare function pointers, it is no longer obvious how we should pass the value outside of the function body to it. We instead have to manually build a closure and one can at this point no longer speak of "first-class" functions.

```
typedef struct {
    int (*f)(int, int, int);
    int *a;
    int *b;
} closure_t;

void map(closure_t *closure, int x[], size_t n) {
    for (int i = 0; i < n; ++i)
        x[i] = (*closure->f)(*closure->a, *closure->b, x[i]);
}

int f(int a, int b, int x) {
    return a * x + b;
}

void main() {
    int l[] = {1, 2, 3, 4, 5};
    int a = 3;
    int b = 1;
    closure_t closure = {f, &a, &b};
    map(&closure, l, 5);
}
```

Also note that the `map` is now specialized to functions referring to two `int`s outside of their environment. This can be set up more generally, but requires more boilerplate code. If `f` would have been a nested function we would still have run into the same problem and this is the reason they are not supported in C.[6]

Higher-order functions: returning functions as results

When returning a function, we are in fact returning its closure. In the C example any local variables captured by the closure will go out of scope once we return from the function that builds the closure. Forcing the closure at a later point will result in undefined behaviour, possibly corrupting the stack. This is known as the upwards funarg problem.

Assigning functions to variables

Assigning functions to variables and storing them inside (global) datastructures potentially suffers from the same difficulties as returning functions.

```
f :: [[Integer] -> [Integer]]
f = let a = 3
        b = 1
     in [map (\x -> a * x + b), map (\x -> b * x + a)]
```

Equality of functions

As one can test most literals and values for equality, it is natural to ask whether a programming language can support testing functions for equality. On further inspection, this question appears more difficult and one has to distinguish between several types of function equality:[7]

Extensional equality

Two functions *f* and *g* are considered extensionally equal if they agree on their outputs for all inputs ($\forall x.\ f(x) = g(x)$). Under this definition of equality, for example, any two implementations of a stable sorting algorithm, such as insertion sort and merge sort, would be considered equal. Deciding on extensional equality is undecidable in general and even for functions with finite domains often intractable. For this reason no programming language implements function equality as extensional equality.

Intensional equality

Under intensional equality, two functions *f* and *g* are considered equal if they have the same "internal structure". This kind of equality could be implemented in interpreted languages by comparing the source code of the function bodies (such as in Interpreted Lisp 1.5) or the object code in compiled languages. Intensional equality implies extensional equality (under the assumption that the functions do not depend on the value of the program counter.)

Reference equality

Given the impracticality of implementing extensional and intensional equality, most languages supporting testing functions for equality use reference equality. All functions or closures are assigned a unique identifier (usually the address of the function body or the closure) and equality is decided based on equality of the identifier. Two separately defined, but otherwise identical function definitions will be considered unequal. Referential equality implies intensional and extensional equality. Referential equality breaks referential transparency and is therefore not supported in pure languages, such as Haskell.

Type theory

In type theory, the type of functions accepting values of type A and returning values of type B may be written as $A \rightarrow B$ 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.

Language support

Functional programming languages, such as Scheme, ML, Haskell, F#, and Scala, all have first-class functions. When Lisp, one of the earliest functional languages, was designed, not all aspects of first-class functions were then properly understood, resulting in functions being dynamically scoped. The later Common Lisp dialect does have lexically scoped first-class functions.

Many scripting languages, including Perl, Python, PHP, Lua, Tcl/Tk, JavaScript and Io, have first-class functions.

For imperative languages, a distinction has to be made between Algol and its descendants such as Pascal, the traditional C family, and the modern garbage-collected variants. The Algol family has allowed nested functions and higher-order taking function as arguments, but not higher-order functions that return functions as results (except Algol 68, which allows this). The reason for this was that it was not known how to deal with non-local variables if a nested-function was returned as a result (and Algol 68 produces runtime errors in such cases).

The C family allowed both passing functions as arguments and returning them as results, but avoided any problems by not supporting nested functions. (The gcc compiler allows them as an extension.) As the usefulness of returning functions primarily lies in the ability to return nested functions that have captured non-local variables, instead of top-level functions, these languages are generally not considered to have first-class functions.

Modern imperative languages often support garbage-collection making the implementation of first-class functions feasible. First-class function have often only been supported in later revisions of the language, including C# 2.0 and Apple's Blocks extension to C, C++ and Objective-C. C++11 has added support for anonymous functions and closures to the language, but because of the non-garbage collected nature of the language, special care has to be taken for non-local variables in functions to be returned as results (see below).

Language		Higher-order functions		Nested functions		Non-local variables		Notes
		Arguments	Results	Named	Anonymous	Closures	Partial application	
Algol family	ALGOL 60	Yes	No	Yes	No	Downwards	No	Have function types.
	ALGOL 68	Yes	Yes[8]	Yes	Yes	Downwards[9]	No	
	Pascal	Yes	No	Yes	No	Downwards	No	
	Ada	Yes	No	Yes	No	Downwards	No	
	Oberon	Yes	Non-nested only	Yes	No	Downwards	No	
	Delphi	Yes	Yes	Yes	2009	2009	No	
C family	C	Yes	Yes	No	No	No	No	Has function pointers.
	C++	Yes	Yes	C++11[10]	C++11[11]	C++11[11]	C++11	Has function pointers, function objects. (Also, see below.) Explicit partial application possible with `std::bind`.
	C#	Yes	Yes	Using anonymous	2.0 / 3.0	2.0	3.0	Has delegates (2.0) and lambda expressions (3.0).
	Objective-C	Yes	Yes	Using anonymous	2.0 + Blocks[12]	2.0 + Blocks	No	Has function pointers.
	Java	Partial	Partial	Using anonymous	Java 8	Java 8	No	Has anonymous inner classes.
	Go	Yes	Yes	Using anonymous	Yes	Yes	Yes[13]	
	Limbo	Yes	Yes	Yes	Yes	Yes	No	
	Newsqueak	Yes	Yes	Yes	Yes	Yes	No	
	Rust	Yes	Yes	Yes	Yes	Yes	No	
Functional languages	Lisp	Syntax	Syntax	Yes	Yes	Common Lisp	No	(see below)
	Scheme	Yes	Yes	Yes	Yes	Yes	SRFI 26[14]	
	Clojure	Yes	Yes	Yes	Yes	Yes	Yes	

	ML	Yes	Yes	Yes	Yes	Yes	Yes	
	Haskell	Yes	Yes	Yes	Yes	Yes	Yes	
	Scala	Yes	Yes	Yes	Yes	Yes	Yes	
Scripting languages	JavaScript	Yes	Yes	Yes	Yes	Yes	ECMAScript 5	Partial application possible with user-land code on ES3 [15]
	Lua	Yes	Yes	Yes	Yes	Yes	Yes[16]	
	PHP	Yes	Yes	Using anonymous	5.3	5.3	No	Partial application possible with user-land code.
	Perl	Yes	Yes	6	Yes	Yes	6[17]	
	Python	Yes	Yes	Yes	Expressions only	Yes	2.5[18]	(see below)
	Ruby	Syntax	Syntax	Unscoped	Yes	Yes	1.9	(see below)
Other languages	Fortran	Yes	Yes	Yes	No	No	No	
	Io	Yes	Yes	Yes	Yes	Yes	No	
	Maple	Yes	Yes	Yes	Yes	Yes	No	
	Mathematica	Yes	Yes	Yes	Yes	Yes	No	
	MATLAB	Yes	Yes	Yes	Yes[19]	Yes	Yes	Partial application possible by automatic generation of new functions.[20]
	Smalltalk	Yes	Yes	Yes	Yes	Yes	Partial	Partial application possible through library.
	Swift	Yes	Yes	Yes	Yes	Yes	Yes	

C++

C++11 closures can capture non-local variables by reference (without extending their lifetime), by copy construction or by move construction (the variable lives as long as the closure does). The former potentially avoids an expensive copy and allows to modify the original variable but is unsafe in case the closure is returned (see dangling references). The second is safe if the closure is returned but requires a copy and cannot be used to modify the original variable (which might not exist any more at the time the closure is called). The latter is safe if the closure is returned and does not require a copy but cannot be used to modify the original variable either.

Java

Java 8 closures can only capture final or "effectively final" non-local variables. Java's function types are represented as Classes. Anonymous functions take the type inferred from the context. Method references are limited. For more details, see Anonymous Functions: Java Limitations

Lisp

Lexically scoped Lisp variants support closures. Dynamically scoped variants do not support closures or need a special construct to create closures.[21]

In Common Lisp, the identifier of a function in the function namespace cannot be used as a reference to a first-class value. The special operator `function` must be used to retrieve the function as a value: `(function foo)` evaluates to a function object. `#'foo` exists as a shorthand notation. To apply such a function object, one must use the `funcall` function: `(funcall #'foo bar baz)`.

Python

Explicit partial application with `functools.partial (https://docs.python.org/library/functools.html#functools.partial)` since version 2.5, and `operator.methodcaller (https://docs.python.org/library/operator.html#operator.methodcaller)` since version 2.6.

Ruby

The identifier of a regular "function" in Ruby (which is really a method) cannot be used as a value or passed. It must first be retrieved into a `Method` or `Proc` object to be used as first-class data. The syntax for calling such a function object differs from calling regular methods.

Nested method definitions do not actually nest the scope.

Explicit currying with `[1] (http://www.ruby-doc.org/core-1.9.3/Proc.html#method-i-curry)`.

See also

- Defunctionalization
- eval
- First-class message
- Kappa calculus – a formalism which excludes first-class functions
- Man or boy test
- Partial application

Notes

1. Abelson, Harold; Sussman, Gerald Jay (1984). *Structure and Interpretation of Computer Programs*. MIT Press. Section 1.3 Formulating Abstractions with Higher-Order Procedures (http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-12.html#call_footnote_Temp_121). ISBN 0-262-01077-1.
2. Programming language pragmatics (http://www.worldcat.org/oclc/222529448), by Michael Lee Scott, section 11.2 "Functional Programming".
3. Roberto Ierusalimschy; Luiz Henrique de Figueiredo; Waldemar Celes. "The Implementation of Lua 5.0" (http://www.lua.org/doc/jucs05.pdf) (PDF).
4. Burstall, Rod; Strachey, Christopher (2000). "Understanding Programming Languages" (https://web.archive.org/web/20100216060948/http://www.cs.cmu.edu/~crary/819-f09/Strachey67.pdf) (PDF). *Higher-Order and Symbolic Computation*. **13** (52): 11–49. doi:10.1023/A:1010052305354 (https://doi.org/10.1023%2FA%3A1010052305354). Archived from the original on February 16, 2010. (also on 2010-02-16
5. Joel Moses. "The Function of FUNCTION in LISP, or Why the FUNARG Problem Should be Called the Environment Problem" (https://dspace.mit.edu/handle/1721.1/5854). MIT AI Memo 199, 1970.
6. "If you try to call the nested function through its address after the containing function has exited, all hell will break loose." (GNU Compiler Collection: Nested Functions (https://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Nested-Functions.html#Nested-Functions))
7. Andrew W. Appel (1995). "Intensional Equality ;=) for Continuations" (http://www.cs.princeton.edu/~appel/papers/conteq.pdf).
8. Tanenbaum, A.S. (1977). "A comparison of PASCAL and Algol 68" (http://comjnl.oxfordjournals.org/content/21/4/316.full.pdf) (PDF). *The Computer Journal*. **21** (4): 319. doi:10.1093/comjnl/21.4.316 (https://doi.org/10.1093%2Fcomjnl%2F21.4.316).
9. http://python-history.blogspot.nl/2009/04/origins-of-pythons-functional-features.html?showComment=1243166621952#c702829329923892023
10. Nested functions using lambdas/closures (http://stackoverflow.com/a/4324829)

11. Doc No. 1968 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1968.pdf): V Samko; J Willcock, J Järvi, D Gregor, A Lumsdaine (February 26, 2006) *Lambda expressions and closures for C++*
12. https://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/Blocks/Articles/00_Introduction.htı
13. "2 examples in Go that you can have partial application" (https://play.golang.org/p/lZHXrX-yR6).
14. http://srfi.schemers.org/srfi-26/srfi-26.html
15. http://ejohn.org/blog/partial-functions-in-javascript/
16. "Lua Code for Curry (Currying Functions)" (http://tinylittlelife.org/?p=249). 2010-07-23. Retrieved 2016-04-05.
17. http://perlgeek.de/blog-en/perl-5-to-6/28-currying.html
18. https://docs.python.org/whatsnew/2.5.html#pep-309-partial-function-application
19. http://www.mathworks.co.uk/help/matlab/matlab_prog/anonymous-functions.html
20. https://stackoverflow.com/questions/9154271/partial-function-evaluation-in-matlab
21. Closures in ZetaLisp (https://common-lisp.net/project/bknr/static/lmman/fd-clo.xml)

References

- Leonidas Fegaras. "Functional Languages and Higher-Order Functions" (http://lambda.uta.edu/cse5317/l12.ppt). CSE5317/CSE4305: Design and Construction of Compilers. University of Texas at Arlington.

External links

- First-class functions (http://rosettacode.org/wiki/First-class_functions) on Rosetta Code.
- Higher order functions (http://www.ibm.com/developerworks/linux/library/l-highfunc/index.html) at IBM developerWorks

Retrieved from "https://en.wikipedia.org/w/index.php?title=First-class_function&oldid=799699398"