The Essence of Closures

A language design perspective

M. Anton Ertl
anton@mips.complang.tuwien.ac.at
TU Wien

Abstract
Closures are originally associated with lexically scoped name binding. However, in the course of implementing closures in Gforth, it turned out that the actual function (the essence) of closures is to communicate data between closure creation and the closure execution (with the closure call usually being far from the closure creation). This paper presents a simple language extension for C: two-stage parameter passing, implemented with flat closures; the first stage creates a closure, the second stage calls it. Nested functions and access to outer locals are not needed.

1 Introduction

In the summer of 2018 I worked with Bernd Paysan on adding "closures" to Gforth [Ertl and Paysan, 2018]. At the time Gforth had local variables and nested definitions (known as quotations), but outer local variables were (and are) not visible inside quotations, in order to avoid the implementation complications of closures. But we wanted to allow programmers to do in Gforth what they do with closures in other languages.

I started out with the lexical scoping idea with some adaptions to avoid garbage collection and with explicit capture of outer locals for simpler implementation in terms of flat closures, and wrote a draft version of a paper about that [Ertl, 2018]. Then Bernd Paysan set out to implement these ideas, came up with further simplifications, and after a lot of back-and-forth we arrived at the final design and implementation. As a result, the paper had to be almost completely rewritten [Ertl and Paysan, 2018].

One interesting aspect is that the result is not just easier to implement, but also often more convenient to use than the original lexical-scoping-inspired syntax. As an example, here is a definition of `+field` using the draft syntax:

```
: +field ( u1 u "name" -- u2 )
    create over {: u1 :} +
    ['] alloth <[{: : u1 :} drop u1 + ;] set-does> ;
```

and the syntax of the final paper:

```
: +field ( u1 u "name" -- u2 )
    create
    over [{: u1 :}d drop u1 + ;] set-does>
    + ;
```

I will not explain the syntax in detail here, but just explain the decisive difference: In the draft version is was necessary to create a local `u1` with `{: u1 :}` that could then be passed (explicitly) into the closure (with `: u1`), while in the final version the value is passed as a parameter at closure creation and only turned into a local in the closure (`[{: u1 :}d`). Since then we have added a variant that does not require defining a local and expresses the same functionality as:

```
: +field ( u1 u "name" -- u2 )
    create
    over [n:d nip + ;] set-does>
    + ;
```

In this variant closure creation takes one item from the stack and stores it in the closure, and the closure call pushes this item.

Of course, these examples and the examples in the papers are hard to understand for a non-Forth audience, so I wrote this paper with a wider audience in mind, and use C with appropriate (unimplemented) extensions in the rest of this paper.

The main point of this paper is that it can be (and, in our experience, is) better for the implementation complexity, for the language, and for the programmers to provide what programmers want to do with closures more directly than by accessing outer locals in nested functions.

This paper may be of little interest to well-versed implementors of functional languages, but it may inspire language designers of other languages to widen their perspective, looking for other solutions than just implementing nested functions with access to outer locals. It may help them to find a better solution, without a detour as long as the one I have taken.

Section 2 gives an example where closures are useful. Section 3 introduces functions with two-stage parameter passing as a language-level way to build flat

closures directly. Section 4 shows how to implement the equivalent of writing to an outer local variable. Section 5 discusses how to manage the memory needed for closures. Finally, Section 6 describes related work.

A note on terminology: The term *closure* has been originally used for a lambda expression with (not really) free variables where these variables are bound by the lexically-scoped environment, then by language implementors for the data structure that implements this concept, and by programmers when they create such a thing, pass it around, or call it. It has seen quite a bit of expansion in meaning in that process, and this paper continues with that expansion, by using *closure* to mean any function/procedure/method that has a parameter that is determined at run-time[1], but is not in the parameter list of calls to the closure.

2 Motivation

When a function g takes another function f as parameter, g typically calls f with a certain number of arguments, and normally the argument is generated by f according to its internal logic. A classical example is a numeric integration function, which in C would go something like this:

```
double numint(double l, double h, double (*f)(double))
{
  ... (*f)(x);
  ...
  return result;
}
```

`numint()` calls `(*f)(x)` repeatedly with different x in order to approximate $\int_l^h f(x)dx$. Now consider the case that we want to compute $\int_l^h x^{-y}dx$, where y is a run-time parameter that is not changed inside `numint`. We would have to implement

```
double numint2(double l, double h,
               double (*f)(double,double), double y)
{
  ... (*f)(x,y);
  ...
  return result;
}
```

[1]This excludes passing through a global variable, which is bound at compile time, and makes the whole thing non-reentrant.

And additional versions of numint would be necessary for passing more parameters or parameters with other types.

Instead, we would like to bind y to a function at run-time and pass a function pointer to `numint()` that can be called with only an x argument. Closures provide this capability.

3 Two-stage parameter passing

A way to define such a closure is to pass parameters in two stages: In the first stage we pass y, creating the closure; we can then pass the closure to `numint()`, where it is called (repeatedly) with various x parameters. A C syntax extension for this might look as follows:

```
double g(double y)(double x) {
  return pow(x,-y);
}

/* inside a different function */
  r1 = numint(a, b, g(2.0));
  r2 = numint(a, b, g(3.0));
  r3 = numint(a, b, g(2.5));
```

The first `numint()` call contains a call to `g()`, which is evaluated first, and produces a closure where $y = 2.0$. Inside `numint()` the closure is then called with various values for x.

One benefit of this approach is that we can pass a value directly instead of having to assign it to an outer local first. Admittedly, with outer locals a function `g()` can be constructed that is called in a very similar way, but that function would pass a closure outwards (which requires explicit deallocation in a language like C), whereas at least in the `numint()` case stack allocation is sufficient for a closure created by the first-stage call to `g()`.

Implementationwise, the closure consists of the code address of `g` and the value of y. This kind of representation comes out of flat-closure-conversion [Dybvig, 1987], one possible way to implement a language with nested functions with access to outer locals.[2] So what we do here is to write the code such that flat closures are created directly.

[2]The mechanical conversion from nested functions to flat closures also shows that two-stage parameter passing is as powerful as nested functions.

This means that we cannot directly add a third stage, but (if we need that) have to do it by having a two-stage function that calls another two-stage function and passes the arguments of the first two stages to the second function as arguments:

```
int f1(int i1, int i2)(int i3) {
  return i1*i2*i3;
}
int(*)(int) f2(int i1)(int i2) {
  return f1(i1,i2);
}
```

Of course, if it turns out that this kind of extension is used a lot in your language, it could be extended into something more sophisticated, along the lines of what functional programming languages like ML do with currying and a more sophisticated compiler.

4 Changeable data

Until now these closures only allow passing data by value. What if we want to have read and write access to shared data? Well, the C way is to pass a pointer to the data. A simple example of that is:

```
int counter(int *p)(void) {
  return (*p)++;
}

/* inside a different function */
  int *c1p = malloc(sizeof(int));
  int *c2p = malloc(sizeof(int));
  *c1p = 5; /* initialize counters */
  *c2p = 0;
  int (*c1)() = counter(c1p);
  int (*c2)() = counter(c2p);
  int (*c1a)() = counter(c1p);
  printf("%d %d\n",(*c1)(),(*c2)()); // 5 0
  printf("%d\n",(*c1a)());           // 6
  printf("%d %d\n",(*c1)(),(*c2)()); // 7 1
```

In this example memory for two counters is created with `malloc()`, and two closures (`c1` and `c1a`) access one of the counters, while a third closure (`c2`) accesses

the other one. Each call to one of the closures increases the corresponding counter and returns its previous value.

When implementing languages with nested functions with access to outer locals and flat-closure conversion, this kind of code comes out of assignment conversion [Dybvig, 1987]. Again, here we write this code directly.

Explicitly allocating the memory for the counters is a bit cumbersome, and depending on the language you are designing, you may or may not want to provide a more convenient way to write this.

Moreover, your base language may not allow to pass pointers to memory around or another way may be preferable. E.g., in C++ it would be more convenient to use references rather than pointers to the memory. In Pascal, Modula-2, or Oberon I would probably use the `VAR` parameter syntax (and its pass-by-reference semantics) for this purpose.

5 Closure memory

Closures need memory at run-time, at least for storing (in our example) y, possibly also a pointer to the code for computing x^{-y}, and possibly some native code (called a trampoline) so that the closure can be involved like a regular function pointer.

In many languages closures are garbage-collected; others, like Algol 60, Pascal, and C++ only allow calling closures in a way that allows them to be stack-allocated.[3] In both cases the programmer does not have to deal with allocation and deallocation of the memory.

C (and Forth) does not have garbage collection, but instead uses either stack allocation and deallocation or `malloc()` and `free()` (or user-constructed memory management, such as region-based allocation or reference-counting). We could require the same restrictions as C++ to get by with stack allocation, but that is insufficient in many usage cases.

Therefore, closure memory is managed by the programmer, just like other memory. The programmer decides whether closures are allocated on the stack, or with which allocator they are allocated and destroyed.

In Gforth we have the general closure constructor that takes the allocator as parameter, and also shorthands for the common cases: stack, heap (`allocate` (like `malloc()`) and `free`), and dictionary (essentially allocation until the end of the process). I leave it to you to find a nice syntax for C or the language you are designing.

[3] One usually does not call closures in languages with this restriction “closures”, but here I do.

6 Related work

Our earlier work [Ertl and Paysan, 2018] describes a similar Forth extension in more detail, including performance results, and implementation data. At the time of that paper the closure implementation in Gforth cost 78 source lines of code (it has grown a little since then).

A number of other languages provide similar features, some through nested functions with access to outer locals, others with more special syntax. Of particular note are the C++11 lambdas[4], which inspired my first (too-complicated) approach: They are based on accessing outer locals, but allow capturing the outer locals explicitly, and allow to control whether a variable is captured by-value or by-reference. One major difference is that we need to assign the value to the outer local first rather than passing it as parameter as in the present approach. Interestingly, C++ does not offer explicit memory management of closures, and restricts its programmers to stack-related limitations.

Our implementation ideas were based on flat-closure conversion approach [Dybvig, 1987, Section 4.4] in combination with assignment conversion [Dybvig, 1987, Section 4.5]. A later work [Keep, Hearn, and Dybvig, 2012] discusses the kinds of optimizations that implementors of nested functions should perform. But the basic implementation ideas inspired the language side of the Gforth extension, and eventually the present paper, and there such optimizations are unnecessary (or left to the programmer).

However, there are also other ways to implement access to outer locals. Static link chains and the display [Fischer and LeBlanc, 1988] are particularly well-known. They keep each local in only one place, but have relatively complex and sometimes slow ways to access them.

7 Conclusion

It is not necessary to implement nested function with access to outer locals in order to provide the features to programmers that they want. Instead, functions with two-stage parameter passing allow a direct implementation in terms of flat closures, and this can even be nicer to use than accessing outer locals.

[4] https://en.wikipedia.org/wiki/Anonymous_function#C++_%28since_C++11%29

References

Dybvig, R. Kent (Apr. 1987). "Three implementation models for scheme". PhD thesis. University of North Carolina at Chapel Hill. URL: http://agl.cs.unm.edu/~williams/cs491/three-imp.pdf.

Ertl, M. Anton (2018). "General locals". Draft version of the published paper [Ertl and Paysan, 2018] that was mostly rewritten. URL: http://www.euroforth.org/ef18/drafts/ertl.pdf.

Ertl, M. Anton and Bernd Paysan (2018). "Closures — the Forth way". In: *34th EuroForth Conference*, pp. 17–30. URL: http://www.complang.tuwien.ac.at/papers/ertl%26paysan.pdf.

Fischer, Charles N. and Richard J. LeBlanc (1988). *Crafting a compiler*. Menlo Park, CA: Benjamin/Cummings.

Keep, Andrew W., Alex Hearn, and R. Kent Dybvig (2012). "Optimizing closures in O(0) time". In: *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming, Scheme 2012, Copenhagen, Denmark, September 9-15, 2012*. Ed. by Olivier Danvy. ACM, pp. 30–35. ISBN: 978-1-4503-1895-2. URL: http://doi.acm.org/10.1145/2661103.