Efficient Closure Conversion in Lang F

Student: Anthony Castiglia Advisor: Matthew Fluet

December 17, 2013

1 Introduction

Higher-order programming languages treat functions as first class values, in that functions are treated the same as other values. This treatment allows functions to be passed as arguments to and returned from other functions, leading to the notion of "higher-order functions," that is, those that take functions as parameters or return functions as values. Many functional programming idioms rely on these first-class and higher-order functions to perform basic programming tasks. For example, the commonly used *map* function (which can be found or easily implemented in almost any functional language) takes as arguments a function f and a list ℓ and returns a new list ℓ', which is constructed by applying f to each element of ℓ. While these high-level language features allow a more abstract, expressive manner of writing programs, they are typically not available in languages targeted by compilers for higher-order languages. To properly implement first-class functions in lower-level languages, a data structure called a *closure* is used to model these lexically-scoped, first-class functions.

The lower-level languages (e.g. C, assembly, JVM bytecode) typically targeted by compilers of higher-order languages typically do not support nested or first-class functions. As a result, compiling a a higher-order language (e.g. ML, Scheme, Haskell) to one of these lower-level languages requires source translations that do two things. First, to preserve the meaning of programs with nested, lexically-scoped functions, the program undergoes a process called *closure conversion*. In this process a referencing environment is constructed for each function that maps the function's free variables to their bound values [4]. All references within the function body to free variables are then made to instead reference the function's environment. The new function code (or a pointer to it) is then packaged with its referencing environment into a data structure called a *closure* [3] [7]. A second translation, sometimes referred to as *lambda lifting*, is also performed to hoist each newley-closed function to the top level scope of the program, thus eliminating nested scopes.

Each function called at runtime will then require its referencing environment me made available, either by loading the environment onto the stack [3], or by providing a pointer to a heap-allocated environment data structure, as described in this work. Since function calls and returns are used heavily in higher-order programs, there have been a number of methods reported to optimize these through more efficient closure representations [2] [5] [6]. Simple attempts at optimization, like using so-called *linked closures* can yield modest reductions in memory usage, as demonstrated in this work, but can cause "space leaks" to occur when used with simple, although admittedly contrived examples as demonstrated in this work as well as by Shao and Appel [6], and in real programs, such as compilers [6].

```
val a = 1
fun f () =
  let val b = 2
      fun g () =
        let val c = 3
            fun h () = a + b + c
         in h
        end
   in g ()
  end

val a = 0
val func = f ()
val result = func ()
```

Figure 1: Nested functions

In this work, two simple closure implementations are examined in the context of Lang F, a simple System F based functional programming language. The first, linked closures, as described by Landin [3], in which each closure contains the free variables that are in the same local scope as its function, and a pointer to its enclosing environment are discussed and shown to provide in some situations modest reductions in memory usage of compiled programs, but in other cases to case drastic "space leaks" in which pointers to enclosing arguments are retained even after they are no longer needed, preventing the memory allocated for said environments from being reclaimed by the garbage collector. The second implementation, flat closures, as described by Cardelli [1] in which each closure contains a copy of each of its free variables, is shown to be immune to the space leaks caused by the linked closure implementation, but to use slightly more memory in some cases by storing the same values in multiple closures rather than sharing them. The possible consequences of storing many copies of the same objects across multiple flat closures are shown, however to be largely mitigated by the implementation details of the Lang F virtual machine.

2 Background

2.1 Nested functions and lexical scope

Nested function declarations, and thus nested scopes are also allowed in higher-order languages. For example, in figure 1, the function `h` is nested twice, first in `g`, and then in `f` in turn. In this particular program, `h` is returned by `g` and then by `f` by calling `f ()`. Returning `h` as a value from `f ()` allows `h` to be called outside of the scope in which it was declared. Calling `h` outside of the body of `f` in figure 1 (by calling `func`, which is bound to a copy of `h`) raises the question of what `func ()` should evaluate to. The definition of `h` states that

```
fun h () = a + b + c
```

or, that `h` should return the sum of what `a`, `b` and `c` evaluate to. These values depend on the scoping rules imposed by the language. In a lexically scoped language

```
a + b + c
```

is equivalent to

```
1 + 2 + 3
```

Here `a` evaluates to 1, despite being bount to 0 in the scope in which `func` is called. This is because, according to lexical scope, a function's free variables, those which are not declared locally within the function body or taken as parameters, are bound within the function body to the values to which they were bound at the function's declaration. This differs from the notion of dynamic scope, where variables evaluate to their current in-scope binding. The difference is less obvious in languanges without first-class functions. Since no function can be passed as a value out of the scope in which it is declared, all of the function's free variables are necessarily in scope any time the function is called, and as such are guaranteed to stored somewhere in the runtime stack. Evaluating the values of free variables in this case is then just a matter of determining where in the stack each variable binding is stored, and retrieving the value at that location.

When functions are first class, their use is no longer restricted to the scope in which they are declared. Instead, a function may be passed out of its declaring scope and called elsewhere in the program, where the function's free variables are no longer guaranteed to be in scope. Since variables that are no longer in scope do not necessarily persist on the stack, a different strategy must be adopted to preserve their bindings for use in the function.

2.2 Closures

A closure is a representation of a λ-expression paired with the *environment* in which it is evaluated [3] [7]. More generally, a closure is a data structure comprised of a function's code or definition, and an environment that maps the function's free variables to their correct bound values. For example, consider the function `f` in figure 1. The function `f` has only one free variable, namely `a`, so `f`'s closure's environment need only map `a` to 1, `a`'s bound value at the time of `f`'s declaration. If F is the closure for `f`, G is the closure for `G` and H is the closure for `h`, then conceptually

$$\begin{aligned} F &= (\texttt{f}, \{\texttt{a} \mapsto 1\}) \\ G &= (\texttt{g}, \{\texttt{a} \mapsto 1, \texttt{b} \mapsto 2\}) \\ H &= (\texttt{h}, \{\texttt{a} \mapsto 1, \texttt{b} \mapsto 2, \texttt{c} \mapsto 3\}) \end{aligned}$$

Function closures for are typically implemented for each function as a pair containing a pointer to the function's machine code and a record containing the function's free variables [5].

2.3 Lang F language and compiler

Lang F is an ML-based functional language based on the syntax of Standard ML (SML). The language consists of a stack-based virtual machine (VM) that executes Lang F object files, and a compiler written in SML that generates bytecode instructions for the virtual machine. The compiler has six major components: a scanner, parser, type checker, two separate syntax translators and a bytecode generator.

The compiler can be built with either an ML-Lex and ML-Yacc generated scanner and parser, respectively, or with an MLULex and MLAntlr scanner and parser. The third phase, the type checker, takes the typed abstract syntax tree (AST) from the parser and checks it against a set

of typing rules based on System F, the simply-typed, polymorphic lambda calculus. In the next phase, the type-checked AST is translated to the Core Intermediate Representation (CoreIR). This translation makes simplifications to the Lang F grammar by replacing conditionals with `case` expressions, sequence expressions (e.g. $s1;s2$) are converted to `let` expressions (e.g. `let`$v1 = s1$*in*$let \ldots$`end`), and anonymous functions are only allowed one parameter. For example, `fn ['a] (x : 'a) (y : 'a) => ...` becomes `fn ['a] x : 'a => fn ['a] y : 'a => ...`.

The CoreIR AST is then translated to Representation-Location Intermediate Form (RepLocIR) in the fourth phase of the compiler. RepLocIR replaces all variable identifiers with explicit variable locations, describing where each variable is located in relation to where it is referenced. This concept is explained further in the Implementation section of this report.

The fifth and final phase of the Lang F compiler is the VM code generator. This phase translates the RepLocIR source into bytecode instructions targeting the Lang F virtual machine, as described in the next subsection.

2.4 Lang F virtual machine

The Lang F virtual machine (VM) uses 32-bit words for all values. Values can be 31-bit tagged integers (represented by a 32-bit word) or 32-bit pointers to heap-allocated objects, stack positions or bytecode instructions. Upon loading an object file, the VM loads the object file's literal table and C-function table, which comprise the VM's immutable state. The mutable part of the VM's internal state consists of the call stack and heap, the frame and environment pointers and the program counter. The environment pointer (EP) points to the current function's environment (a record in the heap) and the frame pointer points to the base of the current stack frame. The program counter is a pointer to the next bytecode instruction to be executed.

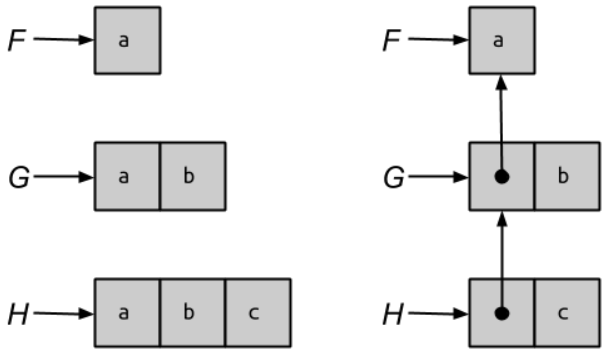
The VM uses a simple garbage collector to reclaim memory occupied in the heap by "dead" objects, those with no pointers referencing them from live objects. To accomplish this, the garbage collector scans the stack for pointers to heap objects and copies any objects referenced from these pointers to a temporary location. The heap is then cleared to remove all of the dead objects, and the cached live objects are returned to the heap. The garbage collector is invoked any time a heap allocation would cause the size of the heap to exceed the amount of system memory allocated for the heap.

3 Implementation

⟨*varloc*⟩ ::= Global (int, int)
  | Local (int)
  | Param
  | Self (int)

Figure 2: Variable locations in RepLocIR

The implementation details of both closure conversion strategies depend on the notion of a variable location in RepLocIR. CoreIR is still a higher-order language, and so it still has nested scopes, and in-scope variables are resolved by name. In RepLocIR however, variable's names are



(a) Flat closures (b) Linked closures

Figure 3: Closure implementation strategies

replaced their variable locations, relative to where the program is in its execution. There are four types of variable locations (as shown in Figure 2 in RepLocIR: Global, Local, Param and Self.

Global represents a variable whose definition is in an enclosing scope as the current location. Two integers are associated with each Global, call them p and n. First, p denotes how many enclosing scopes away from the current location the Global resides in the environment, and n denotes the location in the p^{th} scope where the Global resides. For example, if v is converted to Global (0, 2), then v is the first or second variable (depending on the type of closure being used) in the immediatly enclosing environment.

Local, Param and Self locations are a bit more straightforward. Local (n) denotes the n^{th} local variable in the current environment. Param denotes the parameter to the current function (recall that RepLocIR functions have only one variable), and Self (i) denotes the i^{th} mutually recursive function in the current function's group of mutually recursive functions.

Closure conversion in the Lang F compiler is performed during the conversion from CoreIR to RepLocIR, and during the conversion from RepLocIR to VM bytecode. A function declaration in CoreIR consists of a list of records, one for each mutually recursive function in the declaration, each consisting of a name, a type and a lambda expression. In RepLocIR, a function declaration consists of a list of records for each mutually recursive function in the declaration, paired with a list of variable locations (the function declaration's environment). Each record consists of a integer specifying where the function's closure should be stored in its environment and a lambda expression. The second part of closure conversion in Lang F occurs during the VM code generation phase. It is here that closures are actually allocated in and retrieved from memory (or rather the VM instructions to do so are generated).

3.1 Flat closures

The default closure strategy, flat closures (Figure 3a) for the Lang F compiler is to allocate each function closure as a one-dimensional heap record containing copies of all of the function's free variables. To do this, for each of the function's free variables, a new variable location is created in the function's RepLocIR environment, namely, a Global $(0, i)$, where i is the index for each free variable, i.e. which "slot" it will occupy in the function's closure. The 0 in the first field of the pair is just a placeholder, and is used for generating linked closures without having to modify the RepLocIR specification.

When the VM bytecode is generated to declare the function, all free variables in the function's closure are first loaded onto the stack from the current environment (in which the function is declared). The top n variables are then popped from the stack and stored in a heap record, a pointer to which is then pushed to the stack.

3.2 Linked closures

The creation of linked closures (Figure 3b) differs slightly from that of flat closures. First, in the construction of RepLocIR environments, free variables' locations in their environment (the one in which the function is declared) are considered when generating locations for use within the function's closure. Variable location values are converted according to the following rules:

```
case VarLoc of
     Global (p, i) => Global (p+1, i)
   | Local or Param or  => Global (0, i + 1)
```

So a variable with a location of Global (p, i) in the enclosing environment becomes Global $(p+1, i)$, since it will already be allocated in the enclosing environment, and therefore will not need to be allocated in the function's closure. Param and Local variables become Global $(0, i)$, since they are not yet allocated in the enclosing function's closure. The 0 then signals to the code generator that these values must be stored in the function's closure.

In the code generation phase, two things differ from the code generation for flat closures. First, loading global variables from an enclosing environment requires climbing up a chain of pointers until the closure in which the variable is actually stored is reached, i.e. where the variable's location is a Global $(0, i)$ for some i. Second, to generate a function declaration's closure, before pushing the function's free variables to the stack, the current environment pointer is pushed to provide a reference to the enclosing environment from the function's closure.

3.3 Statistics and heap profiling

To measure the performance of programs compiled with each closure strategy, simple statistic reporting was implemented in the Lang F virtual machine. Counters for the number of bytes allocated, number of bytes deleted (via garbage collection) and the number of instructions executed were added to the VM. When profiling or statistic reporting is enabled, the memory counters are updated each time the VM performs an allocation or garbage collection, and the instruction counter is incremented each time a bytecode instruction is executed. If profiling is enabled, at each allocation and garbage collection the VM prints the current instruction count, the total bytes allocated and the total bytes freed. The current number of live bytes in the heap can then be computed by subtracting the number of bytes freed from the number of bytes allocated. If statistic reporting

```
val x1 : Integer = 0
val x2 : Integer = 1
val x3 : Integer = 2
val x4 : Integer = 3
val x5 : Integer = 4
val x6 : Integer = 5
val x7 : Integer = 6
val x8 : Integer = 7
val x9 : Integer = 8

fun f (u : Unit) : (Unit -> Integer) =
  let
      fun g (u : Unit) : Integer = x1+x2+x3+x4+x5+x6+x7+x8+x9
   in g
  end

fun loop (i : Integer) (acc : List [Unit -> Integer])
    : List [Unit -> Integer] =
  if i < 1
    then acc
    else loop (i - 1) (Cons [Unit -> Integer] {f Unit, acc})
;
loop n (Nil [Unit -> Integer])
```

Figure 4

is enabled, the total number of instructions executed, bytes allocated and bytes freed are printed when the running program terminates. Using this information, performance data was collected and analyzed for different programs using each closure conversion strategy.

3.4 Test cases

A unit test was devised to highlight the potential for higher memory usage by flat closures, due to duplication of values in many closures. Figure 4 shows a Lang F program (with some utility function and datatype declarations omitted) written to highlight the potential for higher memory usage when using flat closures due to the duplication of values in multiple closures. The `loop` function is used to build a list of closures of `g` created by calling `f Unit`, which, when called returns a new function with `x1`, `x2`, ..., `x9` in its closure.

A second test, shown in Figure 5, was devised to show the potential for "space leaks" when using linked closures. A `loop` function is defined the same way as in the previous test (shown in Figure 4) which again creates a list of `n` closures, this time of `h`, which is nested inside of `g`, in turn inside `f`.

```
fun f (u : Unit) : (Unit -> Integer) =
  let
    val l : Array [Integer] = array [Integer] m 0
    fun g (u : Unit) : Unit -> Integer =
      let val a : Integer = l ! 0
          fun h (u : Unit) : Integer = a
       in h
      end
  in
    g Unit
  end

fun loop (i : Integer) (acc : List [Unit -> Integer])
    : List [Unit -> Integer] =
  if i < 1
    then acc
    else loop (i - 1) (Cons [Unit -> Integer] {f Unit, acc})
;
loop n (Nil [Unit -> Integer])
```

Figure 5

4 Results and discussion

Results for the first unit test (whose partial code is shown in Figure 4) are shown in Figure 6. Figure 6 shows a heap profile for $n = 100$, i.e. the program generates 100 closures. Since each flat closure requires its own copy of each of its function's free variables, while linked closures can share objects from their parents' closures, the heap size increases more rapidly in the case of flat closures, since each one contains a copy of each `xi` (for $i \in \{1..9\}$). This is expected, and shows a simple case where the use of flat closures results in higher overall memory usage.

The results from the second test are given in Figure 7. In this case, flat closures perform much better than linked closures. Since each linked closure generated by `h` contains a pointer to `g`, which contains an array of 1000 integers, the memory occupied by each copy of `g`'s closure cannot be reclaimed by the garbage collector, even though none of the values in it are needed by `h`, or anywhere else in the program. This is an example of the type of space leak described by Shao and Appel [6]. Additionally, looking closely at the shape of the profile curves, the periodic shapes reveal the behaviour of the garbage collector in each case. For the case of flat closures, the saw-tooth shape of the curve shows when the garbage collector reclaims the array generated at each iteration of the loop. The linked closure profile curve shows a staircase shape, showing allocation for the array on each loop iteration, but no reclaiming of that memory. This behaviour is expected, since each closure in the list generated by `loop` maintains a pointer to the generated array.

It would seem then, from the results given that flat closures are better in some situations, but worse in others. Looking again at the heap profile for the first test (Figure **??**), one can easily see that the difference in final heap sizes is much smaller than in the second test (Figure **??**).

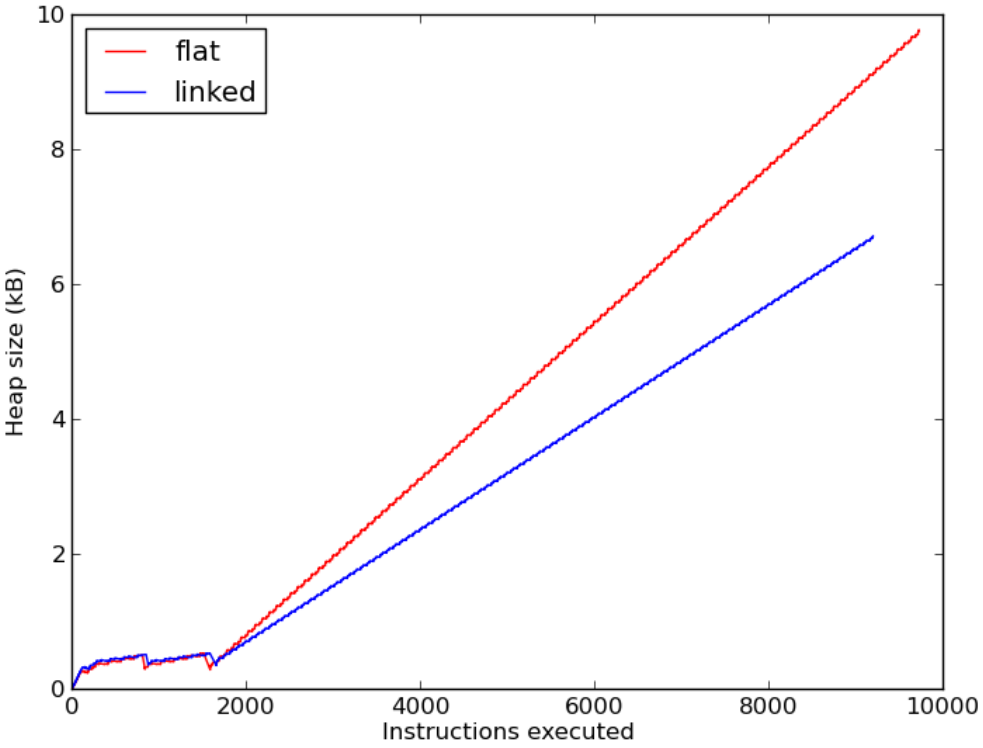


Figure 6: Heap profile for generating 100 closures, each containing 9 integers

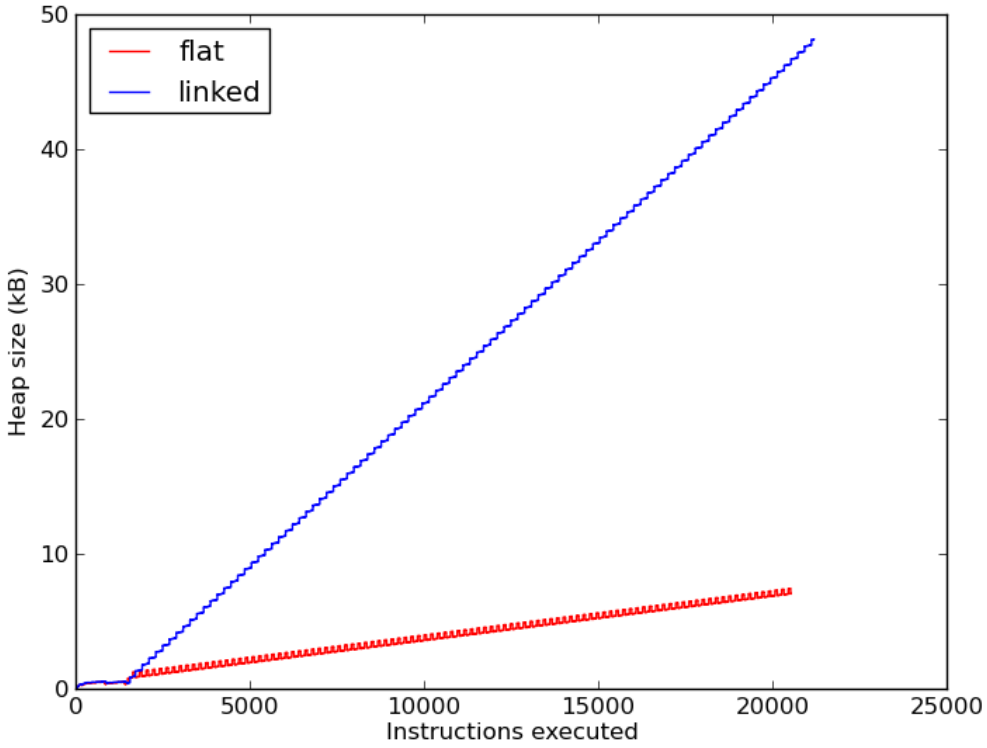


Figure 7: Heap profile for generating 100 closures, each containing an array of 1000 integers

This is reasonable, since each closure in the first test only contains a maximum of nine 32-bit integers. What is less obvious, however, is that even if the objects referenced in each closure were much larger, for example, if they were large arrays, the difference between final heap sizes for each closure strategy would still not change much. Because large objects like arrays and lists are heap-allocated, each closure would still only contain an integer number of 32-bit pointers to these objects, so the difference in memory usage would not increase dramatically.

Given the results of these tests, it is concluded that the potential slight reduction in memory afforded by the use of linked closures in certain situations is not worth the risk of dramatic space leaks that can occur as a result of linking closures together, as these leaks could be catastrophic in certain situations. For example, in the second test, had the loop been allowed to continue beyond 100 iterations, the VM would have eventually run out of memory and terminated.

5 Future work

In the future, it might be worthwhile to look into more sophisticated closure conversion strategies, like the one proposed by Shao and Appel [6], in which control flow analysis is performed on the program prior to closure conversion to determine which values can be safely shared between closures, and which values need to be duplicated to prevent space leaks like the ones seen with linked closures. It would also be good to look at profiling data from a larger variety of test programs for each closure conversion strategy.

References

[1] Cardelli, L. Compiling a functional language. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming* (New York, NY, USA, 1984), LFP '84, ACM, pp. 208–217.

[2] Cejtin, H., Jagannathan, S., and Weeks, S. Flow-directed closure conversion for typed languages. In *Proceedings of the 9th European Symposium on Programming Languages and Systems* (London, UK, UK, 2000), ESOP '00, Springer-Verlag, pp. 56–71.

[3] Landin, P. J. The mechanical evaluation of expressions. *The Computer Journal 6*, 4 (1964), 308–320.

[4] Moses, J. The function of function in lisp or why the funarg problem should be called the environment problem. *ACM SIGSAM Bulletin*, 15 (1970), 13–27.

[5] Shao, Z., and Appel, A. W. Space-efficient closure representations. *SIGPLAN Lisp Pointers VII*, 3 (July 1994), 150–161.

[6] Shao, Z., and Appel, A. W. Efficient and safe-for-space closure conversion. *ACM Trans. Program. Lang. Syst. 22*, 1 (Jan. 2000), 129–161.

[7] Sussman, G. J., and Steele Jr, G. L. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation 11*, 4 (1998), 405–439.