

Closure Implementation in BitC[†]

Preliminary Implementation Note

Jonathan Shapiro, Ph.D.
Systems Research Laboratory
Dept. of Computer Science
Johns Hopkins University

September 3, 2005

Abstract

BitC is a statically typed higher-order programming language intended (in part) for systems programming. A key decision in such languages is the representation of procedure objects, especially closures, and (in consequence) calling convention and interactions with the garbage collector. In this note I try to capture some ideas about how to do all of this efficiently.

I should add that my current thoughts in this regard are somewhat unhappy. Efficient implementation of closures would appear to preclude compilation to C, which is most unfortunate.

These notes are preliminary. They should not be viewed as a commitment to a particular implementation, nor as advice concerning feasible implementations.

1 Issues

In languages having first-class procedure objects, the representation of closures is a foundational design decision in the language implementation. Some of the questions that are raised in this design process include:

1. Are closures heap allocated or stack allocated?

It has been argued that stack-allocated frames are more efficient, but I am unaware of conclusive measurements one way or the other, with the possible exception of Appel and Shao [4] I suspect that stack allocation is faster, primarily because many modern processors provide special handling of top of stack caching. Heap allocation requires garbage collection (GC), and in practice assumes an incremental collection strategy.

Because BitC needs to be compilable in a non-collected mode (subject to language restrictions), it is strongly desirable to stack-allocate continuations.

2. Is the native calling convention respected?

On most modern machines, the native calling convention uses registers to pass the first k arguments. This has been conclusively shown to improve performance, but it presents a variety of hazards in a garbage collected environment. In particular, some collector designs will now be required to collect registers conservatively, which has implications for the storage allocator design.

In many respects, procedure call is simplified if the native calling convention is abandoned in favor of a single-argument convention.

3. In environments providing collection, do we wish to support concurrent GC?

If the BitC runtime is concurrently multithreaded, it is probably necessary to support concurrent collection. This makes the register-based object reference issue even more of a challenge.

[†] Copyright © 2005, Jonathan S. Shapiro.

2 Ground Work

Before describing what I think we should do in BitC, let me start with some groundwork.

2.1 Baseline Approach

Jeff Siskind provides a wonderful description of what he calls the *baseline approach* [14]:

1. Each procedure has a closure.
2. The closure for each procedure contains a *variable slot* for each variable bound by that procedure.
3. The closure for each procedure contains a *parent slot*, providing a link to the closure of the lexically enclosing procedure. For a top-level procedure, the parent slot contains a null pointer.
4. A procedure object contains a *closure pointer slot*, which points to the lexically enclosing closure for that procedure. The closure pointer for top-level procedures is null.
5. A procedure object contains a *code-pointer slot*, a pointer to the code object for that procedure.
6. The code object for a procedure has a *variable parameter* for each variable bound by that procedure.
7. The code object for a procedure has a *parent parameter*, a pointer to the closure for the immediate lexically surrounding procedure. For a top-level procedure, the value of this pointer is null.
8. Procedure calls indirect to the code object pointed to by the procedure object.
9. *Variable Passing*: A procedure call passes each argument to its corresponding variable parameter in the code object.
10. *Parent Passing*: A procedure call passes the closure pointer of the target procedure object to the parent parameter of the code object.
11. Each code object contains a *closure pointer*, a local variable that holds a pointer to its closure. Each code object begins with a preamble that allocates a closure and stores a pointer to that closure in the closure pointer.
12. *Variable spilling*: The preamble spills each variable parameter into the corresponding variable slot of the closure.
13. *Parent spilling*: The preamble spills the parent parameter into the parent slot of the closure.
14. Variables are referenced indirectly via the closure pointer.

2.2 Conventional Optimizations

Several optimizations to this approach are so universal that they are included in nearly every real implementation:

- Closure entries for identifiers bound at global scope can be omitted.
- Procedure calls to statically known target procedures do not need to indirect through a procedure object.
- Only those variables that are actually closed by inner procedures are included in the closure at a given lexical scope. Non-closed variables are stack allocated.
- If a given lexical scope has no closed variables, it can be “dropped out” of the linked list of closures.
- In the presence of a more sophisticated analysis phase, some closures may be stack-allocated rather than heap-allocated. Those compilers that have an analysis phase good enough to support this optimization generally do it.

Siskind identifies some optimizations that require greater care than he implies. For example, he notes that a top-level procedure doesn't need a passed closure at all, and this argument can be eliminated. Unfortunately, this is not true. Statically detected invocations of such a procedure will work fine, but if the procedure object reference is stored into a structure, there is no way for the caller to know whether a closure pointer is desired. In practice, implementing the optimization that Siskind proposes requires a small bit of trampoline code (to pop the unnecessary closure argument) that is used as the code pointer in the procedure object.

When reading Siskind's proposed optimizations, it is important to recognize that he speaks in the context of the STALIN compiler, which performs whole-program analysis.

2.3 Calling Convention Issues

On machines that use register-based calling conventions, it is worth noting that the limit on the number of registers used is generally no more than three. There are good reasons for this:

- Using more registers tends to induce register spills in the caller.
- Every register reserved in the calling convention potentially requires additional saves in the callee if further recursive calls are made.

The main complication introduced by register-based calling conventions is concurrent collection. The concurrent collector needs to know which (if any) registers may be collection roots. Note that this knowledge is really only needed at those points where the collector might preempt the main thread of control. Note further that the problem is mitigated if a mostly-copying collector implementation is feasible [5].

It is possible to thread a middle path: design the code emission so that any object pointer residing in a register also resides someplace else that the collector can reach. This does not directly support copying collectors, but it does mitigate the need to keep the additional data structures needed for mostly-copying collection to work.

There is one other issue that may work significantly in our favor: the call frame size of a procedure in BitC is purely fixed. The language does not support anything analogous to `alloca()`, which actually turns out to be very helpful.

3 BitC Issues

There are several issues to consider that are specific to BitC.

3.1 Non-Collected Mode

For BitC, we very much want to do stack allocation of closures wherever possible. We *must* do so in non-collected mode, and when compiling in this mode the compiler enforces additional restrictions to *guarantee* that heap-allocated closures are not necessary. In particular, escaping lambdas are not permitted at all. This significantly reduces the expressive power of the language, because it precludes things like the `make-counter` pattern:

```
(define (make-counter)
  (let ((n 0))
    (lambda ()
      (set! n (+ n 1))
      (- n 1))))
```

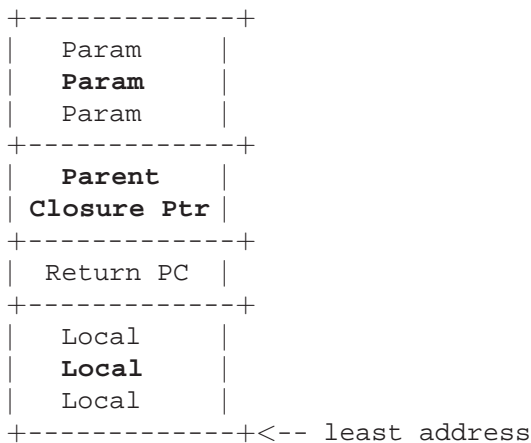
However, it still permits some very useful *downward* uses of nested procedures, such as:

```
(map (lambda (x) ...) some-tree)
```

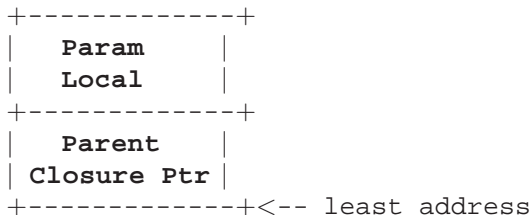
It is tempting to try to come up with an amalgam of these styles wherein allocation of this form could be permitted. Unfortunately, it impacts the calling convention and the code emission strategy, and the performance implications of adopting a GC-friendly code generation strategy for non-GC code seem likely to be quite unpleasant.

3.2 Mutable Variables of Value Type

In languages like SML [12] and Haskell, closure formation is made fairly straightforward by the fact that all mutable cells reside in the heap. Consider a typical SML call frame, where the bold entries are to be copied into a closure:

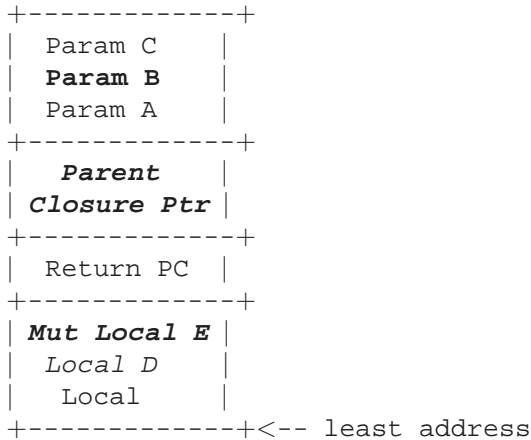


There is no need to consider mutability, and there is no real benefit here in copy avoidance, since the number of uncopied locals and parameters is likely to be larger than the number copied. The structure will get copied into a heap-allocated closure with the closure pointer at a well-known offset:



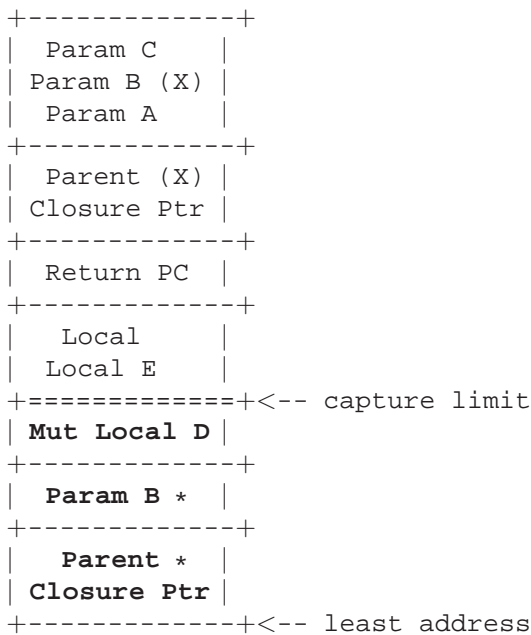
There isn't even a need for the parent procedure to make references via this closure, since all of the elements of the closure itself are (shallowly) constant. It is also permissible (and sometimes desirable) to let different subprocedures receive differently specialized closures that copy different subsets of the lexically enclosing activation.

In a language with closed variables of mutable value type, however, things are rather more delicate. Consider a similar frame, wherein bold items are closed by one nested procedure, italic items by a second, and there is an object of mutable value type that is referenced by both:



It is permissible to rearrange things in the parent by copy, but *only* if this is done before any access is made to the mutable cell(s). It is *not* permissible to make distinct subsets of the closure for nested procedures, because each must be able to see mutations made by the other to the mutable local. While we could heap-migrate the mutable value in a garbage collected environment, this is not an option for the non-collected mode in BitC, and it largely eliminates the benefits of closure stack allocation if it is done. This means that in practice all nested procedures will end up sharing a single closure record.

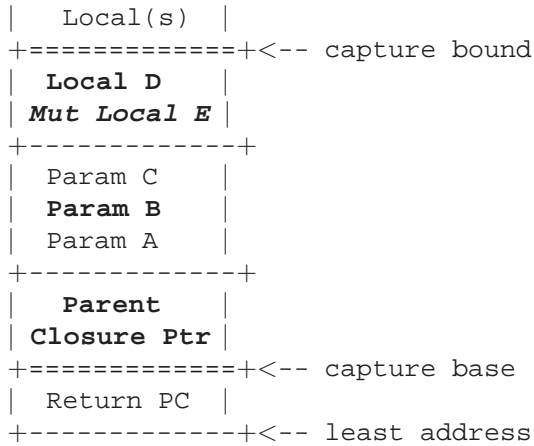
Fortunately, we *can* sort the locals so that the ones that are captured are gathered together. If this is done, and if an early copy is performed at procedure entry, then we will end up with the following frame:



If the frame is stack allocatable, then only the parent pointer requires relocation.

If we are prepared to abandon compatibility with the native calling convention, another option is possible: have the *caller* reserve the space for the callee locals, and also for the closure parent pointer. If this is done, our original frame would look like:



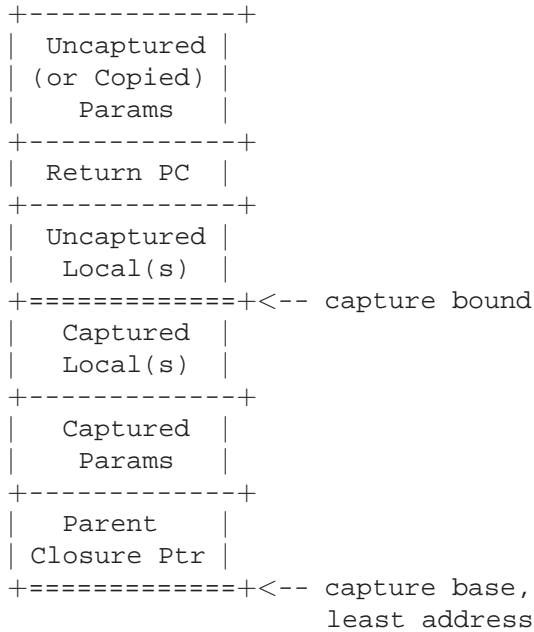


Whether this is desirable depends heavily on the ratio between captured and uncaptured argument bytes. It has the significant *disadvantage* that it cannot be expressed in C.

4 Issues for Generating C

The main issue for generating C concerns the relationship between the C code and the garbage collector. The problem is that arguments may reside in registers that the collector cannot find.

My current plan to deal with this is decidedly not pretty. The proposal is to overload the use of the closure pointer to point to the entire parent frame, with the end result that we form a heap-like linked call stack using the hardware machine stack. After closure rearrangement, each frame will look like:



Note that the parent closure pointer now doubles as a parent frame pointer, which is intentional. This pointer is unconditionally passed at procedure call. The purpose of passing this pointer is to ensure that all of these items are aliased, and that register spills will occur on procedure call. This permits the collector to safely walk the frame chain to find all of the garbage collection roots.

Regrettably, a C-based implementation must copy the parameters. This is because the calling convention may have registerized some of them, and there may not be any stack space allocated to back those parameters. Regrettably, we cannot rely on the compiler to spill them for us predictably.

I cannot help but think that this is all rather a lot of work purely for the sake of avoiding conservative or mostly-copying collection, and that it is probably misguided.

References

- [1] **NOT USED** —: American National Standard for Information Systems, Programming Language C ANSI X3.159-1999, 2000.
- [2] **NOT USED** —: *IEEE Standard for Binary Floating-Point Arithmetic*, 1985, ANSI/IEEE Standard 754-1985.
- [3] **NOT USED** —: *IEEE Standard for Radix-Independent Floating-Point Arithmetic*, 1987, ANSI/IEEE Standard 854-1987.
- [4] Andrew W. Appel and Zhong Shao. “An Empirical Study of Stack vs. Heap Cost for Languages with Closures.” *Journal of Functional Programming*, **1**(1), January 1993
- [5] Joel F. Bartlett. *Mostly-Copying Garbage Collection Picks Up Generations and C++*. WRL Technical Note TN-12. Digital Equipment Corporation, 1989.
- [6] **NOT USED** Jacques Garrigue. “Relaxing the Value Restriction.” *Proc. International Symposium on Functional and Logic Programming*. 2004.
- [7] **NOT USED** Anita K. Jones. *Protection in Programmed Systems*, Doctoral Dissertation, Department of Computer Science, Carnegie-Mellon University, June 1973.

- [8] **NOT USED** Mark Jones. “Type Classes With Functional Dependencies.” *Proc. 9th European Symposium on Programming* (ESOP 2000). Berlin, Germany. March 2000. Springer-Verlag Lecture Notes in Computer Science 1782.
- [9] **NOT USED** M. Kaufmann, J. S. Moore. *Computer Aided Reasoning: An Approach*, Kluwer Academic Publishers, 2000.
- [10] **NOT USED** Richard Kelsey, William Clinger, and Jonathan Rees (Ed.) *Revised⁵ Report on the Algorithmic Language Scheme*, ACM SIGPLAN Notices, 33(9), pp 26–76, 1998.
- [11] **NOT USED** David MacQueen, “Modules for Standard ML.” *Proc. 1984 ACM Conference on LISP and Functional Programming*, pp. 198–207, 1984.
- [12] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised* The MIT Press, May 1997.
- [13] **NOT USED** J. S. Shapiro, J. M. Smith, and D. J. Farber. “EROS, A Fast Capability System” *Proc. 17th ACM Symposium on Operating Systems Principles*. Dec 1999, pp. 170–185. Kiawah Island Resort, SC, USA.
- [14] Jeffrey Mark Siskind. “*Flow-Directed Lightweight Closure Conversion*” Technical Report 99-109R, NEC Research Institute, Inc., Dec. 1999.
- [15] **NOT USED** Zhong Shao and Andrew W. Appel. “Space-Efficient Closure Representations.” *Proc. LISP and Functional Programming*, pp. 150-161, 1994.
- [16] **NOT USED** N. Wirth and K. Jensen. *Pascal: User Manual and Report*, 3rd Edition, Springer-Verlag, 1988