An Unexceptional Implementation of Continuations

Joseph Marshall
Google, Inc.
jmarshall@alum.mit.edu

Abstract

In 'Continuations from Generalized Stack Inspection', Pettyjohn et al. show how to translate a program into a form that allows it to capture and restore its own stack without requiring that the target machine provide stack manipulation primitives. They demonstrate an implementation that uses the native exception handling mechanism to propagate captured control state down the stack.

The technique described appears promising, but performs poorly in practice and is not a reasonable alternative to simply transforming the program to continuation passing style and allocating all continuations in the heap.

I have developed variation on the technique that avoids the use of the exception handler. This variation performs quite well.

In this paper, I compare two Scheme interpreters. The first interpreter is written in C and explicitly manages its own continuations on the heap. The second interpreter is written in C# and runs on the .NET CLR. The second interpreter uses the implicit continuations of the implementation language and the variation of the generalized stack inspection in order to capture and restore the stack. The difference in performance is negligible.

1. Introduction

Continuations are an abstraction of flow control in a computer program. Most computer languages enforce the restriction that flow control between subroutines is strictly last-in, first-out. This allows the implementation to use a simple push-down stack for continuation management. Computer hardware typically provides hardware to manage some part of the memory as a stack. It also provides primitive continuation management in the form of call and return instructions.

Languages that support first-class continuations present a unique challenge to the implementor. It is in general impossible to statically determine the lifetime of a continuation, so use of a push-down stack would lead to incorrect behavior. Yet even these languages the default flow of control is stack-like.

Many implementation strategies for first-class continuations have been developed, and several of these strategies attempt to make at least partial use of a stack in order to take advantage of the simpler allocation and deallocation protocol and to use hardware to accelerate calls and returns. A popular technique is to assume that all continuations are stack-like and to make a snapshot of the stack when a continuation is captured. This technique performs well when continuation capture is a fairly uncommon operation.

Low-level languages, such as C, permit access to the stack through use of pointer arithmetic. Higher level languages, such as Java or C# prohibit arbitrary stack access. Pettyjohn et al.(3) show how to create an equivalent of a snapshot of the stack by annotating the procedures in the call chain. They describe a technique that uses the exception handling mechanism to pass control down the stack.

2. Review of Technique

The technique described in Pettyjohn et al. is somewhat complicated, so a review is in order. For the sake of brevity we will omit a number of details. Readers are encouraged to refer to the original paper for an in-depth explanation.

We start with the observation that although the CLR strictly prevents inspection of the stack, a method has complete access to its own stack frame and can perform any task it wishes such as reifying the frame to the heap. We therefore introduce into each method an additional control path that extracts the dynamic state of the method and appends it to a data structure. To capture a continuation, we throw a special exception to return control to the method along the alternate control path. After appending the dynamic state, the method re-throws the exception. This causes the entire stack to be emptied and the corresponding chain of reified frames to be built. A handler installed at the base of the stack is the ultimate receiver of the exception and it creates a first-class continuation object in the heap using the chain of reified frames.

The task of modifying a method to add a new control path involves several steps. Fortunately, these steps are commonly used as part of the compilation process, so the task may be easier than it first appears. There are six steps to the process.

1. Assignment Conversion - We convert any mutable shared bindings on the stack to allocate mutable cells on the heap. (By 'shared bindings', we mean those bindings that are lexically visible in nested contexts.) This avoids problems with unsharing that may occur when the stack is reified.
2. ANF Conversion - The code is converted to A-normal form [Flanagan93]. This exposes the temporary values and linearizes the control flow by replacing compound expressions with an equivalent sequence of primitive expressions and variable bindings. After ANF conversion, all procedure calls will either be the right-hand side of an assignment statement or a return statement.
3. Live variable analysis - We determine which variables hold active dynamic state. Unused variables are not copied when the continuation is captured.
4. Procedure Fragmentation - We split each method at the procedure call boundaries. This allows us to 're-enter' a method right after each call site.

5. Closure conversion - The continuation frames are closed over the live variables in the original method. We construct an explicit object that represents the closure. The 'body' of the closure is a procedure fragment.
6. Code annotation - Each procedure call is annotated by wrapping an exception handler around the call. This intercepts the special exception thrown for reifying the stack, constructs the closure object from the live variables, appends it to the accumulated stack frame chain, and re-throws the exception.

This technique introduces some overhead along the normal control path. Each procedure call is wrapped with an exception handler and it takes a small amount of time to establish and de-establish the handler. However, a program that has been converted in this way is not much slower than the original provided that it never actually attempts to capture a continuation. (But why would one go through the effort of making this possible without intending to use it?)

Unfortunately, exception handling on the CLR is extremely slow. A simple test shows that returning control to the caller by throwing can easily be more than three thousand times slower than returning control with a return instruction. Given the poor performance when capturing continuations it seemed unlikely that this technique would be competitive with the far more popular methods of keeping control state in the heap.

3. A Scheme Implementation

The author has written his own Scheme implementation for the purposes of experimenting with various interpretation and compilation techniques. This implementation is based on the MIT implementation of Scheme, but it runs on the .NET CLR.

MIT Scheme has two components: a library of Scheme code that provides various high-level services, and a virtual machine interpreter. The virtual machine, called the 'microcode' (an early implementation was literally the microcode for the Scheme chip), interprets a tagged tree structure called 'S-Code'. The 'S-Code' is essentially a realization of the abstract syntax tree of the Scheme language. The microcode recursively descends the S-Code and at each node it dispatches to a handler that interprets that node.

The current version of the MIT Scheme microcode is written in C. In order to achieve tail recursion, the main loop of the interpreter is contained entirely in a single C function. The continuations necessary for executing the Scheme program are managed in a large static array that represents the stack.

The author wrote a new version of the interpreter in C#. It is not a port of the C interpreter, but a new design. However, the new interpreter is able to directly read the binary image of the MIT Scheme S-code. The primitive API provided by the C interpreter is also duplicated so the existing high-level Scheme code can be run without modification.

The initial implementation of the C# interpreter used the standard trick of a 'trampoline' to achieve tail recursion. When a method wished to transfer control to another method, it returned control to the trampoline which then invoked the target method. This ensures that the C# stack remains at a fixed depth. The Scheme continuations were simply allocated on the heap.

The interpreter worked, but the performance was disappointingly poor. Profiling the application revealed that continuation manipulation was the primary bottleneck. This is hardly surprising, but Appel and others have suggested that heap allocation coupled with a decent garbage collector is competitive with stack allocation(1). Under these particular circumstances, this does not seem to be the case.

Miller and Rozas(6), however, argue that Appel's analysis abstracts out an important detail of the implementation of the stack: that a stack is implemented with a single pointer into a contiguous block of memory. Taking this into account, they predict a noticeable difference in performance and observe a measured performance improvement of up to a factor of three when the stack is used rather than the heap.

In order to improve the performance of the Scheme interpreter, the author rewrote the code to use the technique described in Pettyjohn et al. However, to avoid the incredible overhead of exception handling, the author modified the technique. Although the modified code is not novel (and rather obvious), it combines nicely with three other standard tricks.

4. Modification of Technique

While the CLR supports tail recursive calls, the C# compiler does not emit the instruction. To get tail recursion in C#, the author used trampolines. But rather than have a single trampoline invoked at each control transfer, the code was modified to have a trampoline at each recursive call to EVAL in the interpreter. Calls that are not tail recursive (so-called 'subproblem' calls) are invoked using the standard method call mechanism of C#. Calls that are tail recursive return to the caller and use the trampoline to do the transfer of control. Invoking a continuation simply involves returning a value to the caller.

The original interpreter trampoline was coded like this:

```
Control expr = initialExpression;

while (true) {
    expr = expr.Step();
}
```

Control is an abstract base class that all classes associated with evaluation would inherit from. The Step method would perform part of the evaluation specific to the particular subclass of Control, and then return a new Control object to be stepped. Since no state whatsoever was held on the stack, any dynamic state had to be encapsulated in a heap object prior to returning to the trampoline. This results in a ferocious amount of object construction.

In the new code, each call site has a trampoline. But the caller now needs to distinguish between returning in order to release the callee's stack frame versus returning in order to return the result of a computation. This could be done in a number of ways ranging from returning a distinguished type, returning multiple values, putting a token in a global variable, etc. The author chose to take advantage of the CLR's 'out' and 'ref' parameters. The call site is therefore like this:

```
Control expr = initialExpression;
object answer;

while (expr.EvalStep (out answer, ref expr)) { };
```

In other words, the EvalStep method returns a boolean. If the boolean value is false, the subproblem has finished its task and produced and answer. This is stored in the ANSWER variable. The ANSWER variable is passed as a reference to the EvalStep method, so it can be modified by the callee. If, on the other hand, the returned boolean value is true, this indicates that additional computation is to be performed. The new control expression has been stored in the EXPR variable. The EXPR variable is also passed by reference so the callee may modify it. (The difference between 'out' and 'ref' is simply that 'out' parameters are allowed to be uninitialized when passed.)

The callee method would look like this:

```
boolean EvalStep (out object answer, ref Control expr) {

  < code that implements the expression >

  if (...) {
      // Return an answer.
```

```
        answer = 33;
        return false; // false means no futher computation
    } else {
        expr = nextControlObject;
        return true; // true means tail call request.
    }
}
```

There is one further change. Since this a Scheme interpreter, each evaluation method needs an environment structure to hold the lexical variables. This is also supplied as a ref argument.

With these changes, we achieve two important things: our continuations are now implemented directly as CLR continuations, and, by using the 'ref' and 'out' parameters, we no longer need to allocate temporary storage to return the multiple items used in maintaining the interpreter state. In essence, the interpreter state variables holding the expression and environment are allocated in the callee and can thus survive the return through the trampoline.

We could have chosen one of a number of different permutations or variations on the above. For example, we could have had the answer be returned directly and the boolean flag returned as an 'out' parameter. However, the calling convention of COM components, which is important to the Windows operating system, returns a status word as the return value and uses 'out' parameters to handle the values. It seems likely that the C# compiler would take care to ensure that this pattern of usage performs no more poorly than the alternatives.

But we have neglected the first-class continuations. We could wrap each call site with an exception handler, but we want to avoid that. Given that we already test the return value on each return, it would not be much burden to add an additional test. We create a unique singleton object that acts as a signal to capture the continuation and test for it when the callee returns. The call site now looks like this:

```
object temp;
Control expr = initialExpression;
Environment env = initialEnvironment;

while (expr.EvalStep (out temp, ref expr, ref env)) { };
if (temp == Interpreter.UnwindStack) {
    <capture the current frame>
    answer = Interpreter.UnwindStack;
    return false;
    }
```

The value returned is tested against the distinguished value to see if we are attempting to capture the stack. If so, we capture our frame and propagate the return value. While this test must be performed after each call, it is very lightweight — comparable to the cost of establishing and de-establishing an exception handler. On the other hand, the cost of transferring control to the path that captures the stack now involves a return and conditional branch and is therefore tens of thousands of times quicker.

One further trick completes the task. We need to propagate the unwinding state as we capture the stack. In the original design, the unwind state was encapsulated within the special exception that was thrown. In the new design, the environment variable turns out to be the most convenient place. This is an unusual choice, but the lexical environment is not needed during unwinding and is present at every call site. The problem is that the type system prevents us from using the environment variable for this unrelated purpose.

Fortunately, a rather large hole in the type system can be exploited in a fairly nasty way. We create a class that inherits from the Environment class and override every inherited method with one that throws a runtime error. Then we add additional methods that support stack unwinding. The resulting object is syntactically an 'Environment' object because it inherits from Environment, but is semantically unrelated because it supports none of the methods usually associated with an environment. This is sufficient to fool the type system into letting us pass the unwind state in the environment variable.

Implementation	Median Time	10log10 Time
MIT Scheme, C microcode	5.100 s	7.08
MIT Scheme, C# microcode	3.928 s	5.94
Common Larceny	18.299 s	12.6

Table 1. Performance Comparison, nboyer Benchmark

Implementation	Median Time	10log10 Time
MIT Scheme, C microcode	0.391 s	-4.08
MIT Scheme, C# microcode	0.414 s	-3.83
Common Larceny	2.340 s	3.69

Table 2. Performance Comparison, earley Benchmark

For a concrete example, the code that currently implements IF expressions is given here:

```
public override bool EvalStep (out object answer,
                               ref Control expression,
                               ref Environment environment)
{
    object ev;
    Control unev = this.predicate;
    Environment env = environment;

    while (unev.EvalStep (out ev, ref unev, ref env)) {};
    if (ev == Interpreter.UnwindStack) {
        ((UnwinderState) env).AddFrame (
            new ConditionalFrame (this, environment));
        environment = env;
        answer = Interpreter.UnwindStack;
        return false;
    }

    if ((ev is bool) && (bool) ev == false) {
        expression = this.alternative;
        answer = null;
        return true;
    }
    else {
        expression = this.consequent;
        answer = null;
        return true;
    }
}
```

5. Results

Tables 1 and 2 gives the time for running two simple Scheme benchmarks. The nboyer is run with an argument of 1, the earley benchmark takes no arguments. These benchmarks are available as part of the Common Larceny distribution.

The changes to the interpreter have the desired effect of using the stack rather than the heap to maintain the continuations. The ability to reflect and reify the stack is provided by the technique in Pettyjohn et al., but the high cost of exception handling is avoided by using a distinguished return value.

The resulting interpreter performed quite a bit better than the original design. Since both the MIT Scheme virtual machine and this interpreter support the exact same api, it is possible to do a side-by-side comparison. We found that in several benchmarks the performance difference between the interpreter written in C and the one written in C# is negligible.

A further comparison is between this version of Scheme on the CLR and Common Larceny(7), another implementation of Scheme for the CLR. Common Larceny compiles Scheme into the .NET IL instructions. It is able to compile entire modules into a single block of code, so control transfer within a module need not use

a trampoline. However, it explicitly manages continuations in the heap. Rather than rely on the .NET garbage collector to free unused continuations, Common Larceny allocates and manages a pool of continuation objects. Nonetheless, both MIT Scheme and the author's reimplementation perform noticeably better than the original release of Common Larceny.

6. Discussion

Why should there be such a performance change between allocating on the stack versus the heap? Miller and Rozas showed that a stack is quicker if you look at highly optimized code, but the high-level Scheme code is removed from the hardware by a layer of interpretation in C and a layer of interpretation and a layer of JIT compilation in C#.

It turns out that difference between stack and heap allocation of continuations is amplified by both modern hardware and virtual machine based languages. The call/return conventions of the virtual machine are very restricted and map directly into the continuations provided by the processor. The processor itself has specialized hardware and caches designed solely to speed access to the stack. The stack pointer itself is a special register. A method call in C# turns into a small number of instructions (an indirect call) in the hardware, and this path has been turned to perform well.

If continuations are managed on the heap, however, this hardware is wasted. In fact, since the hardware is maintaining the underlying continuation structure whether we want it to or not, it hurts performance when it gets in the way (on tail recursive calls, for example). We cannot use the stack pointer to address continuations, so we have to use one of the very few other registers in the processor. Since the other registers are 'general purpose', they cannot be optimized for the extremely common cases of control transfer.

Nor can the processor get hints from the instruction stream. A processor can notice a return instruction when it enters the decoding logic and prepare the stack for the upcoming transfer. If the continuation is on the heap, however, the control transfer occurs via an indirect address stored at an offset from one of the registers. The processor has no easy way of predicting which address will be used.

7. Conclusion

It is very much worthwhile to allocate continuations on the stack, especially when using higher-order virtual machines. The standard technique of using a trampoline interacts well with a simple variation of the stack inspection mechanism described in Pettyjohn et al., and the resulting code has reasonable performance.

A. Appendix 1, Implementations

The version of MIT Scheme used in testing was

```
Release 7.7.90.+ || Microcode 15.1 || Runtime 15.7
Win32 1.8 || SF 4.41 || LIAR/i386 4.118
Edwin 3.116      || SOS 1.8
```

The S-Code interpreter of MIT Scheme is designed to be simple, not fast. It is assumed that performance critical code will be compiled. Earlier versions of the system did not rely on the compiler, and the interpreter had several performance tweaks. Most of these have been removed.

Library routines in the Scheme runtime are compiled in the C version of MIT Scheme, but are interpreted in the C# version. The effect on benchmarking varies depending on whether the benchmark calls routines like MEMQ or ASSQ, or whether it uses explicit code that is similar to MEMQ or ASSQ.

The version of Common Larceny used was

```
CommonLarceny
CLR, Version=2.0.50727.3053
Scheme, Version=1.0.2063.20317
```

This version dates from August 1995.

The C# implementation of MIT Scheme has a number of performance tweaks that can be enabled at compilation time. These tweaks can greatly increase the performance. Most of these tweaks, however, were disabled for these benchmarks in order to more closely duplicate the evaluation strategy of the C implementation of MIT Scheme.

The following flags, however, were enabled:

1. EnableVariableBinding — When enabled, the interpreter will perform a pre-pass over the S-Code to determine the location class of the variables. Variables are divided into arguments (those with lexical depth of zero), lexical variables with depth greater than zero, and global variables. This flag enables only the annotation of the variables and does not affect the evaluation, but it must be present for the other variable to work.
2. EnableArgumentBinding — When enabled, variables that are at lexical depth zero (arguments to the immediately enclosing lambda expression) are evaluated through a special fast code path.
3. EnableGlobalBinding — When enabled, variables that are bound at top-level are cached and a special test is done to check that they are not shadowed.
4. EnableSimpleLambda — When enabled, lambda expressions with a fixed number of arguments (no optional or rest arguments) are given a special tag so that environment construction can be accelerated.
5. EnableStaticLambda — When enabled, lambda expressions that contain no internal definitions or calls to (the-environment) are specially marked. MIT Scheme allows incremental definition to some first-class environments. Dynamically adding incremental definitions can change the location of bound variables. This makes it necessary to perform a deep search on every variable access in case the location changed. Static lambdas do not support incremental definition so variable lookup can assume that the location of bound variables is static.
6. EnableInlinePrimitive1 — When enabled, calls to most primitive procedures of a single argument are performed through a fast path rather than going through the standard APPLY call.
7. EnableInlinePrimitive2 — When enabled, calls to most primitive procedures of two arguments are performed through a fast path rather than going through the standard APPLY call.

An important performance enhancement was left disabled: EnableLexicalAddressing. When this is enabled, the lexical depth and offset of the variable (as determined during the pre-pass) is cached with the variable and a deep search is avoided. However, because MIT Scheme has removed this feature from its interpreter, I disabled it for these benchmarks. A deep search was performed for each lexical variable at a lexical depth greater than zero.

B. Appendix 2, Cost of Throwing

In July 2005, Microsoft eased its restrictions on reporting benchmark figures for the CLR. See http://msdn.microsoft.com/en-us/library/ms973265.aspx

The header on the test output gives the necessary disclosures, and the code is freely available at http://code.google.com/p/jrm-code-project/

```
; LENOVO ThinkPad T61
```

```
; 2 processors
;     GenuineIntel
;     Intel(R) Core(TM)2 Duo CPU     T7700  @ 2.40GHz
;     x64 Family 6 Model 15 Stepping 11
;     2401 MHz
; Working set:  9826304
; Overflow checking is disabled.
; Compiled in RELEASE mode.
; Debugger is not attached.
; Stopwatch is high resolution.  14318180 ticks per second.
;
; 2009-Feb-08 21:11:55
; Microsoft Windows NT 6.0.6001 Service Pack 1
; CLR 2.0.50727.3053
;
; See http://eval.apply.googlepages.com/ for further information.
;
; Unexceptional, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
; Debugging Flags: IgnoreSymbolStoreSequencePoints
; JIT Optimizer enabled
; JIT Tracking disabled

Starting Return test benchmark, 21 iterations...done.
Median return time: 0.000272939717198694
Starting Throw test benchmark, 21 iterations...done.
Median exception time: 1.0473244504539
Ratio = 3837
```

References

[1] Appel, A. W. 1987. Garbage collection can be faster than stack allocation. Inf. Process. Lett. 25, 4 (Jun. 1987), 275-279. DOI= http://dx.doi.org/10.1016/0020-0190(87)90175-X

[2] Fnfrocken, S. 1999. Transparent Migration of Java-Based Mobile Agents. In Proceedings of the Second international Workshop on Mobile Agents K. Rothermel and F. Hohl, Eds. Lecture Notes In Computer Science, vol. 1477. Springer-Verlag, London, 26-37.

[3] Pettyjohn, G., Clements, J., Marshall, J., Krishnamurthi, S., and Felleisen, M. 2005. Continuations from generalized stack inspection. In Proceedings of the Tenth ACM SIGPLAN international Conference on Functional Programming (Tallinn, Estonia, September 26 - 28, 2005). ICFP '05. ACM, New York, NY, 216-227. DOI= http://doi.acm.org/10.1145/1086365.1086393

[4] Schinz, M. and M. Odersky. Tail call elimination on the java virtual machine. In Proc. ACM SIGPLAN BABEL '01 Workshop on Multi-Language Infrastructure and Interoperability., volume 59 of Electronic Notes in Theoretical Computer Science, pages 155- 168. Elsevier, 2001. http://www.elsevier.nl/locate/entcs/ volume59.html.

[5] Sekiguchi, T., T. Sakamoto and A. Yonezawa. Portable implementation of continuation operators in imperative languages by exception handling, volume Advances in Exception Handling Techniques, pages 217-233. Springer-Verlag, 2001.

[6] Miller, J. S. and Rozas, G. J. 1994 Garbage Collection is Fast, But a Stack is Faster. Technical Report. UMI Order Number: AIM-1462., Massachusetts Institute of Technology.

[7] William D Clinger. Common Larceny. Proceedings of the 2005 International Lisp Conference, June 2005, pages 101-107. See http://www.ccs.neu.edu/home/will/papers.html

[8] William Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for first-class continuations. Higher-Order and Symbolic Computation, 12(1):7(45), 1999.