Compiling with Continuations and LLVM

Kavon Farvardin
John Reppy

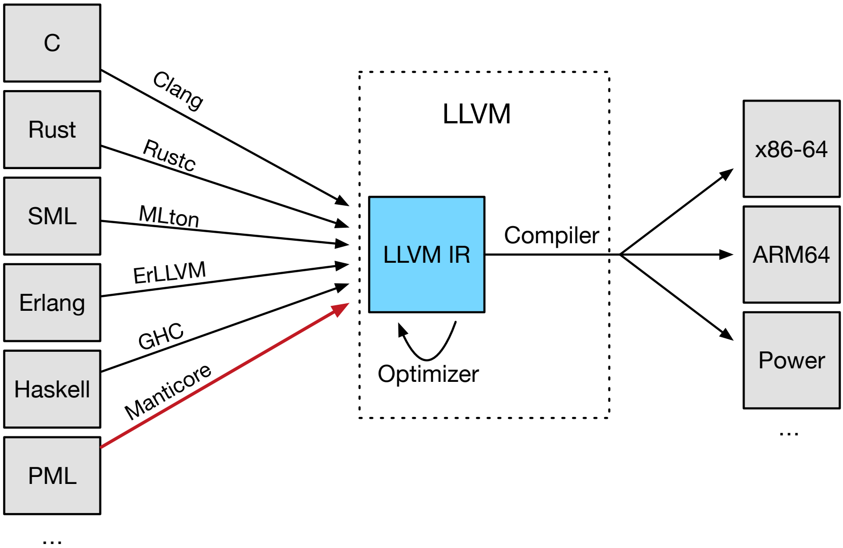
University of Chicago

September 22, 2016

Introduction to LLVM

- De facto backend for new language implementations
- Offers high quality code generation for many architectures
- Active industry development
- Widely used for research
- Includes a multitude of features and tools

The LLVM Landscape

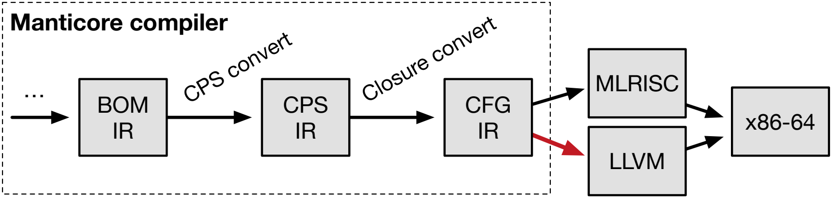


Characteristics of LLVM IR

```
define i32 @factorial(i32 n) {
    isZero = compare eq i32 n, 0
    if isZero, label base, label recurse

base:
    res1 = add i32 n, 1
    goto label final

recurse:
    minusOne = sub i32 n, 1
    retVal = call i32 @factorial(i32 minusOne)
    res2 = mul i32 n, retVal
    goto label final

final:
    res = phi i32 [res1, res2]
    return i32 res
}
```

Manticore's Runtime Model

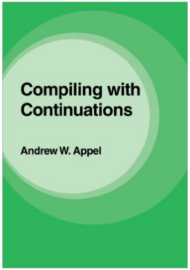
- Efficient first-class continuations are used for concurrency, work-stealing parallelism, exceptions, etc.
- As in *Compiling with Continuations*, return continuations are passed as arguments to functions.
- Continuations are heap-allocated, making `callcc` cheap.
- Functions return by throwing to an explicit continuation.



This Model Poses a Challenge for LLVM

We require

- Efficient, reliable tail calls
- Garbage collection
- Preemption and multithreading
- First-class continuations



?

+

Efficient, Reliable Tail Calls

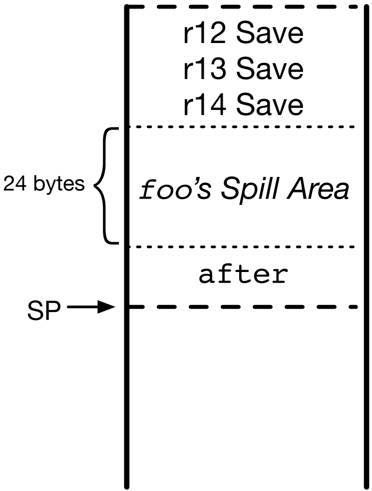
- Tail calls are a major correctness and efficiency concern for us.
- LLVM's tail call support is shaky: the issues are numerous and fixes are hard to come by.

Anatomy of a Call Stack

```
foo:
    push  r12
    push  r13
    push  r14
    sub   sp, 24
; body of foo
    call bar
after:
; body of foo
    add   sp, 24
    pop   r14
    pop   r13
    pop   r12
    ret
```

Prologue

Epilogue



LLVM's Tail Call Optimization

```
foo:
    push r12
    push r13
    push r14
    sub  sp, 24
; body of foo
    call bar   ; <--
    add  sp, 24
    pop  r14
    pop  r13
    pop  r12
    ret        ; <--
```

```
foo:
    push r12
    push r13
    push r14
    sub  sp, 24
; body of foo

    add  sp, 24
    pop  r14
    pop  r13
    pop  r12
    jmp  bar   ; <--
```

Avoiding the Tail Call Overhead

- MLton uses a trampoline, reducing procedure calls.
- GHC's calling convention removes only callee-save instructions.
- We remove *all* overhead with a new calling convention (JWA) **plus** the use of naked functions.

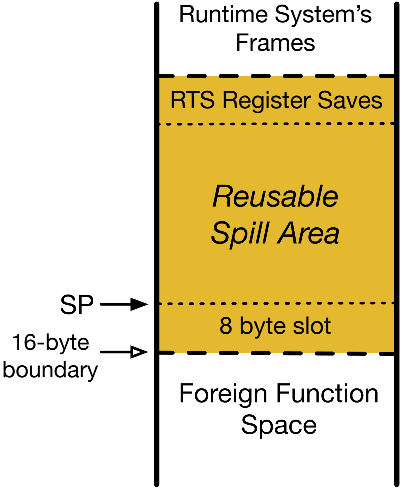
⚠ Naked functions blindly omit all frame setup, *requiring* you to handle it yourself!

GOAL →

```
foo:
; body of foo
    jmp bar
```

Using Naked Functions

- Runtime system sets up frame
- Compiler limits number of spills
- All functions reuse same frame
- FFI calls are transparent



Garbage Collection

- Cannot use LLVM's GC support; assumes a stack runtime model.
- Manticore's stack frame is only for temporary register spills.
- Thus, no new stack format to parse; our GC remains unchanged.
- We insert heap exhaustion checks before LLVM generation.

Example of a Heap Exhaustion Check

```
declare {i64*, i64*} @invoke-gc(i64*, i64*)

define jwa void @foo(i64 allocPtr_0, ... ) naked {
...
    if enoughSpace, label continue, label doGC

doGC:
    roots_0 = allocPtr_0
    ; ... save live vals in roots_0 ...
    allocPtr_1 = getelementptr allocPtr_0, 5 ; bump
    fresh = call {i64*, i64*} @invoke-gc(allocPtr_1, roots_0)
    allocPtr_2 = extractvalue fresh, 0
    roots_1 = extractvalue fresh, 1
    ; ... restore live vals ...
    goto label continue

continue:
    allocPtr_3 = phi i64* [ allocPtr_0, allocPtr_2 ]
    liveVal_1 = phi i64* [ ... ]
...
```

Preemption and Multithreading

- Continuations are a natural representation for suspended threads.
- Multithreaded runtimes must asynchronously suspend execution.
- When using a precise GC, safe preemption is challenging.

Preemption at Garbage Collection Safe Points

Heap tests can be used for preemption:

- Threads keep their heap limit pointer in shared memory.
- We preempt by forcing a thread's next heap test to fail.
- Preempted threads reenter runtime system via `callcc`.
- Non-allocating loops are also given a heap test.

```
fun foo x =
  ...
  if limitPtr - allocPtr >= bytesNeeded
    then foo y
    else (callcc enterRTS ; foo y)
  ...
```

First-class Continuations in LLVM

- Preemptions need to occur in the middle of a function.
- In CwC, we allocate a function closure to capture a continuation.

Problem

LLVM does not have first-class labels to create the closure!

First-class Labels in LLVM

Observations:

- The return address of a *non-tail* call is a label generated at runtime.
- Return conventions for C structs specify a mix of stack/registers.

Solution

We treat the return address like a first-class label by specifying a return convention for C structs that matches calls.

The Jump-With-Arguments Calling Convention

Arguments Passed	Arg 1	Arg 2	Arg 3	Arg 4	…
	↓	↓	↓	↓	
Location of Value	rsi	r11	rdi	r8	…
	↑	↑	↑	↑	
C Struct Returned	Field 1	Field 2	Field 3	Field 4	…

Example of First-class Labels for callcc

```
define jwa void @foo( ... ) naked {
...
preempted:
  env = ; ... save live vars ...
  closPtr = allocPair (undef, env)
  ret = call jwa {i64*, i64*} @genLabel(closPtr, @enterRTS)
  arg1 = extractvalue ret, 0
  arg2 = extractvalue ret, 1
...
}


; call convention:
; rsi = closPtr, r11 = @enterRTS
genLabel:
  pop rax          ; put return addr in rax
  mov rax,(rsi)    ; finish closure
  jmp r11
```

Example of First-class Labels for callcc

```
_foo:
...
preempted:
  ; r10 = env, rsi = closPtr (unintialized)
  mov r10, 8(rsi)
  mov _enterRTS, r11
  call genLabel
  ; return convention:
  ; rsi = arg1, r11 = arg2
  ...


; call convention:
; rsi = closPtr, r11 = @enterRTS
genLabel:
  pop rax          ; put return addr in rax
  mov rax,(rsi)    ; finish closure
  jmp r11
```

Performance Comparison

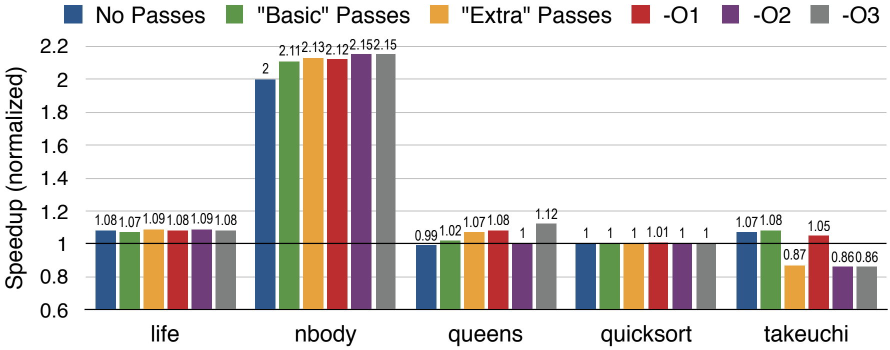
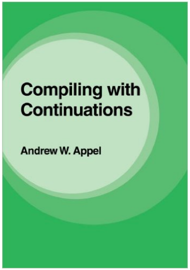


Figure: Execution time speedups over MLRisc when using LLVM codegen.

Conclusion and Future Work

- Hope to apply this to SML/NJ in the future.
- Plan to upstream JWA convention.
- More implementation details in our forthcoming tech report!



+

(with modifications)

`http://manticore.cs.uchicago.edu`