LLVM and Clojure

Runtime Native Code Generation in a Lisp

@timbaldridge
http://github.com/halgari/mjolnir

Why are you doing this?

- I enjoy micro-optimizations

Why are you doing this?

- I enjoy micro-optimizations
- I enjoy writing compilers

Why are you doing this?

- I enjoy micro-optimizations
- I enjoy writing compilers
- There are some modern optimizations not available to programs running on the JVM

Why are you doing this?

- I enjoy micro-optimizations
- I enjoy writing compilers
- There are some modern optimizations not available to programs running on the JVM
- Because I can!

Native code is Complex

- ISAs differ radically
- Processors differ between generations
  - Different sets of registers
  - Different instructions
  - Different methods of encoding instructions
- We live in a multi-ISA world
  - x86 is used in Desktops/Laptops
  - ARM is used in tablets, phones
  - PTX, AMD IL on GPUs

Native code is Complex

- Can we abstract all this complexity?
- Can native code generation be made easier?

Enter: LLVM

- LLVM (Low Level Virtual Machine)
- Abstracts away machine code generation.
- Started in 2000 at University of Illinois
- Several companies hire developers to improve LLVM
- Compilers have been written for: Ada, C, C++, C#, D, Fortran, Objective-C, Haskell, Ruby and many others using LLVM.

What does LLVM look like?

```
static Function *CreateFibFunction(Module *M, LLVMContext &Context) {
  // Create the fib function and insert it into module M. This function is said
  // to return an int and take an int parameter.
  Function *FibF =
    cast<Function>(M->getOrInsertFunction("fib", Type::getInt32Ty(Context),
                                          Type::getInt32Ty(Context),
                                          (Type *)0));

  // Add a basic block to the function.
  BasicBlock *BB = BasicBlock::Create(Context, "EntryBlock", FibF);

  // Get pointers to the constants.
  Value *One = ConstantInt::get(Type::getInt32Ty(Context), 1);
  Value *Two = ConstantInt::get(Type::getInt32Ty(Context), 2);

  // Get pointer to the integer argument of the add1 function...
  Argument *ArgX = FibF->arg_begin();   // Get the arg.
  ArgX->setName("AnArg");            // Give it a nice symbolic name for fun.

  // Create the true_block.
  BasicBlock *RetBB = BasicBlock::Create(Context, "return", FibF);
  // Create an exit block.
  BasicBlock* RecurseBB = BasicBlock::Create(Context, "recurse", FibF);

  // Create the "if (arg <= 2) goto exitbb"
  Value *CondInst = new ICmpInst(*BB, ICmpInst::ICMP_SLE, ArgX, Two, "cond");
  BranchInst::Create(RetBB, RecurseBB, CondInst, BB);

  // Create: ret int 1
  ReturnInst::Create(Context, One, RetBB);

  // create fib(x-1)
  Value *Sub = BinaryOperator::CreateSub(ArgX, One, "arg", RecurseBB);
  CallInst *CallFibX1 = CallInst::Create(FibF, Sub, "fibx1", RecurseBB);
  CallFibX1->setTailCall();

  // create fib(x-2)
  Sub = BinaryOperator::CreateSub(ArgX, Two, "arg", RecurseBB);
  CallInst *CallFibX2 = CallInst::Create(FibF, Sub, "fibx2", RecurseBB);
  CallFibX2->setTailCall();


  // fib(x-1)+fib(x-2)
  Value *Sum = BinaryOperator::CreateAdd(CallFibX1, CallFibX2,
                                         "addresult", RecurseBB);

  // Create the return instruction and add it to the basic block
  ReturnInst::Create(Context, Sum, RecurseBB);

  return FibF;
}
```

Can we do better?

- LLVM's interface is in C++
- Static Single Assignment (SSA)
  - A virtual machine with an infinite number of registers
  - Each register is assigned once
  - Uses blocks for flow control
    - Single entry/exit point for blocks
    - A block can be entered from multiple points and can exit to multiple points
  - Simple, but not easy.

Can we do better?

- Building blocks requires the use of mutable "builder" objects.
- But I want S-Exprs, not statements.
- I want to write in Clojure, not C++
- I want to make writing a compiler easy.
  - or at least easier

Can we do better?

- How do we "slay" the "dragons" of LLVM?

Can we do better?

- How do we "slay" the "dragons" of LLVM?
- With a big magical hammer!

Introducing Mjolnir

- Mjolnir is a library that makes generating code with LLVM and Clojure easier.
- It not only wraps, it extends LLVM to allow for new abstractions and easier construction of code.
- The power of LLVM with the comfort of a dynamic immutable lisp.

Layers of Mjolnir

- LLVM (C++ API)

Layers of Mjolnir

- LLVM-C (C API)
- LLVM (C++ API)

Layers of Mjolnir

- llvmc.clj (JNA Interface)
- LLVM-C (C API)
- LLVM (C++ API)

Layers of Mjolnir

- Mjolnir Expressions (Clojure Records)
- llvmc.clj (JNA Interface)
- LLVM-C (C API)
- LLVM (C++ API)

Layers of Mjolnir

- Constructors (Macros + Functions)
- Mjolnir Expressions (Clojure Records)
- llvmc.clj (JNA Interface)
- LLVM-C (C API)
- LLVM (C++ API)

Layers of Mjolnir

- Constructors (macros + functions)
- Mjolnir Expressions (Clojure records)
- Mjolnir SSA (transforms via datalog)
- llvmc.clj (JNA Interface)
- LLVM-C (C API)
- LLVM (C++ API)

Expressions

```
(ns example
  (:require [mjolnir.types :refer :all]
            [mjolnir.expressions :refer :all]))

(def I64 (->IntType 64))
(def IncFuncType (->FunctionType [I64] I64))

(def fnc (->Fn "inc" IncFuncType ["x"]
           (->IAdd (->Argument 0)
                   (->Const I64 1))))

(def module (->Module "Test" [fnc]))

(build module)
```

Constructors

```
(ns example
    (:require [mjolnir.types :refer :all]
              [mjolnir.constructors-init :as cinit])
    (:alias c mjolnir.constructors))

(c/defn fib [Int64 x -> Int64]
  (c/if (c/< x 2)
    x
    (c/+ (fib (c/- x 1))
         (fib (c/- x 2)))))
```

Demo Time!

- Let's build a compiler (or two).

Native BF Compiler

- Has 30k "slots"
- 8 Instructions

<	**Decrements Index Pointer**
>	**Increments Index Pointer**
+	**Increments value at IP**
-	**Decrements value at IP**
.	**Prints value at IP**
,	**Reads in a char at IP**
[	**Start loop**
]	**End loop**

Hello World in BF

```
++++++++++
[>+++++++>++++++++++>+++>+<<<<-]>++.
>+.+++++++..+++.>++.<<+++++++++++++++.
>.+++.------.--------.>+.>.
```