Building Small Native Languages with Haskell

Paris Functional Programming Meetup
Berlin Haskell Meetup
Haskell Basics

- Higher Order Functions
- Polymorphism
- Typeclasses
- Higher-kindred types
- Monoids
- Monads
- Applicatives
- State Monad
- Writer Monad
- Monad Transformers
github.com/sdiehl/paris-fp

http://www.stephendiehl.com/llvm/

Longer form of this talk.
Motivation

The smallest lambda calculus derivative language that compiles to native code.

Virtual machines (SECD, CAML, Z-Machine, G-machine) are interesting points in the design space, but industrial uses often have different constraints.

Many use-cases in image processing, signal processing, numerical computing, query planning, data analytics that require the ability to generate fast native code dynamically from a mathematical description of the kernel.
rivendell% ./petit

_ _ _ ___| |(_)___| |
| `'\ / -_ ) _| | _| Minimalist Language
| .__\_\_\_\|\_\_\_\|\_ Type :help for help
|_/|

>> :load example.pet
>> ...

rivendell% ./petit -i example.pet -o example
./example
LLVM

LLVM is a collection of modular and reusable compiler and toolchain technologies.
This Talk is in 3 Languages

Host Language

Haskell

Surface Language

```
data Expr
    = Var Name
    | App Expr Expr
    | Lam [Name] Expr
    | Let Name Expr Expr
    | Lit Literal
    | If Expr Expr Expr
    | Prim Prim

deriving (Show, Eq, Ord)
```

Target Language

LLVM
data Expr
  = Var Name
  | App Expr Expr
  | Lam [Name] Expr
  | Let Name Expr Expr
  | Lit Literal
  | If Expr Expr Expr
  | Prim Prim
  deriving (Show, Eq, Ord)

data Literal
  = LInt Int
  | LBool Bool
  deriving (Show, Eq, Ord)

data Prim
  = Add
  | Sub
  | Mul
  | Eql
  deriving (Eq, Ord, Show)

data Decl
  = FunDecl Name [Name] Expr
  deriving (Eq, Show)

data Program = Program [Decl]
  deriving (Show, Eq)
Parser

Not shown because parsers are boring. Google "parsec tutorial"
Functional Compilation

The lambda calculus does not admit a naive translation into LLVM due to the presence of currying, partially applied functions, and partially saturated functions.

Via transformations we can manipulate the program into a form that is amenable to translation to low level machine code.

*Can always turn functional code into an explicitly compilable form using continuation passing transformations. This is of limited usefulness in practice and is difficult to optimize on modern CPUs.*
**Closure conversion** adjusts all call sites to be saturated (adding extra arguments to calls) so as the function does not introduce a closure for the lifted lambda expression.

```haskell
type Env = Map Name Name
type Lift a = WriterT [Decl] (State (Int, Env)) a

\x -> (+x)   =>   \x y -> x + y   Closure Conversion
```
Lambda lifting is a transformation that restructures a program so that functions are defined independently of each other in a global scope. An individual "lift" transforms a local function into a global function:

- Eliminating free variables in the function by adding parameters.
- Moving all functions to fully saturated global functions with all arguments passed.

```
function f(x, y) {
  function g(a, b) {
    return (x + y + a)
  }
  return g(x, y)
}
```

```
function g(a, b, x) {
  return (x + y + a)
}
```

```
function f(x, y) {
  return g(x, y, x)
}
```
Lambda Lifting

\[\text{liftExpr} \;::\; \text{Name} \to \text{Expr} \to \text{Lift Expr}\]

\[\text{liftExpr} \;\text{root} = \begin{cases} \text{Var} \;\text{x} & \to \begin{cases} \text{mx}' & \leftarrow \text{relocate x} \\ \text{case} \;\text{mx}' & \text{of} \\ \text{Just} \;\text{x}' & \rightarrow \text{pure} (\text{Var} \;\text{x}') \\ \text{Nothing} & \rightarrow \text{panic} ("\text{Variable not in scope: }" \leftarrow (\text{toS} \;\text{x})) \end{cases} \\ \text{Lam} \;\text{vs} \;\text{e} & \to \begin{cases} \text{let} \;\text{sc} = \text{Map}\!.\text{fromList} \;[(\text{x},\text{x}) \mid \text{x} \leftarrow \text{vs}] \\ \text{e}' & \leftarrow \text{scope sc} \;\text{(liftExpr} \;\text{root e}) \\ \text{tell} [\text{FunDecl} \;\text{root} \;\text{vs} \;\text{e}'] \\ \text{return} (\text{Var} \;\text{root}) \end{cases} \end{cases}\]

\[\text{App} \;\text{x1} \;\text{x2} & \rightarrow \text{App} \;<\$> \;\text{liftExpr} \;\text{root} \;\text{x1} \;<\star> \;\text{liftExpr} \;\text{root} \;\text{x2}\]
Simple Types

data MType
  = TVar TVar
  | TCon String
  | TArr Type Type
  deriving (Show, Eq, Ord)

typeInt, typeBool :: Type
  typeInt = TCon "Int"
  typeBool = TCon "Bool"

data PType = forall [Name] MType

- Polytypes (σ)
- Monotypes (τ)

const :: forall a b. a -> b -> a
TArr (TVar "a") (TArr (TVar "b") (TVar "a"))
Substitution

\[ \tau \sim \tau': s \]
\[ [s]\tau = [s]\tau' \]

The solver function simply iterates over the set of constraints, composing them and applying the resulting constraint solution over the intermediate solution eventually converting on the most general unifier which yields the final substitution which when applied over the inferred type signature, yields the principal type solution for the expression.

\begin{verbatim}
 type Subst = Map Name MType

 comp :: Subst -> Subst -> Subst
comp s1 s2 = M.union s1 (M.map (subst s1) s2)

 instance Substitutable MType where
  free (TVar n) = S.singleton n
  free (Con _) = S.empty
  free (TArr t1 t2) = free t1 \ S.union\ free t2

 instance Substitutable Env where
  free gamma = S.unions (map free (M.elems gamma))
  subst s gamma = M.map (subst s) gamma
\end{verbatim}
Unification ala Damas-Milner

\[
\begin{align*}
\text{unify} :: \text{Type} \to \text{Type} \to \text{Infer Subst} \\
\text{unify} (l \ `\text{TArr}` r) (l' \ `\text{TArr}` r') &= \text{do} \\
&\quad s1 \leftarrow \text{unify} l l' \\
&\quad s2 \leftarrow \text{unify} (\text{apply} s1 r) (\text{apply} s1 r') \\
&\quad \text{return} (s2 \ `\text{compose}` s1) \\
\text{unify} (\text{TVar} a) t &= \text{bind} a t \\
\text{unify} t (\text{TVar} a) &= \text{bind} a t \\
\text{unify} (\text{TCon} a) (\text{TCon} b) \mid a == b &= \text{return} \text{nullSubst} \\
\text{unify} t1 t2 &= \text{throwError} \$ \text{UnificationFail} t1 t2 \\
\text{bind} :: \text{TVar} \to \text{Type} \to \text{Infer Subst} \\
\text{bind} a t \mid t == \text{TVar} a &= \text{return} \text{nullSubst} \\
&\mid \text{occursCheck} a t &= \text{throwError} \$ \text{InfiniteType} a t \\
&\mid \text{otherwise} &= \text{return} \$ \text{Map.singleton} a t
\end{align*}
\]
Generalization & Instantiation

- **Generalization**: Converting a $\tau$ type into a $\sigma$ type by closing over all free type variables in a type scheme.

- **Instantiation**: Converting a $\sigma$ type into a $\tau$ type by creating fresh names for each type variable that does not appear in the current typing environment.

```
instantiate :: Scheme -> Infer Type
instantiate (Forall as t) = do
    as' <- mapM (const fresh) as
    let s = Map.fromList $ zip as as'
    return $ apply s t
```

```
generalize :: TypeEnv -> Type -> Scheme
generalize env t = Forall as t
    where as = Set.toList $ ftv t `Set.difference` ftv env
```

(a -> Int) => forall a. (a -> Int)
forall a. (a -> Int) => a0 -> Int
Inference Rules

\[
\text{infer :: Env } \rightarrow \text{ Expr } \rightarrow \text{ Infer (Subst, MType)}
\]

\[
\text{infer } \gamma (\text{Var } x) = \text{ do}
\]

\[
\begin{align*}
\sigma & \leftarrow \text{lookupEnv } x \ \gamma \\
\tau & \leftarrow \text{inst } \sigma \\
pure (\text{none, } \tau)
\end{align*}
\]

\[
\text{infer } \gamma (\text{App } e_1 e_2) = \text{ do}
\]

\[
\begin{align*}
(s_1, \tau_1) & \leftarrow \text{infer } \gamma e_1 \\
(s_2, \tau_2) & \leftarrow \text{infer } (\text{subst } s_1 \ \gamma) e_2 \\
\tau_3 & \leftarrow \text{fresh} \\
s_3 & \leftarrow \text{unify } (\text{subst } s_2 \ \tau_1) (\text{TArr } \tau_2 \ \tau_3) \\
pure (s_3 \ \text{`comp` } s_2 \ \text{`comp` } s_1, \ \text{subst } s_3 \ \tau_3)
\end{align*}
\]

\[
\text{infer } \gamma (\text{Lam } x_1 e) = \text{ do}
\]

\[
\begin{align*}
\tau_1 & \leftarrow \text{fresh} \\
ts & \leftarrow \text{replicateM } (\text{length } x_1) \ \text{fresh} \\
\text{let } \gamma' = \text{extends } (\text{zip } x_1 (\text{fmap } (\text{Forall } [])) t) \gamma \\
(te, e') & \leftarrow \text{infer } \gamma' e \\
pure (te, \ \text{subst } te (\text{foldl1'} \ \text{TArr} (\tau_1 : t)))
\end{align*}
\]
Example

\[
\text{compose } f \ g \ x = f (g \ x)
\]

The generated type from the infer function consists again simply of unique fresh variables.

\[
a \rightarrow b \rightarrow c \rightarrow e
\]

Induced by two cases of the T-App rule we get the following constraints:

\[
b \sim c \rightarrow d
\]
\[
a \sim d \rightarrow e
\]

Here \(d\) is the type of \((g \ x)\). The constraints are already in a canonical form, by applying Uni-VarLeft twice we get the following set of substitutions:

\[
b \sim c \rightarrow d
\]
\[
a \sim d \rightarrow e
\]

\[
\text{compose :: forall c d e. } (d \rightarrow e) \rightarrow (c \rightarrow d) \rightarrow c \rightarrow e
\]
The AST then has a complete copy of inferred type and the types of all lambda binders are annotated with type information.

Our program is nothing but toplevel functions that can be called precisely like C functions.

Our program is ready for translation into LLVM.
**Code Generation**

**llvm-general-pure** is a pure Haskell representation of the LLVM IR.

**llvm-general** is the FFI bindings to LLVM required for constructing the C representation of the LLVM IR and performing optimization and compilation.

[https://hackage.haskell.org/package/llvm-general-pure](https://hackage.haskell.org/package/llvm-general-pure)
define i32 @test1(i32 %x, i32 %y, i32 %z) {
    %a = and i32 %z, %x
    %b = and i32 %z, %y
    %c = xor i32 %a, %b
    ret i32 %c
}
LLVM Features

- Abstracts over register allocation
- Abstracts over instruction allocation
- Cross-platform targeting
- Fast compile times
- Built-in JIT compiler
- Reusable Optimizations
- Extensible

Optimizations

- Dead code elimination
- Loop invariant code motion
- Loop strength reduction
- Loop unrolling
- Instruction combination
- Automatic function inlining
- Automatic tail recursion detection
LLVM Types

i1 ; Boolean type
i8 ; char
i32 ; 32 bit integer
i64 ; 64 bit integer
float ; 32 bit
double ; 64 bit

[10 x float] ; Array of 10 floats
[10 x [20 x i32]] ; Array of 10 arrays of 20 integers.

{float, i64} ; structure
{float, {double, i3}} ; nested structure
<{float, [2 x i3]> ; packed structure
<4 x double>
<8 x float>

float* ; Pointer to a float
[25 x float]* ; Pointer to an array
## LLVM Operations

<table>
<thead>
<tr>
<th>Integer/Float</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>shl</td>
<td>add</td>
</tr>
<tr>
<td>lshr</td>
<td>fadd</td>
</tr>
<tr>
<td>ashr</td>
<td>sub</td>
</tr>
<tr>
<td>and</td>
<td>fsub</td>
</tr>
<tr>
<td>or</td>
<td>mul</td>
</tr>
<tr>
<td>xor</td>
<td>fmul</td>
</tr>
<tr>
<td></td>
<td>udiv</td>
</tr>
<tr>
<td></td>
<td>sdiv</td>
</tr>
<tr>
<td></td>
<td>fdiv</td>
</tr>
<tr>
<td></td>
<td>urem</td>
</tr>
<tr>
<td></td>
<td>srem</td>
</tr>
<tr>
<td></td>
<td>frem</td>
</tr>
</tbody>
</table>

- **Unconditional Branch**: Unconditional Branch
- **Conditional Branch**: Conditional Branch
- **Switch**: Switch
- **Return**: Return
- **Phi**: Phi

### Notes
- The table lists a variety of LLVM operations, including logical, arithmetic, and control flow instructions.
- The operations are categorized into integer and floating-point types.
- Each operation is paired with its corresponding LLVM instruction.
Symbols used in an LLVM module are either global or local. Global symbols begin with @ and local symbols begin with %. All symbols must be defined or forward declared.

Instructions in LLVM are either numbered sequentially (%0, %1, ...) or given explicit variable names (%a, %foo, ..). For example the arguments to the following function are named values, while the result of the add instructions unnamed.

```assembly
define i32 @add(i32 %a, i32 %b) {
  %1 = add i32 %a, %b
  ret i32 %1
}
```
Memory

- **load**: Load a typed value from a given reference
- **store**: Store a typed value in a given reference
- **alloca**: Allocate a pointer to memory on the virtual stack

```assembly
%ptr = alloca i32
store i32 3, i32* %ptr
%val = load i32* %ptr
```

The `getelementptr` instruction is used to get the address of a subelement of an aggregate data structure. It performs address calculation only and does not access memory.

```assembly
getelementptr i32* %0, i64 %index
```
Static Single Assignment

Static single assignment form is a property of an intermediate representation which requires that each variable is assigned exactly once, and every variable is defined before it is used.

\[
\begin{align*}
x & := 10; \\
a & := 11; \\
x & := x + a;
\end{align*}
\]

\[
\begin{align*}
x_0 & := 10; \\
a_0 & := 11; \\
x_1 & := x_0 + a_0;
\end{align*}
\]
if \( x < 5 \) {
    x := x + 1;
} else {
    x := x + 2;
}

if \( x < 5 \) {
    x_1 := x_0 + 1;
} else {
    x_2 := x_0 + 2;
}

x_3 := \phi(x_1, x_2)
Branching Logic

```assembly
define i1 @foo() {
    entry:
        br label %next
    next:
        br label %return
    return:
        ret i1 0
}
```

```assembly
define i32 @foo() {
    start:
        br il true, label %left, label %right
    left:
        ret i32 10
    right:
        ret i32 20
}
```
define i32 @foo(i32 %a) {
  entry:
    switch i32 %a, label %default [
      i32 0, label %f
      i32 1, label %g
      i32 2, label %h
    ]
    f:
      ret i32 1
    g:
      ret i32 2
    h:
      ret i32 3
    default:
      ret i32 0
}
define i32 @foo() {
    start:
        br i1 true, label %left, label %right
    left:
        %plusOne = add i32 0, 1
        br label %merge
    right:
        br label %merge
    merge:
        %join = phi i32 [ %plusOne, %left ], [ -1, %right ]
        ret i32 %join
}
Loops

define i32 @count(i32 %n) {
  entry:
    br label %loop

  loop:
    %i = phi i32 [ 1, %entry ], [ %nextvar, %loop ]
    %nextvar = add i32 %i, 1
    %cmptmp = icmp ult i32 %i, %n
    %booltmp = zext i1 %cmptmp to i32
    %loopcond = icmp ne i32 %booltmp, 0
    br i1 %loopcond, label %loop, label %afterloop

  afterloop:
    ret i32 %i
}
newtype LLVM a = LLVM { unLLVM :: State AST.Module a }
  deriving (Functor, Applicative, Monad, MonadState AST.Module )

newtype Codegen a = Codegen { runCodegen :: State CodegenState a }
  deriving (Functor, Applicative, Monad, MonadState CodegenState )

data CodegenState =
  CodegenState { currentBlock :: Name, -- Name of the active block to append to
                 blocks :: Map.Map Name BlockState, -- Blocks for function
                 symtab :: SymbolTable, -- Function scope symbol table
                 blockCount :: Int, -- Count of basic blocks
                 count :: Word, -- Count of unnamed instructions
                 names :: Names -- Name Supply } deriving Show
Basic Blocks

```haskell
data BlockState = BlockState {
    idx :: Int -- Block index,
    stack :: [Named Instruction] -- Stack of instructions,
    term :: Maybe (Named Terminator) -- Block terminator
} deriving Show
```

```assembly
define i64 @n1(i64 %x, i64 %y, i64 %z){
    entry:
        %0 = mul i64 %x, %y
        %1 = add i64 %0, %z
    ret i64 %1
}
```
**Instr Stack**

```haskell
instr :: Instruction -> Codegen Operand

instr ins = do
  n <- fresh
  blk <- current
  let i = stack blk
  let ref = (UnName n)
  modifyBlock $ blk { stack = i ++ [ref := ins] }
  return $ local ref

terminator :: Named Terminator -> Codegen (Named Terminator)

terminator trm = do
  blk <- current
  modifyBlock $ blk { term = Just trm }
  return trm
```
Monadic Interface to LLVM

We can compose monadic functions for each of the machine instructions that give rise to more complex logic that can be composed with each other.

We can write higher-order functions that compose larger pieces of assembly code to build larger abstractions needed for compiler backends.

```haskell
initModule :: AST.Module
initModule = emptyModule "my cool jit"

logic :: LLVM ()
logic = do
  define double "main" [] $ do
    let a = cons $ C.Float (F.Double 10)
    let b = cons $ C.Float (F.Double 20)
    res <- fadd a b
    ret res

main :: IO (AST.Module)
main = do
  let ast = runLLVM initModule logic
  runJIT ast
  return ast
```
Walk typechecked AST, convert inferred types to LLVM types and generate LLVM for the function body logic.

codegenDecl :: Decl -> LLVM ()
codegenDecl (FunDecl fn args body) = do
  let args' = fmap codegenArg args
  define int (toS fn) args' $ do
    forM args $ \a -> do
      let aname = toS a
      assign aname (local (typeOf a) (AST.Name aname))

codegenExpr body
$$\text{codegenExpr} :: \text{Expr} \rightarrow \text{Codegen AST.Operand}$$

$$\text{codegenExpr} = \text{\textbackslash case}$$

$$\text{Var } x \rightarrow \text{ do}$$
  $$\text{getvar } (\text{toS } x)$$

$$\text{App } (\text{Var } f) \text{ arg} \rightarrow \text{ do}$$
  $$\text{arg'} \leftarrow \text{codegenExpr } \text{arg}$$
  $$\text{call } (\text{externf } (\text{fn int [int]}) (\text{AST.Name } (\text{toS } f))) [\text{arg'}]$$
If cond tr fl -> do
  ifthen <- addBlock "if.then"
  ifelse <- addBlock "if.else"
  ifexit <- addBlock "if.exit"

-- %entry
------------------
cond <- codegenExpr cond
test <- icmp IP.EQ true cond
cbr test ifthen ifelse

-- if.then
------------------
setBlock ifthen
trval <- codegenExpr tr
br ifexit
ifthen <- getBlock

-- if.else
------------------
setBlock ifelse
flval <- codegenExpr fl
br ifexit
ifelse <- getBlock

-- if.exit
------------------
setBlock ifexit
phi int [(trval, ifthen), (flval, ifelse)]
let fib n =
  if (n == 0) then 0
  else if (n == 1) then 1
  else ((fib (n-1)) + (fib (n-2)));
let fib n =
    if (n == 0)
        then 0
    else if (n==1)
        then 1
    else ((fib (n-1)) + (fib (n-2)));

n3:
  # BB#0:
    xorl %eax, %eax
    testq %rdi, %rdi
    je .LBB3_2
  # BB#1:
    movl $1, %eax
    cmpq $1, %rdi
    jne .LBB3_3
  # %if.exit
  ret

.LBB3_2:
  .LBB3_3:
  # %if.exit
foreign import ccall "dynamic" haskFun :: FunPtr (IO Int) -> (IO Int)

run :: FunPtr a -> IO Int
run fn = haskFun (castFunPtr fn :: FunPtr (IO Int))

import qualified LLVM.General.ExecutionEngine as EE

EE.withModuleInEngine executionEngine m $ \\
  ee -> do
  mainfn <- EE.getFunction ee (AST.Name "main")
  case mainfn of
    Just fn -> do
      res <- run fn
      putStrLn $ "Evaluated to: " <> show res
    Nothing -> putStrLn "Invalid data returned"
Projects

Optimizations

- More efficient representation of closures.
- Efficient handling of mutually recursive functions.
- Avoiding combinatorial blowup of creating toplevel lambda-lifted functions specialized to every type of call site.
- This approach forces to compiler writers to remove all polymorphism. (Make joke about Go here).

Features

- Custom datatypes mapping to LLVM structs.
- Target a richer Core language (System-F, MLF, Dependently Typed)
- Custom Runtime
The Implementation of Functional Programming Languages
Simon Peyton Jones

Modern Compiler Implementation in ML
Andrew W. Appel

Types and Programming Languages
Benjamin Pierce
Merci!
twitter.com/smdiehl
github.com/sdiehl
stephen@adjoint.io