Compiler Construction

Sibylle Schupp[1]

[1]Institute for Software Systems/Institut für Softwaresysteme
Hamburg University of Technology (TUHH)

SoSe16

Lecture 8

STS
Software
Technology
Systems

Outline

1 Intermediate representations II
- LLVM: The Haskell API
- Static single assignment (SSA)

LLVM: A complete example

```
int ifac(int a) {
        int r;
        int n;

        r = 1;
        n = a;
        while (n > 0) {
                r = r * n;
                n = n - 1;
        }
        return r;
}
```

A complete example (cont')

```
int ifac(int a) {
        int r;
        int n;

        r = 1;
        n = a;
        while (n > 0) {
        ...
}}
```

```
; Function Attrs: nounwind ssp uwtable
define i32 @_Z4ifaci(i32 %a) #0 {
  %1 = alloca i32, align 4
  %r = alloca i32, align 4
  %n = alloca i32, align 4
  store i32 %a, i32* %1, align 4
  store i32 1, i32* %r, align 4
  %2 = load i32* %1, align 4
  store i32 %2, i32* %n, align 4
  br label %3
  ...

}
```

STS Software Technology Systems

A complete example (cont')

```
define i32 @_Z4ifaci(i32 %a) #0 {
  ...
; <label>:3                                             ; preds = %6, %0
  %4 = load i32* %n, align 4
  %5 = icmp sgt i32 %4, 0
  br i1 %5, label %6, label %12

; <label>:6                                              ; preds = %3
  %7 = load i32* %r, align 4
  %8 = load i32* %n, align 4
  %9 = mul nsw i32 %7, %8
  store i32 %9, i32* %r, align 4
  %10 = load i32* %n, align 4
  %11 = sub nsw i32 %10, 1
  store i32 %11, i32* %n, align 4
  br label %3

; <label>:12                                             ; preds = %3
  %13 = load i32* %r, align 4
  ret i32 %13
}
```

Outline

STS Software Technology Systems

The Haskell API

The Haskell API for LLVM is divided into two packages:

- https://hackage.haskell.org/package/llvm-general-pure
- https://hackage.haskell.org/package/llvm-general

In the following, we'll use Stephen Diehl's Kaleidoscope tutorial.

The type BlockState for local code generation

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

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

- A basic block is a stack of instructions.
- In LLVM, the last instruction must be a terminating instruction.
- Named, Instruction, Terminator are types from the LLVM API (see below).

STS Software Technology Systems

The type CodegenState for global code generation

```
type SymbolTable = [(String, Operand)]

data CodegenState
  = CodegenState {
    currentBlock :: Name          -- Active block to append to
  , blocks       :: Map.Map Name BlockState  -- Function
  , symtab       :: SymbolTable -- Function scope ST
  , blockCount   :: Int         -- Count of basic blocks
  , count        :: Word        -- Count of unnamed instructions
  , names        :: Names       -- Name Supply
  } deriving Show
```

- In LLVM, a function is a sequence of basic blocks. Each block can be referred to by name. The sequence is are organized as a map.
- The symbol table associates parameter names and operands.

Instances

```
emptyBlock :: Int -> BlockState
emptyBlock i = BlockState i [] Nothing

entryBlockName :: String
entryBlockName = "entry"

emptyCodegen :: CodegenState
emptyCodegen = CodegenState (Name entryBlockName)
                             Map.empty [] 1 0 Map.empty
```

- The `emptyBlock` contains no instructions. It is indexed: each block starts as empty block and gets populated with instructions afterwards.
- The `emptyCodegen` sets the label of the first block to `''entry''`. All maps and tables are empty.

Stateful, parameterized code generation

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

- The new type Codegen wraps the code generator in the State monad.
- It is parameterized by the subject of code generation.
  - Codegen Operand, Codegen Name, Codegen (Named Terminator), Codegen BlockState
- The type Codegen is an instance of various monadic classes.
  - The deriving clause requires a special compilation flag: GeneralizedNewtypeDeriving.

Stateful programming

```
addBlock :: String -> Codegen Name
addBlock bname = do
  bls <- gets blocks
  ix  <- gets blockCount
  nms <- gets names
  let new             = emptyBlock ix
      (qname, supply) = uniqueName bname nms
  modify $ \s -> s { blocks = Map.insert (Name qname) new bls
                   , blockCount = ix + 1
                   , names = supply
                   }
  return (Name qname)

modifyBlock :: BlockState -> Codegen ()
modifyBlock new = do
  active <- gets currentBlock
  modify $ \s -> s { blocks = Map.insert active new (blocks s) }
```

- A new block is added in-place (with a unique name).
- The active block is modified in-place.

STS Software Technology Systems TUHH

The final state

```
execCodegen :: Codegen a -> CodegenState
execCodegen m = execState (runCodegen m) emptyCodegen

> :type execState
execState :: State s a -> s -> s
> :type runCodegen
runCodegen :: Codegen a -> State CodegenState a
```

- The final state of the computation can be obtained via `execState`, which takes a state constructor and an initial state and returns the final state.
- The function `execCodegen` supplies the proper arguments. It is called from those functions that process the top-level nodes of the input AST.

The LLVM monad

```
newtype LLVM a = LLVM { unLLVM :: State AST.Module a }
  deriving (Functor, Applicative, Monad, MonadState AST.Module )
```

- The Kaleidoscope code generator uses a second monad, `LLVM`. The `LLVM` monad holds the AST of the LLVM representation.
- The type `AST` is defined in the API (llvm-general-pure). The lop-level construct in LLVM is a module: `AST.Module`.
- Code generation is invoked with an empty module, which is then updated with the top-level definitions of the source language.

Changing and returning the state

```
newtype LLVM a = LLVM { unLLVM :: State AST.Module a }
  deriving (Functor, Applicative, Monad, MonadState AST.Module )

addDefn :: Definition -> LLVM ()
addDefn d = do
  defs <- gets moduleDefinitions
  modify $ \s -> s { moduleDefinitions = defs ++ [d] }

runLLVM :: AST.Module -> LLVM a -> AST.Module
runLLVM = flip (execState . unLLVM)
```

- New definitions are added in-place; `moduleDefinitions` is defined in `AST.Module`.
- The final state is obtained via `runLLVM`. The code generator invokes `runLLVM` with the initial, empty module and the completed LLVM tree, and returns the final module.

AST data types for LLVM constructs

https://hackage.haskell.org/package/llvm-general-pure-3.5.0.0/docs/LLVM-General-AST.html

The module `LLVM.General.AST` encapsulates the major LLVM constructs in types. Relevant types and modules include:

- `data Module`
- `data Definition`
- `data Parameter`
- `data BasicBlock`
- module `LLVM.General.AST.Instruction`
- module `LLVM.General.AST.Name`
- module `LLVM.General.AST.Operand`

Example (LLVM API)

```
define ::  Type -> String -> [(Type, Name)] -> [BasicBlock]
          -> LLVM ()
define retty label argtys body = addDefn $
  GlobalDefinition $ functionDefaults {
    name        = Name label
  , parameters  = ([Parameter ty nm [] | (ty, nm) <- argtys],
                   False)
  , returnType  = retty
  , basicBlocks = body
  }
```

- In the source language, functions are defined at the top level.
- The function `define` adds those definitions to the LLVM AST using two functions from the LLVM API.
- `GlobalDefinition` is a constructor of `Definition`. It accepts global variables and functions.
- `functionDefaults` is a helper function that provides default bindings for the `Function` constructor. Here, it overwrites four fields: `name`, `parameter`, `returnType`, `basicBlocks`.

The API for LLVM instructions

https://hackage.haskell.org/package/llvm-general-pure-3.5.0.0/docs/LLVM-General-AST-Instruction.html

The data type `Instructions` is a large sum type. It contains a constructor for each LLVM non-terminator instructions. Examples include

- `Add`, `FAdd`, `Sub`, ...
- `Alloca`, `Store`, `Load`, ...
- `Call`, `Phi` ...
- ...

Code generation for instructions

```
fadd :: Operand -> Operand -> Codegen Operand
fadd a b = instr $ FAdd NoFastMathFlags a b []

call :: Operand -> [Operand] -> Codegen Operand
call fn args = instr $ Call Nothing CC.C [] (Right fn)
                         (toArgs args) [] []

alloca :: Type -> Codegen Operand
alloca ty = instr $ Alloca ty Nothing 0 []

store :: Operand -> Operand -> Codegen Operand
store ptr val = instr $ Store False ptr val Nothing 0 []

load :: Operand -> Codegen Operand
load ptr = instr $ Load False ptr Nothing 0 []
```

- The Kaleidoscope code generator encapsulates each API constructor. An important helper function is `instr`, which pushes instructions on the current basic block stack.

STS Software Technology Systems

Top-level code generation

```
import qualified Syntax as S

codegenTop :: S.Expr -> LLVM ()
codegenTop (S.Function name args body) = do
  define double name fnargs bls
  where
    fnargs = toSig args
    bls = createBlocks $ execCodegen $ do
      entry <- addBlock entryBlockName
      setBlock entry
      forM args $ \a -> do
        var <- alloca double
        store var (local (AST.Name a))
        assign a var
      cgen body >>= ret
```

- Several functions codegenTop exist. They transform the top-level constructs from the source AST to the LLVM (tree) representation.
- For function definitions, an initial block is created and turned into the current block. All parameters are allocated and initialized. Code for the body is generated via cgen, the last statement is bound to ret.

Outline

STS Software Technology Systems

An illegal example

```
 br i1 %1, label %2, label %3

; <label>:2                                        ; preds = %0
  %bla = sub nsw i32 %n, 1
  br label %7

; <label>:3                                        ; preds = %0
  %bla = sub nsw i32 %n, 1
  %4 = sub nsw i32 %n, 1
```

Error message

```
:63:3: error: multiple definition of local value named bla
  %bla = sub nsw i32 %n, 1
  ^
```

STS Software Technology Systems

Static single assignment (SSA)

A program is in SSA form if every variable is

- assigned in exactly once place
- defined before it is used

Why SSA?

- Simplifies dataflow analysis and optimizations
- Represents *def-use chains* of variables efficiently (linear instead of quadratic)
- Intermediate representation by itself

STS Software Technology Systems

SSA forms for IRs

http://www.llvm.org/docs/tutorial/LangImpl7.html#why-is-this-a-hard-problem

Higher programming languages typically do not require SSA. The following program is not in SSA:

```
int G, H;
int test(_Bool Condition) {
  int X;
  if (Condition)
    X = G;
  else
    X = H;
  return X;
}
```

- LLVM and most intermediate languages of compilers require a SSA form.
- Programs that are not in SSA need to be converted (see below for how to do that in LLVM)

SSA conversion: value numbering

Within a basic block, *value numbering* suffices to obtain an SSA form:

- Rename each new definition of a variable
- Update each use to the most recent definition
- Example

```
x = a + b
y = x
x = y + 1
y = x
x = 7 - x
```

```
x1 = a + b
y1 = x1
x2 = y1 + 1
y2 = x2
x3 = 7 - x2
```

STS Software Technology Systems

Problem: join nodes

Cytron, Fig. 6

```
I ← 1
J ← 1
K ← 1
L ← 1
repeat




    if (P)
       then do
          J ← I
          if (Q)
             then L ← 2
             else L ← 3

          K ← K + 1
       end
       else K ← K + 2



    print(I,J,K,L)
```

```
I₁ ← 1
J₁ ← 1
K₁ ← 1
L₁ ← 1
repeat
    I₂ ← φ(I₃,I₁)
    J₂ ← φ(J₄,J₁)
    K₂ ← φ(K₅,K₁)
    L₂ ← φ(L₉,L₁)
    if (P)
       then do
          J₃ ← I₂
          if (Q)
             then L₃ ← 2
             else L₄ ← 3
          L₅ ← φ(L₃,L₄)
          K₃ ← K₂ + 1
       end
       else K₄ ← K₂ + 2
    J₄ ← φ(J₃,J₂)
    K₅ ← φ(K₃,K₄)
    L₆ ← φ(L₂,L₅)
    print(I₂,J₄,K₅,L₆)
```

SSA conversion: ϕ function

The ϕ function is a theoretical construct.

- Conceptually, it is a function that is inserted at the beginning of a join node.
- The number of its arguments depends on the number of predecessor blocks. The arguments depends on all previous renamings of a variable, e.g., $x_2 = \phi(x_3, x_1)$ for the join of an if-else-statement.
- If control flow reaches the join node through the true edge, ϕ represents x_3, otherwise x_1. Similarly for ϕ functions with 3 or more arguments.
- How does ϕ know the flow of control? It does not.
  - If ϕ statement needs to be executed: conversion back
  - But, often use-def relations matter (and ϕ is not executed at all).

STS Software Technology Systems TUHH

Example (the phi instruction in LLVM)

http://www.llvm.org/docs/tutorial/LangImpl7.html

LLVM provides a ϕ command:

```
@G = weak global i32 0   ; type of @G is i32*
@H = weak global i32 0   ; type of @H is i32*

define i32 @test(i1 %Condition)
{
entry:
  br i1 %Condition, label %cond_true, label %cond_false

cond_true:
  %X.0 = load i32* @G
  br label %cond_next

cond_false:
  %X.1 = load i32* @H
  br label %cond_next

cond_next:
  %X.2 = phi i32 [ %X.1, %cond_false ],[ %X.0, %cond_true ]
  ret i32 %X.2
}
```

SSA computation

The core of every SSA computation contains two steps:

- Inserting ϕ functions
- Renaming variables

Yet, inserting ϕ functions in every join block is too expensive. Different SSA algorithms exist to minimize their number:

- Compute dominator tree and dominance frontier
- Include *liveness* information
- Split edges to obtain *unique successor* property

An SSA computation is not trivial. In LLVM, one gets it for free, though.

SSA for free

http://www.llvm.org/docs/tutorial/OCamlLangImpl7.html

How to ensure SSA form? Key idea

- Exploit that memory locations are not in SSA form: `http://llvm.org/docs/LangRef.html#memory-access-and-addressing-operations`
- Key idea: avoid the need for SSA.

From the Kaleidoscope tutorial:

- Each (mutable) variable becomes a stack allocation.
- Each read of the variable becomes a load from the stack.
- Each update of the variable becomes a store to the stack.
- Taking the address of a variable just uses the stack address directly.

Example (SSA generation, step 1)

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

```
@G = global i32 0    ; type of @G is i32*
@H = global i32 0    ; type of @H is i32*

define i32 @test(i1 %Condition)
{
entry:
  %X = alloca i32             ; mutable variable X: stack-allocated
  br i1 %Condition, label %cond_true, label %cond_false

cond_true:
  %X.0 = load i32* @G
  store i32 %X.0, i32* %X    ; Update X
  br label %cond_next

cond_false:
  %X.1 = load i32* @H
  store i32 %X.1, i32* %X    ; Update X
  br label %cond_next
cond_next:
  %X.2 = load i32* %X         ; Read X
  ret i32 %X.2
}
```

Automated optimization

- The extra `alloca`, `store` are inefficient, obviously.
- But: the LLVM optimizer can turn `alloca` expressions automatically in SSA registers and inserts `phi` instructions in an optimized way.
- Use the flag `mem2reg`:

```
llvm-as < $@.ll | opt -mem2reg | llvm-dis
```

- Applied to the example, it eliminates the `alloca` instruction for `X` and inserts one `phi` instruction.

Example (-mem2reg)

```
@G = global i32 0
@H = global i32 0

define i32 @test(i1 %Condition) {
entry:
  br i1 %Condition, label %cond_true, label %cond_false

cond_true:                                  ; preds = %entry
  %X.0 = load i32* @G
  br label %cond_next

cond_false:                                 ; preds = %entry
  %X.1 = load i32* @H
  br label %cond_next

  cond_next:                   ; preds = %cond_false, %cond_true
  %X.01 = phi i32 [ %X.0, %cond_true ], [ %X.1, %cond_false ]
  ret i32 %X.01
}
```

C++ API

http://llvm.org/docs/doxygen/html/classllvm_1_1IRBuilder.html

- The class `IRBuilder` provides the necessary methods: `CreateAlloca`, `CreateStore`, `CreateLoad`
- Mutable variables are of type `AllocaInst`

Example (C++ API)

http://www.llvm.org/docs/tutorial/LangImpl7.html

```
static AllocaInst *CreateEntryBlockAlloca(
    Function *TheFunction,
    const std::string &VarName) {
  IRBuilder<> TmpB(&TheFunction->getEntryBlock(),
                   TheFunction->getEntryBlock().begin());
  return TmpB.CreateAlloca(Type::getDoubleTy(
           getGlobalContext()), 0, VarName.c_str());
}

Value *VariableExprAST::codegen() {
  // Look this variable up in the function.
  Value *V = NamedValues[Name];
  if (!V)
    return ErrorV("Unknown variable name");

  // Load the value.
  return Builder.CreateLoad(V, Name.c_str());
}
```

- Variables are allocated on the stack, and are then loaded from it.

Summary

- Haskell API, monads
- SSA: ϕ function
- LLVM: type-safe, SSA

References

- http://brandon.si/code/the-state-monad-a-tutorial-for-the-confused/
- Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. 13, 4 (October 1991), 451-490. DOI=10.1145/115372.115320 http://doi.acm.org/10.1145/115372.115320

STS Software Technology Systems

TUHH