Compilation 2024

LLVM-- code generation

aslan@cs.au.dk

Programmatic representation of LLVM programs

Ll.ml module, selected parts

```
module S = Symbol

type uid = S.symbol  (* Local identifiers  *)
type gid = S.symbol  (* Global identifiers *)
type tid = S.symbol  (* Named types        *)
type lbl = S.symbol  (* Labels             *)

(* LLVM IR types *)
type ty
  = Void                  (* void              *)
  | I1 | I8 | I32 | I64 (* integer types    *)
  | Ptr of ty             (* t*                *)
   (* … *)
and fty = ty list * ty
```

```
type operand
  = Null                (* null pointer         *)
  | IConst64 of int64 (* integer constant    *)
  | IConst32 of int32 (* integer constant    *)
  | IConst8 of char   (* integer constant    *)
  | IConst1 of bool   (* boolean constant    *)
  | Gid of gid        (* A global identifier *)
  | Id of uid         (* A local identifier  *)

(* Binary operations *)
type bop = Add | Sub | Mul | SDiv
  | SRem | Shl | Lshr | Ashr | And | Or | Xor

(* Comparison operators *)
type cnd = Eq | Ne | Slt | Sle | Sgt | Sge
```

Instructions, terminators, CFG

```
type insn (* Instructions *)
  = Binop of bop * ty * operand * operand      (* bop ty %o1, o2                  *)
  | Alloca of ty                               (* alloca ty                       *)
  | Load of ty * operand                       (* load ty %u                      *)
  | Store of ty * operand * operand            (* store ty %t, ty* %u             *)
  | Icmp of cnd * ty * operand * operand       (* icmp %s ty %s, %s               *)
  | Call of ty * operand * (ty * operand) list (* fn (%1, %2, ...)                *)
  (* … *)

type terminator (* Block terminators *)
  = Ret of ty * operand option (* ret i64 %s                          *)
  | Br of lbl                  (* br label %lbl                       *)
  | Cbr of operand * lbl * lbl (* br i1 %s, label %l1, label %l2 *)
  | Unreachable                (* unreachable -- use sparingly  .*)

(* Basic blocks *)
type block = { insns: (uid option * insn) list; terminator: terminator }

(* Control Flow Graph: a pair of an entry block and a set of labeled blocks *)
type cfg = block * (lbl * block) list
```

Programs

```
(* Function declarations *)
type fdecl = { fty: fty; param : uid list; cfg: cfg }

type gdecl = (* … *) (* Global declaration *)

(* LLVM-- programs *)
type prog
    = { tdecls: (tid * ty) list        (* named types         *)
      ; extgdecls: (gid * ty) list     (* external globals    *)
      ; gdecls: (gid * gdecl) list     (* global data         *)
      ; extfuns: (gid * fty) list      (* external functions *)
      ; fdecls: (gid * fdecl) list     (* code                *)
      }
```

Example

```
let llprog_01 =
  let cfg = { insns = []
            ; terminator = Ret (I64, Some (IConst64 0L))
            }
          , [] in
   { tdecls    = []
   ; extgdecls = []
   ; gdecls    = []
   ; extfuns   = [ (symbol "print_integer", ([I64], Void))
                 ; (symbol "read_integer", ([], I64))]
   ; fdecls = [ (Sym.symbol "dolphin_main",
                  { fty = ([],  I64); param = []; cfg} )]
   }
let () =
   let s = Ll.string_of_prog llprog_01 in
   output_string stdout s
```

```
declare void @print_integer(i64)
declare i64 @read_integer()

define i64 @dolphin_main () {
 ret i64 0
}
```

Example

```
let llprog_02 =
   let insns = [
      Some (symbol "x"), Call (I64, Gid (symbol "read_integer"), [])
   ;  Some (symbol "y"), Binop (Add, I64, Id (symbol "x"), IConst64 1L)
   ;  None             , Call (Void, Gid (symbol "print_integer"), [I64, Id (symbol "y")])
   ] in
  let cfg = { insns
            ; terminator = Ret (I64, Some (IConst64 0L))}, [] in

   (* the rest as in the previous slide *)
```

```
declare void @print_integer(i64)
declare i64 @read_integer()

define i64 @dolphin_main () {
 %x = call i64 @read_integer ()
 %y = add i64 %x, 1
 call void @print_integer (i64 %y)
 ret i64 0
}
```

Example, with some refactoring

```
let llprog_03 =
   let sym_x = symbol "x" in
   let sym_y = symbol "y" in
   let sym_read_integer = symbol "read_integer" in
   let sym_print_integer = symbol "print_integer" in
   let insns = [
      Some sym_x, Call (I64, (Gid sym_read_integer), [])
   ;  Some sym_y, Binop (Add, I64, Id sym_x, IConst64 1L)
   ;  None, Call (Void, Gid sym_print_integer, [I64, Id sym_y])
   ] in
  let cfg = { insns
            ; terminator = Ret (I64, Some (IConst64 0L))}, [] in
  (* the rest as before *)
```

```
declare void @print_integer(i64)
declare i64 @read_integer()

define i64 @dolphin_main () {
 %x = call i64 @read_integer ()
 %y = add i64 %x, 1
 call void @print_integer (i64 %y)
 ret i64 0
}
```

Observations about LL module

- Enforces well-formedness by construction, e.g.:
  - each basic block must return
  - CFG must contain at least one basic block
  - all but the first block must be explicitly labeled
- Verbose, but only necessarily so
  - easy to refactor around
- Pros:
  - Good internal representation for consumption by an LLVM backend (clang or others)
- Cons
  - Not good for representing *incomplete* LLVM programs (or complex fragments)

Considerations for Translation to LLVM

What should the function translating Typed AST statements to LLVM return?

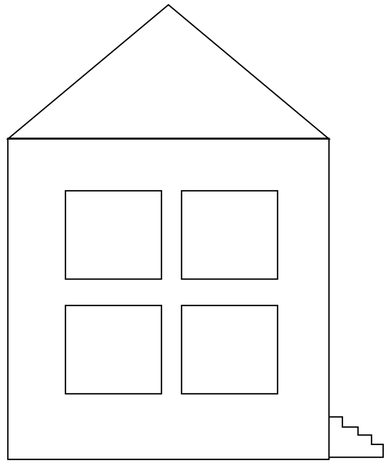
- Just one LLVM instruction?
  - No, because a single source level expression/statement corresponds to many LLVM instructions
- A list of LLVM instructions?
  - No, because some statements, e.g., IfThenElse need to generate basic blocks
- A basic block/list of basic blocks?
  - No, sometimes we don't know what the block terminator is

(Recall from last lecture) Dolphin Phase 1 Typed AST

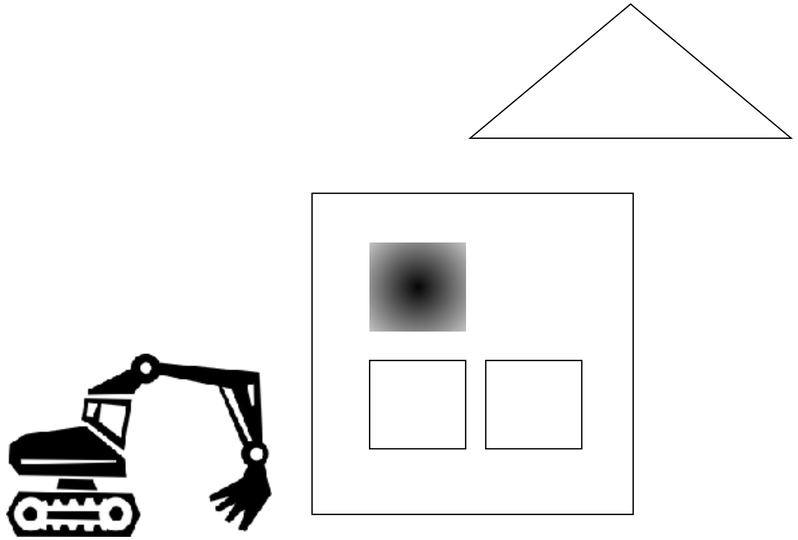
```
(* -- Use this in your solution without modifications *)
module Sym = Symbol

type ident = Ident of {sym : Sym.symbol}

type typ = Void | Int | Bool | ErrorType

type binop = Plus | Minus | Mul | Div | Rem | Lt | Le | Gt | Ge | Lor | Land | Eq
| NEq

type unop = Neg | Lnot

type expr =
| Integer of {int : int64}
| Boolean of {bool : bool}
| BinOp of {left : expr; op : binop; right : expr; tp : typ}
| UnOp of {op : unop; operand : expr; tp : typ}
| Lval of lval
| Assignment of {lvl : lval; rhs : expr; tp : typ}
| Call of {fname : ident; args : expr list; tp : typ}
and lval =
| Var of {ident : ident; tp : typ}

type statement =
| VarDeclStm of {name : ident; tp : typ; body : expr}
| ExprStm of {expr : expr option}
| IfThenElseStm of {cond : expr; thbr : statement; elbro : statement option}
| CompoundStm of {stms : statement list}
| ReturnStm of {ret : expr}

type param = Param of {paramname : ident; typ : typ}

type funtype = FunTyp of {ret : typ; params : param list}

type program = statement list
```

We need a way of representing unfinished LLVM programs

Turns out, only need unfinished CFG representation



Ll.ml



CfgBuilder.ml

+ non-destructive API for modifications

img attribution: https://www.wannapik.com/vectors/35104

Using the CFG Builder directly

```
let llprog_04 = in
   let sym_x = symbol "x" in
   let sym_y = symbol "y" in
   let sym_read_integer = symbol "read_integer" in
   let sym_print_integer = symbol "print_integer" in

   let b1 = empty_cfg_builder in
   let b2 = add_insn (Some sym_x, Call (I64, Gid sym_read_integer, [])) b1 in
   let b3 = add_insn (Some sym_y, Binop (Add, I64, Id sym_x, IConst64 1L)) b2 in
   let b4 = add_insn (None      , Call (Void, Gid sym_print_integer, [I64, Id sym_y])) b3 in
   let b5 = term_block (Ret (I64, Some (IConst64 0L))) b4 in
   let cfg = get_cfg b5 in
   { tdecls    = []
   ; extgdecls = []
   ; gdecls    = []
   ; extfuns   = [ (symbol "print_integer",  ([I64], Void))
                 ; (symbol "read_integer", ([], I64))]
   ; fdecls = [ (Sym.symbol "dolphin_main",
                 { fty = ([],  I64); param = []; cfg}
                )]
   }
```

```
declare void @print_integer(i64)
declare i64 @read_integer()

define i64 @dolphin_main () {
 %x = call i64 @read_integer ()
 %y = add i64 %x, 1
 call void @print_integer (i64 %y)
 ret i64 0
}
```

Considerations for Translation to LLVM (revisited)

What should the function translating Typed AST statements to LLVM look like?

- A: something that takes other auxiliary arguments, e.g., environment for translation, and also a cfg_builder and returns a cfg_builder
- Assuming cfg_builder is the last argument
  - ... -> cfg_builder -> cfg_builder

Dolphin Phase 1 Typed AST

```
(* -- Use this in your solution without modifications *)
module Sym = Symbol

type ident = Ident of {sym : Sym.symbol}

type typ = Void | Int | Bool | ErrorType

type binop = Plus | Minus | Mul | Div | Rem | Lt | Le | Gt | Ge | Lor | Land | Eq
| NEq

type unop = Neg | Lnot

type expr =
| Integer of {int : int64}
| Boolean of {bool : bool}
| BinOp of {left : expr; op : binop; right : expr; tp : typ}
| UnOp of {op : unop; operand : expr; tp : typ}
| Lval of lval
| Assignment of {lvl : lval; rhs : expr; tp : typ}
| Call of {fname : ident; args : expr list; tp : typ}
and lval =
| Var of {ident : ident; tp : typ}

type statement =
| VarDeclStm of {name : ident; tp : typ; body : expr}
| ExprStm of {expr : expr option}
| IfThenElseStm of {cond : expr; thbr : statement; elbro : statement option}
| CompoundStm of {stms : statement list}
| ReturnStm of {ret : expr}

type param = Param of {paramname : ident; typ : typ}

type funtype = FunTyp of {ret : typ; params : param list}

type program = statement list
```

Using the CFG Builder with buildlets

```
let llprog_05 =
    (* … *)
    let i1 = add_insn (Some sym_x, Call (I64, Gid sym_read_integer, [])) in
    let i2 = add_insn (Some sym_y, Binop (Add, I64, Id sym_x, IConst64 1L)) in
    let i3 = add_insn (None      , Call (Void, Gid sym_print_integer, [I64, Id sym_y])) in
    let tr = term_block (Ret (I64, Some (IConst64 0L))) in
    let cfg = get_cfg (tr (i3 (i2 (i1 empty_cfg_builder)))) in

let llprog_06 =
    (* … *)
    let i1 = add_insn (Some sym_x, Call (I64, Gid sym_read_integer, [])) in
    let i2 = add_insn (Some sym_y, Binop (Add, I64, Id sym_x, IConst64 1L)) in
    let i3 = add_insn (None      , Call (Void, Gid sym_print_integer, [I64, Id sym_y])) in
    let composed_block = seq_buildlets [i1; i2; i3; tr] in
    let cfg = get_cfg (composed_block empty_cfg_builder) in
```

CFGBuilder and buildlets

CfgBuilder.ml excerpt

```
type cfg_builder

type buildlet = cfg_builder → cfg_builder

val id_buildlet: buildlet
val seq_buildlets: buildlet list → buildlet

val add_alloca:  uid * ty → buildlet
val add_insn: uid option * insn → buildlet
val term_block: terminator → buildlet
val start_block: lbl → buildlet

val empty_cfg_builder: cfg_builder
val get_cfg: cfg_builder → cfg
```

From Typed AST to LLVM--

- Sources of concerns in the code generation

Going from a Tree data structure to a Linear data structure

Scoping and shadowing

Mutable variables

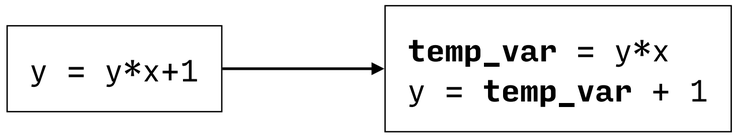
```
var x: int = read_integer ();
var y: int = 0;
if (x == 0) {
    var x: int = read_integer ();
    y = read_integer ();
    y = y*x+1
}
print_integer (x);
print_integer (y);
return;
```

Translation in broad strokes

(not in the syntax of anything particular)

Going from a Tree data structure to a Linear data structure

```
var x: int = read_integer ();
var y: int = 0;
if (x == 0) {
    var x: int = read_integer ();
    y = read_integer ();
    y = y*x+1
}
print_integer (x);
print_integer (y);
return;
```



Need to generate fresh variables as part of the translation

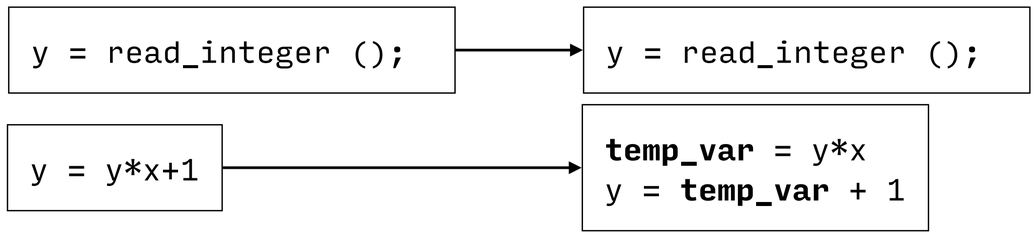
Multiple instructions in the target correspond to one source instruction

Translation in broad strokes

(not in syntax of anything particular)

Mutable variables

```
var x: int = read_integer ();
var y: int = 0;
if (x == 0) {
    var x: int = read_integer ();
    y = read_integer ();
    y = y*x+1
}
print_integer (x);
print_integer (y);
return;
```



Cannot (directly) use SSA registers for variables

Space for mutable variables allocated on the stack
- use LOAD for reading
- use STORE for updating

Point of declaration in the source does not correspond to the point of allocation in the target

once again

Multiple instructions in the target correspond to one source instruction

```
(* allocating the locals    *)
(* in the first basic block *)
%x_loc2 = alloca i64
%y_loc3 = alloca i64
```

...

```
y = read_integer ();
```

→

```
y = read_integer ();
```

→

```
%y_tmp_5 = call @read_integer ()
store i64 %y_loc3 %y_tmp_5
```

```
y = y*x+1
```

→

```
temp_var = y*x
y = temp_var + 1
```

→

```
%y_tmp_6 = load i64, i64* %y_loc3
%x_tmp_7 = load i64, i64* %x_loc2
%temp_var_7 = mult i64 %y_tmp_6 %x_tmp_7
%y_tmp_7 = add i64 %temp_var_7 1
store i64 %y_loc3 %y_tmp_7
```

Cannot (directly) use SSA registers for variables

We need a map from local variables in the source to LLVM operands corresponding to their allocation, i.e.,

$$\texttt{x} \mapsto \texttt{x\_loc2}; \texttt{y} \mapsto \texttt{y\_loc3}$$

Question: why no STORE/LOAD for `temp_var`?

Translation in broad strokes

Scoping and shadowing

```
var x: int = read_integer ();
var y: int = 0;
if (x == 0) {
    var x: int = read_integer ();
    y = read_integer ();
    y = y*x+1
}
print_integer (x);
print_integer (y);
return;
```

```
(* allocating the locals    *)
(* in the first basic block *)
%x_loc1 = alloca i64
%x_loc2 = alloca i64
```

$x \mapsto x\_loc1;$

$x \mapsto x\_loc2;$

$x \mapsto x\_loc1;$

The map for the variables needs to be “in sync” with the scoping rules of the language

Consolidating all the details

```
var x: int = read_integer ();
var y: int = 0;
if (x == 0) {
    var x: int = read_integer ();
    y = read_integer ();
    y = y*x+1
}
print_integer (x);
print_integer (y);
return;
```

Need to generate fresh variables as part of the translation

Multiple instructions in the target correspond to one source instruction

Space for mutable variables allocated on the stack
- use LOAD for reading
- use STORE for updating

Point of declaration in the source does not correspond to the point of allocation in the target

We need a map from local variables in the source to LLVM operands corresponding to their allocation, i.e.,

$$\texttt{x} \mapsto \texttt{x\_loc2}; \texttt{y} \mapsto \texttt{y\_loc3}$$

The map for the variables needs to be “in sync” with the scoping rules of the language

Alternatives

- α-translation into a form where all variable names are unique, before code generation
  - conceptually cleaner
  - extra-work in practice because of a separate translation
    - may not be worth it for as long as the language is small/simple, but definitely worth doing for a larger language
    - needs separate AST representation

α-conversion: the idea

```
var x: int = read_integer ();
var y: int = 0;
if (x == 0) {
    var x: int = read_integer ();
    y = read_integer ();
    y = y*x+1
}
print_integer (x);
print_integer (y);
return;
```

Original

```
var x: int = read_integer ();
var y: int = 0;
if (x == 0) {
    var x2: int = read_integer ();
    y = read_integer ();
    y = y*x2+1
}
print_integer (x);
print_integer (y);
return;
```

All variables have unique names (just renaming)

- obs: this is not SSA, vars can still be modified

```
var x: int (* uninitialized *)
var y: int (* uninitialized *)
var x2: int; (* uninitialized *)

x = read_integer ();
y: int = 0;
if (x == 0) {
   x2 = read_integer ();
   y = read_integer ();
   y = y*x2+1
}
print_integer (x);
print_integer (y);
return;
```

Shadowing is no-longer an issue, declarations are moved up to the start of the program.

- Some decisions need to be made about delayed initialization.
  - This is why this is a pass that follows type-checking, not prior to it.

α-conversion: the idea

```
var x: int (* uninitialized *)
var y: int (* uninitialized *)
var x2: int; (* uninitialized *)

x = read_integer ();
y: int = 0;
if (x == 0) {
   x2 = read_integer ();
   y = read_integer ();
   y = y*x2+1
}
print_integer (x);
print_integer (y);
return;
```

Shadowing is no-longer an issue, declarations are moved up to the start of the program.

- Some decisions need to be made about delayed initialization.
  - This is why this is a pass that follows type-checking, not prior to it.

All declarations are at the top of the program

No declarations elsewhere

The map from locals to alloca-operands is created once and stays unchanged throughout the codegen

No need to “thread” the environment in between statements

Organizing the translation

Following the structure of the (Typed) AST

```
module Sym = Symbol

type ident = Ident of {sym : Sym.symbol}
type typ =| Void | Int | Bool | ErrorType
type binop = | Plus |Minus | Mul | Div | Rem | Lt | Le
             | Gt | Ge | Lor | Land | Eq | NEq
type unop = | Neg | Lnot

type expr =
| Integer of {int : int64}
| Boolean of {bool : bool}
| BinOp of {left : expr; op : binop; right : expr; tp : typ}
| UnOp of {op : unop; operand : expr; tp : typ}
| Lval of lval
| Assignment of {lvl : lval; rhs : expr; tp : typ}
| Call of {fname : ident; args : expr list; tp : typ}
and lval =
| Var of {ident : ident; tp : typ}

type statement =
| VarDeclStm of {name : ident; tp : typ; body : expr}
| ExprStm of {expr : expr option}
| IfThenElseStm of {cond : expr; thbr : statement
                                 ; elbro : statement option}
| CompoundStm of {stms : statement list}
| ReturnStm of {ret : expr}

type param = Param of {paramname : ident; typ : typ}
type funtype = FunTyp of {ret : typ; params : param list}
type program = statement list
```

We have two main syntactic entities: *expr* and *statement*

```
let rec trans_expr ... expr =
  match expr with
  | ...
```

```
let rec trans_stmt ... stm =
  match stm with
  | ...
```

What else should these functions take as arguments and what should they return?

Organizing the environment

- What information to have in there?
- When we know what instruction should be added to the CFG, how do we do it?
  - There are two strategies
    - in-place modifications
    - non-destructive modifications