

Cb AST→LIR Lowering

1 Overview

We need to translate from the AST data structure to the LIR data structure. Note that:

- LIR still uses `Type` just like AST.
- LIR has `Program` and `Function` like AST, but with different field types (mainly because now that we've done validation and ensured there are no duplicates, we can safely use maps instead of vectors).
- LIR has some new data types, specifically for a function's body, that we use to represent a function's control-flow graph (CFG).

Converting everything but function bodies is trivial, the only complicated part is lowering AST statements and expressions into LIR instructions and organizing them into a LIR control-flow graph. Our strategy for translating a function's body is:

1. Lower the statements into a single “translation vector” whose elements are LIR instructions and also labels `Label(name)`, where `name` is a string.
2. Take the resulting vector and use it to construct the control-flow graph for the corresponding LIR function body.

Section 2 describes the lowering algorithm, and Section 3 describes how to convert a translation vector into a control-flow graph suitable for the LIR function body.

2 Lowering Algorithm

During the lowering process we will need several things:

- To create fresh LIR variables (“fresh” means that they are unique within the enclosing function). To do this we will use a helper function that takes a type and (1) creates a new variable of that type, inserting it into the enclosing function's `locals`; and (2) returns that variable to the caller. To make the variable names consistent they should be called `_t<n>`, where `n` is a counter that starts from 1 for each function and increases each time the helper function is called (e.g., `_t1`, `_t2`, etc). We use an underscore in the variable name to ensure that we don't clash with user-defined variable names.
- To create fresh labels. Again we will use a helper function. To make the label names consistent they should be called `lb1<n>`, where `n` is a counter that starts from 1 for each function and increases each time the helper function is called (e.g., `lb11`, `lb12`, etc).
- To look up type information. We will only need to look up type information for LIR variables. To do so: (1) we look in the `locals` of the enclosing function; if it isn't there then (2) we look in the parameters of the enclosing function; if it isn't there then (3) we look in the `globals` for the program. If we're lowering a valid Program then this is guaranteed to find the requested type.

2.1 Lowering Programs

Given a `AST::Program prog`:

- Create an empty `LIR::Program lir`.
- Copy `prog.{globals, externs, structs, functions}` into `lir` (translating into the appropriate data structures) *except* ignore any local variable initializations and leave all function bodies empty.
- For all `func ∈ prog.functions`: add `[func.name → Ptr(Fn(func.params.types, func.rettyp))]` to `lir.globals`; these are the implicit function pointers made explicit.

Now we are ready to lower the function bodies. For each function `func ∈ prog.functions`:

- Create an vector that will hold the emitted instructions/labels for that function (the “translation vector”). Its first and only element should be `Label("entry")`.
- Eliminate local variable initializations by turning them into assignments and adding them to the beginning of `func.stmts` (taking care to preserve the order of the initializations).
- Compute $\llbracket \text{func.stmts} \rrbracket^s$, which will fill in the translation vector.
- Enforce having a single `Return` instruction. If there is more than one `Return` in the translation vector:

```
let EXIT be a fresh label
emit Label(EXIT)
if func returns a value then
  let x be a fresh var with type  $\tau$  s.t. func.rettyp =  $\tau$ 
  emit Return(x)
  replace all previous Return(op) instructions with Assign(x, op); Jump(EXIT)
else
  emit Return(None)
  replace all previous Return(None) instructions with Jump(EXIT)
```

- Take the final translation vector and construct the CFG for `lir.functions[func].body`.

2.2 Lowering Statements

These functions emit LIR instructions into the translation vector without returning anything.

$\llbracket \text{stmts} \rrbracket^s$: $\forall s \in \text{stmts}. \llbracket s \rrbracket^s$

```
 $\llbracket \text{If}(\text{guard}, \text{tt}, \text{ff}) \rrbracket^s$ :
let TT, FF, IF_END be fresh labels
emit Branch( $\llbracket \text{guard} \rrbracket^e$ , TT, FF)
emit Label(TT)
 $\llbracket \text{tt} \rrbracket^s$ 
emit Jump(IF_END)
emit Label(FF)
 $\llbracket \text{ff} \rrbracket^s$ 
emit Jump(IF_END)
emit Label(IF_END)
```

```

[[While(guard, body)]]s:
  let WHILE_HDR, WHILE_BODY, WHILE_END be fresh labels
  emit Label(WHILE_HDR)
  emit Branch([[guard]]e, WHILE_BODY, WHILE_END)
  emit Label(WHILE_BODY)
  [[body]]s
  emit Jump(WHILE_HDR)
  emit Label(WHILE_END)

[[Assign(lhs, RhsExp(e))]]s:
  if lhs is Id(name) then emit Copy(Var(name), [[e]]e)
  else
    let x = [[lhs]]ℓ
    let y = [[e]]e
    emit Store(x, y)

[[Assign(lhs, New(typ, e))]]s:
  if lhs is Id(name) then emit Alloc(Var(name), [[e]]e)
  else
    let w be a fresh var with type &typ
    let x = [[lhs]]ℓ
    emit Alloc(w, [[e]]e)
    emit Store(x, w)

[[Call(callee, args)]]s:
  let aops = ∀a ∈ args. [[a]]e
  if callee is Id(name) and name is an extern then emit CallExt(None, name, aops)
  else
    let NEXT be a fresh label
    if callee is Id(name) and name is a function then emit CallDirect(None, name, aops, NEXT)
    else emit CallIndirect(None, [[callee]]ℓe, aops, NEXT)
    emit Label(NEXT)

[[Break]]s:
  find the nearest previous Label(WHILE_HDR)
  emit Jump(WHILE_HDR)

[[Continue]]s:
  find the nearest previous Branch(−, −, WHILE_END)
  emit Jump(WHILE_END)

[[Return(None)]]s: emit Return(None)

[[Return(e)]]s: emit Return([[e]]e)

```

2.3 Lowering Expressions

These functions emit LIR instructions into the translation vector that will compute the value of the expression being translated, returning a LIR Operand (a variable or constant) containing the final value of the expression.

```
[[Num(n)]]e: Const(n)
```

```
[[Id(name)]]e: Var(name)
```

```
[[Nil]]e: Const(0)
```

```
[[UnOp(Neg, e)]]e:  
  let lhs be a fresh var of type Int  
  emit Arith(lhs, Sub, Const(0), [[e]]e)  
  lhs
```

```
[[UnOp(Deref, e)]]e:  
  let src = [[e]]e  
  let lhs be a fresh var of type  $\tau$  s.t. src:& $\tau$   
  emit Load(lhs, src)  
  lhs
```

```
[[BinOp(op  $\in$  {Add, Sub, Mul, Div}, left, right)]]e:  
  let op1 = [[left]]e  
  let op2 = [[right]]e  
  let lhs be a fresh var of type Int  
  emit Arith(lhs, op, op1, op2)  
  lhs
```

```
[[BinOp(op  $\in$  {Equal, NotEq, Lt, Lte, Gt, Gte}, left, right)]]e:  
  let op1 = [[left]]e  
  let op2 = [[right]]e  
  let lhs be a fresh var of type Int  
  emit Cmp(lhs, op, op1, op2)  
  lhs
```

```
[[ArrayAccess(ptr, index)]]e:  
  let src = [[ptr]]e  
  let idx = [[index]]e  
  let elem be a fresh var of type & $\tau$  s.t. src:& $\tau$   
  let lhs be a fresh var of type  $\tau$  s.t. src:& $\tau$   
  emit Gep(elem, src, idx)  
  emit Load(lhs, elem)  
  lhs
```

```
[[FieldAccess(ptr, fld)]]e:  
  let src = [[ptr]]e  
  let fldp be a fresh var of type & $\tau$  s.t. src:&Structid, id[fld]: $\tau$   
  let lhs be a fresh var of type  $\tau$  s.t. src:&Structid, id[fld]: $\tau$   
  emit Gfp(fldp, src, fld)  
  emit Load(lhs, fldp)  
  lhs
```

```

[[Call(callee, args)]]e:
  let aops =  $\forall a \in \text{args}. \llbracket a \rrbracket^e$ 
  let fun = [[callee]]e
  let lhs be a fresh var of type  $\tau$  s.t.  $\text{fun} : \&(\_) \rightarrow \tau$ 
  if callee is Id(name) and name is an extern then emit CallExt(lhs, name, aops)
  else
    let NEXT be a fresh label
    if callee is Id(name) and name is a function then emit CallDirect(lhs, name, aops, NEXT)
    else emit CallIndirect(lhs, fun, aops, NEXT)
    emit Label(NEXT)
  lhs

```

2.4 Lowering Lvals

These functions emit LIR instructions into the translation vector that will compute the location where a value should be stored and return a variable that contains a pointer to that location. Note that the argument should *not* be an Id.

```

[[Deref(lv)]]ℓ: [[lv]]ℓe

```

```

[[ArrayAccess(ptr, index)]]ℓ:
  let src = [[ptr]]ℓe
  let idx = [[index]]e
  let lhs be a fresh var of type  $\tau$  s.t.  $\text{src} : \tau$ 
  emit Gep(lhs, src, idx)
  lhs

```

```

[[FieldAccess(ptr, fld)]]ℓ:
  let src = [[ptr]]ℓe
  let lhs be a fresh var of type  $\&\tau$  s.t.  $\text{src} : \&\text{Struct}_{id}, id[\text{fld}] : \tau$ 
  emit Gfp(lhs, src, fld)
  lhs

```

2.5 Lowering Lvals as Expressions

These functions lower an Lval as if it were an expression, returning a variable containing the final value. Note that the argument *can* be an Id.

```

[[Id(name)]]ℓe: Var(name)

```

```

[[lv ≠ Id(name)]]ℓe:
  let src = [[lv]]ℓ
  let lhs be a fresh var of type  $\tau$  s.t.  $\text{src} : \&\tau$ 
  emit Load(lhs, src)

```

3 Constructing the Control-Flow Graph (CFG)

We need to take the emitted instructions and organize them into a graph of basic blocks (the CFG). To optimize the resulting CFG a bit, we do the following transformations on the translation vector *before* transforming it into a CFG (iterating the last two optimizations until neither applies):

- If there is a sequence of more than one **Labels**, we remove all but the last **Label** and change any reference to a removed **Label** into a reference to the last **Label** in the sequence.
- If a leader instruction labeled LBL1 is **Jump**(LBL2), we change all terminals that reference LBL1 to reference LBL2.
- If a terminal is **Branch**(_, LBL, LBL) then we transform it into **Jump**(LBL).

Here is the process for constructing the CFG given the translation vector:

1. Identify the *leader* instructions, which are all instructions that immediately follow a **Label**. These are the instructions that will start the basic blocks.
2. For each leader instruction (labeled with **Label**(name)), create a LIR BasicBlock **bb** with label **name**, then copy the leader and all following instructions into **bb**'s statements until we reach the last instruction before the next **Label**; this instruction is the terminal of **bb**.
 - If we lowered correctly then the terminal must be a **Branch**, **Jump**, **Return**, **CallDirect**, or **CallIndirect**, and those instructions can *only* appear as terminals.