C♭ 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`$\langle n \rangle$, 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 `lbl`$\langle n \rangle$, where n is a counter that starts from 1 for each function and increases each time the helper function is called (e.g., `lbl1`, `lbl2`, 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` $\in$ `prog.functions`: add [`func.name` $\to$ `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` $\in$ `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 $[\![\texttt{func.stmts}]\!]^s$, 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 = if vector ends in Label(lbl) then lbl
           else
             let NEW_LBL be a fresh label
             emit Label(NEW_LBL)
             NEW_LBL
if func returns a value then
  let x be a fresh var with type τ s.t. func.rettyp = τ
  emit Return(x)
  replace all other 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.

$[\![\texttt{stmts}]\!]^s$: $\forall \texttt{s} \in \texttt{stmts}\,.\,[\![\texttt{s}]\!]^s$

$[\![\texttt{If(guard, tt, ff)}]\!]^s$:

```
let TT, FF, IF_END be fresh labels
emit Branch([[guard]]^e, TT, FF)
emit Label(TT)
[[tt]]^s
emit Jump(IF_END)
emit Label(FF)
[[ff]]^s
emit Jump(IF_END)
emit Label(IF_END)
```

```
⟦While(guard, body)⟧ˢ:
  let WHILE_HDR, WHILE_BODY, WHILE_END be fresh labels
  emit Jump(WHILE_HDR)
  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
  let direct = callee is Id(name) and name is not shadowed by a local/parameter
  if direct and name is an extern then emit CallExt(None, name, aops)
  else
    let NEXT be a fresh label
    if direct and name is a function then emit CallDirect(None, name, aops, NEXT)
    else emit CallIndirect(None, ⟦callee⟧ˡᵉ, aops, NEXT)
    emit Label(NEXT)

⟦Continue⟧ˢ:
  find the nearest previous Label(WHILE_HDR)
  emit Jump(WHILE_HDR)

⟦Break⟧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.

$[\![\texttt{Num(n)}]\!]^e$: `Const(n)`

$[\![\texttt{Id(name)}]\!]^e$: `Var(name)`

$[\![\texttt{Nil}]\!]^e$: `Const(0)`

$[\![\texttt{UnOp(Neg, e)}]\!]^e$:

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

$[\![\texttt{UnOp(Deref, e)}]\!]^e$:

```
let src = [[e]]^e
let lhs be a fresh var of type τ s.t. src:&τ
emit Load(lhs, src)
lhs
```

$[\![\texttt{BinOp(op} \in \{\texttt{Add, Sub, Mul, Div}\}\texttt{, left, right)}]\!]^e$:

```
let op1 = [[left]]^e
let op2 = [[right]]^e
let lhs be a fresh var of type Int
emit Arith(lhs, op, op1, op2)
lhs
```

$[\![\texttt{BinOp(op} \in \{\texttt{Equal, NotEq, Lt, Lte, Gt, Gte}\}\texttt{, left, right)}]\!]^e$:

```
let op1 = [[left]]^e
let op2 = [[right]]^e
let lhs be a fresh var of type Int
emit Cmp(lhs, op, op1, op2)
lhs
```

$[\![\texttt{ArrayAccess(ptr, index)}]\!]^e$:

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

$[\![\texttt{FieldAccess(ptr, fld)}]\!]^e$:

```
let src = [[ptr]]^e
let fldp be a fresh var of type &τ s.t. src:&Struct_id, id[fld]:τ
let lhs be a fresh var of type τ s.t. src:&Struct_id, id[fld]:τ
emit Gfp(fldp, src, fld)
emit Load(lhs, fldp)
lhs
```

$[\![\texttt{Call(callee, args)}]\!]^{e}$:
```
  let aops = ∀a ∈ args . [[a]]^e
  let direct = callee is Id(name) and name is not shadowed by a local/parameter
  let fun = [[callee]]^e
  let lhs be a fresh var of type τ s.t. fun:&(_)→τ
  if direct and name is an extern then emit CallExt(lhs, name, aops)
  else
    let NEXT be a fresh label
    if direct 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`.

$[\![\texttt{Deref(lv)}]\!]^{\ell}$: $[\![\texttt{lv}]\!]^{\ell e}$

$[\![\texttt{ArrayAccess(ptr, index)}]\!]^{\ell}$:
```
  let src = [[ptr]]^{ℓe}
  let idx = [[index]]^e
  let lhs be a fresh var of type τ s.t. src:τ
  emit Gep(lhs, src, idx)
  lhs
```

$[\![\texttt{FieldAccess(ptr, fld)}]\!]^{\ell}$:
```
  let src = [[ptr]]^{ℓe}
  let lhs be a fresh var of type &τ s.t. src:&Struct_id, id[fld]:τ
  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`.

$[\![\texttt{Id(name)}]\!]^{\ell e}$: `Var(name)`

$[\![\texttt{lv} \neq \texttt{Id(name)}]\!]^{\ell e}$:
```
  let src = [[lv]]^ℓ
  let lhs be a fresh var of type τ s.t. src:&τ
  emit Load(lhs, src)
  lhs
```

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). 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 a terminal instruction; this instruction is the terminal of `bb`.

3. After reading a terminal instruction and ending the current basic block, skip until we hit a new `Label`, which will start the next basic block. We do this skipping because, due to the way we translated and removed multiple returns, it's possible that unreachable instructions are in the vector, and this process will ignore them.

Once we have constructed the CFG, we should remove any unreachable basic blocks (this can arise due to **break**, **continue**, and **return** statements that exit a block of code before we would normally expect). There are still some inefficiencies remaining in our translation—for example, an empty **if** branch will result in an extra jump instruction. That's fine, because we can optimize these inefficiencies away later.