Intermediate Code Generation (ICG)

Transform AST to lower-level intermediate representation

Basic Goals: *Separation of Concerns*

- Generate efficient code sequences for *individual* operations
- Keep it fast and simple: leave most optimizations to later phases
- Provide clean, easy-to-optimize code
- IR forms the basis for code optimization and target code generation

Mid-level vs. Low-level Model of Compilation

- *Both* models can use a machine-*in*dependent IR:
  - AST ⟶ [machine-specific code sequences] ⟶ target code
  - AST ⟶ [machine-independent code sequences] → [machine-level IR] → target code
- Key difference: where does “target instruction selection” happen

Intermediate code generation — Overview

Goal: Translate AST to low-level *machine-independent 3-address IR*

Assumptions

- Intermediate language: RISC-like 3-address code‡
- Intermediate Code Generation (ICG) is independent of target ISA
- Storage layout has been pre-determined
- Infinite number of registers + Frame Pointer (FP)

  Q. *What values can* live *in registers?*

‡ *ILOC: Cooper and Torczon, Appendix A.*

Strategy

1. Simple bottom-up tree-walk on AST
2. Translation uses only local info: current AST node + children
3. Good (local) code is important!

   ⟸ Later passes have less semantic information

   ⟸ E.g., array indexing, boolean expressions, case statements
4. We will discuss important special cases

Code generation for expression trees

Illustrates the tree-walk scheme

- assign a virtual register to each operator
- emit code in *postorder* traversal of expression tree

Notes

- assume tree reflects precedence, associativity
- assume all operands are integers
- `base()` and `offset()` may emit code
- `base()` handles lexical scoping

Support routines

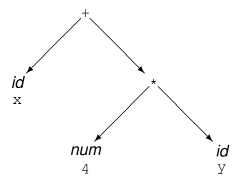
- `base( str )` — looks up `str` in the symbol table and returns a virtual register that contains the base address for `str`
- `offset( str )` — looks up `str` in the symbol table and returns a virtual register that contains the offset of `str` from its base register
- `new_name()` — returns a new virtual register name

Simple treewalk for expressions

```
expr( node )
   int result, t1, t2, t3;
   switch( type of node )
   {
      case TIMES:
         t1 = expr( left child of node );
         t2 = expr( right child of node );
         result = new_name();
         emit( mult, t1, t2, =>, result );
         break;
      case PLUS:
         t1 = expr( left child of node );
         t2 = expr( right child of node );
         result = new_name();
         emit( add, t1, t2, =>, result );
         break;
      case ID:
         t1 = base( node.val );
         t2 = offset( node.val );
         result = new_name();
         emit( loadAO, t1, t2, =>,
         break;
      case NUM:
         result = new_name();
         emit( loadI, node.val, =>
         break;
   }
   return result;
```

Minus & divide follow the same pattern

Code generation



Assume base for x and y is fp

```
loadI    offset of x  => r1
loadAO   fp, r1       => r2     ; r2 ← x
loadi    4            => r3     ; constant
loadi    offset of y  => r4
loadAO   fp, r4       => r5     ; r5 ← y
mult     r3, r5       => r6
add      r2, r6       => r7
```


Mixed type expressions

Mixed type expressions

- E.g., `x + 4 * 2.3e0`
- expression must have a clearly defined meaning
- typically convert to more general type
- complicated, machine dependent code

Typical Language Rule

E.g., `x + 4`, where $(T_{\mathtt{x}} \neq T_4)$:

1. $T_{result} \leftarrow f(+, T_{\mathtt{x}}, T_4)$
2. convert `x` to T_{result}
3. convert `4` to T_{result}
4. add converted values (*yields* T_{result})

Sample Conversion Table

`+`	`int`	`real`	`double`	`complex`
`int`	*int*	*real*	*double*	*complex*
`real`	*real*	*real*	*double*	*complex*
`double`	*double*	*double*	*double*	*complex*
`complex`	*complex*	*complex*	*complex*	*complex*

Array references

Example: `A[i,j]`

Basic Strategy

1. Translate i (may be an expr)
2. Translate j (may be an expr)
3. Translate &A + $[i, j]$
4. Emit load

Index Calculation assuming *row-major* order)

- Let $n_i = high_i - low_i + 1$
- *Simple address expression (in two dimensions)*:
  $base + ((i_1 - low_1) \times n_2 + i_2 - low_2) \times w$
- *Reordered address expression (in k dimensions)*:

$$((...(i_1 n_2 + i_2)n_3 + i_3)...)n_k + i_k) \times w$$
$$+\ base\ -$$
$$w \times ((...((low_1 \times n_2) + low_2)n_3 + low_3)...)n_k + low_k)$$

Optimizing the address calculation

$$((...(i_1 n_2 + i_2)n_3 + i_3)...)n_k + i_k) \times w$$
$$+\ base\ -$$
$$w \times ((...((low_1 \times n_2) + low_2)n_3 + low_3)...)n_k + low_k)$$

Constants

- Usually, all low_i are constants
- Sometimes, all n_i except n_1 (high-order dimension) are constant
  $\Longrightarrow$ final term is compile-time evaluable

Expose common subexpressions

- refactor first term to create terms for each i_r:
  $i_r \times n_{r+1} \times n_{r+2} \times \ldots \times n_k \times w$
- LICM: update r^{th} term only when i_r changes

$\Rightarrow$ *can remove much of the overhead*

Case statements

Basic rules

1. evaluate the controlling expression
2. branch to the selected case
3. execute its code
4. branch to the following statement

Main challenge: finding the right case

Method	*When*	*Cost*
linear search	few cases	**O**($\mid$ *cases* $\mid$)
binary search	sparse	**O**($\log_2(\mid$ *cases* $\mid)$)
jump table	dense	**O**(1), but with table lookup

Implementing Assignment of Mutable Objects

Assignment
```
let x: SomeType <- foo() in
    ...
```

Mutable Objects: Copy pointers

t0 = alloca pointer to SomeType-Object;

t2 = Call foo() ;; *t1 points to result object*

*t0 = t2 ;; *t0, t2 now point to same obj*

...

*Cannot copy object values from *t2 to *t0 because x must refer to the same object that the last statement evaluated in foo().*

E.g., Javascript 6 defines Object.assign(t2) that supports immutable object copy

Options for Implementing Immutable Objects

Immutable Type: Int
```
let x: Int <- foo() + self.n in
    let o: Object <- x in
        o.type_name() ...
```

Option 1: Objects everywhere

- Store primitive type values as first-class objects always
- E.g., Store i32 value as object field
- Load/store the field value before/after arithmetic ops on the object
- Normal object operations for all other ops, e.g., method dispatch

Option 2: Box / unbox only where needed

- Store primitive type values as primitive types, e.g., i32
- Arith. ops work directly on the values
- Object-level ops, e.g., method dispatch:
  - Alloc object before op
  - Copy value into object field (“box”)
  - Perform method dispatch
  - Copy final object field out (“unbox”)
  - Deallocate object if on heap

Options for Objects: How well can they be optimized?

This mainly applies to primitive objects / fields

For Mutable Types: Copy pointers

- Uniform code generator: no special cases for primitives
- Int fields: likely to remain objects in heap

Immutable Types, Option 1: Copy values

- Special cases for primitives: assignment, copies
- Local variables on heap: all promoted to int registers
- Temp objects on heap: all promoted to int registers
- Fields: likely to remain objects in heap

Immutable Types, Option 2: Box/unbox only where needed

- Special cases for primitives: assignment, copies, method dispatch
- Let vars, temps: held in int registers (except when boxed)
- Fields: simple primitive fields in parent object
- Overhead of boxing/unboxing can be eliminated if method calls inlined

Structures

Structure Accesses

`p->x` *becomes* `loadAI` r_p, `offset(x)` $\Rightarrow r_1$

Structure Layout: Key Goals

- All structure fields have constant offsets, fixed at compile-time
- May need padding between fields $\Longleftarrow$ *Why?*
- May need padding at *end of struct* $\Longleftarrow$ *Why?*

Structure Layout Example

```
struct Small { char* p; int n; };
struct Big   { char c; struct Small m; int i };
```

- Assume SparcV9: Byte alignments = $pointer$: 8, int: 4, $short$: 2
- Offsets? $p:$? $n:$? $c:$? $m:$? $i:$?
- sizeof(struct Small) = ? sizeof(struct Big) = ?

Class with single-inheritance

Key Operations and Terminology

- `p.x` — *Field access*
- `p.M`$(a_1, a_2, \ldots, a_n)$ — *Method dispatch*
- C_q `q = (`C_q`) p` — *Downcast*

Terminology and Type-Checking Assumptions

- C_p, C_q : the *static types* of references p, q
- O_p, O_q : the *dynamic types* of the objects p, q refer to
- $O_p \leq C_p$ (in COOL notation), i.e., O_p is lower in inheritance tree.
- `x, M` are valid members of $C_p \Longrightarrow$ valid for O_p
- For downcast, $O_p \leq C_q$ — *When is this checked?*

Class with single-inheritance: Code Generation Goals

Functional Goals

1. Class layouts, run-time descriptors constructed at compile-time
   Note: Class-loading-time in a JVM <u>is</u> compile-time
2. *Same* code sequences must work for any $O_p \leq C_p$
3. Separate compilation of classes
   $\Longrightarrow$ we know superclasses but not subclasses
   $\Longrightarrow$ code sequences, layouts, descriptors must be consistent for all classes

Efficiency Goals

1. Small constant-time code sequences
2. Avoid hashing *[class-name,method-name] $\rightarrow$ func ptr*
3. Minimize #indirection steps
4. Minimize #object allocations for temporaries
5. Two important optimizations: *inlining*, *dynamic $\rightarrow$ static dispatch*

Single-inheritance: Example

Runtime Objects

- *Class record*: One per class
- *Method table*: One per class
- *Object record*: One per instance

COOL Classes

```
class C1 (* inherits Object *) { x1: Int, y1: Int; M1(): Int };
class C2 inherits C1 { x2: Int; M2(): Int };
class C3 inherits C2 { x3: Int; M1(): Int; M3(): Int };
```

Class Records for Example

Class Records in C

```
struct ClassC1 { struct ClassObject* p; VTableC1 { $...$ }; int 1; }
struct ClassC2 { struct ClassC1* p;     VTableC2 { $...$ }; int 2; }
struct ClassC3 { struct ClassC2* p;     VTableC3 { $...$ }; int 3; }
```

Single-inheritance: Example (continued)

Method Tables for Example

Method Tables in C

```
struct VTableC1 { <Object methods>; int()* M1_C1; };
struct VTableC2 { <Object methods>; int()* M1_C1; int()* M2_C2; };
struct VTableC3 { <Object methods>; int()* M1_C3; int()* M2_C2; int()* M3_C3;
```

Object Records for Example

Object Records in C

```
struct ObjObject { void* classPtr;      /* no fields */ };
struct ObjC1     { struct ObjObject p1; int x1; int y1; };
struct ObjC2     { struct ObjC1     p2; int x2; };
struct ObjC3     { struct ObjC2     p3; int x3; };
```

Compare layouts of these object records:

ObjObject:	{ classPtr }
ObjC1:	{ classPtr; x1; y1 }
ObjC2:	{ classPtr; x1; y1; x2 }
ObjC3:	{ classPtr; x1; y1; x2; x3 }

Single-inheritance: Example (continued)

Code Sequence for Field Access

```
(* r2:  C2 <- new C3; r3:  C3 <- new C3*)
x:  Int <- r2.x1 + r3.x1;
```

Code Sequence for Method Dispatch

```
(* r3:  C1 <- new C3 *)
x:  Int <- r3.M1()
```


Runtime Safety Checks

Fundamental cost for safe *languages*:
Java, ML, Modula, Ada, . . .

Loads and Stores

- Initialize all pointer variables (including fields) to NULL
- Check $(p$ `!= 0`$)$ before every load/store using p *optimize for locals!*

Downcasts

- Record class identifier in class object
- Before downcast C_q `q = `(C_q)` p`: Check $O_p \leq C_q$

Array References

- *Empirical evidence*: These are by far the most expensive run-time checks
- Record size information just *before* array in memory
- Before array reference `A[`$\mathtt{expr}_0$`, ..., `$\mathtt{expr}_{n-1}$`]`: *optimize!*
  Check $(lb_i \leq \mathtt{expr}_i)$, $(\mathtt{expr}_i \leq ub_i), \forall\, 0 \leq i \leq n-1$

Separation of Concerns: Principles

Read: *The PL.8 Compiler, Auslander and Hopkins, CC82.*

Fundamental Principles

- Each compiler pass should address one goal and leave other concerns to other passes.
- Optimization passes should use a common, standardized IR.
- *All* code (user or compiler-generated) optimized uniformly

Key Assumptions

- register allocator does a great job $\Rightarrow$ *simplifies optimizations*
- optimization phase does a great job $\Rightarrow$ *simplifies translation*
- little or no special case analysis
- global data-flow analysis is worthwhile

Separation of Concerns: Optimizations and Examples

Optimization Passes in PL.8 Compiler

- Dead Code Elimination (DCE)
- Constant Propagation (CONST)
- Strength reduction
- Reassociation
- Common Subexpression Elimination (CSE)
- Global Value Numbering (GVN)
- Loop Invariant Code Motion (LICM)
- Dead Store Elimination (DSE)
- Control flow simplification (Straightening)
- Trap Elimination
- Peephole optimizations

Separation of Concerns: Examples

- ICG ignores common opts: DCE, CSE, LICM, straightening, peephole
- CSE and LICM ignore register allocation
- Instruction scheduling ignores register allocation

Separation of Concerns: Tradeoffs

Advantages

- Simple ICG: *bottom-up, context-independent*
- Opts. can ignore register constraints
- Each pass can be simpler $\Longrightarrow$ *more reliable, perhaps faster*
- Each optimization pass can be run multiple times.
- Sequences of passes can be run in different orders.
- Each pass gets used nearly every time $\Longrightarrow$ more reliable
- User-written and compiler-generated code optimized uniformly

Disadvantages

- Requires robust optimization algorithms
- Requires strong register allocation
- Compilation time?

22-1

EXTRA SLIDES

REMAINING SLIDES ARE EXTRA TOPICS

23-1

Whole arrays as procedure parameters

Three main challenges

1. Finding extents of all dimensions (including highest if checking bounds)
2. Passing non-contiguous section of larger array, e.g., Fortran 90:
   Formal Param: `F(:, :)`
   Actual Param (whole array): `A(1:100, 1:100)`,
   Actual Param (array section): `A(10:50:2,20:100:4)`
3. Passing an array by value

Language design choices

- C, C++, Java, Fortran 77: problem (1) is trivial and (2,3) don't exist
- Fortran 90, 95, ... : problems (1) and (2) are non-trivial

Passing whole arrays by value

- making a copy is extremely expensive
  $\Longrightarrow$ pass by reference, *copy-on-write* if value modified
- most languages (including call-by-value ones) pass arrays by reference

Whole arrays by reference

Finding extents

- pass a pointer to a *dope vector* as parameter:
  $[l_1, u_1, s_1, l_2, u_2, s_1, \ldots l_k, u_k, s_k]$
- stuff in all the values in the calling sequence
- generate address polynomial in callee
- interprocedural optimizations can eliminate this:
  - inlining
  - procedure specialization (aka cloning)
  - single caller

Passing non-contiguous section of larger array

- Fortran 90 requires that *section must have regular stride*
  $\Longrightarrow$ dope vector with strides is sufficient

Function calls in Expressions

Key issue: Side Effects

- Evaluation order is important
- Example: `func1(a) * globalX * func2(b)`
- Register save/restore will preserve intermediate values

Use standard calling sequence for each call

- set up the arguments
- generate the call and return sequence
- get the return value into a register

Boolean & relational expressions

Boolean expressions

boolean	→	`not` *or-term*
	|	*or-term*
or-term	→	*or-term* `or` *and-term*
	|	*and-term*
and-term	→	*and-term* `and` *value*
	|	*value*
value	→	`true`
	|	`false`
	|	*rel-term*

Relational expressions

rel-term	→	*rel-term rel-op expr*
	|	*expr*
rel-op	→	$<$ | $\leq$ | $=$ | $\neq$ | $\geq$ | $>$
expr	→	. . . (rest of expr grammar)

Short-circuiting Boolean Expressions

What is "short circuiting"?

Terms of a boolean expression can be evaluated until its value is established. Then, any remaining terms must not be evaluated.

Example

```
if (a && foo(b)) ...
```

call to `foo()` should not be made if a is false.

Basic Rules

- once value established, stop evaluating
- `true or` ⟨expr⟩ is `true`
- `false and` ⟨expr⟩ is `false`
- order of evaluation must be observed

Note: If order of evaluation is unspecified, short-circuiting can be used as an optimization: reorder by cost and short-circuit

Relations and Booleans using numerical values

Numerical encoding

- assign a value to `true`, such as `1` (`0x00000001`) or `-1` (`0xFFFFFFFF`)
- assign a value to `false`, such as `0`
- use hardware instructions — `and`, `or`, `not`, `xor`
- *Select values that work with the hardware* (*not 1 & 3*)

Example: **b or c and not d**

```
    t1 ← not d
    t2 ← c and t1
    t3 ← b or t2
```

Example: **if (a $<$ b)**

```
     if (a < b) br l1
     t1 ← false
     br l2
l1:  t1 ← true
l2:  nop    ; now use result
```

Can represent relational as boolean!
⇒ Integrates well into larger boolean expressions

Relationals using control flow

Encode using the program counter

- encode answer as a position in the code
- use conditional branches and hardware comparator
- along one path, relation holds; on other path, it does not

Example: `if (a < b) stmt`$_1$ `else stmt`$_2$

Naïve code:

```
         if (a < b) br lthen
         br lelse
lthen:   code for stmt1
         br lafter
lelse:   code for stmt2
         br lafter
lafter:  nop
```

After branch folding:

```
         if (a < b) br lthen
lelse:   code for stmt2
         br lafter
lthen:   code for stmt1
lafter:  nop
```

Path lengths are balanced.

Booleans using control flow

***Example*:**

```
if (a<b or c<d and e<f)
    then stmt1
    else stmt2
```

Naïve code:

```
        if (a < b) br lthen
        br l1
l1:     if (c < d) br l2
        br lelse
l2:     if (e < f) br lthen
        br lelse
lthen:  stmt1
        br lafter
lelse:  stmt2
        br lafter
lafter: nop
```

After branch folding:

```
        if (a < b)  br lthen
        if (c >= d) br lelse
        if (e < f)  br lthen
lelse:  stmt2
        br lafter
lthen:  stmt1
lafter: nop
```

It cleans up pretty well.

Control Flow vs. Numerical Representations: Tradeoffs

Hardware Issues

- *Condition code (CC)* registers: encode comparisons
- *Conditional moves*: use CC regs as boolean values
- *Predicated instructions*: use boolean values for conditional execution (instead of "control flow")

Tradeoffs

- Control flow works well when:
  - Result is only used for branching
  - Conditional moves and predicated execution are not available *or* code in branches is not appropriate for them
- Numerical representation works well when:
  - Result must be materialized in a variable
  - Result is used for branching but conditional moves or predicated execution are available and appropriate to use for code in branches

Control-flow constructs

Branches are common and expensive. Efficient inner loops are critical.

Examples

`if-then-else`: see Boolean/relational expressions
`do`, `while` or `for` loops
`switch` statement

Loops

- Convert to a common representation
- `do`: evaluate iteration count *first*
- `while` and `for`: test and backward branch at bottom of loop
  - $\Longrightarrow$ simple loop becomes a single basic block
  - $\Longrightarrow$ backward branch: easy to predict