Intermediate Representations

February 19, 2001

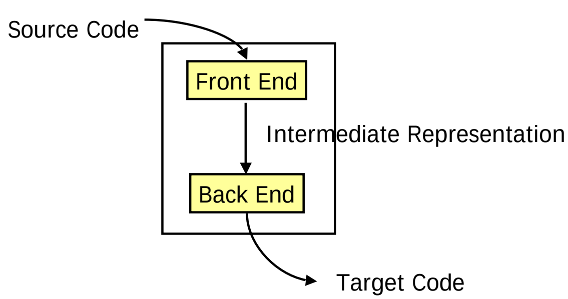
CS 132: Compiler Design

Where are we?

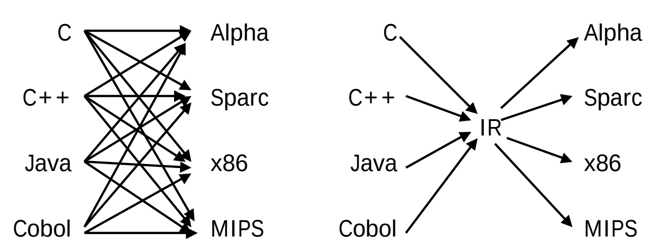
- We have talked about
  - Reading in concrete syntax
  - Generating abstract syntax
  - Analyzing and annotating the abstract syntax
    - Typechecking
    - Escaping variables
- What's next?

The Ends of a Compiler

- Most modern compilers are broken into halves:



Advantage of an IR



Choosing an IR

- Question: What should the IR be?
  - Abstract syntax?
  - SPARC assembly?
  - Something else?
- Choice is an art, not a science
  - Many possible representations
  - Different tradeoffs
  - Useful for different purposes
  - A single compiler may use several IR's

High vs. Low-Level Representations

- High-level
  - Primitives for complex operations, such as
    - Memory allocation
    - Looping constructs and conditionals
    - Procedure call and return
    - Complicated arithmetic expressions
    - Pattern matching
    - Exception raising and handling

High vs. Low-Level Representations

- Low-level
  - Primitives very simple (close to target code)
    - Simple arithmetic and tests
    - Jumps and branches
    - Memory loads and stores
    - May assume infinitely many variables/registers

(Unchecked) Access to a 20×10 array

Higher-level ⟷ Lower-level

```
x ← a[i,j+2]
```

```
t1 ← j+2
t2 ← i*20
t3 ← t1+t2
t4 ← 4*t3
t5 ← &a
t6 ← t5+t4
x  ← *t6
```

```
r1 ← *(fp-4)
r2 ← r1+2
r3 ← *(fp-8)
r4 ← r3*20
r5 ← r4+r2
r6 ← 4*r5
r7 ← fp-216
x  ← *(r6+r7)
```

Why Low-Level Representation?

- Simplicity
- Exposes more details, opportunities for optimization
- Less source-language specific

Checked Access to 10-word arrays

Higher-level ⟷ Lower-level

```
x ← a[i]
y ← b[i]
```

```
if i < 0 goto error
if i > 9 goto error
t1 ← &a
t2 ← 4*i
t3 ← t1+t2
x  ← *t3
if i < 0 goto error
if i > 9 goto error
t4 ← &b
t5 ← 4*i
t6 ← t4+t5
y  ← *t6
```

```
r1 ← *(fp-4)
r2 ← r1<0
bgt r2,error
r3 ← r1>9
bgt r3,error
...etc...
```

Why High-Level Representation?

- Otherwise information lost or obscured
  - Translate `for` and `while` to tests and branches
    - Loses loop structure of the code
  - Expand ML record creation to memory allocation + initialization (assignments to memory)
    - If the record is never used, then we don't have to create it at all. Less obvious that assignments can be eliminated.
- May want to delay decisions
  - Whether a variable is stored in memory or a register
  - What order to evaluate subexpressions
  - Code generation (`min` and `max`, or conditional branches)
  - Data representations (memory layout)

Pair Creation

Higher-level ⟷ Lower-level

```
p ← (999,998)
```

```
t1  ← alloc 12
t2  ← 0x83
*t1 ← t2
t3  ← t1+4
t4  ← 999
*t3 ← t4
t5  ← t1+8
t6  ← 998
*t5 ← t6
```

```
  hp ← hp-12
  r1 ← hp<lp
  bz r1, ok
  r1 ← 12
  call GC
ok:
  r2  ← 0x83
  *hp ← r2
  r3  ← hp+4
  r4  ← 999
  *r3 ← r4
  r5  ← hp+8
  r6  ← 998
  *r5 ← r6
```

A Common Choice

- Three-address code (a.k.a. *quadruples*)
  - Pseudo-assembly: very simple operations
  - Arbitrarily many temporaries
  - Conditional jumps to labels
  - Primitives for call/return
  - Memory load, stores

```
  i ← 1
  j ← 1
  k ← 0
L1:
  if k >= 100 goto done
  if j >= 20 goto L2
  j ← i
  k ← k + 1
  goto L3
L2:
  j ← k
  k ← k + 2
L3:
  j ← j - 1
  goto L1
```

- Instructions may be stored as long sequence, or in a more structured form

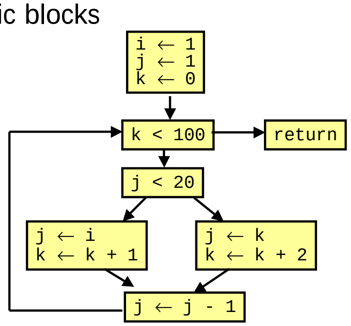
Basic Blocks

- A *basic block* is a maximal sequence of instructions that is only entered at the first instruction and which may leave the sequence at the last instruction

- An *extended basic block* is a maximal sequence of instructions that is entered only at the first instruction

Using Basic Blocks

- Control flow can be made explicit by making a DAG of basic blocks



Static Single Assignment

- Makes certain optimizations easier
- Idea: code only has one definition of any variable

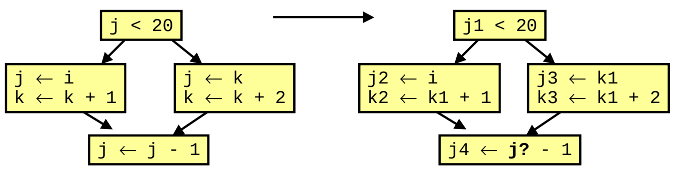
```
j ← 0
k ← 2
k ← k + 1
j ← j + k
```

→

```
j1 ← 0
k1 ← 2
k2 ← k1 + 1
j2 ← j1 + k2
```

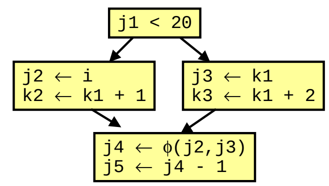
Static Single Assignment

- What do we do about join nodes?

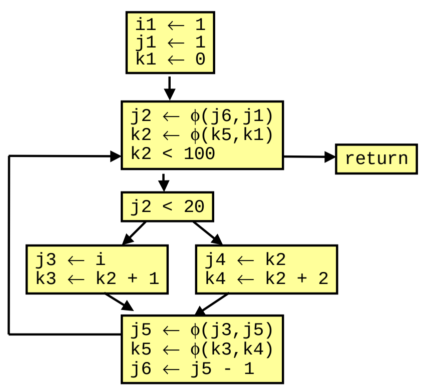


Static Single Assignment

- Answer: Insert ϕ-functions



Full SSA Example



Other Intermediate Representations

- Lambda-calculus based
  - Direct representations (like core ML)
    - e.g., A-normal form

```
fun f(i,j2,k2) =
  if (k2 < 100) then
     if (j2 < 20) then
        let val j3 = i
            val k3 = k2+1
        in f(i,j3,k3) end
     else
        let val j4 = k2
            val k4 = k2+2
        in f(i,j4,k4) end
  else
     (i,j2,k2)
```

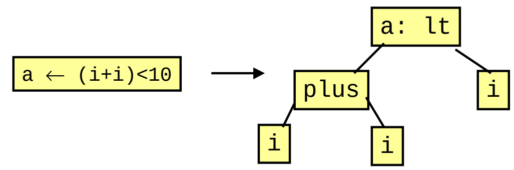
Other Intermediate Representations

- Lambda-calculus based
  - Continuation-based (a.k.a. CPS)
    - Every function takes a continuation function as an argument: what to do with its result
    - No function ever returns
    - Natural stackless implementation

```
add (t1, t2, fn t3 =>
mul (t3, 4,  fn t5 =>
add (t5, 1,  fn t6 =>
t6)))
```

Other Intermediate Representations

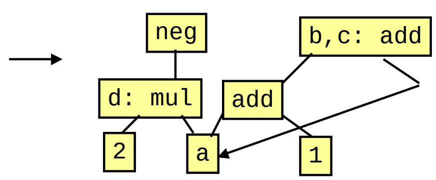
- Tree based



Other Intermediate Representations

- DAG based

```
c ← a
b ← a+1
c ← 2*a
d ← -c
c ← a+1
c ← b+a
d ← 2*a
b ← c
```



Tree Intermediate Language

```
signature TREE =
sig
   datatype exp = CONST of int
                | NAME  of Temp.label
                | BINOP of binop * exp * exp
                | MEM   of exp
                | CALL  of exp * exp list
                | ESEQ  of stm * exp
   and stm      = MOVE  of exp * exp
                | EXP   of exp
                | JUMP  of exp * Temp.label list
                | CJUMP of relop * exp * exp *
                              Temp.label * Temp.label
                | SEQ   of stm * stm
                | LABEL of Temp.label
   and binop = PLUS | MINUS | ... | RSHIFT | XOR | ...
   and relop = EQ | NE | LT | GT | ... | ULT | UGT | ...
end
```

Complication

- Some abstract syntax expressions return a result, and others don't.
- What should the type of our expression-translating function return?
  - `Tree.exp` ?
  - `Tree.stm` ?
  - Something else?

Appel's Answer

Define the type `Translate.exp` which can act like an expression or a statement as necessary.

```
signature TRANSLATE =
sig
    type exp
end
```