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

Source Code

Front End

Intermediate Representation

Back End

Target Code

C
C++
Java
Cobol
Alpha
Sparc
x86
MIPS

C
C++
Java
Cobol
Alpha
Sparc
x86
MIPS
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 20x10 array

Higher-level  →  Lower-level

\[
\begin{align*}
x & \leftarrow a[i, j+2] \\
t1 & \leftarrow j+2 \\
t2 & \leftarrow i*20 \\
t3 & \leftarrow t1+t2 \\
t4 & \leftarrow 4*t3 \\
t5 & \leftarrow &a \\
t6 & \leftarrow t5+t4 \\
x & \leftarrow *t6 \\
r1 & \leftarrow *(fp-4) \\
r2 & \leftarrow r1+2 \\
r3 & \leftarrow *(fp-8) \\
r4 & \leftarrow r3*20 \\
r5 & \leftarrow r4+r2 \\
r6 & \leftarrow 4*r5 \\
r7 & \leftarrow fp-216 \\
x & \leftarrow *(r6+r7)
\end{align*}
\]
**Why Low-Level Representation?**

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

**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)

**Checked Access to 10-word arrays**

**Pair Creation**
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
- 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
Static Single Assignment

- What do we do about join nodes?

\[
\begin{align*}
    j & \leftarrow i \\
    k & \leftarrow k + 1 \\
    j & \leftarrow k \\
    k & \leftarrow k + 2 \\
    j & \leftarrow j - 1 \\
\end{align*}
\]

\[
\begin{align*}
    j & \leftarrow i \\
    k & \leftarrow k + 1 \\
    j & \leftarrow k \\
    k & \leftarrow k + 1 \\
    j & \leftarrow j - 1 \\
\end{align*}
\]

\[
\begin{align*}
    j & \leftarrow i \\
    k & \leftarrow k + 1 \\
    j & \leftarrow k1 \\
    k & \leftarrow k1 + 2 \\
    j & \leftarrow j2 - 1 \\
\end{align*}
\]

\[
\begin{align*}
    j1 & \leftarrow 20 \\
    j2 & \leftarrow 1 \\
    k2 & \leftarrow k1 + 1 \\
    j3 & \leftarrow 1 \\
    k3 & \leftarrow k1 + 2 \\
    j4 & \leftarrow j3 - 1 \\
\end{align*}
\]

\[
\begin{align*}
    j & \leftarrow j2 - 1 \\
    k & \leftarrow k2 + 1 \\
    j & \leftarrow k2 \\
    k & \leftarrow k2 + 2 \\
    j & \leftarrow j3 - 1 \\
\end{align*}
\]

\[
\begin{align*}
    j1 & \leftarrow 20 \\
    j2 & \leftarrow 1 \\
    k2 & \leftarrow k1 + 1 \\
    j3 & \leftarrow 1 \\
    k3 & \leftarrow k1 + 2 \\
    j4 & \leftarrow j3 - 1 \\
\end{align*}
\]

\[
\begin{align*}
    j1 & \leftarrow 20 \\
    j2 & \leftarrow 1 \\
    k2 & \leftarrow k1 + 1 \\
    j3 & \leftarrow 1 \\
    k3 & \leftarrow k1 + 2 \\
    j4 & \leftarrow j3 - 1 \\
\end{align*}
\]

Full SSA Example

\[
\begin{align*}
    i1 & \leftarrow 1 \quad j1 \leftarrow 1 \quad k1 \leftarrow 0 \\
    j2 & \leftarrow \phi(j1,j1) \quad k2 \leftarrow \phi(k1,k1) \\
    j3 & \leftarrow 1 \quad k3 \leftarrow k2 + 1 \\
    j4 & \leftarrow k2 \quad k4 \leftarrow k2 + 2 \\
    j5 & \leftarrow \phi(j3,j5) \quad k5 \leftarrow \phi(k3,k4) \\
    j6 & \leftarrow j5 - 1 \\
\end{align*}
\]

\[
\begin{align*}
    i1 & \leftarrow 1 \quad j1 \leftarrow 1 \quad k1 \leftarrow 0 \\
    j2 & \leftarrow \phi(j3,j5) \\
    j3 & \leftarrow 1 \quad k3 \leftarrow k2 + 1 \\
    j4 & \leftarrow k2 \quad k4 \leftarrow k2 + 2 \\
    j5 & \leftarrow \phi(j3,j5) \quad k5 \leftarrow \phi(k3,k4) \\
    j6 & \leftarrow j5 - 1 \\
\end{align*}
\]

\[
\begin{align*}
    i1 & \leftarrow 1 \quad j1 \leftarrow 1 \quad k1 \leftarrow 0 \\
    j2 & \leftarrow \phi(j3,j5) \\
    j3 & \leftarrow 1 \quad k3 \leftarrow k2 + 1 \\
    j4 & \leftarrow k2 \quad k4 \leftarrow k2 + 2 \\
\end{align*}
\]

\[
\begin{align*}
    i1 & \leftarrow 1 \quad j1 \leftarrow 1 \quad k1 \leftarrow 0 \\
    j2 & \leftarrow \phi(j3,j5) \\
    j3 & \leftarrow 1 \quad k3 \leftarrow k2 + 1 \\
    j4 & \leftarrow k2 \quad k4 \leftarrow k2 + 2 \\
\end{align*}
\]

\[
\begin{align*}
    i1 & \leftarrow 1 \quad j1 \leftarrow 1 \quad k1 \leftarrow 0 \\
    j2 & \leftarrow \phi(j3,j5) \\
    j3 & \leftarrow 1 \quad k3 \leftarrow k2 + 1 \\
    j4 & \leftarrow k2 \quad k4 \leftarrow k2 + 2 \\
\end{align*}
\]

Other Intermediate Representations

- Lambda-calculus based
  - Direct representations (like core ML)
    - e.g., A-normal form
  
\[
\text{fun } f(i,j2,k2) = \\
\text{if } (k2 < 100) \text{ then} \\
\text{ let val } j3 = i \text{ in } f(i,j3,k2) \text{ end} \\
\text{else} \quad \text{if } (j2 < 20) \text{ then} \\
\quad \text{ let val } j4 = k2 \text{ in } f(i,j4,k2) \text{ end} \\
\quad \text{else} \quad \text{return } (i,j2,k2) \\
\text{else} \quad \text{return } (i,j2,k2) \\
\]
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

• DAG based

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

Other Intermediate Representations

• Tree based

```
a ← (i+1)<10
da: lt
b: plus
i
i
```

Tree Intermediate Language

```
signature TREE =
sig
datatype exp = CONST of int
  NAME of Temp.label
  BINOP of binop * exp * exp
  MUL of exp
  CALL of exp * exp list
  ESEQ of stm * 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
```