CS202 Compiler Construction

April 1, 2003

Dataflow analysis

Assignment 9

Announcements:
1. CS Seminar this Friday, on use of types in compilation
2. Article in this month’s ACM Computing Surveys on types in PLs
3. Guest lecture Thursday (Jeff Blaisdell, alias analysis)

4. New course policy:

Because homeworks hard, almost everyone taking grad credit…

NO PROJECT!

Final exam
Additional homework, time permitting
Undergrad(s): drop two homework grades
Generated code uses temps
   A figment of compiler; not real target entity
   Registers assigned by register allocation
Three analyses needed for reg alloc:
Build control flow graph
   What instructions can come next?
Solve data flow equations
   Where is this value used? defined?
Build interference graph
   Which values are in use at same time?

Control flow identifies what instructions can follow or precede an instruction

Linear within basic block
Between blocks:
   determined by jumps
   next/prev on basic blocks
Next instr(s) after jump, always first instr in some block(s)

Control flow represented by control flow graph (cfg)
  Directed graph
Directed graph
   Has nodes
   Has “one-way” edges labeled to/from

cfg is directed graph
   Each node is instruction or statement
   Each edge leads to a possible successor
Variable is live if
Value might be used again
Before being assigned again

Liveness range of variable:
Set of stmts or instrs where it is live

Consider i and j
Assume no more usage of i or j

i = 7;
j = 3;
k = i+j;
m = k-i;
i = m+k;
j = 0;
First liveness range of $i$

starts with $i = 7$;

\[
\begin{align*}
  i &= 7; \\
  j &= 3; \\
  k &= i+j; \\
  m &= k-i; \\
  i &= m+k; \\
  j &= 0;
\end{align*}
\]

First liveness range of $i$

continues through $m = k-i$;

\[
\begin{align*}
  i &= 7; \\
  j &= 3; \\
  k &= i+j; \\
  m &= k-i; \\
  i &= m+k; \\
  j &= 0;
\end{align*}
\]

Second definition of $i$

is just $i=m-k$;

\[
\begin{align*}
  i &= 7; \\
  j &= 3; \\
  k &= i+j; \\
  m &= k-i; \\
  i &= m+k; \\
  j &= 0;
\end{align*}
\]
First liveness range of $j$
is two statements

\[
i = 7; \\
m = k - i; \\
i = m + k; \\
j = 0;
\]

Second liveness range of $j$
is only $j = 0$;

\[
i = 7; \\
m = k - i; \\
i = m + k; \\
j = 0;
\]

$j$ is not alive during middle 2 stmts

Loop liveness more complicated

\[
i = 3 \\
\text{loop:} \\
j = i/2 \\
\text{if } (j == 0) \text{ goto exit} \\
i = j - 1 \\
goto \text{loop} \\
exit:
\]

Usage of $i$ at top comes from 2 places
2 parts of same liveness range

if (j == 0) goto exit

exit:
Either assignment may give i for j=i/2
i is not alive during test of j

Data flow based on sets
associated with each stmt/instr
Statements define values
def set
Statements use values
use set
Statements import set of values
in set
Statements export set of values (that may
be used again before redefinition)
out set

Data flow computed over cfg
Data flows across edges

At each node n:
Sets satisfy two dataflow equations
Want smallest satisfying sets

\[ \text{in}[n] = \text{use}[n] \cup (\text{out}[n] \setminus \text{def}[n]) \]
\[ \text{out}[n] = \bigcup_{s \in \text{next}(n)} \text{in}[s] \]

\[ \times \text{min}(\text{in}[n]) \]
Note that we use conservative approximation to solutions

out set may contain values that, in fact, are not used again

But if its used again, will definitely be in out set

May be more efficient solution, but don’t want to risk affecting semantics
Does that remind you of something?

Interesting fact: Recent research has shown dataflow analysis, certain type analyses equivalent

---

Solve data flow equations iteratively

For each statement/instr:
use and def explicit in statement/instr
Initialize in and out to minimum sets:
in = use
out = {}

Adjust for errors
Propagate around graph

---

foreach node n
in[n] = use[n]
out[n] = {}
do
changed = false
foreach node n
  out[n] = union(in[s] foreach s in succ(n))
in[n] = use[n] U (out[n] - def[n])
if (out or in changed) changed = true
while changed
\begin{itemize}
\item \texttt{i = 3}
\item loop: \texttt{j = i/2}
\item if \texttt{j == 0} goto exit
\item \texttt{i = j - 1}
\item goto loop
\end{itemize}

use and def sets given

\begin{itemize}
\item in: \{ \}
\item out: \{ \}
\item use: \{ \}
\item def: \{ i \}
\item in: \{ i \}
\item out: \{ \}
\item use: \{ i \}
\item def: \{ j \}
\item in: \{ j \}
\item out: \{ \}
\item use: \{ j \}
\item def: \{ \}
\item in: \{ j \}
\item out: \{ \}
\item use: \{ \}
\item def: \{ \}
\item in: \{ \}
\item out: \{ \}
\item use: \{ \}
\item def: \{ \}
\end{itemize}

initialize in and out sets: in=use, out={}
\textbf{Algorithm Example}

\begin{itemize}
  \item $i = 3$
  \item loop: $j = i/2$
  \item if ($j == 0$) goto exit
  \item $i = j - 1$
  \item goto loop
\end{itemize}

\textbf{In:} $\text{in}$ \hspace{1cm} \textbf{Use:} $\text{in}$ \hspace{1cm} \textbf{Out:} $\text{out}$ \hspace{1cm} \textbf{Def:} $\text{in}$

\textbf{Out:} $\text{out} = \bigcup \text{in[\text{s}], in[n] = use[n] U (out[n] - def[n])}$

CS 210-26

algorithm example
Completes 1 iteration of main loop
many sets changed

Need to make another iteration
nothing will change now

All done after two iterations (second changes nothing)
Chose statements in reverse order why?

Inter-node equation
    each node depends on next

Compute out then in
    Single change propagates all way back

Order can increase efficiency

What if analysis based on opposite dependency?

Use reverse ordering

Choose ordering based on direction of dependency

Assignment 9 out today
Code selection
    Using maximal munch algorithm

All work in SparcTranslator.java
Define translate functions
    for IR statement types
    for IR expr types
Back end is machine independent
   Except for isolated functions
   All live in Sparc*.java

How about instructions
   How can they be machine independent?

Abstracted into class Instr
   Represents one *assembly* instruction

Instr is not an abstract class
   Instructions are direct instances

Includes string
   Lists instruction name
   Describes instruction formatting

Instructions must support analysis
   Not just printing

Instrs have:
   Two lists of operands
      sources  all operands used
      targets  all operands modified
   List of labels
      Everywhere control could go next
   Modifies flag
      Is target simply copy of source?
      Important for e.g. register allocation
Operands are Locations
Location is interface

Includes IRTemp, IRReg, IRConstant

IRConstants not always valid operands
Must “fit” in instruction

String on instruction controls display
Indicates instruction
Indicates where operands go
  %s0 replaced with first source operand
  %t1 replaced with second target operand
  %l1 replaced with second label

“add %s0, %s1, %l0”

Lists support analysis

cfg built from labels

Data flow
  uses is sources
  def is targets

Used in reg alloc
Because of analysis, must include all
operands
Even if not appearing in format
Include implicit registers
Include false label for branch
Not mentioned explicitly
Not in format string
Does not appear in assembly

SparcTranslator.java
Contains all work to be done
Two kinds of translate functions
Translate for statement
Takes subclass of IRSmt
May take nextStmt — ignore
Returns list of instructions

Translate for expression
Takes subclass of IRExpr
Takes list of expressions

Returns Location
May append instructions to list

Guarantees value in Location
After instrs executed
Maximal munch
   Used by both forms of translate

Greedy algorithm
   Find largest subtree that matches
   Call translate on expr for rest

Try possible matches in order
   First should be largest
   Last should be smallest

Stop at first match

---

Translate on expressions
   Often recurses

Remember, append instrs to front on way up
   Otherwise not correct order

Strange recursion in
   translate(IRBinExpr,InstrList):
      IRExpr.translate(InstrList, this)

---

Why not call SparcTranslator.translate?
   Need double dispatch
   Want to minimize code
      Varies both on translator and expr

Call to IRExpr.translate
   Chooses correct expr type
   Calls to TargetTranslator.translate
      Gets correct target machine

Useful trick for single dispatch langs
You have to complete four cases:
- translate on IRCJump
- translate on IRBinExpr
- translate on IRCConvert
- translate on IRUnExpr

Use others as examples

Extra credit option

Efficient switch translation
Extra credit option

IR for switch uses full IRJump
default label goes to default
or end of switch if none
Each case x adds case to IRJump
x becomes nthValue
case label becomes nthLabel

switch (i)    JUMP i,[1,3],[L1,L2],L3
{case 1: L1:
  f(1);    CALL f,1
  break;    JUMP L4
  case 3: L2:
  f(3);    CALL f,3
  break;    JUMP L4
  default: L3:
  f(0);    CALL f,0
  break;    JUMP L4
} L4:
What instruction for that first JUMP?
Certainly no single instruction
(not on the sparc at least)

Two different approaches
jump table
series of if’s

Current implementation is jump table

Intuition:

Create array of jump target addr in memory

Use case value as index into array

Jump table code looks like

cmp i,1
blt L3
cmp i,3
bgt L3
sethi T1,%hi(L5)
or T1,%lo(L5),T2
sub i,1,T3
ld [T2+T4],T5
jmp T5
nop
Jump table itself comes after jmp

L5:
   .word L1
   .word L3
   .word L2
L1:
   mov 1,%e0
   call f
   ba L4
   nop
   ...

Jump table can implement any switch
Not always efficient
   Slower if numberCases <= 3
   Big table if cases sparse
For example, assume two cases:
   -10000000
   10000000
Requires 80MByte table!

Alternative:
   generate cmp/beq for each case
This approach slower as number cases grow
   Needs average n/2 cmp/beq
Code size gets large
Optimal answer blends both

Use jump table for large, dense spots

Use ifs otherwise
  also to choose jump table

For extra credit:

Implement if construct for switch

Use when \( \leq 4 \) cases or when density of cases \( \leq 1/4 \) of range