Liveness analysis

Simone Campanoni
simonec@eecs.northwestern.edu
Outline

• Introduction to register allocation

• Liveness analysis

• Calling convention
A register allocator structure

Register allocator

Map heuristic

f without varX

Current f, varX

Spill

f without variables
A graph-coloring register allocator structure

Register allocator

Code analysis

Graph coloring

f without variables and with registers

Spill

spill(f, var, prefix)

f with var spilled

f with

f
Task: From Variables to Registers

`:MyVeryImportantFunction 0

%MyVar1 <- 2
%MyVar2 <- 40
%MyVar3 <- %MyVar1
%MyVar3 += %MyVar2
print %MyVar3
`

- Why can we map MyVar1 and MyVar3 to r8?
- Why can’t we map MyVar1 and MyVar2 to r8?

We built the interference graph.
To compute it automatically, we need the liveness analysis.
Liveness analysis

**Goal:**
Identify the variables whose values might be used in the future just before and just after a given instruction i, for every i in a function f

\[
\begin{align*}
:myF (0) & { } \\
%myVar1 & <- 2 \\
%myVar2 & <- 40 \\
%myVar3 & <- %myVar1 \\
%myVar3 & += %myVar2 \\
\text{rax} & <- %myVar3 \\
\text{return} & { } \\
\end{align*}
\]

IN (just before) and OUT (just after) sets

\[
\begin{align*}
\text{IN[0]} & { } \\
\{ } & { } \\
{ } & {myVar1} \\
{ } & {myVar1, myVar2} \\
{ } & {myVar3, myVar2} \\
{ } & {myVar3} \\
\end{align*}
\]

IN[0], IN[1], OUT[0], IN[1], OUT[1], IN[2], OUT[1], IN[2], OUT[2], IN[3], OUT[2], IN[3], OUT[3], IN[4], OUT[3], IN[4]

Interference graph

```
myVar1 —— myVar2
  ^        |
  |        v
  myVar3
```

Graph coloring

:myF w/o variables

Live ranges
Outline

• Introduction to register allocation

• Liveness analysis

• Calling convention
Variables in the liveness analysis

• General Purpose (GP) 64-bit registers are seen as variables for the liveness analysis
  • rsp is not included

• Every time we say “variable” in the context of liveness analysis, we mean either L2 variables or GP 64-bit registers

• IN and OUT sets of the liveness analysis includes variables
  • Hence, they include L2 variables or GP 64-bit registers
  • IN[i] = {r10}
Execution path

Let $i$ be an instruction, we need to identify the set of variables with values that will be used just before and just after $i$ along all possible execution paths that include $i$. 

It is a predecessor of

It is a successor of
Successors of an instruction

\[ i ::= w \leftarrow s \mid w \leftarrow \text{mem} \times M \mid \text{mem} \times M \leftarrow s \mid w \leftarrow \text{stack-arg} \times M \mid \]
\[ w \leftarrow \text{aop} \times t \mid w \leftarrow \text{sop} \times s \mid w \leftarrow \text{sop} \times N \mid \text{mem} \times M \leftarrow t \mid \text{mem} \times M \leftarrow -t \mid w \leftarrow \text{mem} \times M \mid w \leftarrow \text{mem} \times M \mid \]
\[ w \leftarrow t \leftarrow \text{cmp} \times t \mid \text{cjump} \times t \leftarrow \text{cmp} \times t \times \text{label} \mid \text{label} \mid \text{goto} \times \text{label} \mid \]
\[ \text{return} \mid \text{call} \times u \times N \mid \text{call} \times \text{print} \times \text{t} \mid \text{call} \times \text{input} \times 0 \mid \text{call} \times \text{allocate} \times 2 \mid \text{call} \times \text{tensor-error} \times \text{F} \mid \]
\[ w++ \mid w-- \mid w @ w \times w \times E \]

An instruction \( i \) that has only one successor \( s \) and \( s \) is the instruction stored just after \( i \)

\[
\text{rdi} \leftarrow 5 \\
\text{call} \times \text{print} \times 1 \\
\text{r10} \leftarrow \text{rax} \times 5
\]
Successors of an instruction (2)

\[ i ::= \begin{align*}
& w <- s \mid \text{mem x M} \mid \text{mem x M} <- \text{s} \mid w <- \text{stack-arg M} \mid \\
& w \text{ aop t} \mid w \text{ sop sx} \mid w \text{ sop N} \mid \text{mem x M} += t \mid \text{mem x M} -= t \mid w += \text{mem x M} \mid w -= \text{mem x M} \mid \\
& w <- t \text{ cmp t} \mid \text{cjump t cmp t label} \mid \text{label} \mid \text{goto label} \mid \\
& \text{return} \mid \text{call u N} \mid \text{call print 1} \mid \text{call input 0} \mid \text{call allocate 2} \mid \text{call tensor-error F} \mid \\
& w++ \mid w-- \mid w @ w w E
\end{align*} \]

An instruction \( i \) that has only one successor \( s \) but \( s \) is not necessarily the instruction stored just after \( i \)

\[
\begin{align*}
& \text{goto :MY_LABEL_0} \\
& \text{rdi} <- 5 \\
& \text{call print 1} \\
& \text{goto :MY_LABEL_0} \\
& :\text{MY_LABEL_0}
\end{align*}
\]
Successors of an instruction (3)

\[ i ::= w \leftarrow s | w \leftarrow \text{mem } x \text{ M} | \text{mem } x \text{ M} \leftarrow s | w \leftarrow \text{stack-arg } M | \]
\[ w \text{ aop } t | w \text{ sop } s | w \text{ sop } N | \text{mem } x \text{ M} += t | \text{mem } x \text{ M} -= t | w += \text{mem } x \text{ M} | w -= \text{mem } x \text{ M} | \]
\[ w \leftarrow \text{t cmp } t | \text{cjump } t \text{ cmp } t \text{ label} | \text{label} | \text{goto } \text{label} | \]
\[ \text{return} | \text{call } u \text{ N} | \text{call } \text{print } 1 | \text{call } \text{input } 0 | \text{call } \text{allocate } 2 | \text{call } \text{tensor-error } F | \]
\[ w++ | w-- | w @ w \text{ w } E \]

An instruction \( i \) that has no successor
Successors of an instruction (4)

\[ i ::= w \leftarrow s \mid w \leftarrow \text{mem} \times M \mid \text{mem} \times M \leftarrow s \mid w \leftarrow \text{stack-arg} \times M \mid w \leftarrow \text{aop} \times t \mid w \leftarrow \text{sop} \times s \mid w \leftarrow \text{sop} \times N \mid \text{mem} \times M \leftarrow t \mid \text{mem} \times M \leftarrow t \mid w \leftarrow \text{mem} \times M \mid w \leftarrow \text{mem} \times M \mid w \leftarrow \text{t cmp} \times t \mid \text{cjump} t \times \text{cmp} \times t \times \text{label} \mid \text{label} \mid \text{goto} \times \text{label} \mid \text{return} \mid \text{call} u \times N \mid \text{call print} \times 1 \mid \text{call input} \times 0 \mid \text{call allocate} \times 2 \mid \text{call tensor-error} \times F \mid w \leftarrow \text{w++} \mid w \leftarrow \text{w--} \mid w \leftarrow \text{w} \times \text{w} \times E \]

An instruction \( i \) that has two successors

\[ \text{cjump rax} < 5 : \text{L1} \]
\[ \text{rdi} \leftarrow 1 \]
\[ \text{rsi} \leftarrow 3 \]
\[ : \text{L1} \]
Now with knowledge about paths and successors we can compute IN and OUT sets of each instruction of a function automatically.
Liveness analysis

A variable is **alive** at a particular point in the program if its value at that point will be used in a path that starts from there (the future).

A variable is **dead** if it is not alive.

- To compute liveness at a given point, we need to look into the future
- A variable $v$ is alive at a given point of a program $p$ if
  - Exist a directed path from $p$ to an use of $v$ and
  - that path does not contain any definition of $v$

```plaintext
%v1 <- 5
%v3 <- 3
%v2 <- %v1
return
```

```plaintext
%v1 <- 5
%v1 <- 3
%v2 <- %v1
return
```
Liveness analysis algorithm

1. Identify which variables are define and which ones are read (used) by an instruction
   • GEN and KILL sets (local information)

2. Specify how instructions transmit live values around the program
   • How to compute IN and OUT sets from GEN and KILL sets (global information)

3. Iterate (2) until nothing (i.e., IN and OUT set) changes
   • Notice that (1) is performed only once!
   • GEN and KILL sets are constants and, therefore, path independent!
GEN and KILL sets

• GEN[i] = {all variables read (used) by instruction i}
  \%myVar3 <- \%myVar1  // GEN[i] = {\%myVar1}

• KILL[i] = {all variables defined by instruction i}
  \%myVar3 <- \%myVar1  // KILL[i] = {\%myVar3}

\%myVar3 += \%myVar1
KILL[i] = {\%myVar3}  GEN[i] = {\%myVar1, \%myVar3}
GEN and KILL sets: more examples

- GEN[i] = \{all variables read (used) by instruction i\}
- KILL[i] = \{all variables defined by instruction i\}

rdi++
KILL[i] = \{rdi\}
GEN[i] = \{rdi\}
GEN and KILL sets: more examples

- GEN[i] = {all variables read (used) by instruction i}
- KILL[i] = {all variables defined by instruction i}

```c
 cjump rdi <= %v2 :true
KILL[i] = {}
GEN[i] = {rdi, %v2}
```
Liveness analysis algorithm

1. Define which variables are define and which ones are read (used) for each instruction
   • GEN and KILL sets

2. Specify how instructions transmit live values around the program
   • How to compute IN and OUT sets from GEN and KILL sets

3. Iterate (2) until nothing changes
IN and OUT sets

• IN[i] = \{all variables live right before instruction i\}
  \[ IN[i] = GEN[i] \cup (OUT[i] – KILL[i]) \]

• OUT[i] = \{all variables live right after instruction i\}
  \[ OUT[i] = \bigcup_{s \text{ a successor of } i} IN[s] \]
Algorithm

for (each instruction $i$) {
    GEN[$i$] = ...
    KILL[$i$] = ...
}

for (each instruction $i$)  IN[$i$] = OUT[$i$] = { };
do{
    for (each instruction $i$) {
        IN[$i$] = GEN[$i$] U (OUT[$i$] – KILL[$i$])
        OUT[$i$] = \bigcup_{s \text{ a successor of } i} \text{IN}[s]
    }
} while (changes to any IN or OUT occur);
Outline

• Introduction to register allocation

• Liveness analysis

• Calling convention
The reason why call and return instructions must be treated with the above special rules will be explained at the next class.
Code example

(:myF
  0
  %a <- 2  // 1
  rax <- %a // 2
  return  // 3
)
<table>
<thead>
<tr>
<th></th>
<th>GEN</th>
<th>KILL</th>
</tr>
</thead>
<tbody>
<tr>
<td>call u N</td>
<td>{ u,</td>
<td>{ caller save registers }</td>
</tr>
<tr>
<td></td>
<td>args used }</td>
<td></td>
</tr>
<tr>
<td>call RUNTIME N</td>
<td>{ args used }</td>
<td>{ caller save registers }</td>
</tr>
<tr>
<td>return</td>
<td>{ rax,</td>
<td>{ }</td>
</tr>
<tr>
<td></td>
<td>callee save registers }</td>
<td></td>
</tr>
</tbody>
</table>
Registers

Arguments
- rdi
- rsi
- rdx
- rcx
- r8
- r9

Result
- rax

Caller save
- r10
- r11
- r8
- r9
- rax
- rcx
- rdi
- rdx
- rsi

Callee save
- r12
- r13
- r14
- r15
- rbp
- rbx
# Registers

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Result</th>
<th>Caller save</th>
<th>Callee save</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdi</td>
<td>rax</td>
<td>r10</td>
<td>r12</td>
</tr>
<tr>
<td>rsi</td>
<td></td>
<td>r11</td>
<td></td>
</tr>
<tr>
<td>rdx</td>
<td></td>
<td>r8</td>
<td></td>
</tr>
<tr>
<td>rcx</td>
<td></td>
<td>r9</td>
<td></td>
</tr>
<tr>
<td>r8</td>
<td></td>
<td>rax</td>
<td></td>
</tr>
<tr>
<td>r9</td>
<td></td>
<td>rcx</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>rdi</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>rdx</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>rdx</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>rsi</td>
<td></td>
</tr>
</tbody>
</table>

Let’s assume we only have 1 callee save register (for keeping the example as simple as possible)
### Code example

```c
(:myF
  0
  %a <- 2  // 1
  rax <- %a // 2
  return  // 3
)
```

<table>
<thead>
<tr>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>%{a}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%{a}</td>
<td>{}</td>
<td>%{a}</td>
<td></td>
</tr>
<tr>
<td>{rax, r12}</td>
<td>{}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**GEN[i]** = {all variables read (used) by instruction i}

**KILL[i]** = {all variables defined by instruction i}
Algorithm

for (each instruction $i$) {
    $\text{GEN}[i] = \ldots$
    $\text{KILL}[i] = \ldots$
}

for (each instruction $i$)  $\text{IN}[i] = \text{OUT}[i] = \{ \}$;
do{
    for (each instruction $i$) {
        $\text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i])$
        $\text{OUT}[i] = \bigcup_{s \text{ a successor of } i} \text{IN}[s]$
    }
} while (changes to any $\text{IN}$ or $\text{OUT}$ occur);
## Code example

```plaintext
(:myF
  0
  %a <- 2    // 1
  rax <- %a  // 2
  return    // 3
)
```

<table>
<thead>
<tr>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>%a</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>%a</td>
<td>rax</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>rax, r12</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
</tr>
</tbody>
</table>
Algorithm

for (each instruction $i$) {
    \text{GEN}[i] = ...
    \text{KILL}[i] = ...
}

for (each instruction $i$) \text{IN}[i] = \text{OUT}[i] = \{ \};

do{
    for (each instruction $i$){
        \text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i])
        \text{OUT}[i] = \bigcup_{s \text{ a successor of } i} \text{IN}[s]
    }
} while (changes to any \text{IN} or \text{OUT} occur);
Code example

(:myF

0

%a <- 2     // 1
rax <- %a    // 2
return       // 3
)

\[
\text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i])
\]

\[
\text{OUT}[i] = \bigcup_{s \text{ a successor of } i} \text{IN}[s]
\]
Code example

(:myF
  0
  %a <- 2   // 1
  rax <- %a // 2
  return    // 3
)

<table>
<thead>
<tr>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>{%a}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>{%a}</td>
<td>{rax}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>{rax, r12}</td>
<td>{}</td>
<td>{rax, r12}</td>
<td>{}</td>
</tr>
</tbody>
</table>

\[
\text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i]) \\
\text{OUT}[i] = \bigcup_{s \text{ a successor of } i} \text{IN}[s]
\]
(myF
0
%a <- 2     // 1
rax <- %a   // 2
return      // 3
)

IN[i] = GEN[i] ∪ (OUT[i] − KILL[i])
OUT[i] = ∪_{s \text{ a successor of } i} IN[s]
## Code example

```plaintext
(:myF
  0
  %a <- 2  // 1
  rax <- %a  // 2
  return  // 3
)
```

<table>
<thead>
<tr>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>{ %a}</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>{ %a}</td>
<td>{ rax}</td>
<td>{ %a, r12}</td>
<td>{ rax, r12}</td>
</tr>
<tr>
<td>{ rax, r12}</td>
<td>{ }</td>
<td>{ rax, r12}</td>
<td>{ }</td>
</tr>
</tbody>
</table>

\[ \text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i]) \]
\[ \text{OUT}[i] = \bigcup_{s \text{ a successor of } i} \text{IN}[s] \]
Code example

(:myF
  0
  %a <- 2  // 1
  rax <- %a  // 2
  return  // 3
)

\[
\begin{align*}
\text{IN}[i] &= \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i]) \\
\text{OUT}[i] &= \bigcup_{s \text{ a successor of } i} \text{IN}[s]
\end{align*}
\]
Code example

\[
\text{ (:myF 0)
%a <- 2 \quad // 1
rax <- %a \quad // 2
return \quad // 3
)}
\]

\[
\begin{array}{|c|c|c|c|}
\hline
\text{GEN} & \text{KILL} & \text{IN} & \text{OUT} \\
\hline
\{ \} & \{%a\} & \{r12\} & \{%a, r12\} \\
\{%a\} & \{rax\} & \{%a, r12\} & \{rax, r12\} \\
\{rax, r12\} & \{\} & \{rax, r12\} & \{\} \\
\hline
\end{array}
\]

\[
\text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i])
\]

\[
\text{OUT}[i] = \bigcup_{s \text{ a successor of } i} \text{IN}[s]
\]
Algorithm

for (each instruction $i$) {
    GEN[$i$] = ...
    KILL[$i$] = ...
}

for (each instruction $i$) IN[$i$] = OUT[$i$] = { };
do{
    for (each instruction $i$){
        IN[$i$] = GEN[$i$] U (OUT[$i$] – KILL[$i$])
        OUT[$i$] = $U_s$ a successor of $i$ IN[$s$]
    }
} while (changes to any IN or OUT occur);
### Code example

```plaintext
(:myF
  0
  %a <- 2   // 1
  rax <- %a  // 2
  return    // 3
)
```

<table>
<thead>
<tr>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>{%a}</td>
<td>{r12}</td>
<td>{%a, r12}</td>
</tr>
<tr>
<td>{%a}</td>
<td>{rax}</td>
<td>{%a, r12}</td>
<td>{rax, r12}</td>
</tr>
<tr>
<td>{rax, r12}</td>
<td>{}</td>
<td>{rax, r12}</td>
<td>{}</td>
</tr>
</tbody>
</table>

\[
\text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i])
\]
\[
\text{OUT}[i] = \bigcup_{s \text{ a successor of } i} \text{IN}[s]
\]
Algorithm

for (each instruction $i$) {
    \text{GEN}[i] = ... \\
    \text{KILL}[i] = ... \\
}
for (each instruction $i$) \text{IN}[i] = \text{OUT}[i] = \{ \};
do{
    for (each instruction $i$){
        \text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] – \text{KILL}[i]) \\
        \text{OUT}[i] = \bigcup_s \text{a successor of } i \text{ IN}[s] \\
    }
} while (changes to any IN or OUT occur);
### Code example

```plaintext
(:myF
  0
  %a <- 2  // 1
     { }     {%a}    {r12}   {%a, r12}
  rax <- %a // 2
     {%a}    {rax}    {%a, r12} {rax, r12}
  return   // 3
     {rax, r12}  { }    {rax, r12}  { }
)
```

- Variables within the same set are alive at the same time at that point in the code.
- Hence, they cannot be placed in the same register.
Homework #1

• Compute the IN and OUT sets of all instructions of an L2 function given as input

(:myF
0
%myVar1 <- 5
%myVar2 <- 0
%myVar2 += %myVar1
return
)

Your work

IN[0] → (r13 r15 rax r14 rbp r12 rbx )
IN[1] → (r13 r15 rax r14 rbp r12 %myVar1 rbx )
IN[2] → (r13 r15 rax r14 rbp %myVar2 r12 %myVar1 rbx )
IN[3] → (r13 r15 rax r14 rbp r12 rbx )
)
(out
OUT[0] → (r13 r15 rax r14 rbp r12 %myVar1 rbx )
OUT[1] → (r13 r15 rax r14 rbp %myVar2 r12 %myVar1 rbx )
OUT[2] → (r13 r15 rax r14 rbp r12 rbx )
OUT[3] → ( )
))
Testing your homework #1

- Under L2/tests/liveness there are the tests you have to pass
- A new compiler argument: `-l`
  - Check `L2compiler.cpp` on Canvas
- To test:
  - To check all tests: `make test_liveness`
  - To check one test: `./liveness test/liveness/test1.L2f`
- Check out each input/output for each test if you have doubts
  - For example, the correct output for the test `test/liveness/test1.L2f` is `test/liveness/test1.L2f.out`
Debugging suggestion

• Don’t forget you have our L2 compiler binary

• So, to help you debug your work:
  • you can write your own test (a new MyTest.L2f)
  • Generate the output of our L2 compiler by invoking
    ```
    ./liveness MyTest.L2f > MyTest.L2f.out
    ```
  • Compare our output with the output generated by your L2 compiler