Liveness analysis

Simone Campanoni
simonec@eecs.northwestern.edu
A graph-coloring register allocator structure

```
spill(f, var, prefix)
```

**Register allocator**
- Code analysis
- Graph coloring

**Spill**

`f` without variables and with registers
Outline

• Code analyses required

• Liveness analysis

• Calling convention
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 \( f \):

\[
\begin{align*}
&\text{:myF 0 0} \\
&0: \text{myVar1} <- 2 \\
&1: \text{myVar2} <- 40 \\
&2: \text{myVar3} <- \text{myVar1} \\
&3: \text{myVar3} += \text{myVar2} \\
&4: \text{rax} <- \text{myVar3} \\
\end{align*}
\]

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

\[
\begin{align*}
\text{IN[0]} &= \{ \} \\
\text{OUT[0], IN[1]} &= \{ \text{myVar1} \} \\
\text{OUT[1], IN[2]} &= \{ \text{myVar1, myVar2} \} \\
\text{OUT[2], IN[3]} &= \{ \text{myVar3, myVar2} \} \\
\text{OUT[3], IN[4]} &= \{ \text{myVar3} \}
\end{align*}
\]

**Live ranges**

**Interference graph**

Graph coloring

:w/o variables
Steps

1. Compute IN and OUT sets

2. Compute interference graph from IN and OUT sets
Generating the interference graph

- 1 node per variable or register
- Connect each pair of variables that belong to the same IN or OUT set
- Connect a register to all other registers
- ... (see Interference graph slides for other steps)

**Meaning of an edge:** 2 connected nodes must use different registers

\[
\begin{align*}
\{\} & \quad \text{r11} \\
\{\text{myVar1}\} & \quad \text{r10} \\
\{\text{myVar1, myVar2}\} & \\
\{\text{myVar3, myVar2}\} & \\
\{\text{myVar3}\} & \\
\text{myVar1} & \quad \text{myVar2} \\
\text{myVar3} & \\
\text{Graph coloring} & \quad \text{:myF w/o variables that satisfies the interferences}
\end{align*}
\]
Outline

• Code analyses required

• Liveness analysis

• Calling convention
Execution path

 (:myF 0 0
  rdi <- 5
  call print 1
  return
 )

 (:myF2 1 0
  cjump rdi < 4 :true :false
  :false
  :false
  rdi <- 5
  call print 1
  return
  :true
  rdi <- 7
  call print 1
  return
 )

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
Liveness analysis

A variable is **live** at a particular point in the program if its value at that point will be used in the future (dead, otherwise)

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

\[
\begin{align*}
v_1 & \leftarrow 5 \\
v_3 & \leftarrow 3 \\
v_2 & \leftarrow v_1 \\
\text{return} \\
v_1 & \leftarrow 5 \\
v_1 & \leftarrow 3 \\
v_2 & \leftarrow v_1 \\
\text{return}
\end{align*}
\]
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 (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

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

```
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\}

cjump rdi <= v2 :true :false
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

- \( \text{IN}[i] = \{ \text{all variables live right before instruction } i \} \)
  \[
  \text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i])
  \]
  \[
  \begin{align*}
  &\text{If } \text{OUT}[i] = \{} \text{ then } \\
  &\quad \text{IN}[i] = \{v1\}
  \end{align*}
  \]

- \( \text{OUT}[i] = \{ \text{all variables live right after instruction } i \} \)
  \[\text{OUT}[i] = \bigcup_{s \text{ a successor of } i} \text{IN}[s]\]
  \[
  \begin{align*}
  &\text{If } \text{OUT}[i] = \{} \text{ then } \\
  &\quad \text{IN}[i] = \{v1\}
  \end{align*}
  \]

- If \( \text{OUT}[i] = \{v2\} \) then
  \[\text{IN}[i] = \{v1\}\]

- If \( \text{IN}[i+1] = \{v1\}, \text{ then } \text{IN}[i+j] = \{v2\}, \text{ then } \text{OUT}[i] = \{v1,v2\}\]

- \( i : \text{v2} \leftarrow \text{v1}\)

- If \( \text{OUT}[i] = \{} \text{ then } \text{IN}[i] = \{v1\}\)

- \( i : \text{cjump v = 1 :s1 :s2} \)

- \( i+1: \text{v2} \leftarrow \text{v1} \)

- If \( \text{IN}[i+1] = \{v1\} \) then

- \( i+j: \text{v3} \leftarrow 5 \)

- \( i+1: \text{v2} \leftarrow \text{v1} \)

- \( \text{OUT}[i] = \{v1\} \)
Algorithm

for (each instruction $i$) {
    GEN[$i$] = ...
    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);
Outline

• Code analyses required

• Liveness analysis

• Calling convention
Code example

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

• Are GEN and KILL sets correct?
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 0
  a < - 2  // 1
  rax < - a  // 2
  return  // 3
)

• Are GEN and KILL sets correct?
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 $i$){
        $\text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i])$
        $\text{OUT}[i] = \bigcup_s$ a successor of $i$ $\text{IN}[s]$
    }
} while (changes to any $\text{IN}$ or $\text{OUT}$ occur);
### Code example

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

<table>
<thead>
<tr>
<th></th>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{}</td>
<td>{a}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>2</td>
<td>{a}</td>
<td>{rax}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>3</td>
<td>{rax}</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
</tr>
</tbody>
</table>

- Are GEN and KILL sets correct?

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

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

• Are GEN and KILL sets correct?

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

```plaintext
(:myF
  0 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}</td>
<td>{}</td>
<td>{rax}</td>
<td>{}</td>
</tr>
</tbody>
</table>

- Are GEN and KILL sets correct?

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

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

- Are GEN and KILL sets correct?

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

```plaintext
(:myF
  0 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}</td>
<td>{rax}</td>
</tr>
<tr>
<td>{rax}</td>
<td>{}</td>
<td>{rax}</td>
<td>{}</td>
</tr>
</tbody>
</table>

- Are GEN and KILL sets correct?

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

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

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

<table>
<thead>
<tr>
<th></th>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN[i]</td>
<td>GEN[i] U (OUT[i] – KILL[i])</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUT[i]</td>
<td>U_{s a successor of i} IN[s]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 IN or OUT occur);
## Code example

```plaintext
(:myF
  0 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>{a}</td>
</tr>
<tr>
<td>{a}</td>
<td>{rax}</td>
<td>{a}</td>
<td>{rax}</td>
</tr>
</tbody>
</table>

- Are GEN and KILL sets correct?

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

(:myF
  0 0
a <- 2    // 1  { }    {a}    { }    {a}
  rax <- a  // 2 {a}    {rax}   {a}    {rax}
  return    // 3  {rax}  { }    {rax}  { }
)
Steps

1. Compute IN and OUT sets

2. Compute interference graph from IN and OUT sets
Generating the interference graph

• 1 node per variable or register

• Connect each pair of variables that belong to the same IN or OUT set
• Connect a register to all other registers
• ... (see Interference graph slides for other steps)

• Meaning of an edge: 2 connected nodes must use different registers
Code example

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

- Graph coloring can assign r12 to a
Code example

(:myF
  0 0
  r12 <- 2  // 1
  rax <- r12  // 2
  return  // 3
)

• Are GEN and KILL sets correct?
• Graph coloring can assign r12 to a
• Is there any problem?
# 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>r13</td>
</tr>
<tr>
<td>rdx</td>
<td></td>
<td>r8</td>
<td>r14</td>
</tr>
<tr>
<td>rcx</td>
<td></td>
<td>r9</td>
<td>r15</td>
</tr>
<tr>
<td>r8</td>
<td></td>
<td>rax</td>
<td>rbp</td>
</tr>
<tr>
<td>r9</td>
<td></td>
<td>rcx</td>
<td>rbx</td>
</tr>
</tbody>
</table>
Code example

(:myF
  0 0
  r12 <- 2    // 1
  rax <- r12  // 2
  return      // 3
)

• The calling convention counts as definitions and uses
• When adding them as such, we automatically enforce the calling convention

• Are GEN and KILL sets correct?
• Graph coloring can assign r12 to a
• Is there any problem?
## Calling convention in GEN/KILL

<table>
<thead>
<tr>
<th></th>
<th>GEN</th>
<th>KILL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>call u N</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>call RUNTIME N</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>return</strong></td>
<td>{ rax, callee save registers}</td>
<td>{ }</td>
</tr>
</tbody>
</table>
Return instruction in a 2 registers CPU

\[ \text{:myF 0 0} \]
\[ \text{a <- 2} \]
\[ \text{return} \]

**w/o calling convention**

\[ \text{a \rightarrow r12} \]
\[ \text{r10} \]

**Graph coloring**

\[ \text{:myF 0 0} \]
\[ \text{(r12 <- 2)} \]
\[ \text{(return)} \]

---

**Callee-save:**

\[ \text{r12} \]

**Caller-save:**

\[ \text{r10} \]
Calling convention in GEN/KILL

<table>
<thead>
<tr>
<th></th>
<th>GEN</th>
<th>KILL</th>
</tr>
</thead>
<tbody>
<tr>
<td>call u N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>call RUNTIME N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return</td>
<td>{ rax, callee save registers}</td>
<td>{ }</td>
</tr>
</tbody>
</table>
Call instructions

• Which register should we use for a?
  r10

• Is it correct? (r10 is a caller save register)

(:myF 0 0
 a <- 2
 call :f 0
 a *= a
 rax <- a
 return
)

(:myF 0 0
 r10 <- 2
 call :f 0
 r10 *= r10
 rax <- r10
 return
)
## Calling convention in GEN/KILL

<table>
<thead>
<tr>
<th></th>
<th>GEN</th>
<th>KILL</th>
</tr>
</thead>
<tbody>
<tr>
<td>call u N</td>
<td>{ u, args }</td>
<td>{ caller save registers, rax }</td>
</tr>
<tr>
<td>call RUNTIME N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return</td>
<td>{ rax, callee save registers}</td>
<td>{ }</td>
</tr>
</tbody>
</table>
### Calling convention in GEN/KILL

<table>
<thead>
<tr>
<th></th>
<th>GEN</th>
<th>KILL</th>
</tr>
</thead>
<tbody>
<tr>
<td>call u N</td>
<td>{ u, args }</td>
<td>{ caller save registers, rax }</td>
</tr>
<tr>
<td>call RUNTIME N</td>
<td>{ args}</td>
<td>{ caller save registers, rax }</td>
</tr>
<tr>
<td>return</td>
<td>{ rax, callee save registers}</td>
<td>{ }</td>
</tr>
</tbody>
</table>
Homework #1

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

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

Your work

( (in
  (r13 r15 rax r14 rbp r12 rbx  )
  (r13 r15 rax r14 rbp r12 myVar1 rbx  )
  (r13 r15 rax r14 rbp myVar2 r12 myVar1 rbx  )
  (r13 r15 rax r14 rbp r12 rbx  )
) (out
  (r13 r15 rax r14 rbp r12 myVar1 rbx  )
  (r13 r15 rax r14 rbp myVar2 r12 myVar1 rbx  )
  (r13 r15 rax r14 rbp r12 rbx  )
  (  )
)))
Testing your homework #1

• Under L2/tests/liveness there are the tests you have to pass

• Two new compiler arguments: -l -g
  • Check L2/src/compiler.cpp

• To test: make test_liveness

• Check out each input/output for each test if you have doubts
  • test/liveness/test1.L2f
  • test/liveness/test1.L2f.out