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

f
A graph-coloring register allocator structure

Register allocator
- Code analysis
- Graph coloring
- Spill
  - spill(f, var, prefix)

f with var spilled

f without variables and with registers
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 \( f \).

```plaintext
(:myF 0
  (%myVar1 <- 2)
  (%myVar2 <- 40)
  (%myVar3 <- %myVar1)
  (%myVar3 += %myVar2)
  rax <- %myVar3
  return )
```

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

- IN[0] = \{\}\n- OUT[0] = IN[0]
- IN[1] = \{myVar1\}
- OUT[1] = \{myVar1\}
- IN[2] = \{myVar1, myVar2\}
- OUT[2] = \{myVar1, myVar2\}
- IN[3] = \{myVar3\}
- OUT[3] = \{myVar3\}
- IN[4] = \{myVar3, myVar2\}
- OUT[4] = \{myVar3, myVar2\}

Live ranges:

- myVar1
- myVar2
- myVar3

Interference graph:

- myVar1 → myVar2
- myVar3

Graph coloring:

- :myF 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
• Connect each pair of variables that belong to the same IN or OUT set
• ... (see Interference graph slides for other steps)

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

```c
{ myF 0
  %myVar1 <- 2
  %myVar2 <- 40
  %myVar3 <- %myVar1
  %myVar3 += %myVar2
  rax <- %myVar3
  return
}

{ }
{myVar1}
{myVar1, myVar2}
{myVar3, myVar2}
{myVar3}
```

```c
myVar1 -- myVar2
myVar3
```

Graph coloring:

```c
:myF w/o variables that satisfies the interferences
```

```c
myVar1, myVar3 => r10
myVar2 => r11
```

```c
(:myF 0 0
  r10 <- 2
  r11 <- 40
  r10 <- r10
  r10 += r11
  rax <- r10
  return
)```
Outline

• Introduction to register allocation

• Liveness analysis

• Calling convention
Variables in the liveness analysis

• 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}`
Let $i$ be an instruction, we need to identify the set of variables with values that will be used just before and just after $i$. It is a predecessor of $i$. It is a successor of $i$. 
Successors of an instruction

\[ i ::= \begin{array}{c}
    w \leftarrow s \\
    w \leftarrow m x M \\
    m x M \leftarrow s \\
    w \leftarrow \text{stack-arg } M \\
    w \ aop \ t \\
    w \ sop \ sx \\
    w \ sop \ N \\
    m x M += t \\
    m x M -= t \\
    w += m x M \\
    w -= m x M \\
    w \leftarrow t \ cmp \ t \\
    \text{cj} \text{jump } t \ cmp \ t \ \text{label} \\
    \text{label} \\
    \text{goto } \text{label} \\
    \text{return} \\
    \text{call } u \ N \\
    \text{call print } 1 \\
    \text{call allocate } 2 \\
    \text{call array-error } 2 \\
    w++ \\
    w-- \\
    w @ w w E
\end{array} \]

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

\[
    \begin{align*}
        \text{rdi} & \leftarrow 5 \\
        \text{call print } 1 \\
        \text{r10} & \leftarrow \text{rax} < 5
    \end{align*}
\]
Successors of an instruction (2)

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

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

\[
\begin{align*}
goto &: \text{:MY_LABEL_0} \\
goto &: \text{:MY_LABEL_0} \\
goto &: \text{:MY_LABEL_0} \\
\end{align*}
\]

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

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

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

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

An instruction \( i \) that has two successors

\[
\text{cjump rax < 5 :L1} \\
\text{rdi} \leftarrow 1 \\
\text{rsi} \leftarrow 3 \\
: \text{L1}
\]
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 a use of $v$ and
  - that path does not contain any definition of $v$

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

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

- \( \text{KILL}[i] = \{\text{all variables defined by instruction } i\} \)
  
  \[
  \%\text{myVar3} \leftarrow \%\text{myVar1} \quad // \text{KILL}[i] = \{\text{myVar3}\}
  \]

\[
\%\text{myVar3} += \%\text{myVar1} \\
\text{KILL}[i] = \{\text{myVar3}\} \quad \text{GEN}[i] = \{\text{myVar1, myVar3}\}
\]

\% is dropped
GEN and KILL sets: more examples

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

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

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

cjump rdi <= %v2 :true

\( \text{KILL}[i] = \{ \} \)

\( \text{GEN}[i] = \{ \text{rdi, v2} \} \)
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

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

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

\[
\begin{align*}
\text{i: } \%v2 &< \%v1 \\
\text{If OUT}[i] = \emptyset \text{ then } \text{IN}[i] = \{\%v1\}
\end{align*}
\]

\[
\begin{align*}
\text{i: } \%v2 &< \%v1 \\
\text{If OUT}[i] = \{\%v2\} \text{ then } \text{IN}[i] = \{\%v1\}
\end{align*}
\]

\[
\begin{align*}
\text{i : } \text{cjump } \%v = 1 :s2 \\
i+1: & :s1 \\
i+j: & :s2 \\
\text{If } \text{IN}[i+1] = \{\%v1\}, \text{ then } \text{IN}[i+j] = \{\%v2\}, \\
\text{Then OUT}[i] = \{\%v1, \%v2\}
\end{align*}
\]
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[$i$] $\cup$ (OUT[$i$] – KILL[$i$])
        OUT[$i$] = $\bigcup$ a successor of $i$ IN[$s$]
    }
} while (changes to any IN or OUT occur);
Outline

• Introduction to register allocation

• Liveness analysis

• Calling convention
Code example

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

• Are GEN and KILL sets correct?
Algorithm

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

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
%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></td>
<td></td>
<td>{a}</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<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?
Algorithm

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

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

do{
    for (each instruction i){
        IN[i] = GEN[i] \cup (OUT[i] - 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

%a <- 2  // 1

rax <- %a  // 2

return  // 3

)

• Are GEN and KILL sets correct?

<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>{}</td>
<td>{}</td>
</tr>
</tbody>
</table>

IN\[i\] = GEN\[i\] \(\cup\) (OUT\[i\] – KILL\[i\])

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

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

• Are GEN and KILL sets correct?

<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>

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

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

- Are GEN and KILL sets correct?

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

• Are GEN and KILL sets correct?

\[
\begin{align*}
\text{GEN} & : 
\begin{array}{cccc}
\{ \} & \{a\} & \{\}\phantom{1} & \{\}\phantom{1} \\
\{a\} & \{\text{rax}\} & \{a\} & \{\text{rax}\} \\
\{\text{rax}\} & \{\} & \{\text{rax}\} & \{\}
\end{array} \\
\end{align*}
\]

\[
\begin{align*}
\text{KILL} & : 
\begin{array}{cccc}
\{ \} & \{a\} & \{\}\phantom{1} & \{\}\phantom{1} \\
\{a\} & \{\text{rax}\} & \{a\} & \{\text{rax}\} \\
\{\text{rax}\} & \{\} & \{\text{rax}\} & \{\}
\end{array} \\
\end{align*}
\]

\[
\begin{align*}
\text{IN} & : 
\begin{array}{cccc}
\{ \} & \{a\} & \{\}\phantom{1} & \{\}\phantom{1} \\
\{a\} & \{\text{rax}\} & \{a\} & \{\text{rax}\} \\
\{\text{rax}\} & \{\} & \{\text{rax}\} & \{\}
\end{array} \\
\end{align*}
\]

\[
\begin{align*}
\text{OUT} & : 
\begin{array}{cccc}
\{ \} & \{a\} & \{\}\phantom{1} & \{\}\phantom{1} \\
\{a\} & \{\text{rax}\} & \{a\} & \{\text{rax}\} \\
\{\text{rax}\} & \{\} & \{\text{rax}\} & \{\}
\end{array} \\
\end{align*}
\]

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

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

• Are GEN and KILL sets correct?

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

\[
(:\text{myF}
\begin{align*}
0 \\
\%a &\Leftarrow 2 \quad // 1 \\
rax &\Leftarrow \%a \quad // 2 \\
\text{return} &\quad // 3
\end{align*}
\)

• Are GEN and KILL sets correct?

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

\[
\begin{align*}
\text{IN}[i] &= \text{GEN}[i] \cup (\text{OUT}[i] \setminus \text{KILL}[i]) \\
\text{OUT}[i] &= \bigcup_{s \text{ a successor of } i} \text{IN}[s]
\end{align*}
\]
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[$i$] $\cup$ (OUT[$i$] $-$ KILL[$i$])
        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
  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>
<tr>
<td>{rax}</td>
<td>{ }</td>
<td>{rax}</td>
<td>{ }</td>
</tr>
</tbody>
</table>

• Are GEN and KILL sets correct?

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

- Graph coloring can assign r12 to a
Code example

(:myF
  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>rdi</td>
<td>r11</td>
<td>r13</td>
</tr>
<tr>
<td>rdx</td>
<td>rdx</td>
<td>r8</td>
<td>r14</td>
</tr>
<tr>
<td>rcx</td>
<td>rdx</td>
<td>r9</td>
<td>r15</td>
</tr>
<tr>
<td>r8</td>
<td>r8</td>
<td>rax</td>
<td>rbp</td>
</tr>
<tr>
<td>r9</td>
<td>r9</td>
<td>rcx</td>
<td>rbx</td>
</tr>
</tbody>
</table>
Code example

(:myF
  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>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>
Return instruction in a 2 registers CPU

(:myF 0
 %a <- 2
 return
 )

Callee-save: r12

Caller-save: r10

w/o calling convention

Graph coloring

(:myF 0
 r12 <- 2
 return
 )

w/ calling convention

Graph coloring

(:myF 0
 r10 <- 2
 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></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
  %a <- 2
  call :f 0
  %a *= %a
  rax <- %a
  return
)

(:myF 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>
</table>
| call u N | \{ u, 
|          | args used}                            | \{ caller save registers\}               |
| call RUNTIME N |                                  |                                           |
| return   | \{ rax, 
|          | callee save registers\}               | \{ \}                                   |
## 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 used}</td>
<td>{ caller save registers}</td>
</tr>
<tr>
<td>call RUNTIME N</td>
<td>{ args used}</td>
<td>{ caller save registers}</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
  %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[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
• Two new compiler arguments: -l –g
  • 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
  • test/liveness/test1.L2f
  • test/liveness/test1.L2f.out