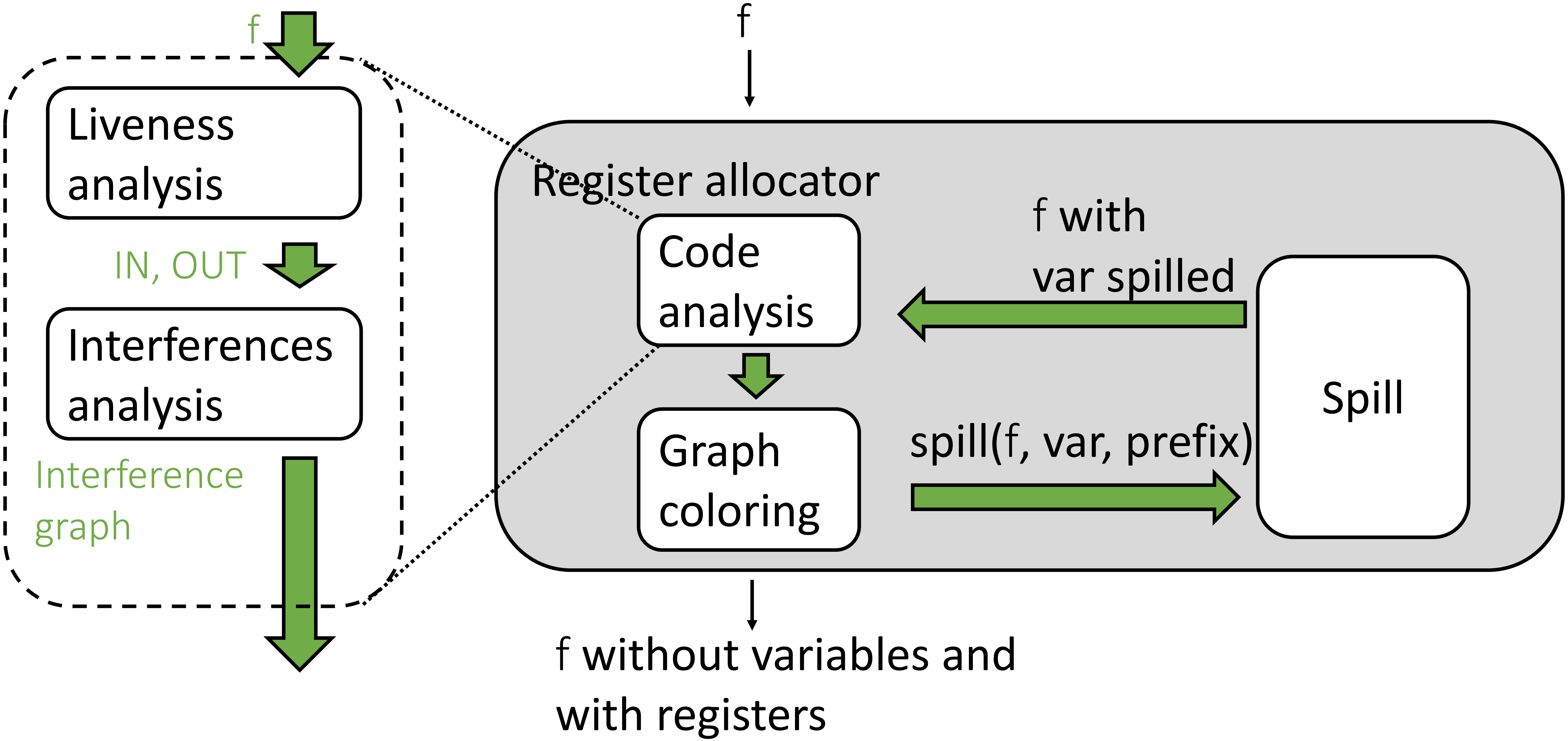
Compiler Construction

Interference graph

Simone Campanoni
simone.campanoni@northwestern.edu

A graph-coloring register allocator structure



Outline

- What is the interference graph
- Algorithm to build the interference graph
- Calling convention

The interference graph

- The Graph coloring algorithm assigns variables to registers

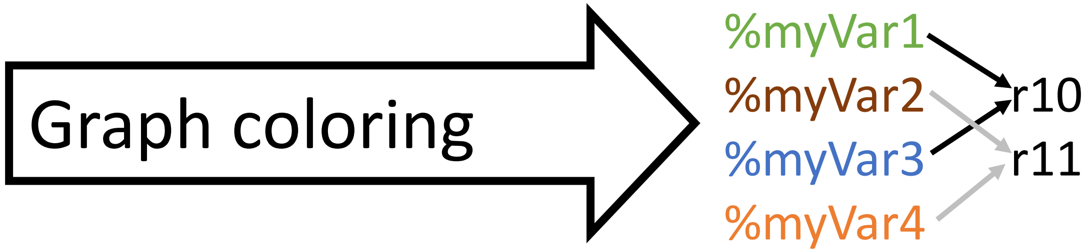
  %myVar1 <- 5 ⇨ r10 <- 5

- This transformation must preserve:
  - The original code semantics
  - The constraints of the target architecture *(e.g., the second operand of the shift operation must be a constant or rcx)*
- These constraints are encoded in the interference graph
- Nodes: variables
- Edges: interferences
- **Meaning of an edge: 2 connected nodes must use different registers**

- Next we are going to learn the algorithm that automatically compute the interference graph
- The algorithm adds edges for different categories of constraints, one category at a time
- We will motivate each category of constraints by showing when the algorithm is incorrect if such category is not considered

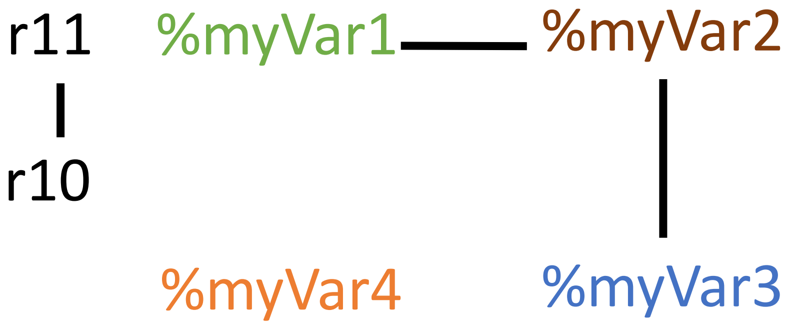
Generating the interference graph

- 1 node per variable
- GP registers are considered variables
- Connect each pair of variables that belong to the same IN or OUT set
- Connect a GP register to all other registers (even those not used by f)
- And …



Is this correct?

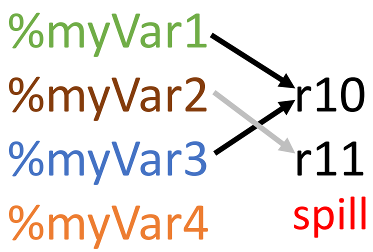
```
(@myF 0
------------------- { }
 %myVar1 <- 2
------------------- {%myVar1}
 %myVar2 <- 40
------------------- {%myVar1, %myVar2}
 %myVar3 <- %myVar1
------------------- {%myVar1, %myVar2}
 %myVar4 <- 42
------------------- {%myVar3, %myVar2}
 %myVar3 += %myVar2
------------------- {%myVar3}
 print %myVar3
)
```



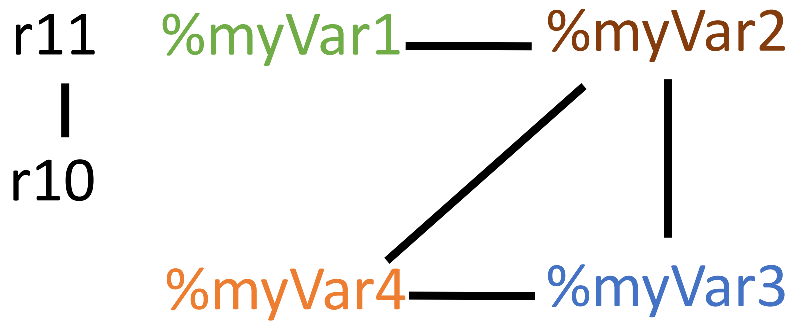

```
(@myF 0 0
 r10 <- 2
 r11 <- 40
 r10 <- r10
 r11 <- 42
 r10 += r11
 print r10
)
```

Generating the interference graph (2)

- 1 node per variable
- GP registers are considered variables
- Connect each pair of variables that belong to the same IN or OUT set
- Connect a GP register to all other registers (even those not used by f)
- Connect variables in KILL[i] with those in OUT[i]
  - Necessary for dead code that defines a variable




```
(@myF 0
 ------------------------- { }
 %myVar1 <- 2
 ------------------------- {%myVar1}
 %myVar2 <- 40
 ------------------------- {%myVar1, %myVar2}
 %myVar3 <- %myVar1
 ------------------------- {%myVar1, %myVar2}
 %myVar4 <- 42
 ------------------------- {%myVar3, %myVar2}
 %myVar3 += %myVar2
 ------------------------- {%myVar3}
 print %myVar3
)
```



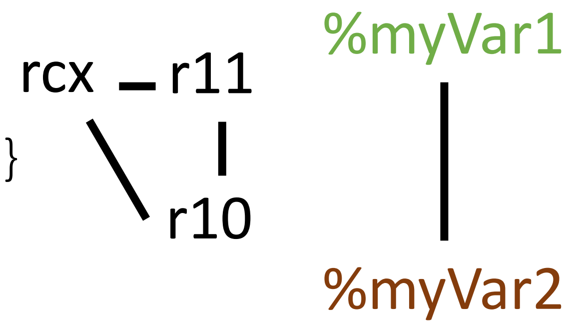

```
(@myF 0 1
  r10 <- 2
  r11 <- 40
  r10 <- r10
  mem rsp 0 <- 42
  r10 += r11
  print r10
)
```

Generating the interference graph (3)

- 1 node per variable
- GP registers are considered variables
- Connect each pair of variables that belong to the same IN or OUT set
- Connect a GP register to all other registers (even those not used by f)
- Connect variables in KILL[i] with those in OUT[i]
  - Necessary for dead code that defines a variable

Is this correct?

```
(@myF 0
------------------------------ { }
 %myVar1 <- 1
------------------------------ {%myVar1}
 %myVar2 <- 2
------------------------------ {%myVar1, %myVar2}
 %myVar2 <<= %myVar1
------------------------------ { }
)
```



%myVar1 → r10
%myVar2 → r11

```
(@myF 0 0
  r10 <- 1
  r11 <- 2
  r11 <<= r10
)
```

Constrains in the target language L1

- The L1 instruction x sop sx is limited to only shifting by the value of rcx (or by a constant)
- This must be encoded in the interference graph
- Add interference edges to disallow the illegal registers when building the interference graph
- For example, consider the following example:

a <<= b

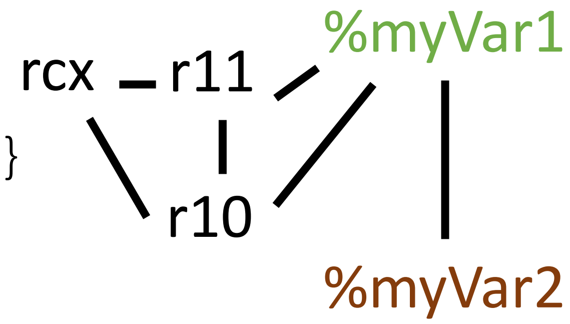
we need to add edges between b and every register except rcx

This ensures b will end up in rcx (or spilled)

Generating the interference graph (3)

- 1 node per variable
- GP registers are considered variables
- Connect each pair of variables that belong to the same IN or OUT set
- Connect a GP register to all other registers (even those not used by f)
- Connect variables in KILL[i] with those in OUT[i]
  - Necessary for dead code that defines a variable
- Handle constrained arithmetic via extra edges

```
(@myF 0
----------------------------- { }
 %myVar1 <- 1
----------------------------- {%myVar1}
 %myVar2 <- 2
----------------------------- {%myVar1, %myVar2}
 %myVar2 <<= %myVar1
----------------------------- { }
)
```



%myVar1 → rcx
%myVar2 → r11

```
(@myF 0
 rcx <- 1
 r11 <- 2
 r11 <<= rcx
)
```

Outline

- What is the interference graph
- Algorithm to build the interference graph
- Calling convention

The relation between Interference graph, calling convention, and liveness analysis

- Finally, we can understand why we had the following rules baked within the Liveness analysis

	GEN	KILL
call u N	{ u, args used}	{ caller save registers}
call RUNTIME N	{ args used}	{ caller save registers}
return	{ rax, callee save registers}	{ }

Let’s assume we don’t treat call and return instructions with special rules.

In other words, let’s assume we don’t embed the calling convention within the Liveness analysis

Code example

	GEN	KILL	IN	OUT
(@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)  {
        GEN[i] = …
        KILL[i] = …
}
for (each instruction i)  IN[i] = OUT[i] = { };
do{
        for (each instruction i){
                IN[i]     = GEN[i] ∪(OUT[i] – KILL[i])
                OUT[i] = ∪_{s a successor of i} IN[s]
        }
} while (changes to any IN or OUT occur);
```

Code example

	GEN	KILL	IN	OUT
(@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)  {
        GEN[i] = …
        KILL[i] = …
}
for (each instruction i)  IN[i] = OUT[i] = { };
do{
        for (each instruction i){
                IN[i]     = GEN[i] ∪(OUT[i] – KILL[i])
                OUT[i] = ∪_{s a successor of i} IN[s]
        }
} while (changes to any IN or OUT occur);
```

Code example

	GEN	KILL	IN	OUT
(@myF				
0				
%a <- 2 // 1	{ }	{%a}	{ }	{ }
rax <- %a // 2	{%a}	{rax}	{ }	{ }
→ return // 3	{rax}	{ }	{ }	{ }
)				

- Are GEN and KILL sets correct?

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

$$OUT[i] = \cup_{s \text{ a successor of } i} IN[s]$$

Code example

	GEN	KILL	IN	OUT
(@myF				
0				
%a <- 2 // 1	{ }	{%a}	{ }	{ }
rax <- %a // 2	{%a}	{rax}	{ }	{ }
→ return // 3	{rax}	{ }	{rax}	{ }
)				

- Are GEN and KILL sets correct?

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

$$OUT[i] = \cup_{s \text{ a successor of } i} IN[s]$$

Code example

	GEN	KILL	IN	OUT
(@myF				
0				
%a <- 2 // 1	{ }	{%a}	{ }	{ }
rax <- %a // 2	{%a}	{rax}	{ }	{ }
return // 3	{rax}	{ }	{rax}	{ }
)				

- Are GEN and KILL sets correct?

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

$$OUT[i] = \cup_{s \text{ a successor of } i} IN[s]$$

Code example

Code	GEN	KILL	IN	OUT
(@myF				
0				
%a <- 2 // 1	{ }	{%a}	{ }	{ }
rax <- %a // 2	{%a}	{rax}	{%a}	{rax}
return // 3	{rax}	{ }	{rax}	{ }
)				

- Are GEN and KILL sets correct?

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

$$OUT[i] = \cup_{s \text{ a successor of } i} IN[s]$$

Code example

	GEN	KILL	IN	OUT
(@myF				
0				
%a <- 2 // 1	{ }	{%a}	{ }	{ }
rax <- %a // 2	{%a}	{rax}	{%a}	{rax}
return // 3	{rax}	{ }	{rax}	{ }
)				

- Are GEN and KILL sets correct?

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

$$OUT[i] = \cup_{s \text{ a successor of } i} IN[s]$$

Code example

	GEN	KILL	IN	OUT
(@myF				
0				
%a <- 2 // 1	{ }	{%a}	{ }	{%a}
rax <- %a // 2	{%a}	{rax}	{%a}	{rax}
return // 3	{rax}	{ }	{rax}	{ }
)				

- Are GEN and KILL sets correct?

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

$$OUT[i] = \cup_{s \text{ a successor of } i} IN[s]$$

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] ∪(OUT[i] – KILL[i])
                OUT[i] = ∪_{s a successor of i} IN[s]
        }
} while (changes to any IN or OUT occur);
```

Code example

	GEN	KILL	IN	OUT
(@myF				
0				
%a <- 2 // 1	{ }	{%a}	{ }	{%a}
rax <- %a // 2	{%a}	{rax}	{%a}	{rax}
→ return // 3	{rax}	{ }	{rax}	{ }
)				

- Are GEN and KILL sets correct?

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

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] = U_{s a successor of i} IN[s]
        }
} while (changes to any IN or OUT occur);
```

Code example

	GEN	KILL	IN	OUT
(@myF				
0				
%a <- 2 // 1	{ }	{%a}	{ }	{%a}
rax <- %a // 2	{%a}	{rax}	{%a}	{rax}
return // 3	{rax}	{ }	{rax}	{ }
)				

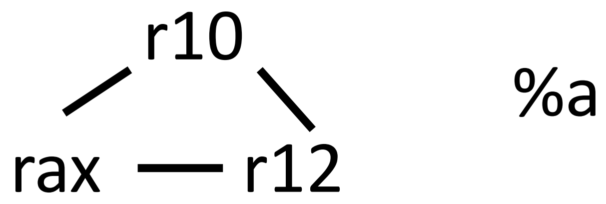
Steps

1. Compute IN and OUT sets

2. Compute interference graph from IN and OUT sets

Code example

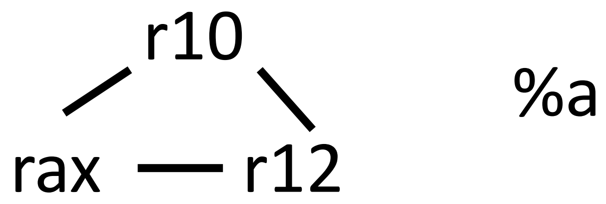
	GEN	KILL	IN	OUT
(@myF				
0				
%a <- 2 // 1	{ }	{%a}	{ }	{%a}
rax <- %a // 2	{%a}	{rax}	{%a}	{rax}
return // 3	{rax}	{ }	{rax}	{ }
)				



- Graph coloring can assign r12 to %a

Code example

```
(@myF
  0
  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?

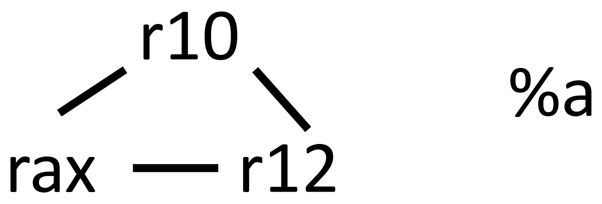
Registers

Arguments	Result	Caller save	Callee save
rdi	rax	r10	r12
rsi		r11	r13
rdx		r8	r14
rcx		r9	r15
r8		rax	rbp
r9		rcx	rbx
		rdi	
		rdx	
		rsi	

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

	GEN	KILL
call u N		
call RUNTIME N		
→ return	{ rax, callee save registers}	{ }

Return instruction in a 2 registers CPU

```
(@myF 0
  %a <- 2
  return
)
```

w/o calling convention →

%a   r12

r10 — r12

Graph coloring →

~~(@myF 0~~
~~r12 <- 2~~
~~return~~
~~)~~

Callee-save:
r12

Caller-save:
r10

w/ calling convention →

%a — r12

r10 — r12

Graph coloring →

```
(@myF 0
  r10 <- 2
  return
)
```

Calling convention in GEN/KILL

	GEN	KILL
call u N		
call RUNTIME N		
return	{ rax, callee save registers}	{ }

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

	GEN	KILL
call u N	{ u, args used}	{ caller save registers}
call RUNTIME N		
return	{ rax, callee save registers}	{ }

Homework #2

- Compute the interference graph of an L2 function given as input
- Implement the spiller (see Spilling.pdf) for an L2 function given as input
- Implement the full L2 compiler (including the graph coloring algorithm, see Graph_coloring.pdf)

Homework #2

- Compute the interference graph of an L2 function given as input

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

test/interference/test1.L2f

The order between rows doesn't matter

Your work needs to print to std::cout

```
%myVar1 %myVar2 r12 r13 r14 r15 rax rbp rbx
...
rsi r10 r11 r12 ... rbp rbx rcx rdi rdx
```

A node in the interference graph

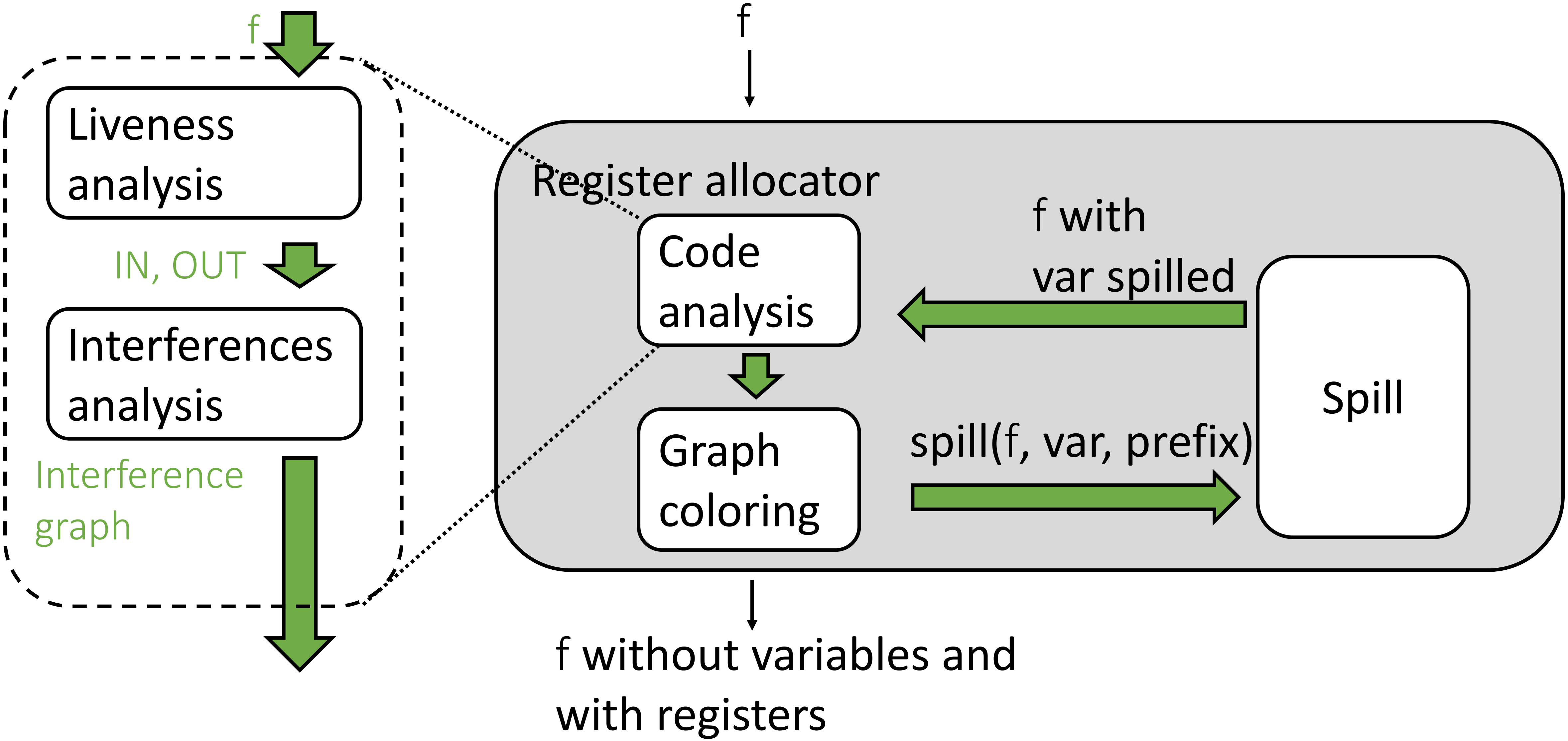
Nodes connected with the first one (the order between them doesn't matter)

test/interference/test1.L2f.out

Testing the interference graph of your homework #2

- **Under** L2/tests/interference **there are the tests you have to pass**
- To test:
  - **To check all tests:** make test_interference
  - **To check one test:** ./interference test/interference/test1.L2f
- Check out each input/output for each test if you have doubts
  - test/interference/test1.L2f
  - test/interference/test1.L2f.out

A graph-coloring register allocator structure



Always have faith in your ability

Success will come your way eventually

Best of luck!