Compilation (#8):

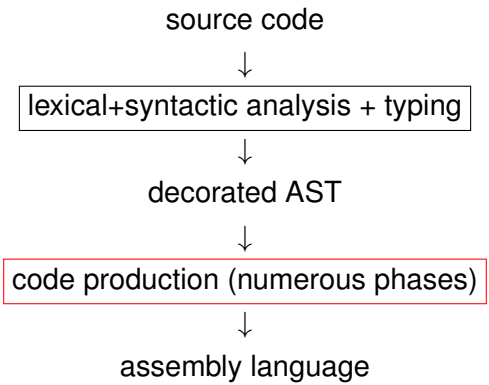
Register Allocation + Data Flow Analyses

C. Deleuze & L. Gonnord

Grenoble INP/Esisar

2022-2023

Where are we ?



- We work on IRs (Middle-end).

Credits

Fernando Pereira's course on register allocation:

http://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/RegisterAllocation.pdf

What for ?

- Finding storage locations to the values manipulated by the program ▶ registers or memory.
- registers are fast but in small quantity.
- memory is plenty, but slower access time.

▶ A good register allocator should strive to keep in registers the variables used more often.

> "Because of the central role that register allocation plays, both in speeding up the code and in making other optimizations useful, it is one of the most important - if not the most important - of the optimizations."

Hennessy and Patterson (2006) - [Appendix B; p. 26]

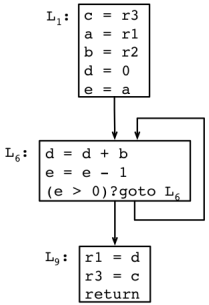
What for?

Expected behavior of **register allocation**:

- Input: a CFG with basic blocks with 3-address code (and pseudo-registers, aka temporaries)
- Output: same CFG but without temporaries:
  - replace with physical registers as much as possible.
  - if not **spill**, ie allocate a place in memory.
  - all copies assigned to the same physical registers (“moves”) can be removed: **coalescing** (**optional**).

Register constraints

Some variable are assigned to some specific registers (compiler, architecture constraints)



- r1,r2,r3 are used to pass function arguments here (thus should be reserved in the allocation)

The key notion: liveness

Observation

Two variables that are simultaneously **alive** must be assigned different registers.

(formal definition of alive follows)

Register assignment is NP-complete

Problem to solve

Given P a program and K general purpose registers, is there an assignment of the variables P in registers, such that (i) every variable gets at least one register along its entire live range, and (ii) simultaneously live variables are given different registers ?

Gregory Chaitin has shown, in the early 80's, that the register assignment problem is NP-Complete (register allocation via coloring, 1981)

3-phase algorithm

- **Liveness analysis**
  - When is a given value necessary for the rest of the computation?
- **Interference graph**
  - A graph that encodes which temporaries cannot be mapped to the same location.
- **Graph coloring** then register allocation.
  - The effective allocation: physical registers and stack allocation for temporaries.

Liveness analysis

In the sequel we call **variable** a temporary or a physical register.

Definition (Alive Variable)

In a given program point, a variable is said to be alive if the value it contains may be used in the rest of the execution.

"May", and non-decidable property $\Rightarrow$ overapproximation.

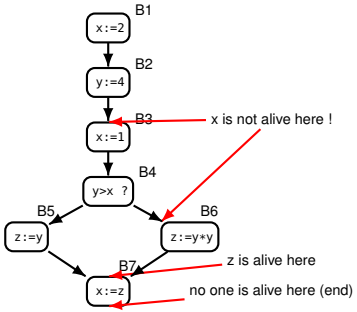
Important remark: here a block = a statement/program point. We have the same kind of analyses with block=basic block.

An example for live ranges

Definition

A variable is **live** at the exit of a block if there exists a path from the block to a use of the variable that does not redefine the variable.

```
x:=2;
y:=4;
x:=1;
if (y>x)
   then z:=y
   else z:=y*y ;
x:=z;
```



- The information flow is **backward**: from uses to definitions.

Data flow expressions

Definition

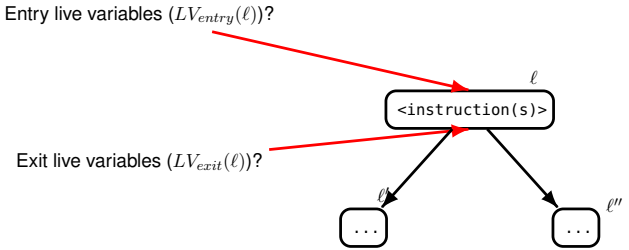
A variable that appears on the left hand side of an assignment is **killed** by the block. Tests do no kill variables.

Definition

A **generated** variable is a variable that appears in the block (read-only!)

- We precompute the sets : $kill_{LV}(block)$ and $gen_{LV}(block)$ for all blocks.

Dataflow equations



$$LV_{exit}(\ell) = \begin{cases} \emptyset & \text{if } \ell \text{ is final} \\ \bigcup_{\ell' \in succ_G(\ell)} LV_{entry}(\ell') & \text{else} \end{cases}$$

$$LV_{entry}(\ell) = \left(LV_{exit}(\ell) \backslash kill_{LV}(\ell)\right) \cup gen_{LV}(\ell)$$

Dataflow equations: solving

Important Remark

Fixpoint equations $X = F(X)$!

Here:

- Initialise LV sets to $\emptyset$.
- Compute LV_{entry} sets, then LV_{exit}, and continue.
- Stop when a fix point is reached.

▶ (vector of) Sets are strictly growing, and the live range set is at most the set of all variables, thus **this algorithm terminates**.

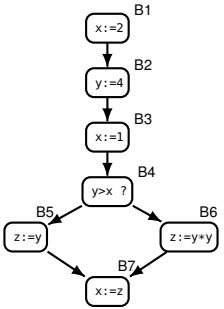
Steps

$LV_{entry}(\ell)$ denoted by $In(\ell)$, $LV_{exit}(\ell)$ by $Out(\ell)$ initialisation to emptysets is not depicted.

			Step 1		Step 2		Step 3 (stable)
ℓ	$kill(\ell)$	$gen(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$
1	$\{x\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
2	$\{y\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{y\}$	$\emptyset$
3	$\{x\}$	$\emptyset$	$\emptyset$	$\{x, y\}$	$\{y\}$	$\{x, y\}$	$\{y\}$
4	$\emptyset$	$\{x, y\}$	$\{x, y\}$	$\{y\}$	$\{x, y\}$	$\{y\}$	$\{x, y\}$
5	$\{z\}$	$\{y\}$	$\{y\}$	$\{z\}$	$\{y\}$	$\{z\}$	$\{y\}$
6	$\{z\}$	$\{y\}$	$\{y\}$	$\{z\}$	$\{y\}$	$\{z\}$	$\{y\}$
7	$\{x\}$	$\{z\}$	$\{z\}$	$\emptyset$	$\{z\}$	$\emptyset$	$\{z\}$

Final result and use

Backward analysis and we want the smallest sets, here is the final result : (we assume all vars are dead at the end).

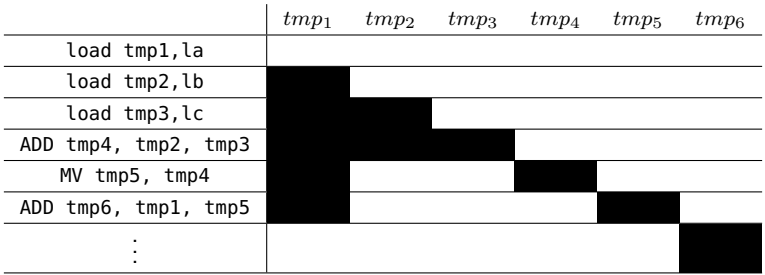


ℓ	$LV_{entry}(\ell)$	$LV_{exit}(\ell)$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{y\}$
3	$\{y\}$	$\{x, y\}$
4	$\{x, y\}$	$\{y\}$
5	$\{y\}$	$\{z\}$
6	$\{y\}$	$\{z\}$
7	$\{z\}$	$\emptyset$

- Use : Dead code elimination !

Step 2: Interferences

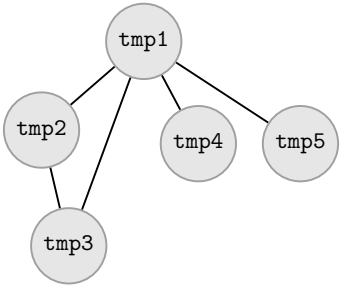
Here is the output of the liveness analysis for $a + (b + c)$:



- `tmp1` is in conflict with `tmp2` (because of instruction 3) denoted by $tmp_1 \bowtie tmp_2$.

Interference graph

The relation $\bowtie$ defines a conflict/interference graph:



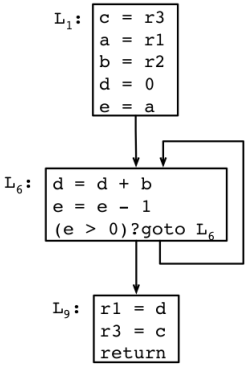
We want a **correct allocation** with respect to $\bowtie$:

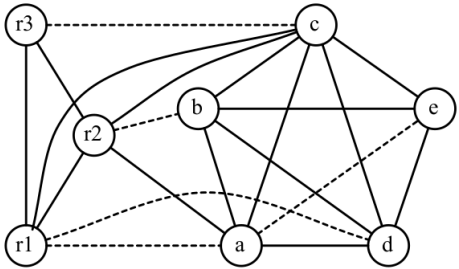
$tmp_1 \bowtie tmp_2 \implies Alloc(tmp_1) \neq Alloc(tmp_2)$.

- Graph coloring.

Running example

Important: in this example consider the r_i as temporary registers, like the others.

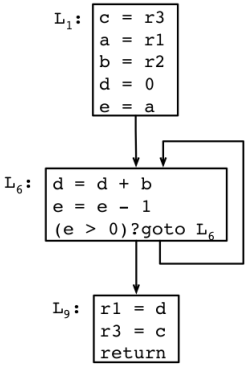


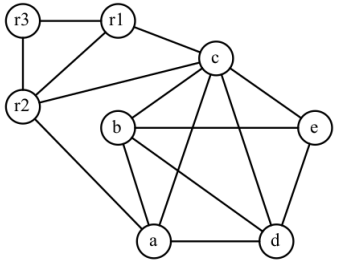


Dashed edges represent moves!

Running example

Important: in this example consider the r_i as temporary registers, like the others.





Let's look at the graph without moves first

Kempe's simplification algorithm 1/2

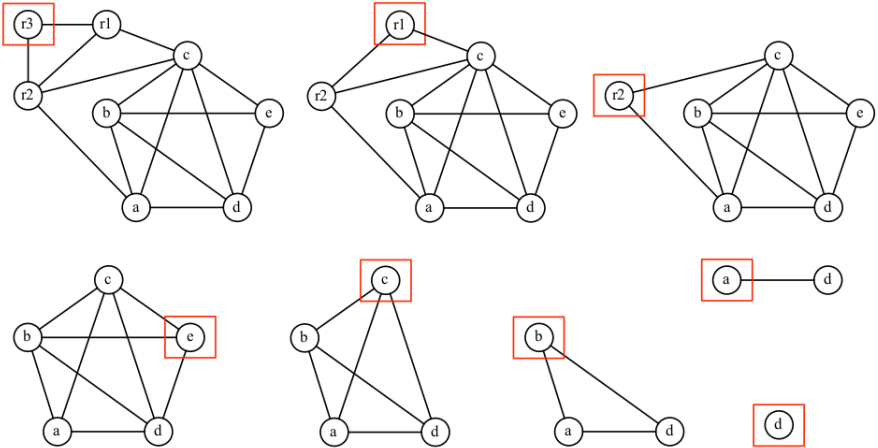
On the interference graph (without coalesce edges):

Proposition (Kempe 1879)

Suppose the graph contains a node m with fewer than K neighbours. Then if $G' = G \setminus \{m\}$ can be K-colored, then G can be K-colored as well.

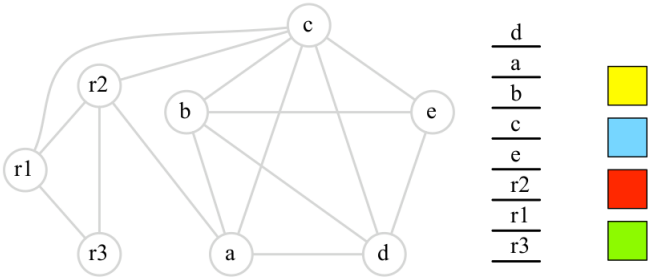
- Pick a low degree node, and remove it, and continue until remove all (the graph is K-colorable) or ...

Kempe's simplification algorithm 2/2

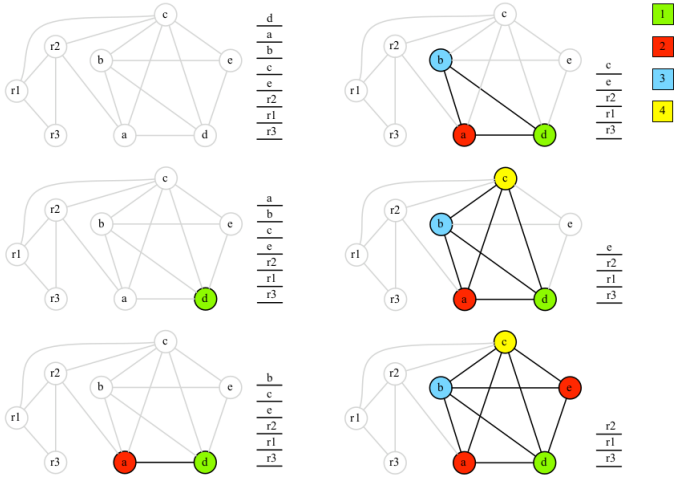


Let's color! ("Kempe's heuristic")

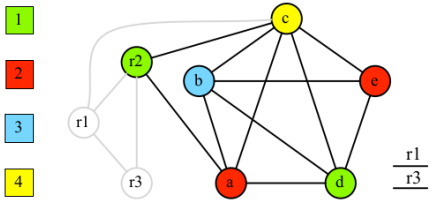
- We assign colors to the nodes greedily, in the reverse order in which nodes are removed from the graph.
- The color of the next node is the first color that is available, i.e. not used by any neighbour.

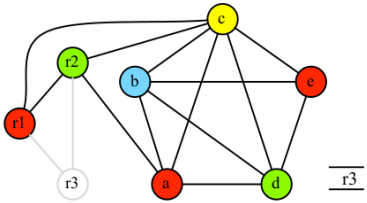


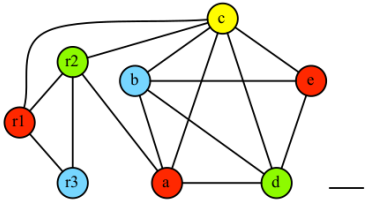
Greedy coloring example 1/2



Greedy coloring example 2/2







On the number of colors (K)

In the last example, we chose K=4, and this is nice, because the graph is 4-colorable.

- The given heuristic may fail to color the graph with K colors: it doesn't mean that the graph is not K-colorable (**heuristic**!).
- We can chose:
  - either to eliminate the "non-colorable node" of the graph and continue with the other nodes inside the node stack.
  - either to augment the K parameter.

Recall memories - Final code generation

With a 3 address code + allocation, rewrite each 3 address instruction into “real code”:

- Each temporary is rewritten into his allocated physical register.
- If the temporary is in memory (Spilling), we generate code with appropriate loads and stores.

If the graph was not successfully colored

Non-colored variables[1] are named **spilled temporaries**.

There are many solutions to handle spilled variables.

[1] either not colored at all or colored with number >K

A naive solution: also color memory!

- Launch the coloration algorithm with an infinite number of colors:
  - first colors are mapped to registers (used in priority by the coloring algorithm)
  - other colors are mapped to offsets in the stack, i.e. spilled to memory
- Drawback: we need a few registers to implement the spilling

More sophisticated: Live range splitting

Idea: Modify the code to lower the number of simultaneously alive registers. Invent 2 versions of the same variable (**live-range splitting**), and modify the code into:

```
ADD temp51, temp4, temp3
STORE temp51, [locationinmemory] # replace with actual location
..
LOAD temp52, [locationinmemory]  #same
ADD temp6, temp52, #5
```

- But now we have to allocate these two new variables!

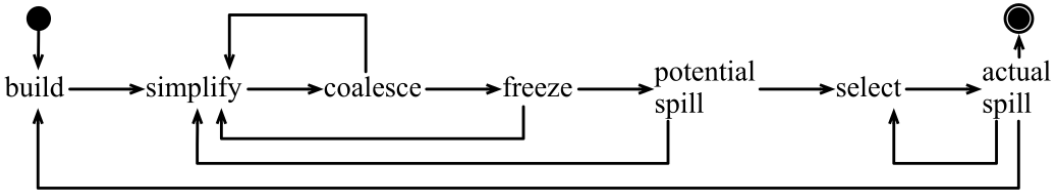
We relaunch the coloring algorithm. This is called **Iterative Register Allocation**.

To go further: Iterative Register Coalescing[2]

Two new optimizations to improve register allocation further

1. Register coalescing
2. Clever spilling

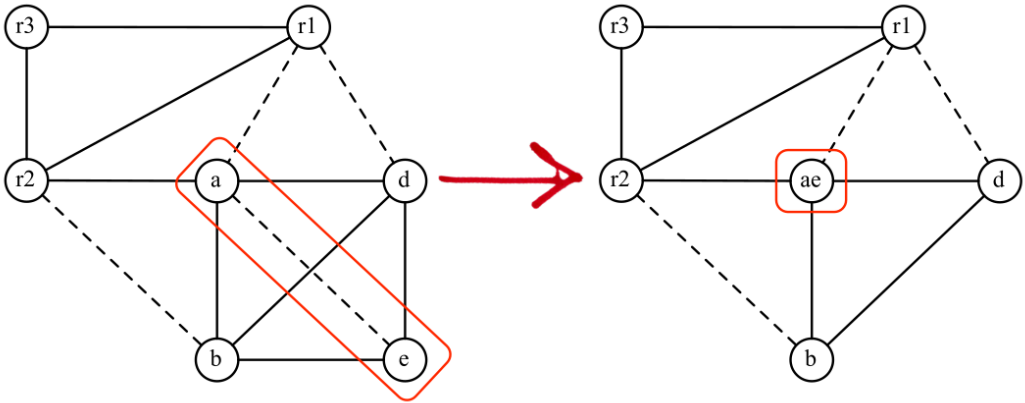
An iterative algorithm with many steps:



[2] Iterated Register Coalescing, TOPLAS (1996)

Iterative Register Coalescing – Coalescing

Coalescing consists of collapsing two move related nodes together

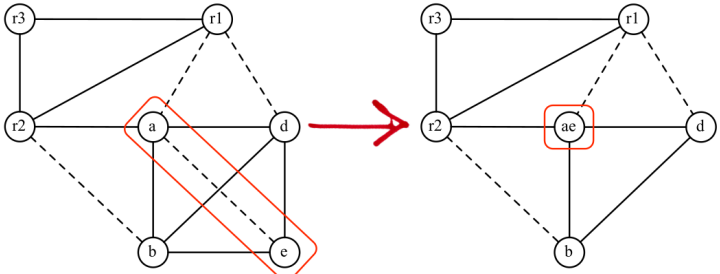


Iterative Register Coalescing – Coalescing

Two heuristics for coalescing safely:

Briggs Nodes a and b can be coalesced if the resulting node ab will have fewer than K neighbors of high degree (i.e., $degree \geqslant K$ edges)

George Nodes a and b can be coalesced if, for every neighbor t of a, either t already interferes with b, or t is of low degree.



Iterative Register Coalescing – Spilling

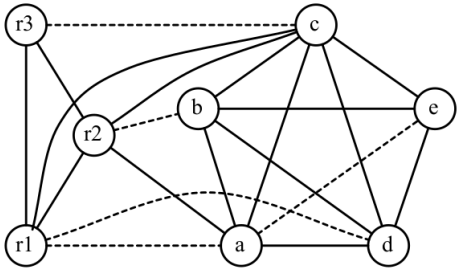
- How to choose which variables to spill ? This is actually really hard:
  - We want to spill variables that are less used dynamically
  - We only have static information

We can use a heuristic:

```
SPILLCOST(v)
  cost = 0
  foreach definition or use in block B
    cost += 10N/D, where
      N is B's loop nesting factor
      D is v's degree in the interference graph
```

Other Algorithms

- **Linear scan**: greedy coloring of interval graphs. (see Fernando Pereira's slides on register allocation: 18 to 35)
- Plenty of other heuristics for spilling.



Bilan

1. Register allocation - Intro
2. Live-range Information: Liveness Analysis
3. Register Allocation with graph coloring
   - Conflict (Interference) Graph
   - Coloring
   - Spilling strategies