Intermediate Representations for Register Allocation

Patryk Zadarnowski

27 June 2003

University of New South Wales

Sydney

The Plan

① Overview

② Graph coloring

- the $degree < k$ rule (the good news)
- the bad news
- optimal coloring

③ Preparation and recovering from trouble

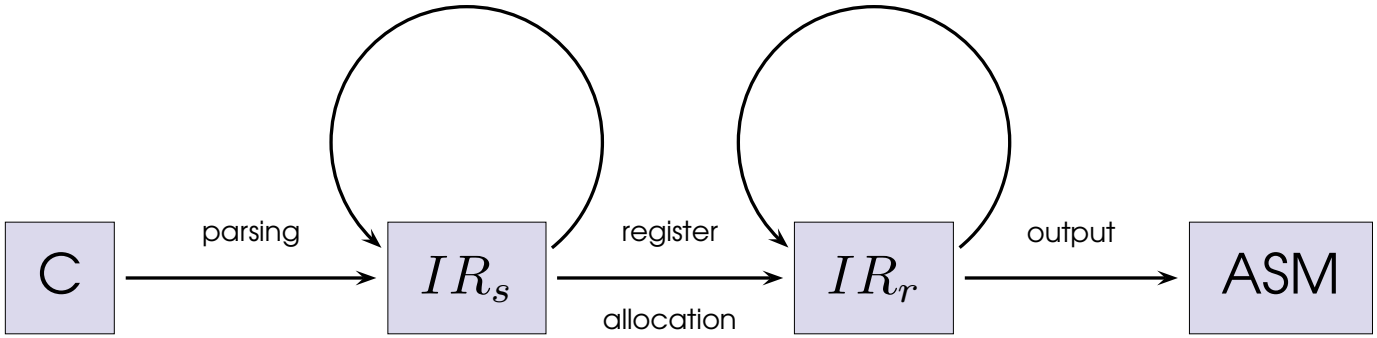
- Variable webs
- Register coalescing

④ Cleaning up the mess

- Control Flow Graphs
- Static Single Assignment Form
- Value/State Dependance Graphs
- Exotic Approaches

Motivation

Structure of a typical compiler:



- Different representations required for symbolic code (IR_s) and register-allocated code (IR_r)
- Optimizations performed on both IR_s and IR_r
- ✗ Hampers code reuse
- ✗ Makes verification cumbersome
- ✗ Introduces phase ordering problems

Overview of Register Allocation

- Register Allocation: select the set of variables that will reside in registers at a point in the program.
- Register Assignment: assign physical register to hold values of variables during evaluation.

Overview of Register Allocation

➜ Register Allocation: select the set of variables that will reside in registers at a point in the program.

➜ Register Assignment: assign physical register to hold values of variables during evaluation.

① Greedy heuristics-based local approaches:

- Use the structure of the program.
- Greedily allocate variables to registers based of frequency of use, beginning with inner loops.
- When combined with heuristics, quite workable.

Overview of Register Allocation

- ➜ Register Allocation: select the set of variables that will reside in registers at a point in the program.
- ➜ Register Assignment: assign physical register to hold values of variables during evaluation.

① Greedy heuristics-based local approaches:

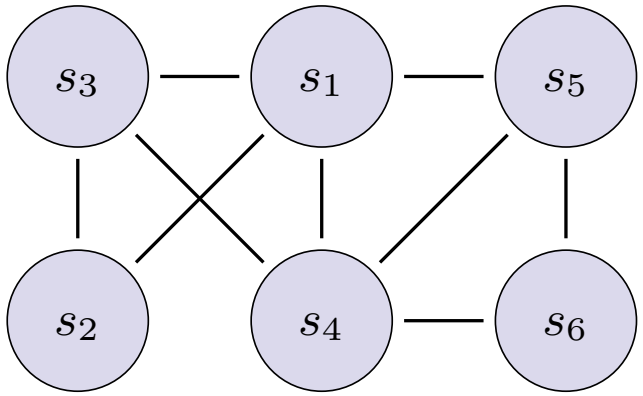
- Use the structure of the program.
- Greedily allocate variables to registers based of frequency of use, beginning with inner loops.
- When combined with heuristics, quite workable.

② Register assignment by graph coloring.

Graph Coloring — I

- Construct register-interference graph for all variables in a procedure/program:
- Nodes represent symbolic registers.
- Edges connect two nodes if one is live at the point where other is defined:

$$
\begin{aligned}
s_1 &\leftarrow \texttt{arg}_1 \\
s_2 &\leftarrow \texttt{arg}_2 \\
s_3 &\leftarrow \texttt{add}(s_1, s_2) \\
s_4 &\leftarrow \texttt{add}(s_1, s_3) \\
s_5 &\leftarrow \texttt{mul}(s_1, 1) \\
s_6 &\leftarrow \texttt{mul}(s_4, 2) \\
&\texttt{ret}(s_5, s_6)
\end{aligned}
\quad \longrightarrow \quad
\]

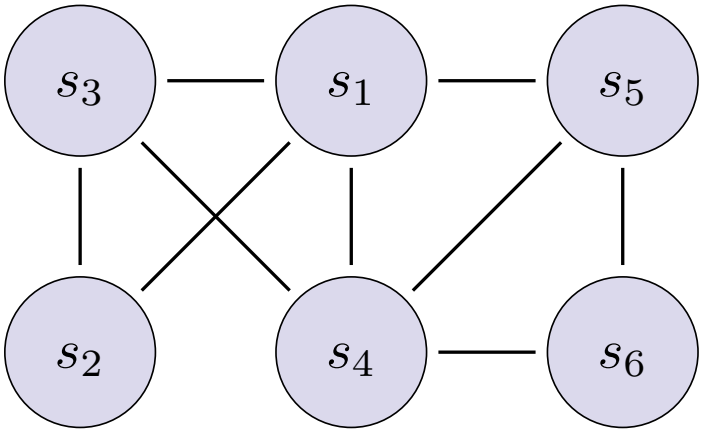


- Colour the nodes using k colours in such a way that no two adjacent nodes have the same colour (the map coloring problem.)

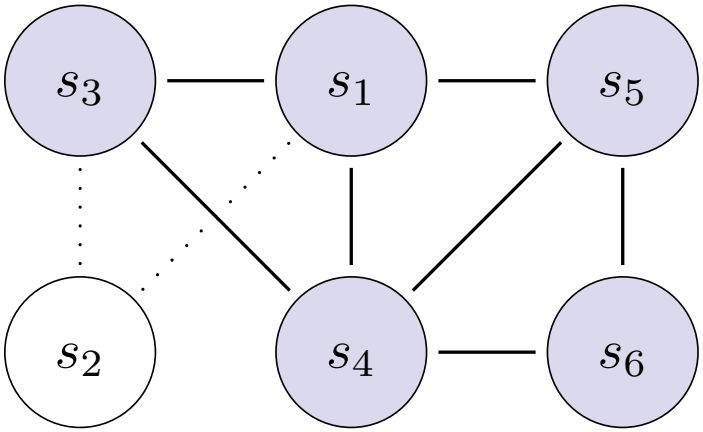
Graph Colouring — II

- Graph coloring is NP-complete in general, but:
- The $degree < k$ rule: A graph containing a node with less than k neighbours is k-colourable iff the graph without that node is k-colourable.
- The following greedy algorithm works extremely well in practice:
  1. Find a node with less than k neighbours. Remove it from the graph.
  2. Repeat until the graph has only one node left. Colour that node.
  3. Begin adding the removed nodes back to the graph, in reverse order.
  4. Each time we add a node back, colour it with a colour different from any of its neighbours.

Graph Colouring Example

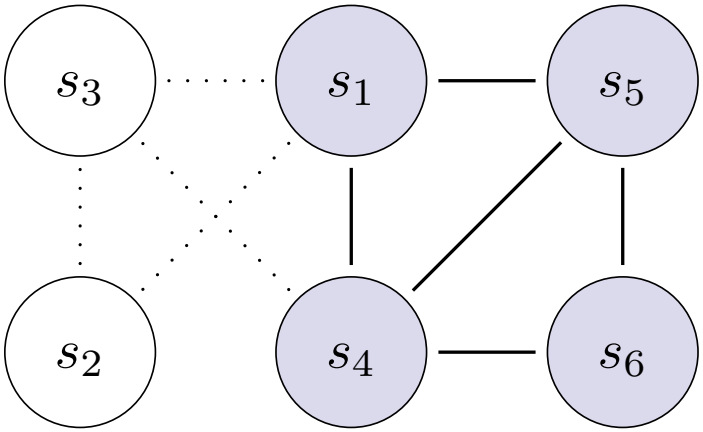


Graph Colouring Example



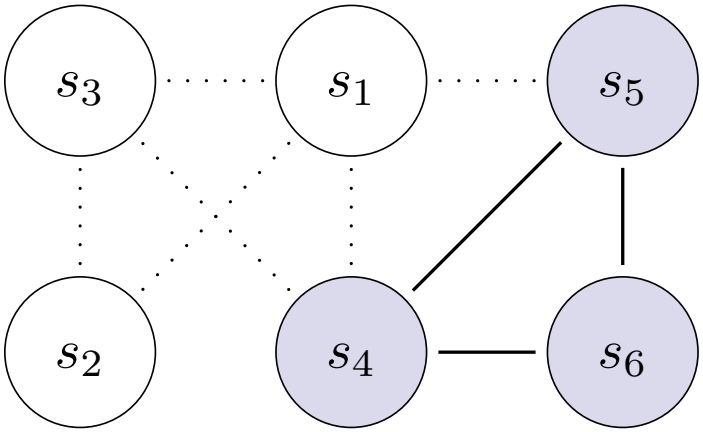
Removed nodes: s_2

Graph Colouring Example



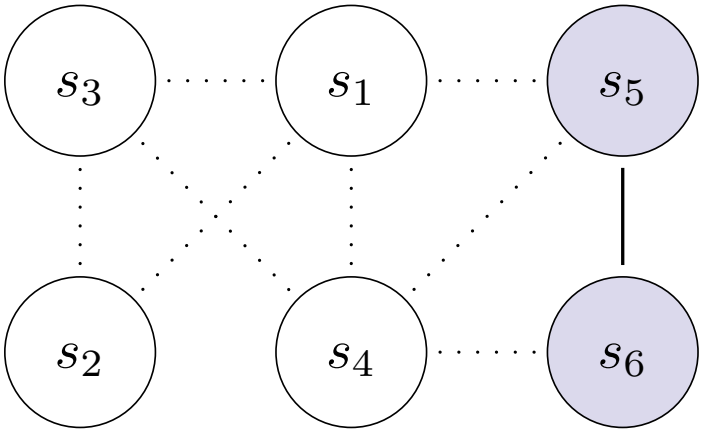
Removed nodes: s_2, s_3

Graph Colouring Example



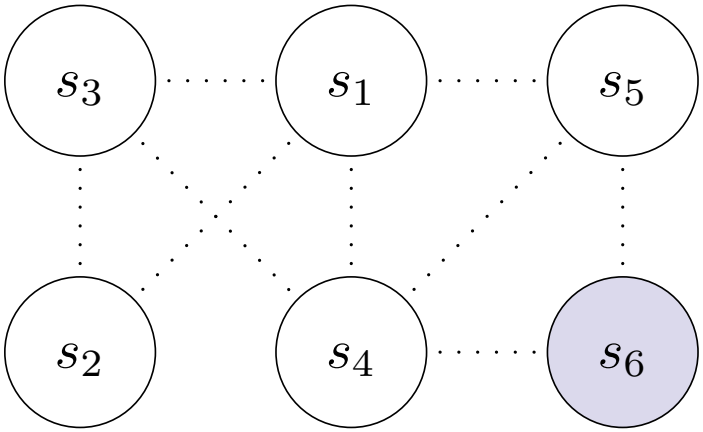
Removed nodes: s_2, s_3, s_1

Graph Colouring Example



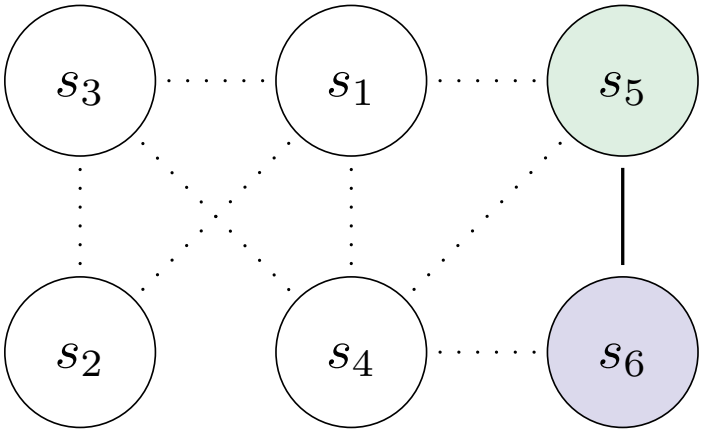
Removed nodes: s_2, s_3, s_1, s_4

Graph Colouring Example



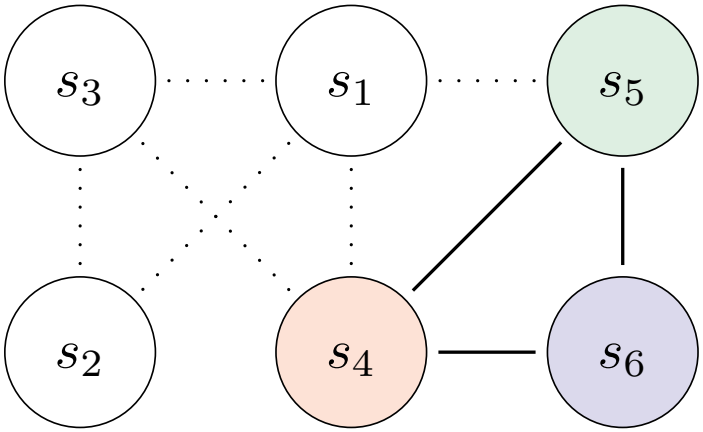
Removed nodes: s_2, s_3, s_1, s_4, s_5

Graph Colouring Example



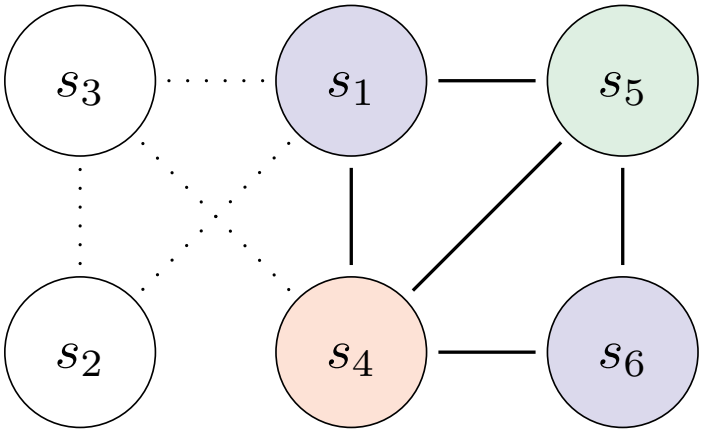
Removed nodes: s_2, s_3, s_1, s_4

Graph Colouring Example



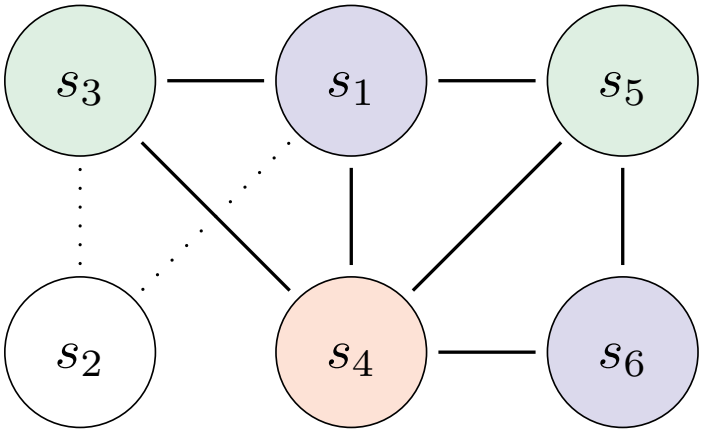
Removed nodes: s_2, s_3, s_1

Graph Colouring Example



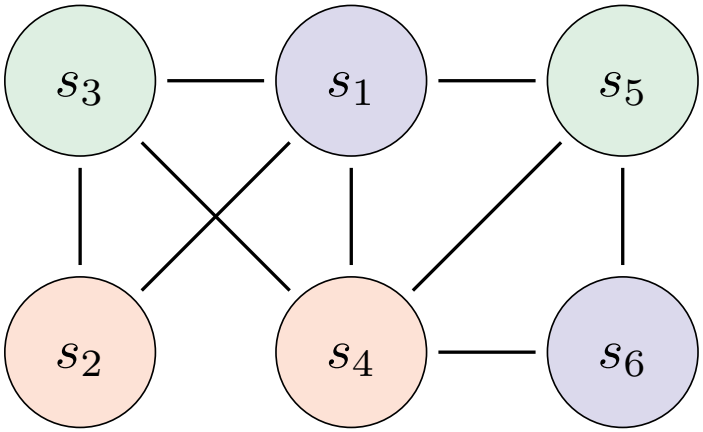
Removed nodes: s_2, s_3

Graph Colouring Example



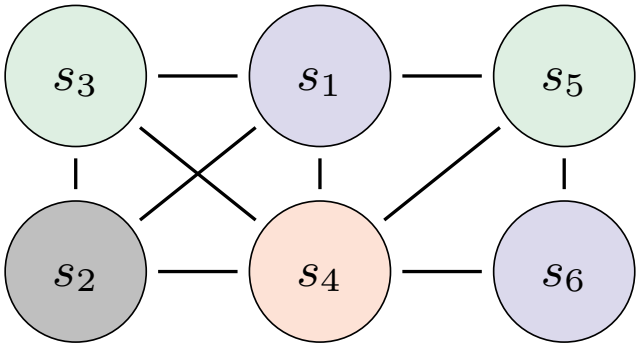
Removed nodes: s_2

Graph Colouring Example

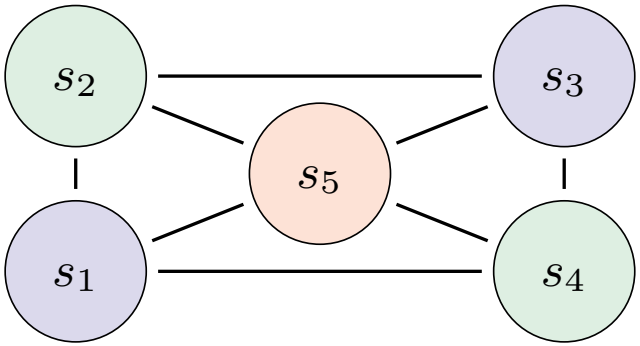


The Bad News

- We still haven't solved the whole problem!
- Some graphs are simply not colourable:



- Others can't be reduced using the $degree < k$ rule:



- But mostly, the technique works suprisingly well. Why?!

Optimal Graph Colouring — I

Interference graphs are special!

Some Terminology:

- The smallest k for which a graph G is colourable is called its chromatic number $\chi(G)$.
- A safe lower bound for $\chi(G)$ is the clique number $\omega(G)$.
- A graph G in which, for every subgraph G', $\chi(G') = \omega(G')$ is called perfect.
- A graph G for which $\chi(G) = \omega(G)$ is called 1-perfect.
- For most graphs, $\chi(G) - \omega(G)$ is large.

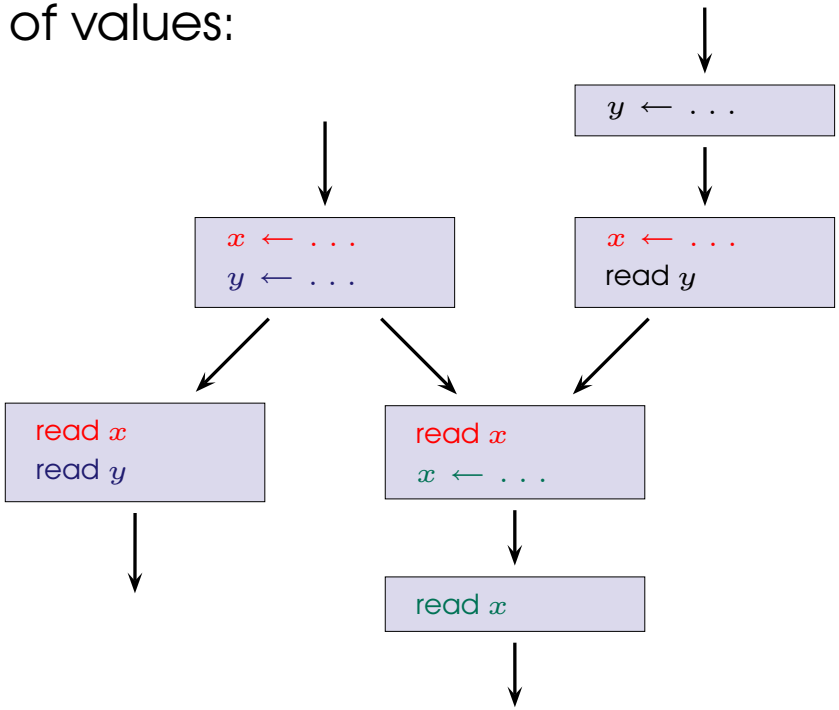
Optimal Graph Colouring — II

- Andersson investigates 27,921 real-life interference graphs and found that all of them are 1-perfect.
- Although finding $\omega(G)$ is NP-complete, there exists the so-called Shannon function $\vartheta(G)$, such that:
  1. $\omega(G) \leq \vartheta(\overline{G}) \leq \chi(G)$
  2. For all graphs, $\vartheta(G)$ is computable in polynomial time.
- Nobody knows how to compute $\vartheta(G)$ in polynomial time for an arbitriary graph.
- Andersson presents a backtracking-search solution that performs optimal coloring of an arbitriary graph and has approximately $O(V)$ complexity for 1-perfect graphs.

Preparing for Register Assignment — I

The Allocatable Objects:

- ➜ Cannot use source variables (name reuse.)
- ➜ Cannot use definition points (multiple definitions may have a common use.)
- ➜ Instead, combine definitions with a common use into maximal webs of values:



Preparing for Register Assignment — II

Register Coalescing (Subsumption):

- ➜ A variety of copy propagation
- ➜ Eliminates assignments of the form $x \leftarrow y$ when x and y do not interfere by replacing all occurances of x with y
- ➜ Easy to perform given the interference graph
- ➜ The interference graph can be updated incrementally for each eliminated assignment.

Recovery Strategies

Recovering from $degree < k$:

- ➜ Don't panic just yet; pick a node for removal based on some plausible heuristics and hope for the best.
- ➜ If we run out of colours while poping nodes back into the graph, spill something and repeat the process.

When things get really crowded, we can try:

① moving some instructions closer to their use
(undoing loop invariant code motion)

② cloning a calculation to avoid keeping its result
(undoing common subexpression elimination)

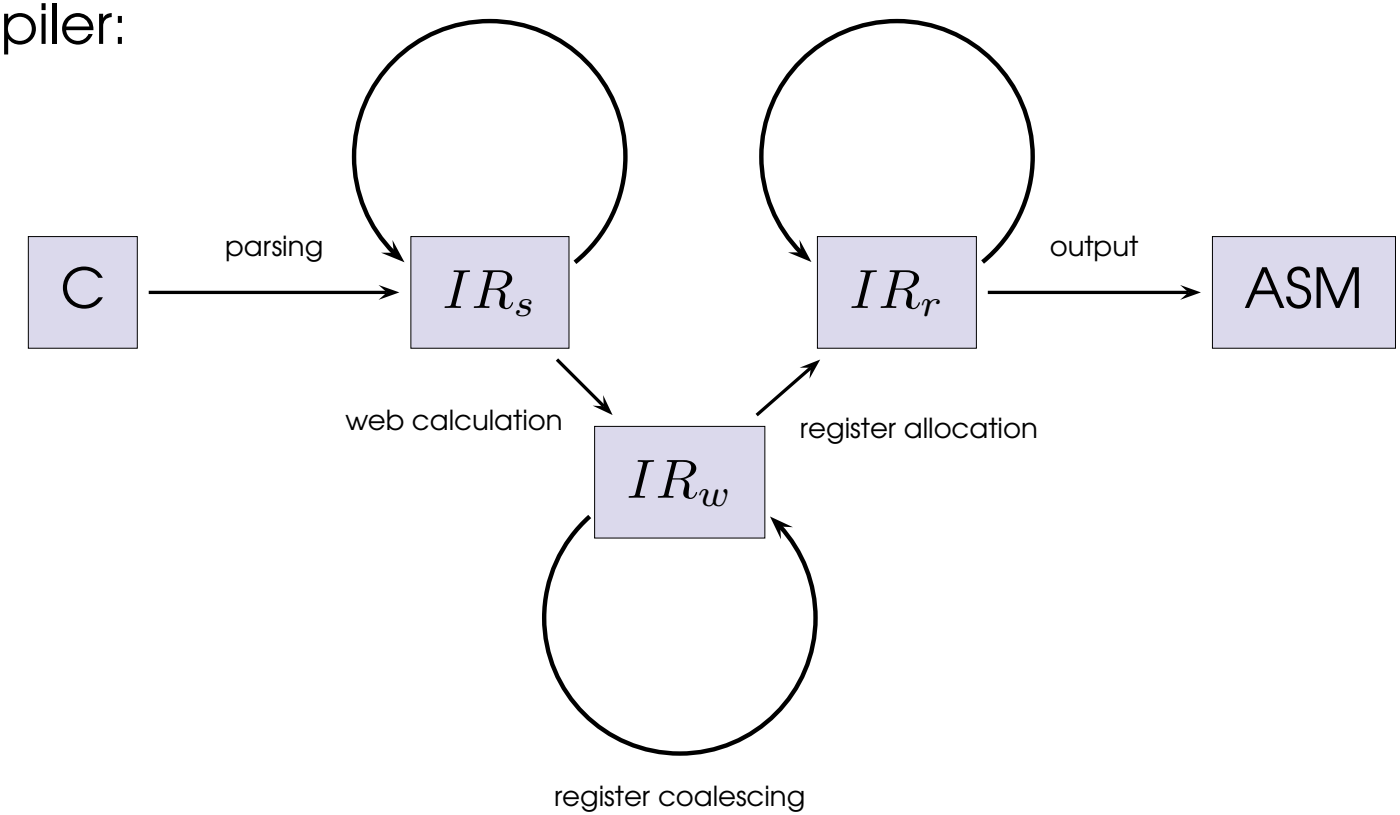
③ collapsing unrolled loops

. . .

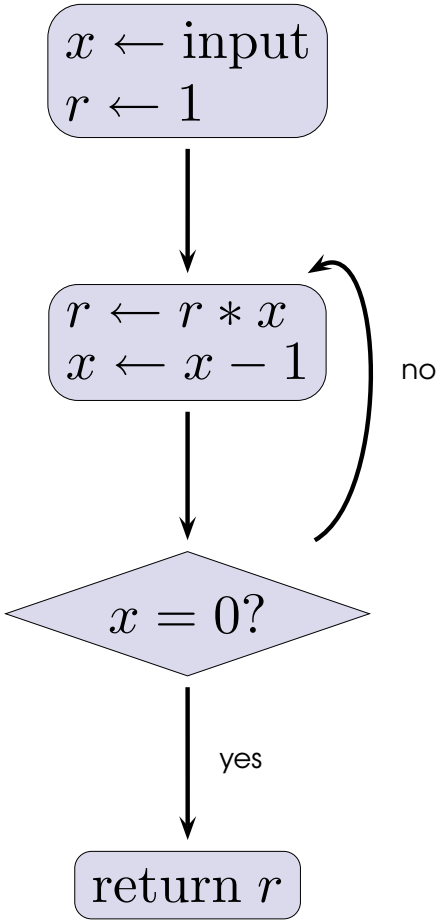
④ spilling some variables into memory.

Cleaning up The Mess

Graph coloring gave us a systematic method of performing register assignment, but it is combined with *ad hoc* register allocation and spilling, complicating the structure of the compiler:



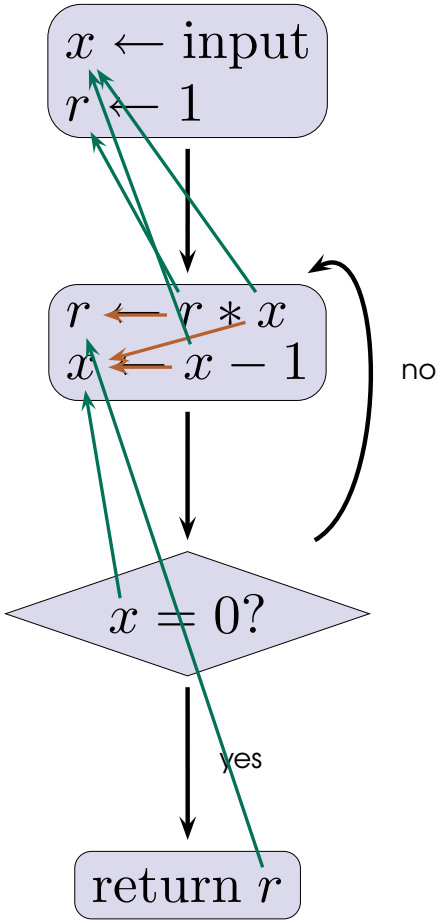
Control Flow Graphs



Overview:

- ☑ All variables are updatable, just like physical registers.
- ☑ Same representation for both IR_s and IR_r.
- ☒ Emphasis on *control flow*, introduces artificial register dependencies.

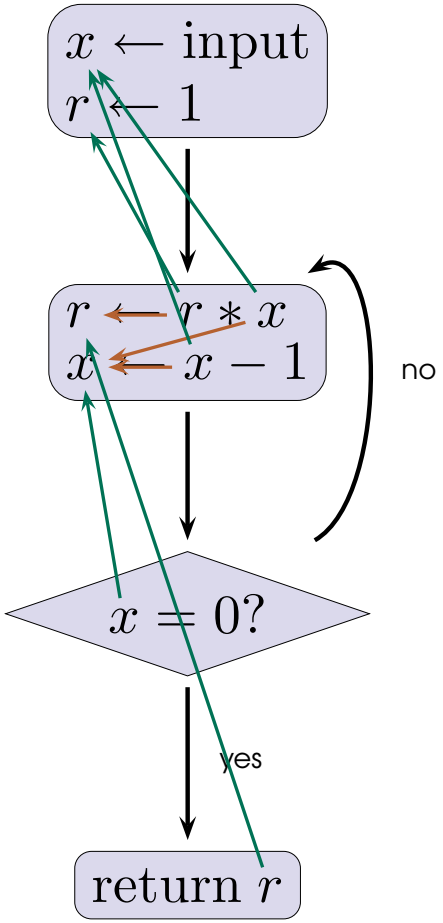
Control Flow Graphs



Data Flow:

- For web calculation, data flow information is required
- Represented by DU-chains linking definitions (assignments) with their respective uses.
- ☑ Usually encoded efficiently as bit arrays.

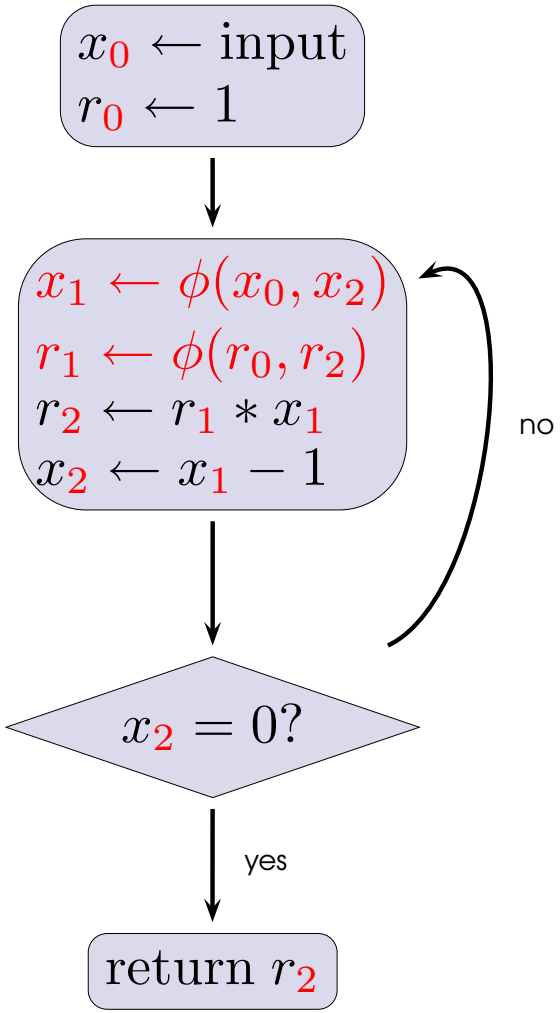
Control Flow Graphs



Problems:

- ✗ Dense representation of data flow.
- ✗ Expensive to compute and update.
- ✗ Makes many common optimizations prohibitively expensive.
- Went out of fashion in mid-1980's.

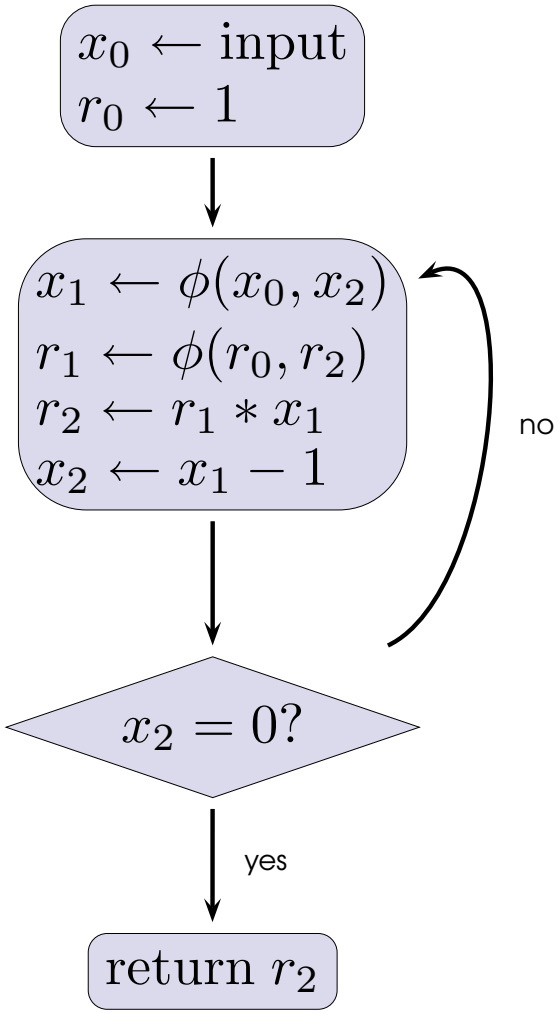
Static Single Assignment Form



Overview:

- Each variable has exactly one defining occurence
- Values from different control flows merged via ϕ functions
- ☑ Efficient sparse representation of data flow
- ☑ Split variables improve accuracy of analysis
- The most popular representation used in compiles today.

Static Single Assignment Form



Problems:

- ✗ Cannot be used for IR_r
- ✗ Control flow still in our way
- ✗ ϕ nodes introduce new problems:
  - translate to extra `mov`'s
  - complicates web calculations
  - cannot express register coalescing
  - live-range identification expensive
- ☑ Efficient $O(n\alpha(n))$ algorithm exists which combines all of the above.

VALUE STATE DEPENDENCE GRAPHS — I

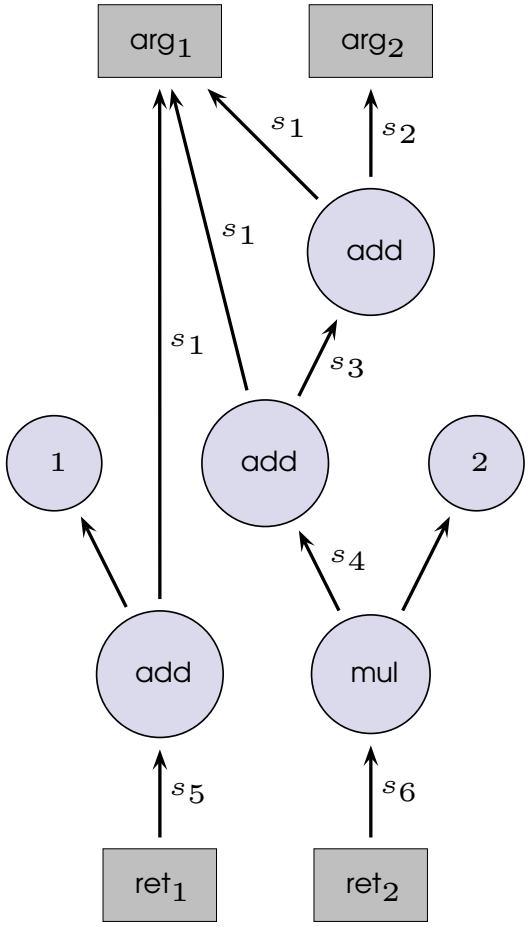
- A new intermediate representation designed specifically for register allocation
- Intended to remove the control-flow bottleneck
- Loosely based on the Gated Static Assignment Form
- Register allocation performed "in reverse":
  1. First, the program is restructured to make it colourable
  2. Then, register allocation is performed by graph colouring.
- Register colouring is the final stage before code generation; all optimizations are performed on the VSDG.

VALUE STATE DEPENDENCE GRAPHS — II

- ➜ Represents all necessary data and state dependencies explicitely.
- ➜ Procedures represented by a graph $\langle N, E_v, E_s, N_0, N_\infty \rangle$ where:
  - N is a set of nodes
  - E_v is a set of value dependence edges
  - E_s is a set of state dependence edges
  - N_0 is a unique entry node
  - N_∞ is a unique exit node
- ➜ Nodes can represent instructions (value definitions) or special structural nodes $\gamma(C, T, F)$ (lazy conditionals) and $\theta(C, I, R, L, X)$ (natural loops.) Loops are nasty.
- ➜ Except for θ nodes, VSDG is an acyclic graph.

Value State Dependence Graphs — III

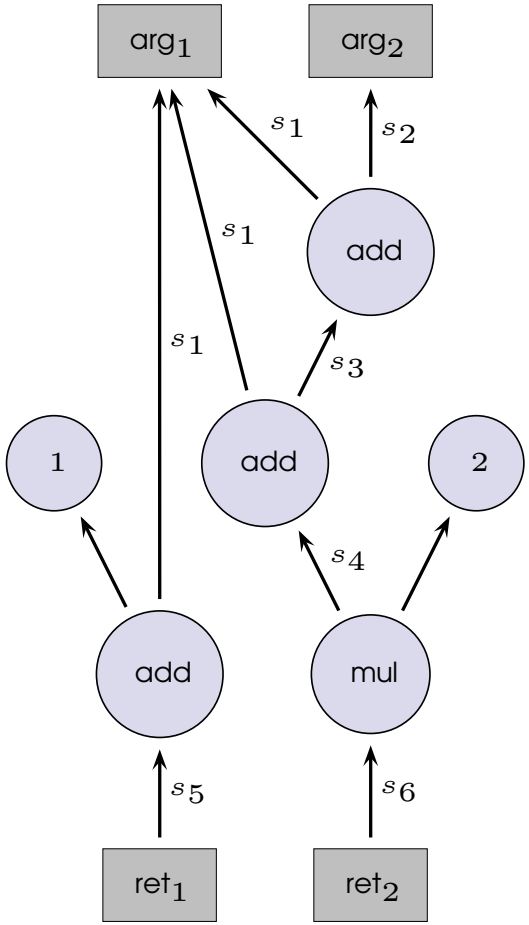
$$
\begin{aligned}
s_1 &\leftarrow \texttt{arg}_1 \\
s_2 &\leftarrow \texttt{arg}_2 \\
s_3 &\leftarrow \texttt{add}(s_1, s_2) \\
s_4 &\leftarrow \texttt{add}(s_1, s_3) \\
s_5 &\leftarrow \texttt{mul}(s_1, 1) \\
s_6 &\leftarrow \texttt{mul}(s_4, 2) \\
&\texttt{ret}(s_5, s_6)
\end{aligned}
\quad \longrightarrow
$$



Value State Dependence Graphs — IV

Benefits:

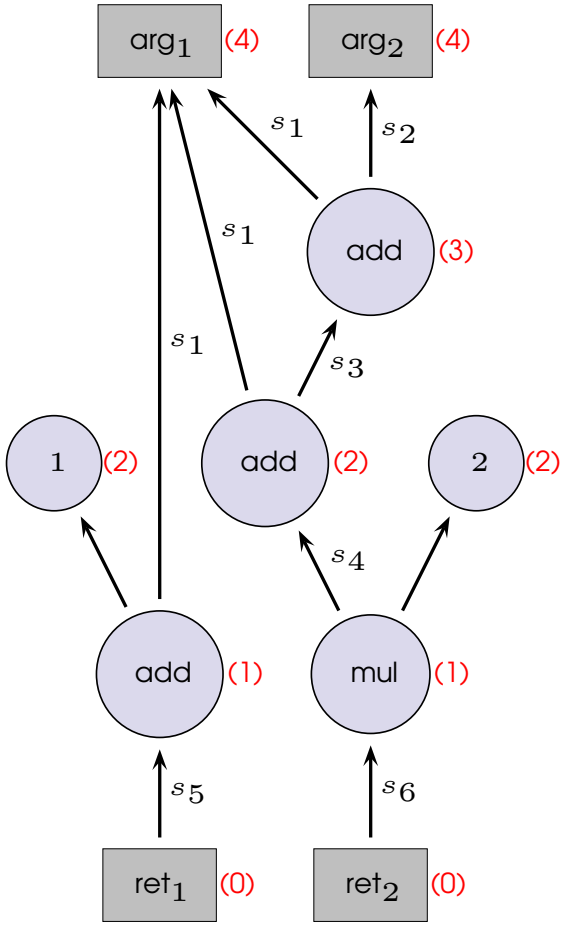
- ☑ Sparse representation of of data flow
- ☑ No implicit control flow
- ☑ Has a strong normalizing effect
- ☑ Can represent program before and after register allocation
- ☑ Can be serialized to correspond to a CFG



VALUE STATE DEPENDENCE GRAPHS — V

Register Allocation Algorithm:

1. Calculate the DFR (maximal distance from N_0) for each node in the graph.
2. Partition the graph into "cuts": sets of nodes with the same DFR.
3. Calculate the set of live nodes for each cut: $S_{>d} \cap pred_v(S_{\leq d})$
4. Apply a forward snow-plough graph reshaping algorithm ensuring the size of each set is less than k.



REGISTER ALLOCATION BY PROOF TRANSFORMATIONS — I

- ➜ A new approach not based on graph coloring.
- ➜ Designed to systematically combine all of the register allocation into a single formal framework.
- ➜ Based on the judgments as types correspondence: programs may be regarded as proofs of their types.
- ➜ Easily incorporated into a static type system to unify pre- and post-allocation representations into a single language.

Register Allocation by Proof Transformations — II

➜ Imperative regarded as a sequent-style proof system which deduces typing properties of code, eg:

$$\Gamma, x : \tau \vdash \texttt{ret}(x) : \tau \qquad \frac{\Gamma' \vdash B : \tau}{\Gamma \vdash I; B * \tau}$$

➜ The method uses types to represent live ranges of variables.

➜ Register allocation is a transformation from a proof system with implicit structural rules to one with explicit ones.

➜ Liveness analysis is done by type inference.

➜ Register assignment is done by insertion of explicit memory-register moves and register allocation/deallocation instructions.

A promising new approach which incorporates all issues involved in register allocation into a single formal framework!

THE END