50006 - Compilers - (Prof Kelly) Lecture 7

Oliver Killane

15/04/22

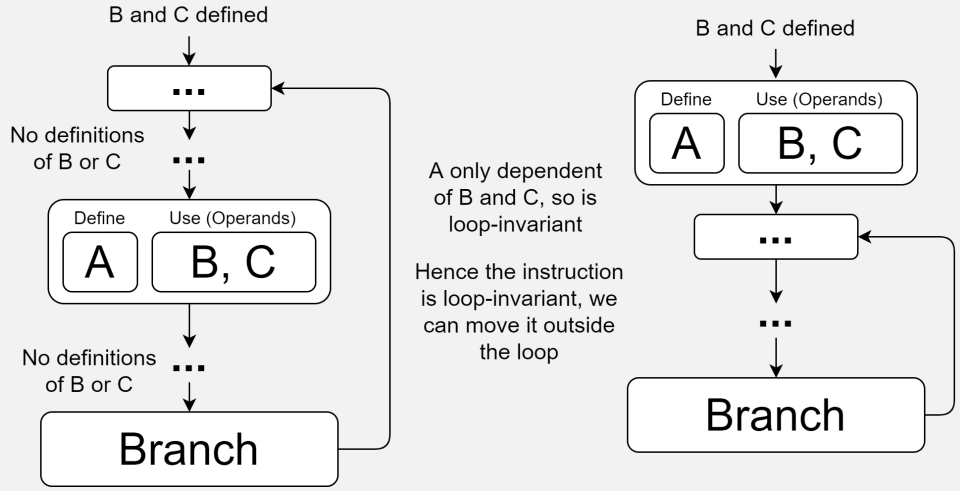
Loop Invariant Code Motion

Lecture Recording

Lecture recording is available here

Definition: Loop Invariant

An instruction is **loop-invariant** if its operands are only defined outside of the loop. Hence the value it defines is not loop-invariant (same for every iteration) and hence it may be possible to move the instruction outside the loop.



Finding Loop-Invariant Instructions

Using a control flow graph, each node is an instruction. We attempt to find definition nodes of form:

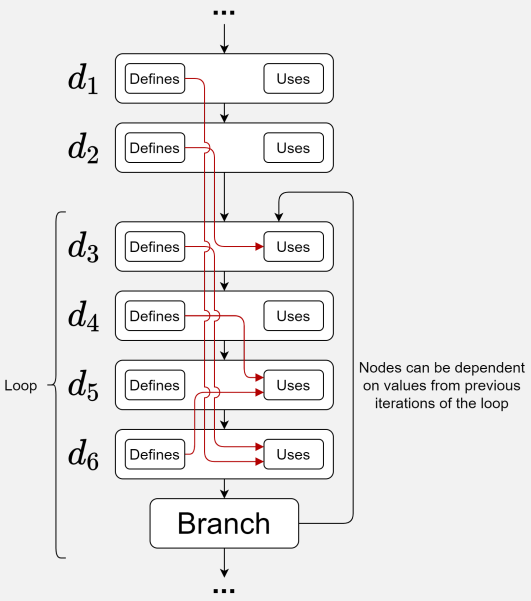
$$\underbrace{d}_{\text{Node ID}} \; : \; t_d := (\underbrace{u_1 \bullet u_2}_{\text{Binary Op}} \mid \underbrace{u_1}_{\text{Copy Op}} \mid \underbrace{c}_{\text{Constant}})$$

Where t_d is the destination, and u_i is for temporary variables used.

d is **loop-invariant** if every definition of a $u_i \in uses(d)$ that reaches d is outside the loop.

Example: Basic Loop

We can see in this loop that some definitions are used as operands. Including from previous iterations of the loop.

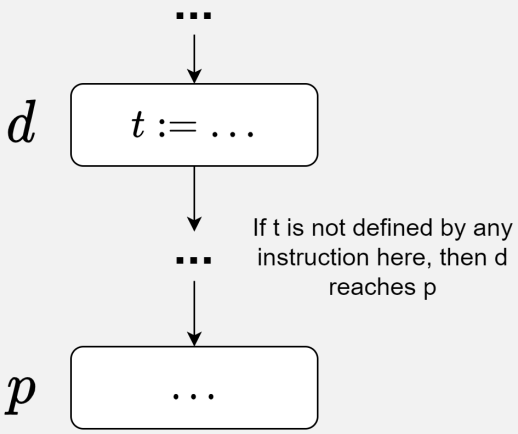


Definitions

We refer to a node/instruction defining some variable t as a definition of t.

Definition: Reach

A **definition** d reaches p if there is a path from $d \to p$ where d is not killed.



For a node n we have several sets containing other nodes/instructions:

$$\begin{aligned} Gen(n) &= \{n\} = \text{the set of definitions generated by the node} \\ Kill(n) &= \text{Set of all definitions of } t \text{ except for } n \\ ReachIn(n) &= \text{Set of definitions reaching up to } n \\ ReachOut(n) &= \text{Set of definitions reaching after } n \end{aligned}$$

We can calculate the **reach-in** and **reach-out** sets as:

$$ReachIn(n) \triangleq \bigcup_{p \in Pred(n)} ReachOut(p)$$

$$ReachOut(n) \triangleq Gen(n) \cup (ReachIn(n) \setminus Kill(n))$$

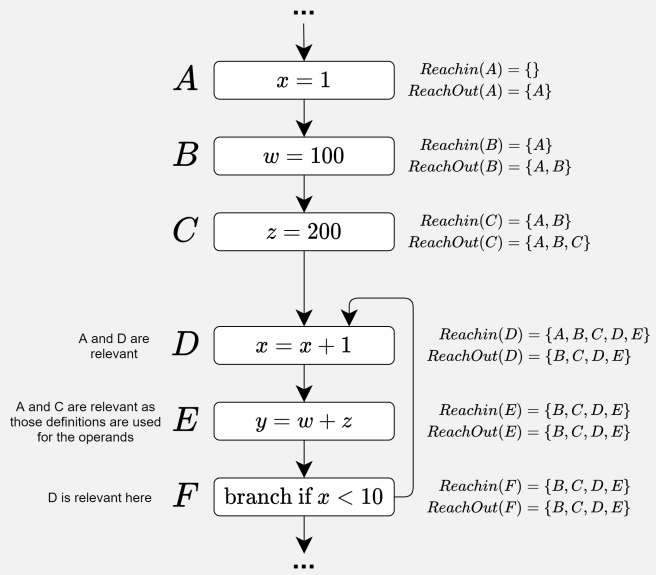
This is a **forward analysis** as definitions reach forwards.

```
1  def get_reach_sets(program):
2
3      # set all reachin, reachout sets to {}
4      for instr in program:
5          instr.reachin = []
6          instr.reachout = []
7
8      changed_set = True
9      while changed_set:
10         # until there is no change, continue to update the sets. This eventually
11         # terminates as the sets are finite, and at each step they can only remain
12         # the same size (no change) or grow.
13
14         changed_set = False
15
16         for instr in program:
17             # Reachin(n) = all predecessor's reachouts
```

```
18                new_reachin = union([pred.reachout for pred in instr.preds])
19                if new_reachin != instr.reachin:
20                    changed = True
21                    instr.reachin = new_reachin
22
23                # Reachout(n) = Gen(n) U (Reachin(n) \ Kill(n))
24                new_reachout = instr.gen + (instr.reachin - instr.kill)
25                if new_reachout != instr.reachout:
26                    changed = True
27                    instr.reachout = new_reachout
```

Form the **reaching definitions**, we can reduce each set to the **relevant reaching definitions**, by considering only the reachins that are actually used by the instruction (for operands).

Example: Basic Reaching Definitions

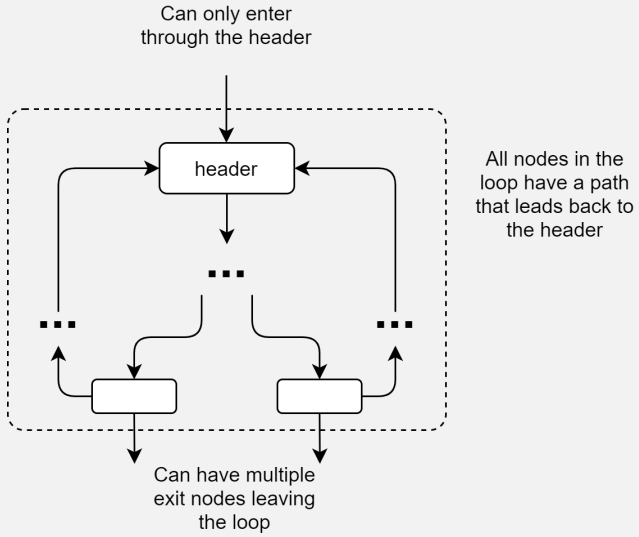


Identifying Loops

Definition: Loop

Given a set S of nodes that are part of a loop.

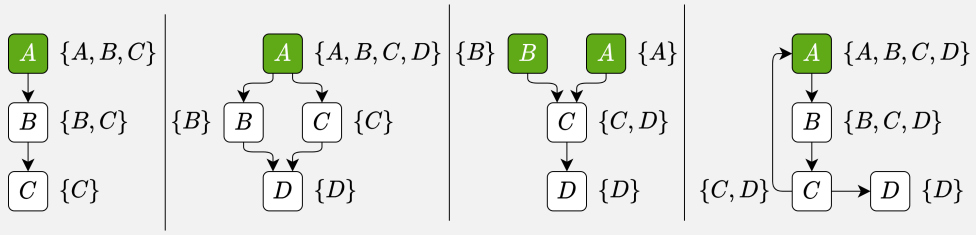
- There is a single header node $h \in S$.
- There is a path from h to any other node in S.
- There is a path from any node in S to h.
- All non-loop nodes can only directly connect to the header h, and no other nodes in the loop.



Definition: Dominator

Node d dominates node n if every path from the start node to n goes through d (Note that every Node dominates itself).

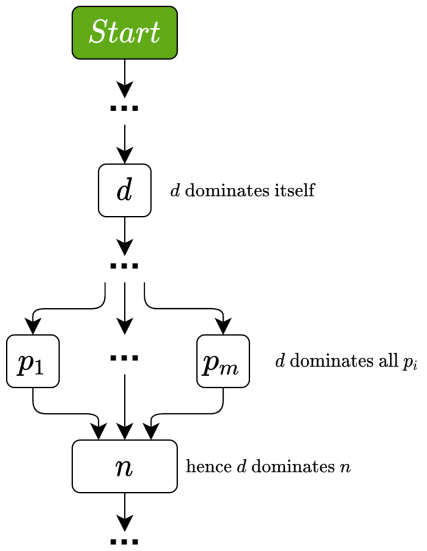
Hence for each node we can get the set of nodes it **dominates** (as shown below), or the set of nodes that **dominate** it.



To find all the nodes that **dominate** a given node:

1. Set all *Dom* sets to the set of all nodes.

2. Apply the rules (as below), as the start node will have a set of $\{start\}$ this will propagate, reducing the sizes of the sets for other nodes.
3. Once the sets stop changing, we have our solution.

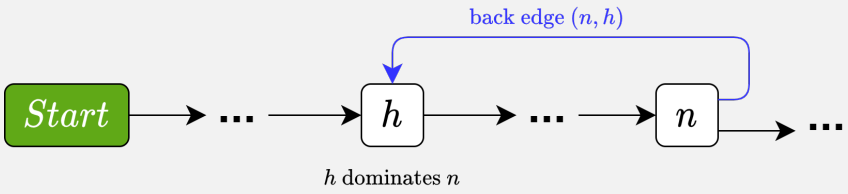


$$Doms(start) = \{start\}$$

$$Doms(n) = \{n\} \cup \left(\bigcap_{p \in preds(n)} Doms(p) \right)$$

Definition: Back Edges

An edge in the **control flow graph** from $n \to h$, where h dominates n is a **back edge**.



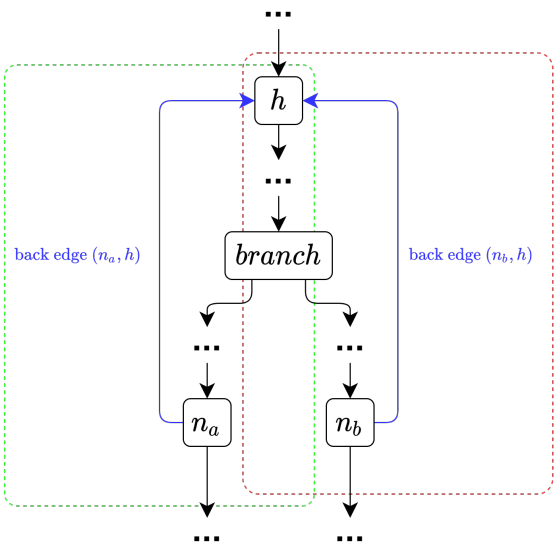
Definition: Natural Loop

The **natural loop** of a **back-edge** (n, h) is the set of nodes S such that:

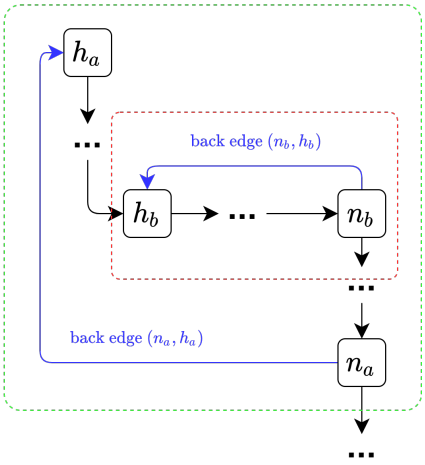
- All nodes $x \in S$ are dominated by h
- For all nodes $x \in S$ (except h), there is a path from $x \to n$ that does not contain h.

This represents a loop, with the header node h.

Multiple Loops



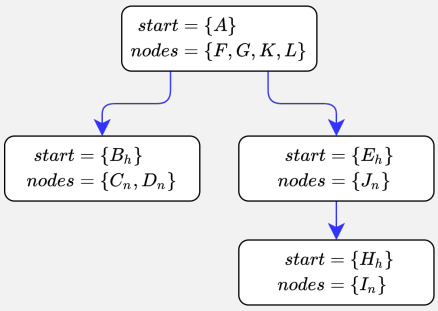
Could be one or two loops, we interpret it as a single loop.

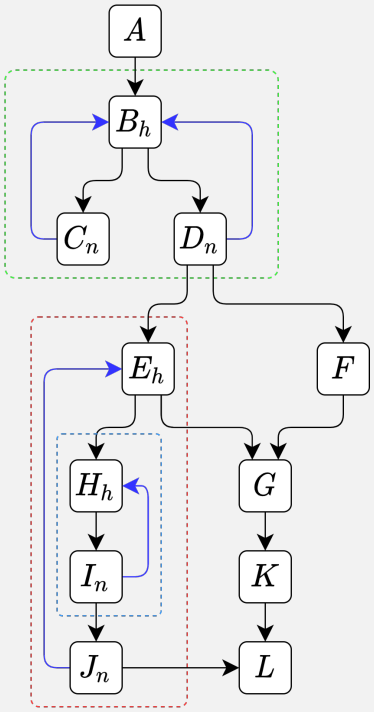


Nested Loops

Definition: Control Tree

We can construt a tree to show which loops are nested, what the headers and final nodes in each loop are.

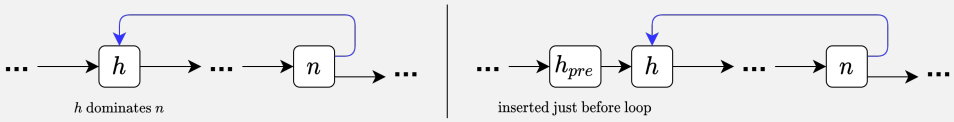




Hoisting Instructions

Definition: Pre-Header

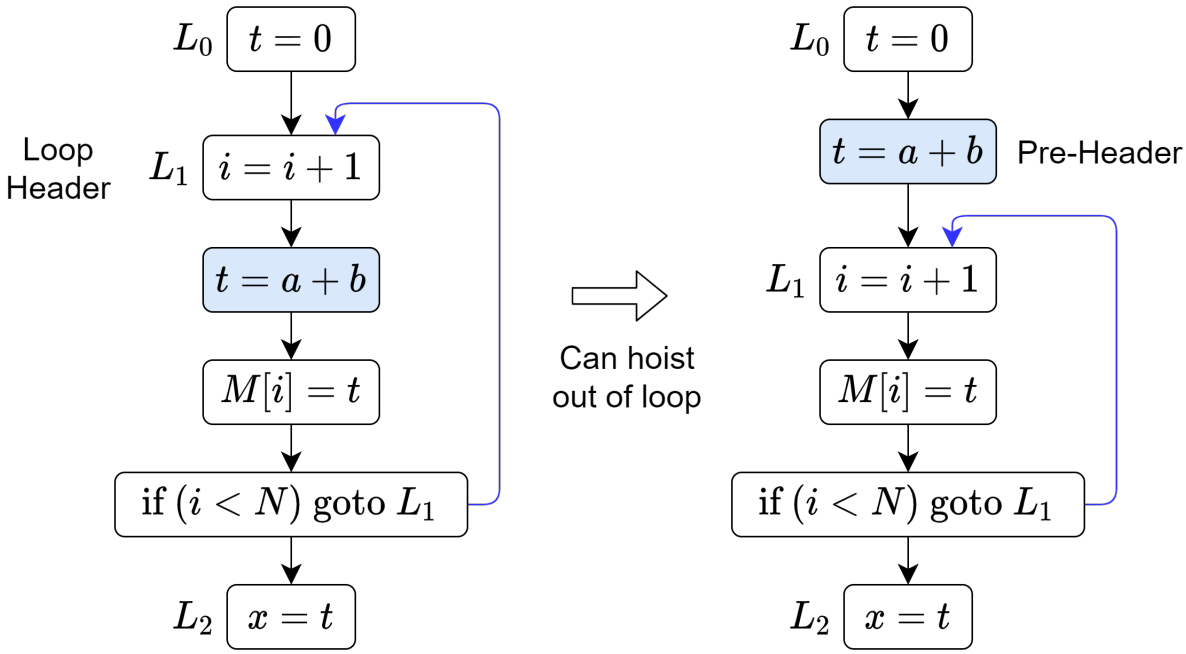
A node inserted immediately before the **header** node of a **natural loop**.

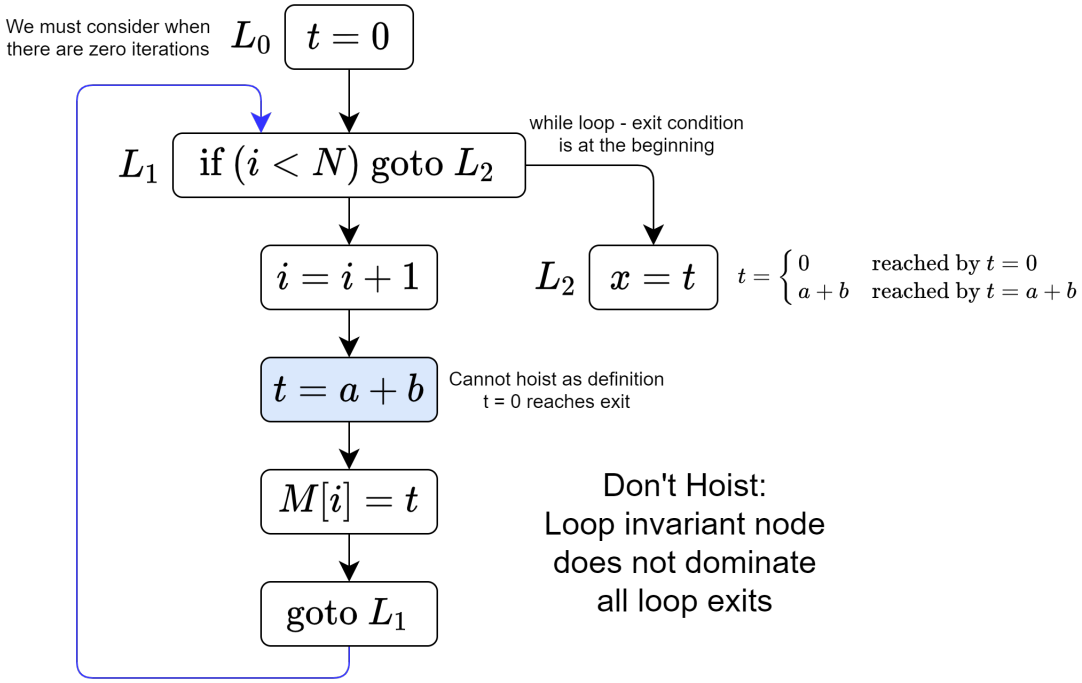


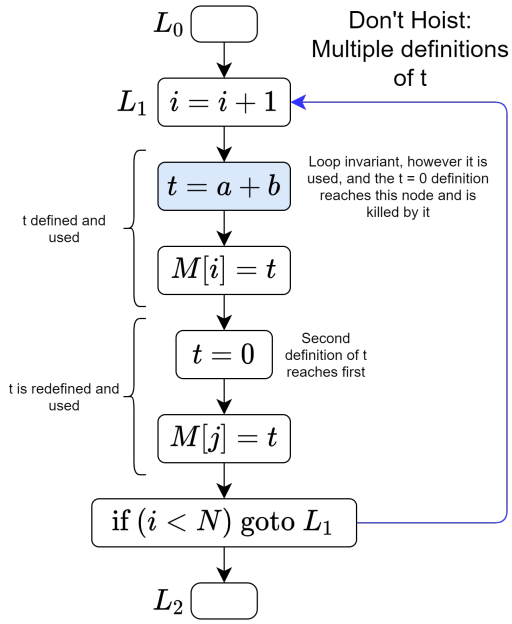
Lecture Recording

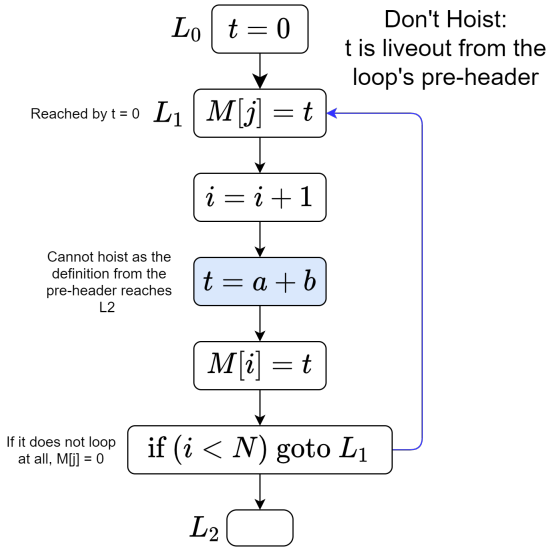
Lecture recording is available here

Consider the following examples:


L_0 $t = 0$
Loop Header
L_1 $i = i + 1$
$t = a + b$
$M[i] = t$
if $(i < N)$ goto L_1
L_2 $x = t$
Can hoist out of loop
L_0 $t = 0$
$t = a + b$
Pre-Header
L_1 $i = i + 1$
$M[i] = t$
if $(i < N)$ goto L_1
L_2 $x = t$


We must consider when there are zero iterations
L_0 $t = 0$
L_1 if $(i < N)$ goto L_2
while loop - exit condition is at the beginning
$i = i + 1$
L_2 $x = t$
$t = \begin{cases} 0 & \text{reached by } t = 0 \\ a + b & \text{reached by } t = a + b \end{cases}$
$t = a + b$
Cannot hoist as definition t = 0 reaches exit
$M[i] = t$
goto L_1
Don't Hoist:
Loop invariant node does not dominate all loop exits





Given a node $d: \ t = a \bullet b$ the conditions for hoisting are:

1. **All reaching definitions used by d occur outside the loop**
   We must not use data defined inside the loop (otherwise not invariant). Use **reaching definition** analysis for this.
2. **Loop invariant node must dominate all loop exits**
   The pre-header dominates all exits, hence in order to be moved to the pre-header, the invariant node must also. Use **dominators** analysis for this.
3. **There can only be one definition of t**
   Count the definitions.
4. **t cannot be liveout from the loop's pre-header**
   If t is live out from the pre-header, it means that t is used somewhere in the loop (t is also defined in the loop), or the value of t from the pre-header is used after the loop (e.g a while loop when no iterations are run). Hence we cannot hoist it. Use **live range** analysis for this.

The basic process of hoisting loop-invariant instructions out of a loop is:

1. Compute **dominance** sets for each node.
2. Use **dominance** sets to identify natural loop and their headers.
3. Compute the reaching sets for nodes.
4. Use relevant reaching definitions to identify loop-invariant code.
5. Attempt the loop invariant code to a **pre-header**.
6. Check that the semantics of the program are not altered.

This process can potentially be repeated (e.g hoisting out of several layers of nested loops).

Static Single Assignment

Lecture Recording

Lecture recording is available here

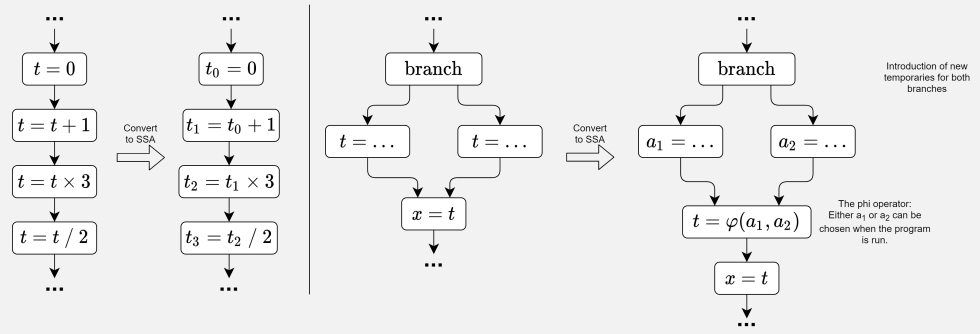
Definition: Static Single Assignment (SSA)

A representation avoiding side conditions by allowing only a single assignment to each temporary (temporaries are immutable).

- Simplifies many optimisation problems
- Requires a potentially complex translation (to and from SSA)
- Makes many optimisation passes more efficient.
- Widely used in compilers.

Each Reassignment of a variable is renamed, this splits all live ranges.

- Live ranges split (temporary is live from definition to last use, with no gaps in liveness).
- Each variable has only one reaching definition.

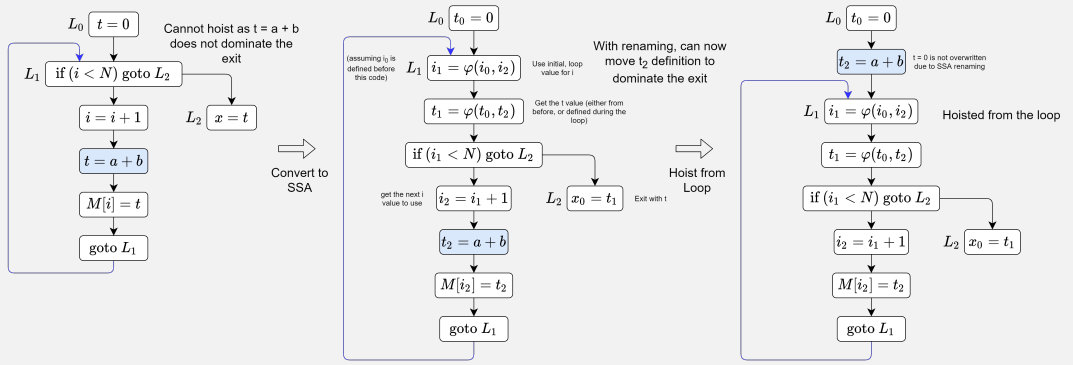


The **phi function** is used for branching. A **phi** statement $\varphi(a_1, a_2)$ means either a_1 or a_2 could be used.

By renaming variables, we can eliminate many of the loop hoisting issues (particularly for variable redefinitions in the loop)

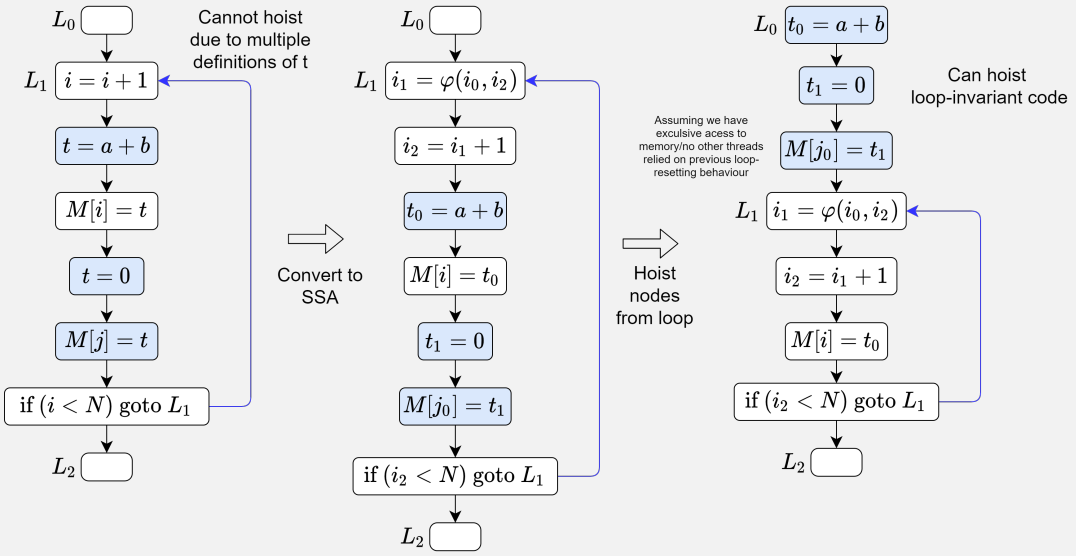
Example: Hoist a node that does not dominate the exit

The renaming done by SSA allows us to move a node so that it dominates the exit (allowing us to hoist it from the loop), without altering the semantics of the program.

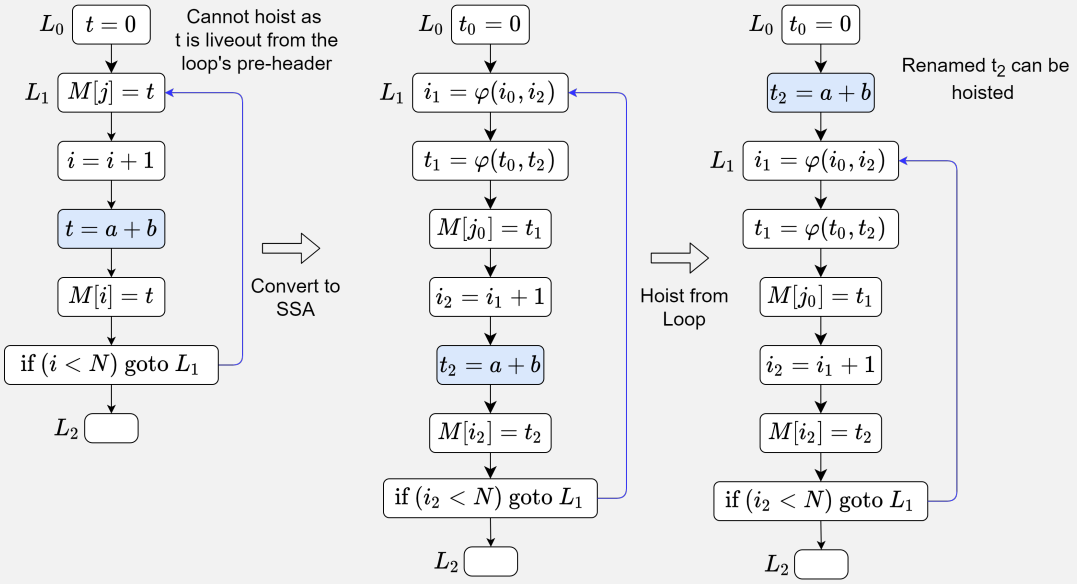


Example: Hoist with multiple definitions of t

Again, renaming from SSA allows us to hoist.



Example: Hoist when t is liveout from the Pre-Header



Once in **SSA** form, we can reassess the requirements for hoisting:

1. **All reaching definitions used by d occur outside the loop**
   (Same as prior to **SSA**). Use **reaching definitions** analysis for this.
2. **Loop invariant node must dominate all loop exits**
   No longer an issue.
3. **There can only be one definition of t**
   Guarenteed by **SSA** form.
4. **t cannot be liveout from the loop's pre-header**
   Cannot occur with **SSA** due to single assignment.

Other Compiler Optimisations

There is a wide selection of optimisation techniques that can be applied, some general (e.g inlining), while others are architecture specific (optimising instruction selection).

Induction Variables	Strength Reduction	Induction variable Elimination
Rewriting Comparisons	Array Bounds Check Elimination	Common Sub-Expressions
Partial Redundance Elimination	Loop Unrolling	Inlining
Dead Code Elimination	Rematerialisation	Tail-Call Optimisation

While conventional compilers attempt to reduce work to be done at runtime, **restructuring compilers** determine the optimal order to do a computation (hence re-structuring the computation).

For example adding parallelism to loops, or changing the order nesting of loops (e.g to allow for SMID instructions, or better cache locality).

Higher-Level Optimisations

Higher level programming language features require care to be optimised.

- **Subtype Polymorphism**
  Given some $x.f()$ we may be able to determine the method called at compile time (allowing inlining, removing loads required to go to x's **Method Lookup Table**) or be able to inline selection (if statements/switches) to determine what the type is and run code accordingly.
- **Pattern Macthing**
  A feature of languages such as Haskell, Prolog and Rust. Compile should determine the optimal order of tests (that still respects the language semantics - i.e matching order)
- **Memory Management**
  Managing heap/dunamic variables, either not doing management (left to the programmer - C), compiler using programmer provided information (and inferring as in Rust), or garbage collectors at runtime.
- **Lazy Evaluation**
  Moving evaluations of expressions to their uses (so no redundant computation is done), or constructing closures to represent complex computations.
- **Arrays**
  Optimising overloads (e.g + to concatenate arrays), and the underlying structure (e.g determining optimal array/list structure at compile time for the programmer). Allowing for efficient access to arrays containing types (e.g arrays of objects - array of pointers, or array of structs).

  Allowing array slicing (e.g $A[3..10]$) without requiring lots of copying of data, or restricting usage of the array the slice is taken from.