Optimizing Compiler Notes

Clemens Koza, 01126573

Contents

Revision history 3
About this document 3
1. The DFA framework 4
2. Gen/Kill analysis specification 4
2.1. Reaching definitions 4
2.2. Available Expressions 5
2.3. Live variables 5
2.4. Very busy expressions 5
3. Constant propagation 6
3.1. Data domain & extended domain 6
3.2. Variables, terms and operators 7
3.3. (Extended) interpretation, evaluation function, state transformer 7
3.4. Simple constants 8
3.5. Copy constants 8
3.6. Linear constants 8
3.7. Q constants (Kam & Ullman) 9
3.8. Finite constants 9
3.9. Conditional constants 9
3.10. Value graph approach 9
4. Partial redundancy elimination (PRE) 10
4.1. Code motions and admissibility 10
4.1.1. Safety example 11
4.2. Busy code motion (BCM) 11
4.3. Lazy code motion (LCM) 11

Revision history

- no “published” version yet.

About this document

This document is based on Uwe Egly’s Optimizing Compilers lecture at Vienna University of Technology; particularly from taking it in the 2022/23 winter term and refreshing my knowledge on it mid-2023. Corrections and additions are welcome as pull requests at

https://github.com/SillyFreak/optimizing-compilers-notes

This document leaves out several details and was written primarily for me, but I hope it is useful for other people as well. Among the things one reading it should keep in mind:

- This is not a replacement to Prof. Egly’s slides or the lecture in any way. Much of the information may require context and/or preliminaries found in the slides or in the lecture.
- I liberally skip proofs. They make up a large part of the professor’s slides, and I don’t think it would be very useful to repeat them here.
- In general, I skip formal definitions where they were basic enough for me; for example, I gloss over structure and semantics of programs, which are important for most program analyses. Your mileage may vary.
- I skip chapters 1 (Motivation) and 2 (Classical Gen/Kill Data Flow Analyses) in favor of starting with the DFA framework, which subsumes that material anyway.
- The document is in general still incomplete.
- I am fallible and make errors or overlook some points’ importance.

If you have questions, feel free to reach out on Github. I may at least occasionally be motivated enough to answer questions, extend the document with explanations based on your question, or help you with adding it yourself.

1. The DFA framework

In the general data flow analysis framework, a DFA problem on an edge-labelled flow graph G is formulated through a lattice with corresponding local semantics and some additional meta-information. The lattice is formed in some *carrier set*, $\mathcal{C}$:

$$\mathcal{S}_G = \left(\hat{\mathcal{C}}, [\![\ ]\!], c_s, d\right)$$
$$\text{where } \hat{\mathcal{C}} = (\mathcal{C}, \sqsubseteq, \sqcap, \sqcup, \bot, \top)$$
$$[\![\ ]\!] : E \to \mathcal{C} \to \mathcal{C}$$
$$c_s \in \mathcal{C}$$
$$d \in \{fw, bw\}$$

$[\![\ ]\!]$ is the DFA functional that returns for each edge a function defining the semantics of that edge. For example, if $\imath_e = \text{skip}$, then $[\![e]\!] = Id_{\mathcal{C}} = \lambda c.\, c$. For other instructions, the semantics will vary by DFA specification. Whether the DFA functional is (for each edge) only monotonic or also distributive determines whether the MaxFP solution will only approximate or even coincide with the MOP solution.

d is the direction of the problem, and c_s is the initial information at the start/end node, depending on the direction.

2. Gen/Kill analysis specification

Gen/kill or bitvector analyses store a single bit of information for each node of the control flow graph, i.e. $\mathcal{C} := \mathbb{B}$. The term *bitvector* analysis reflects that n such analyses (e.g. the availability of n expressions) can be carried out efficiently at the same time by combining boolean values into a bitvector, resulting in $\mathcal{C} := \mathbb{B}^n$. When doing so, the relevant connectives (e.g. $\leq$, $\wedge$) are adapted to point-wise application (e.g. $\leq_{pw}$, $\wedge_{pw}$). These two formulations are ultimately equivalent.

Depending on the quantification of the problem, the boolean lattice or its inverse is used:

$$\hat{\mathbb{B}} := (\mathbb{B}, \wedge, \vee, \leq, \text{false}, \text{true}) \quad \text{for universally quantified problems}$$
$$\widehat{\mathbb{B}_\vee} := (\mathbb{B}, \vee, \wedge, \geq, \text{true}, \text{false}) \quad \text{for existentially quantified problems}$$

Hasse diagram of $\hat{\mathbb{B}}$ Hasse diagram of $\widehat{\mathbb{B}_\vee}$

The DFA functional and direction depend on the specific analysis. In the DFA functionals, we will refer to some basic boolean functions:

$$Cst_{\text{true}} = \lambda b.\, \text{true}$$
$$Cst_{\text{false}} = \lambda b.\, \text{false}$$
$$Id_{\mathbb{B}} = \lambda b.\, b$$

2.1. Reaching definitions

A defintion of some variable v is an assignment to that variable at a specific edge $\hat{e}$. The reach of a definition flows forward in the flow graph, until another definition shadows it.

$$\mathcal{S}_G = \left(\widehat{\mathbb{B}_\vee}, [\![\ ]\!], b_s, fw\right)$$

$$\text{where } [\![e]\!] = \begin{cases} Cst_{\text{true}} & \text{if } e = \hat{e}\text{, i.e. this is the site of the definition} \\ Cst_{\text{false}} & \text{if } \imath_e \text{ redefines } v \\ Id_{\mathbb{B}} & \text{otherwise} \end{cases}$$

2.2. Available Expressions

An expression becomes available when it is computed and stops being available when a variable appearing in it is redefined. This is again a forward analysis. An expression needs to be available on all paths to be available at a node.

$$\mathcal{S}_G = \left(\hat{\mathbb{B}}, [\![\ ]\!], b_s, fw\right)$$

$$\text{where } [\![e]\!] = \begin{cases} Cst_{\text{false}} & \text{if } \imath_e \text{ redefines a variable appearing in the expression} \\ Cst_{\text{true}} & \text{if } \imath_e \text{ computes the expression} \\ Id_{\mathbb{B}} & \text{otherwise} \end{cases}$$

2.3. Live variables

A variable v is live at a node if it is later used on some paths before being redefined. The information flows backwards from the use sites, until a definition site is encountered.

$$\mathcal{S}_G = \left(\widehat{\mathbb{B}_\vee}, [\![\ ]\!], b_e, bw\right)$$

$$\text{where } [\![e]\!] = \begin{cases} Cst_{\text{true}} & \text{if } \imath_e \text{ reads/uses } v \\ Cst_{\text{false}} & \text{if } \imath_e \text{ redefines } v \\ Id_{\mathbb{B}} & \text{otherwise} \end{cases}$$

2.4. Very busy expressions

An expression is very busy at a node if, no matter what path is taken from the node, the expression is always used before any of the variables occurring in it is redefined. The information flows backwards from the use sites, until a definition site of one of the variables in the expression is encountered.

$$\mathcal{S}_G = \left(\hat{\mathbb{B}}, [\![\ ]\!], b_e, bw\right)$$

$$\text{where } [\![e]\!] = \begin{cases} Cst_{\text{true}} & \text{if } \imath_e \text{ computes/uses the expression} \\ Cst_{\text{false}} & \text{if } \imath_e \text{ redefines a variable appearing in the expression} \\ Id_{\mathbb{B}} & \text{otherwise} \end{cases}$$

3. Constant propagation

We now want to know whether a variable (or term) has, at some node, a known constant value. Both the variable and term version are undecidable, so we need conservative approximations.

Let's consider a few examples of pseudocode. In each example, we are interested in whether the printed variable is constant:

```
a = 0          a = 0          if p: a = 0    if p: a = 0      b = a * 0
b = a          b = a + 1      else: a = 0    else: return     print(b)
print(b)       print(b)       print(a)       print(a)
```

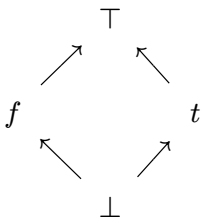
In the first example, `a` is constant and `b` is assigned this variable's value. Naturally, we want an analysis that recognizes that `b` has the constant value zero. In the second example, we are dealing with an arithmetic expression that contains only constant operands. Again, it is desirable to recognize `b` always has value one.

The third and fourth example demonstrates how joining control flow needs to be considered: as variable `a` is known to be *the same* constant in both branches of example 3, its value after the conditional is that same value. As one branch *diverges* in example 4, the other branch(es) determine the value after the conditional.

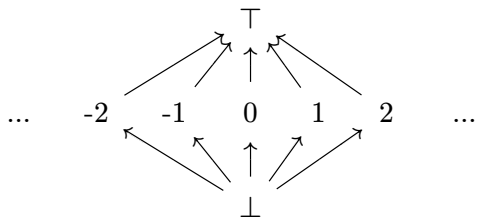
The last example shows how certain operations can result in constant results with non-constant operands: `a` is not known, yet `b` is known to be zero.

3.1. Data domain & extended domain

To reason about values that variables/terms can take, we first need a data domain $\mathbb{D}$ (for which we require a distinguised element $\bot \in \mathbb{D}$). While in $\widehat{\mathbb{B}}$ and $\widehat{\mathbb{B}_\vee}$, the two values true and false are related through $\sqsubseteq$, for constant propagation, they have no meaning except for being distinct. We therefore work on a *flat lattice* $\mathcal{FL}_\mathbb{D} = (\mathbb{D}', \sqsubseteq, \sqcap, \sqcup, \bot, \top)$, with the extended domain $\mathbb{D}' = \mathbb{D} \cup \{\top\}$ containing another distinguished element:



Hasse diagram of $\mathcal{FL}_\mathbb{B}$



Hasse diagram of $\mathcal{FL}_\mathbb{Z}$

We can now talk about *states*, which assign a value to each variable (this is why we require $\bot \in \mathbb{D}$: even variables not defined at some program points need to be covered). A state is thus a function $\sigma : \mathbf{V} \to \mathbb{D}$. We also define DFA states that allow all values of the extended domain: $\sigma : \mathbf{V} \to \mathbb{D}'$ and the sets of all (extended) states:

$$\Sigma = \{\sigma \mid \sigma \in \mathbf{V} \to \mathbb{D}\}$$
$$\Sigma' = \{\sigma \mid \sigma \in \mathbf{V} \to \mathbb{D}'\}$$

This latter set is significant because we can construct a lattice on it by *pointwise ordering* according to the ordering of $\mathcal{FL}_{\mathbb{D}'}$:

$$\sigma \sqsubseteq_{\Sigma'} \sigma' \quad \text{iff } \forall v \in \mathbf{V} : \sigma(v) \sqsubseteq_{\mathcal{FL}_{\mathbb{D}'}} \sigma'(v)$$

Or in words: two DFA states are related according to the lattice's ordering iff all variable assignments in these states are related. This results in pointwise meet and join operations as well, and two distinguished top and bottom states:

$$\begin{aligned}\sigma \sqcap_{\Sigma'} \sigma' &= \lambda v.\, \sigma(v) \sqcap_{\mathcal{FL}(\mathbb{D})} \sigma'(v) \\ \sigma \sqcup_{\Sigma'} \sigma' &= \lambda v.\, \sigma(v) \sqcup_{\mathcal{FL}(\mathbb{D})} \sigma'(v) \\ \sigma_\bot &= \lambda v.\, \bot \\ \sigma_\top &= \lambda v.\, \top\end{aligned}$$

$$\widehat{\Sigma'} = (\Sigma', \sqsubseteq, \sqcap, \sqcup, \sigma_\bot, \sigma_\top)$$

Consider some control flow examples again in light of this flat lattice:

```
if p: a = true      if p: a = true      if p: a = true
else: a = false     else: a = true      else: return
print(a)            print(a)            print(a)
```

Before exiting the conditional, we have two states σ_1 and σ_2, and right after a state $\sigma_3 = \sigma_1 \sqcap \sigma_2$. According to the lattice, we get

$$\begin{aligned}&\text{Example 1:} && \sigma_1(a) = \text{true},\ \sigma_2(a) = \text{false} && \Rightarrow \sigma_3(a) = \bot \\ &\text{Example 2:} && \sigma_1(a) = \text{true},\ \sigma_2(a) = \text{true} && \Rightarrow \sigma_3(a) = \text{true} \\ &\text{Example 3:} && \sigma_1(a) = \text{true},\ \sigma_2(a) = \top && \Rightarrow \sigma_3(a) = \text{true}\end{aligned}$$

We can see that $\bot$ means the value is unknown (not constant) and $\top$ means the value does not exist (due to the divergence of the branch).

Since the $\top$ value only represents impossibilities, we define another set ${\Sigma'}_{\text{Init}}$ excluding those states that contain $\top$ variable assignments:

$${\Sigma'}_{\text{Init}} = \{\sigma \in \Sigma' \mid \forall v \in \mathbf{V}.\, \sigma(v) \neq \top\}$$

3.2. Variables, terms and operators

It is no longer sufficient (as it was for bitvector analyses) to just look at *what* variables appear in expressions; to know whether an expression is constant, and if so what contstant it evaluates to, we need to consider *how* an expression combines variables. We therefore consider the disjoint sets $\mathbf{V}$ of variables, $\mathbf{C}$ of constants, $\mathbf{O}$ of operators, and $\mathbf{T}$ of terms. The set of terms contains the syntaxtically possible combinations of the other three. On top of these, the semantics can be defined.

3.3. (Extended) interpretation, evaluation function, state transformer

The (extended) interpretation I_0 (${I_0}'$) describe the semantics of constants and operators; they can be thought of as "overloaded" for these two kinds of parameters:

$$I_0 : \begin{cases}\mathbf{C} \to \mathbb{D} \\ \mathbf{O} \to (\mathbb{D}^k \to \mathbb{D}) & \text{for a } k\text{-ary operator}\end{cases}$$

$${I_0}' : \begin{cases}\mathbf{C} \to \mathbb{D}' \\ \mathbf{O} \to (\mathbb{D}'^k \to \mathbb{D}') & \text{for a } k\text{-ary operator}\end{cases}$$

These functions satisfy that

- if some of the operands of an operator are $\bot$, the operation's result is $\bot$; and
- (for the extended interpretation) otherwise, if some of the operands of an operator are $\top$, the operation's result is $\top$.

Based on that, the (extended) evaluation function $\mathcal{E}$ ($\mathcal{E}'$) is defined:

$$\mathcal{E} : \mathbf{T} \to (\Sigma \to \mathbb{D})$$
$$\mathcal{E}' : \mathbf{T} \to (\Sigma' \to \mathbb{D}')$$

Intuitively, these functions recursively apply themselves to parts of the expression, grounding themselves in I_0 ($I_0{}'$) for constants and operators, and in the current state for variables.

The exact form of these functions depends on the kind of constant propagation. For example, the term `a * 0` is not a simple constant, but it is a linear constant.

Likewise, the (extended) state transformer Θ (Θ') transforms an (extended) state into another based on the local semantics of an instruction:

$$\Theta : \Sigma \to \Sigma$$
$$\Theta' : \Sigma' \to \Sigma'$$

With this, we can finally formulate a DFA specification for constant propagation:

$$\mathcal{S}_G = \left(\widehat{\Sigma'}, [\![\ ]\!], \sigma_s, fw\right)$$
$$\text{where } [\![e]\!] = \Theta'_{\iota_e}$$
$$\sigma_s \in \Sigma'_{\text{Init}}$$

The difference between constant propagation specifications lies in how the extended evaluation function $\mathcal{E}'$ deals with different kinds of terms:

3.4. Simple constants

The simple constant analysis evaluates terms containing only constants and constant variables to non-$\perp$ values:

$$\mathcal{E}'(t)(\sigma) = \begin{cases} \sigma(v) & \text{if } t \equiv v \in \mathbf{V} \\ I_0{}'(c) & \text{if } t \equiv c \in \mathbf{C} \\ I_0{}'(\text{op})(\mathcal{E}'(t_1)(\sigma), ..., \mathcal{E}'(t_k)(\sigma)) & \text{if } t \equiv (\text{op}, t_1, ..., t_k) \end{cases}$$

3.5. Copy constants

The copy constant analysis only deals with terms that are either constants themselves, or are just a variable reference. Other terms are assumed to be not constant. In other words, *copying* one variable value into another variable is the only kind of constant *propagation* that is performed:

$$\mathcal{E}'(t)(\sigma) = \begin{cases} \sigma(v) & \text{if } t \equiv v \in \mathbf{V} \\ I_0{}'(c) & \text{if } t \equiv c \in \mathbf{C} \\ \perp & \text{otherwise} \end{cases}$$

3.6. Linear constants

The linear constants analysis limits itself to linear arithmetic terms: it considers terms of the form $c * v \oplus d$ (with $c, d \in \mathbf{C}$, $v \in \mathbf{V}$, $\oplus \in \{+, -\}$) specifically (as well as other semantically equivalent forms such as $d \triangleq 0 * v + d$, $c * v \triangleq c * v + 0$, etc.):

$$\mathcal{E}'(t)(\sigma) = \begin{cases} \sigma(v) & \text{if } t \equiv v \in \mathbf{V} \\ \mathcal{E}_{\text{SC}}(t)(\sigma) & \text{if } t \equiv c * v \oplus d \\ \perp & \text{otherwise} \end{cases}$$

where $\mathcal{E}_{\text{SC}}$ is the extended evaluation function for simple constants.

3.7. Q constants (Kam & Ullman)

The Q constants analysis does not use a different evaluation function; instead, the MaxFP algorithm is modified slightly. Consider these code examples and a simple constant analysis on them:

```
if p:
  a = 2
  b = 3
else:
  a = 2
  b = 3
c = a + b
print(c)
```

```
if p:
  a = 2
  b = 3
else:
  a = 3
  b = 2
c = a + b
print(c)
```

In the first example, both a and b are constants after the conditional, but in the first example they are not; $a + b$, however, is a constant not detected by the simple constant analysis. Recall:

- Before exiting the conditional, we have σ_1 with $a = 2, b = 3$ and σ_2 with $a = 3, b = 2$.
- We compute $\sigma_3 = \sigma_1 \sqcap \sigma_2$ and find that in this state, $a = \bot, b = \bot$.
- We evaluate $\mathcal{E}'(a + b)(\sigma_3) = \bot$.

The Q constants approach is now to perform the meet operation later:

- Before exiting the conditional, we have σ_1 with $a = 2, b = 3$ and σ_2 with $a = 3, b = 2$.
- We evaluate $\mathcal{E}'(a + b)(\sigma_1) = 5, \mathcal{E}'(a + b)(\sigma_2) = 5$.
- We compute the result $5 \sqcap 5 = 5$ and thus detected the result as a constant.

Taking the meet "lazily" after the evaluation results in more evaluations overall, but improves precision. This approach can be generalized to taking the meet more than one step later.

3.8. Finite constants

In contrast to the CP analyses so far, the finite constants analysis is based on *terms*: the problem illustrated for Q constants was that while previous analyses kept track of variable values (e.g. of a and b), the value of terms such as $a + b$ was not remembered. If we stored $a + b = 5$ as part of the information at the end of both branches, the MaxFP algorithm would naturally preserve and forward that information.

There are infinitely many possible terms, but fortunately, only a finite set of them needs to be considered (why?), leading to the name *finite constants*.

3.9. Conditional constants

In previous examples, the predicate variable p was left as an unknown; if however a condition is itself constant, this means we don't need to join control flows and can avoid the lossy meet operation at the end altogether.

To do so, we need to extend our domain to include boolean values: $\mathbb{D}_{\mathbb{B}} = \mathbb{D} \cup \mathbb{B}$, and extend our definitions to allow for comparisons and logical operators, so that boolean terms can be formed. On top of this, we use the flat lattice $\mathcal{FL}_{\mathbb{D}_{\mathbb{B}}}$ and state lattice $\widehat{\Sigma}'$

3.10. Value graph approach

TODO

4. Partial redundancy elimination (PRE)

PRE is an optimization that avoids recomputing the same value multiple times. The tool for this is moving computations to one or more earlier points in the flow graph (particularly outside a loop the original computation appeared in). Whenever these stored results are used, none of the variables used in the computation must have changed in the meantime ("correctness"). The three goals which can be traded are

- performance: minimize the number of computations done at runtime
- register pressure: minimize the amount of time a result needs to be held in a register
- code size: minimize the number of instructions in the flow graph

Classically, there is an additional requirement for the code motion: no program path that originally did not compute the value may be made to compute the value after code motion. This is called "safety", because the extra computation could result in a failure even though the branch originally containing the failing code wasn't taken.

Trading these in different ways leads to a taxonomy of multiple code motion strategies, for example:

- **Busy code motion** (BCM) moves calculations to the earliest opportunity. It is computationally optimal.
- **Lazy code motion** (LCM) moves calculations to the latest opportunity. It is computationally and lifetime optimal.
- **Sparse code motion** (SpCM) moves calculations to the latest position where the resulting program is code size optimal. Computational and lifetime optimality is not achieved, but no program of the same size is better.

4.1. Code motions and admissibility

A code motion transformation (CM) of some candidate expression $t \in \mathbf{T}$ on a *node-labelled* flow graph is defined by

- $Insert_{CM}(n)$: whether a computation of the form $h := t$, where h is a new variable name, is inserted at the entry of node n;
- $Repl_{CM}(n)$: whether occurrences of t are replaced by references to h at n.

Obviously, not all CMs lead to correct code, thus we define admissibility: we want CMs that satisfy the following properties:

- Safety: a computation is only inserted on paths where the original program also computed the term. That means an insertion only happens at a node n where the term is either available (= "up-safe": it has been computed without redefinition in the original program on all paths leading to n), or very busy (= "down-safe": it will be used without redefinition on all paths coming from n).
- Correctness: whenever h is used, there is a definition of h before that and h's operands have not been redefined since then.

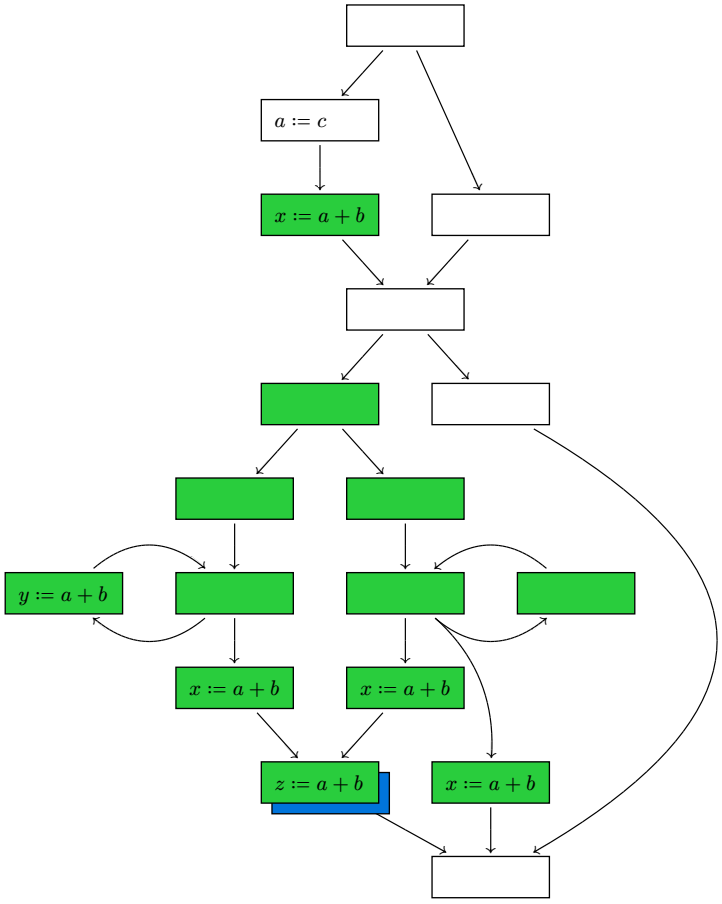
$$\forall n \in N.\, Insert_{CM}(n) \Rightarrow Safe(n)$$
$$\forall n \in N.\quad Repl_{CM}(n) \Rightarrow Correct_{CM}(n)$$

$$\text{where}\quad Safe(n) = Available(n) \vee VeryBusy(n)$$
$$Correct_{CM}(n) = \forall \langle n_1, ..., n_k \rangle \in \mathbf{P}[s, n].\, \exists i.$$
$$Insert_{CM}(n_i) \wedge Transp^{\forall}(\langle n_i, ..., n_{k-1} \rangle)$$

Where $Transp(n)$ means that t's operands are not redefined at n and $Transp^{\forall}(p)$ extends that to apply to all nodes on path p.

4.1.1. Safety example

What follows is a node-labelled flow graph where the up- and down safety of the expression $a + b$ is highlighted in blue and green, respectively. The highlighted nodes are thus the ones where computations of the form $h := a + b$ may be inserted.



4.2. Busy code motion (BCM)

BCM uses the earliestness principle to decide where to put computations, then eliminates redundant computations (i.e. those that can be replaced by referencing the new variable, e.g. h). Earliestness is defined as follows:

$$Earliest(n) = Safe(n) \wedge \begin{cases} \text{true} & \text{if } n \text{ is the start node} \\ \bigvee_{m \in pred(n)} \neg\, Transp(m) \vee \neg\, Safe(m) & \text{otherwise} \end{cases}$$

The BCM transformation is thus defined by

$$Insert_{BCM}(n) = Earliest(n)$$
$$Repl_{BCM}(n) = Comp(n)$$

BCM is computationally optimal, but maximizes register pressure. Code size can grow.

4.3. Lazy code motion (LCM)

LCM is both computationally and lifetime optimal. It moves computations to the latest possible point that still results in the minimum number of computations.

We first need the notion of delayability: a computation (of t) can be delayed to node n when all paths to n inlcude the earliest node for the computation, and these paths do not include a computation between the earliest node (inclusive) and n (exclusive).

$$\begin{aligned} Delayed(n) = \forall p \in \mathbf{P}[s,n].\, \exists i \leq \lambda_p.\\ Earliest(p_i) \wedge \neg Comp^{\exists}(p\lbrack i, \lambda_p[) \end{aligned}$$

Similarly to $Earliest$, we now define $Latest$:

$$Latest(n) = Delayed(n) \wedge \left(Comp(n) \vee \bigvee_{m \in\, succ(n)} \neg\, Delayed(m) \right)$$

This leads is to a precursor of LCM: the ALCM transformation is computationally and *almost* lifetime optimal.

$$\begin{aligned} Insert_{ALCM}(n) &= Latest(n)\\ Repl_{ALCM}(n) &= Comp(n) \end{aligned}$$

As a simple example for how this is not lifetime optimal yet, take the simple example $x := a + b$. While there is no improvement to achieve, ALCM would still hoist the computation to directly before the original assignment: $h := a + b; x := h$. Note that the "uselessness" of the hoisting depends on the CM transformation being applied.

We thus need to further consider a notion of *isolatedness* regarding a specific CM transformation: a node n is isolated if all paths from there to the end node that contain a replacement of t at some later node p_i satisfy that a computation was inserted at some node between n (exclusive) and p_i (inclusive).

We are specifically interested in isolatedness relative to the BCM transformation:

$$\begin{aligned} Isolated_{CM}(n) = \forall p \in \mathbf{P}[n,e].\, \forall 1 < i \leq \lambda_p.\\ Repl_{CM}(p_i) \Rightarrow Insert^{\exists}_{CM}(p]1,i]) \end{aligned}$$

$$\begin{aligned} Isolated_{BCM}(n) = \forall p \in \mathbf{P}[n,e].\, \forall 1 < i \leq \lambda_p.\\ Comp(p_i) \Rightarrow Earliest^{\exists}(p]1,i]) \end{aligned}$$

A node n is isolated (w.r.t. BCM) if either no path from there computes t (at a later node than n), or if t is computed on some paths, all computations on those paths are preceded by an earliest node that comes after n and before or at the computation.

This allows us to specify the LCM transformation:

$$\begin{aligned} Insert_{LCM}(n) &= Latest(n) \wedge \neg Isolated_{BCM}(n)\\ Repl_{LCM}(n) &= Comp(n) \wedge \neg(Latest(n) \wedge Isolated_{BCM}(n)) \end{aligned}$$