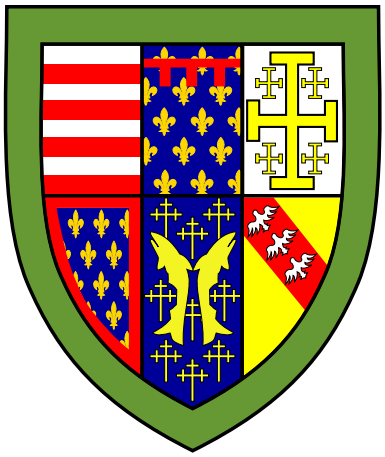
Queens' College Cambridge

Optimizing Compilers



Alistair O'Brien

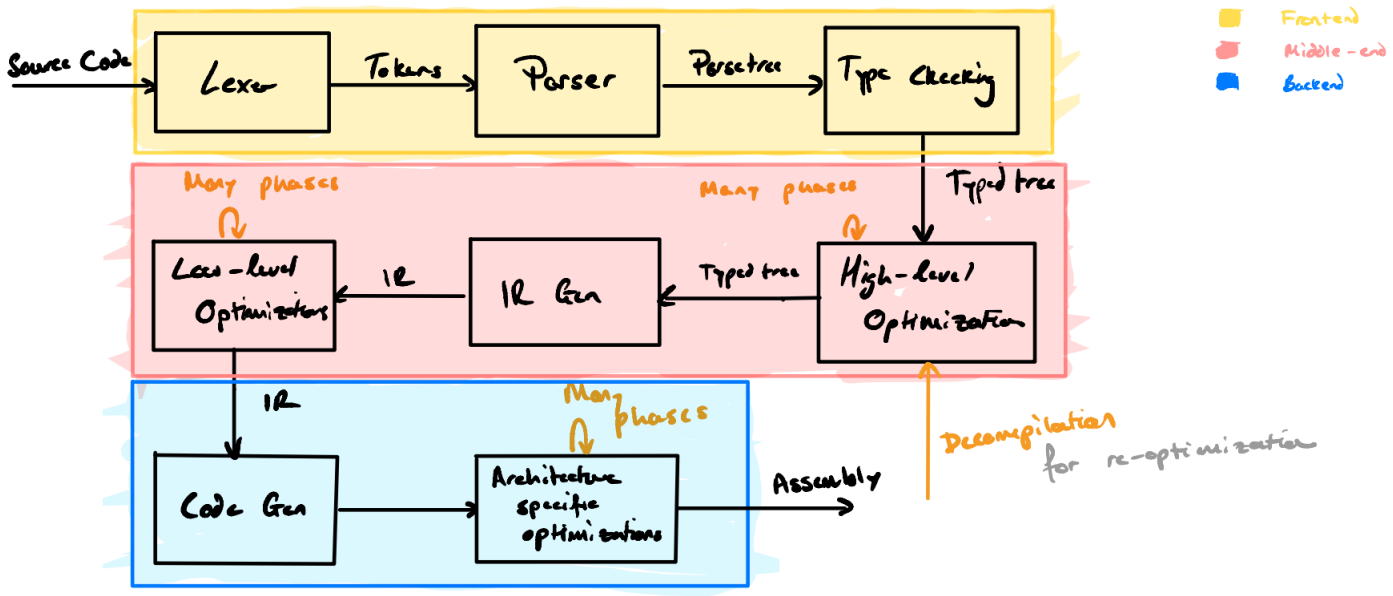
Department of Computer Science

May 31, 2022

Contents

1 Overview

- **Compiler Pipeline**:



- **Optimization** = **Analysis** + **Transformation**
  Analysis finds some property which justifies the **safety** of the *transformation*.

- **Goal**: Programmers write simple, maintainable, portable code. Compiler specializes code for machine, making it faster, etc.

- **Phase Ordering**: Determining order of optimization phases is difficult. Some optimizations work well with others (e.g. CSE + Copy Propagation), others don't (requiring multiple runs).

2 High-Level Optimizations

- High-level optimizations are performed on the *syntax tree* (often used in functional languages).

Definition 2.0.1. (**STLC** $\lambda^{\to}$) The syntax of the simply-typed lambda calculus $\lambda^{\to}$ is given by:

$$\begin{aligned} e ::= {} & x \mid e\ e \mid \lambda x.e && \text{(Classical } \lambda \text{ calculus)} \\ & \mid \rho \mid c && \text{(Primitives and constants)} \\ & \mid \mathsf{let}\ x = e\ \mathsf{in}\ e && \text{(Let bindings)} \\ & \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e && \text{(Control flow)} \end{aligned}$$

- **Many** high-level optimizations are based on the analysis of the simply-typed lambda calculus.
- **Peephole** optimizations are simple syntactic rewriting rules e.g.

$$\begin{aligned} e + 0 &\rightsquigarrow e \\ (e + m) + n &\rightsquigarrow e + (m + n) \\ \mathsf{let}\ x = e\ \mathsf{in}\ \mathsf{if}\ e'\ \mathsf{then}\ C[x]\ \mathsf{else}\ e'' & \\ &\rightsquigarrow \mathsf{if}\ e'\ \mathsf{then}\ \mathsf{let}\ x = e\ \mathsf{in}\ C[x]\ \mathsf{else}\ e'' \quad \text{(for lazy/pure language)} \end{aligned}$$

  Safety of these transformations is determined by contextual equivalence (See IB Semantics).

- **Strength Reduction**:
  Idea: replace expensive operations with cheaper ones e.g.

$$\begin{aligned} x * 2 &\rightsquigarrow x + x \\ x/2 &\rightsquigarrow x \gg 1 \end{aligned}$$

  This form of rewriting is very effective within *loops*.
  Algorithm (for loops):

1. Find the induction variables i, j s.t $i := i \oplus e$ (updated by loop), where $i, j \notin \mathsf{fv}(e)$. Find j s.t $j := e_2 \oplus e_1 \otimes i$ (used within loop), $i, j \notin \mathsf{fv}(e_1, e_2)$.
2. Assuming distributivity between $\oplus, \otimes$: $x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$. Move the assignment to the loop entry: $j := e_2 \oplus e_1 \otimes i$ and add end-of-loop assignment $j := j \oplus (e_2 \oplus e)$.

   Distributivity ensures *safety* of the transformation.

For example:

```
int *v = ...;
for (i = 0; i < 100; i++)
  v[i] = 0; // *(v + i * sizeof(int)) = 0
```

The induction variable i is `i` and j is the expression `v + i * sizeof(int)`. So it is rewritten to:

```
int *v = ...;
for (int *p = v; p < v + 100; p++)
  *p = 0;
```

Problem: Obfuscates the code (and it's intent) $\implies$ decompilation is harder.

2.1 Abstract Interpretation

- **Problem**: Dynamic behavior is infeasible to determine statically (non-termination, etc).
- **Solution**: Execute a *simplified* version of the computation w/ safety property for P:

  P holds in abstract domain $\implies$ P holds in concrete domain

Definition 2.1.1. (**Abstract Interpretation**) Let $D, D^\sharp$ be two arbitrary domains, denoting the *concrete* and *abstract* domains respectively. Let $\mathsf{abs} : D \to D^\sharp$ be the abstraction function.

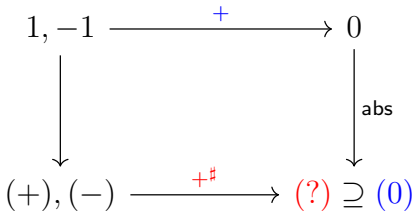
For a syntax $M \in \mathbb{T}$, the concrete and abstract semantics are denoted $[\![\cdot]\!] : \mathbb{T} \rightharpoonup D$ and $[\![\cdot]\!]^\sharp : \mathbb{T} \rightharpoonup D^\sharp$. The *abstract interpretation* of a term $M \in \mathbb{T}$ is given by $[\![\cdot]\!]^\sharp \in D^\sharp$, with the *safety property*:

$$\mathsf{abs}([\![M]\!]) \sqsubseteq [\![M]\!]^\sharp$$

- Intuitively, safety property implies abstract interpretation is an over-approximation of the concrete abstraction.
- **Example**: Addition on integers. $D = \mathbb{Z}$. $D^\sharp = \{(+), (-), (0), (?)\}$. Abstract interpretation:

$+^\sharp$	$(+)$	$(-)$	(0)	$(?)$
$(+)$	$(+)$	$(?)$	$(+)$	$(?)$
$(-)$	$(?)$	$(-)$	$(-)$	$(?)$
(0)	$(+)$	$(-)$	(0)	$(?)$
$(?)$	$(?)$	$(?)$	$(?)$	$(?)$

  Safety property requires that $d \in D^\sharp . d \sqsubseteq (?)$.



- **Example**: Liveness analysis. Live sets are abstract values. Control flow is overapproximated in abstract domain.

2.1.1 Strictness Analysis

- **Idea** : Have a call-by-name language, want to determine parameters that can be passed by value.
  Call-by-value is **more efficient** (avoids constructing thunks, which are allocated on the heap).

Definition 2.1.2. (**Strictness**) A function $f : D_\perp^n \to D_\perp$ is *strict* in x_i (ith parameter) iff

$$\forall (d_j)_{1 \le i \ne j \le n} \in D_\perp . f(d_1, \ldots, d_{i-1}, \perp, d_{i+1}, \ldots, d_n) = \perp$$

- Language is defined by the grammar:

$$\begin{aligned} p &::= \overline{F(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = e} && \text{(Program)} \\ e &::= x_i \mid \rho(e_1, \ldots, e_n) \mid F(e_1, \ldots, e_n) && \text{(Expressions)} \\ \tau &::= \ldots && \text{(Types)} \end{aligned}$$

  p is a program, a sequence of mutually-recursive first-order functions. Operationally, a function F is strict in x_i iff forall $e_1, \ldots, e_n \in \Lambda$

$$e_i \text{ diverges} \implies F(e_1, \ldots, e_n) \text{ diverges}$$

- **Abstract Interpretation**:
  Domain $D \supseteq \{\bot\}$ (Domain must be flat).
  Abstract Domain $D^\sharp = \mathbf{2} = \{0, 1\}$. 0 = definitely diverges, 1 = may/-may not diverge (possible termination).

  Abstract interpretation defines the interpretations for the primitives $\rho^\sharp : \mathbf{2}^n \to \mathbf{2}$ satisfying the property:

$$\begin{aligned} &\rho^\sharp(x_1, \ldots, x_n) = 0 \\ &\iff \forall (d_i)_{i \geq 0} \in D. (\forall 1 \leq i \leq n. x_i = 0 \implies d_i = \bot) \implies \rho(d_1, \ldots, d_n) = \bot \end{aligned}$$

  For example:

$$\begin{aligned} \mathsf{if}^\sharp(b, x, y) &= b \wedge (x \vee y) \\ \mathsf{eq}^\sharp(x, y) &= x \wedge y \end{aligned}$$

  Negation cannot occur in boolean expressions since it would equate to: $\neg x$ terminates $\iff$ x doesn't terminate (Halting problem).

$$\begin{aligned} [\![\Gamma; \Omega \vdash x]\!]^\sharp (\theta^\sharp, \sigma^\sharp) &= \theta^\sharp(x) \\ [\![\Gamma; \Omega \vdash \rho(e_1, \ldots, e_n)]\!]^\sharp (\theta^\sharp, \sigma^\sharp) &= \rho^\sharp(\overline{[\![\Gamma; \Omega \vdash e_i]\!]^\sharp (\theta^\sharp, \sigma^\sharp)}) \\ [\![\Gamma; \Omega \vdash F(e_1, \ldots, e_n)]\!] &= \sigma^\sharp(F)(\overline{[\![\Gamma; \Omega \vdash e_i]\!] (\theta^\sharp, \sigma^\sharp)}) \\ [\![\Omega \vdash F(x_1, \ldots, x_n) = e]\!] (\sigma^\sharp) &= \lambda x_1^\sharp, \ldots, x_n^\sharp \in \mathbf{2}. [\![x_1, \ldots, x_n; \Gamma \vdash e]\!] ([x_i \mapsto x_i^\sharp], \sigma^\sharp) \\ \left[\!\!\left[\Omega \vdash \overline{F(x_1, \ldots, x_n) = e}\right]\!\!\right] (\sigma^\sharp) &= \overline{F \mapsto [\![\Omega \vdash F(x_1, \ldots, x_n) = e]\!] (\sigma^\sharp)} \\ [\![p]\!] &= fix\, ([\![\Omega \vdash p]\!]) \end{aligned}$$

- Interpretation of (user-defined) functions F is $F^\sharp : \mathbf{2}^n \to \mathbf{2}$ satisfying:

  $F^\sharp(x_1, \ldots, x_n) = 0$
  $\implies \forall (d_i)_{i \geq 0} \in D.(\forall 1 \leq i \leq n.x_i = 0 \implies d_i = \bot) \implies F(d_1, \ldots, d_n) = \bot$

  Note difference between ρ ($\iff$) and weaker F ($\implies$).

- Solutions for abstract interpretations of a program p, written $[\![p]\!]$ or $F^\sharp$, computing using *least fixed point*:

```
(* Over-approximation, everything initially diverges *)
for i = 1 to n do
  F♯.(i) ← λ_.0
done;
while (F♯ changes) do
  for i = 1 to n do
    (* Update strictness function, substituting F♯.(i) for F_i *)
    F♯.(i) ← λx.e_i♯
  done
done
```

- *Safety property*:

$$F^\sharp(1, 1, \ldots, 1, \underbrace{0}_{i\text{th parameter}}, 1 \ldots, 1) = 0 \implies F \text{ is strict in } x_i$$

2.2 Constraint-Based Analysis

- **Problem**: Constraint-based analysis is a framework for solving complex abstract interpretation problems – often problems that often require some form of oracle to formalize (e.g. control-flow analysis).

- **Idea**: Define a mapping $[\![\cdot : \cdot]\!] : e \times \alpha \mapsto C_\tau$, then solve the constraint C_τ to obtain the abstract value of type τ.

  Abstract interpretation is given by: $[\![e]\!]^\sharp = \mathsf{solve}(\exists \alpha.\, [\![e : \alpha]\!] \gg \mathsf{decode}\ \alpha)$.

Definition 2.2.1. (**Constraints with a Value**) The syntax of constraints with a value (or τ-constraints) is defined as:

$$\begin{aligned} C ::= {} & \top \mid \bot \mid C \wedge C \mid \exists \alpha.C && \text{(First-order Logic)} \\ & \mid \mathsf{let}\ \overline{x = \lambda\overline{\alpha}.C}\ \mathsf{in}\ C \mid x\ \overline{\alpha} && \text{(Constraint Abstractions)} \\ & \mid A && \text{(Atomic Constraints)} \\ & \mid \mathsf{map}\ C\ f \mid \mathsf{decode}\ \alpha && \text{(Value operators)} \end{aligned}$$

We write C_τ for a constraint of type τ.

- An τ-constraint C *evaluates* to a value of type τ if C is satisfiable. See Dissertation for Semantics.

2.2.1 0th-order Control Flow Analysis

- **Idea**: Each expression e is associated with a (flow) variable α, the set of values (λ-functions) that e could evaluate to (or "flow" from).

  Flow sets can be used to determine set of functions that could be applied $\implies$ control flow analysis for first-class functions / virtual methods (dynamic dispatch).

- Atomic constraints $A ::= \alpha \subseteq \alpha \mid \alpha \subseteq \alpha \mapsto \alpha \mid \{\lambda x.e : \alpha \mapsto \alpha\} \in \alpha$.

- Constraint bindings are of the form (w/ syntactic sugar def):

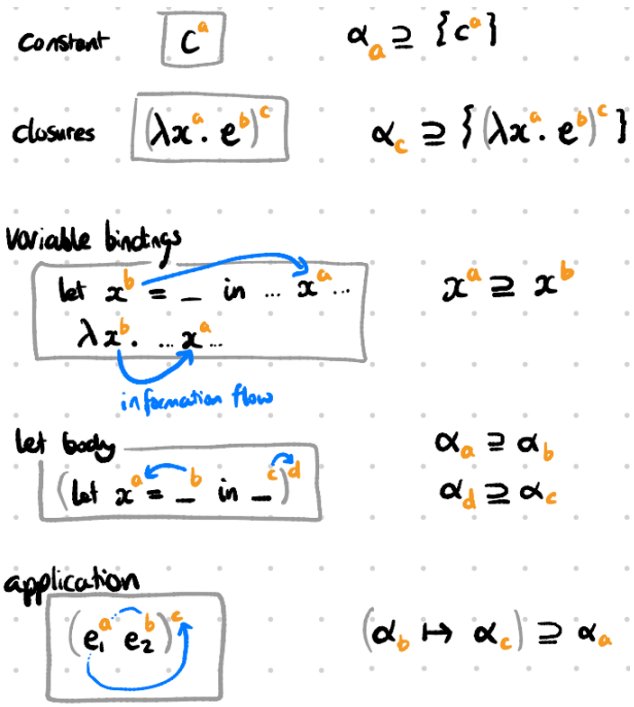
$$\mathsf{def}\ x : \alpha_x\ \mathsf{in}\ C \triangleq \mathsf{let}\ x = \lambda\alpha.\alpha_x \subseteq \alpha\ \mathsf{in}\ C$$

  Semantically $x\ \alpha$ is equivalent to $\alpha_x \subseteq \alpha$.

Definition 2.2.2. (**Constraint generation**) We have the following constraint generation rules for 0th-order CFA (*without value construction*):

$$\begin{aligned} [\![x : \alpha]\!] &= x\ \alpha \\ [\![e_1\ e_2 : \alpha]\!] &= \exists\alpha_1\alpha_2.\ [\![e_1 : \alpha_1]\!] \wedge [\![e_2 : \alpha_2]\!] \wedge \alpha_1 \subseteq \alpha_2 \mapsto \alpha \\ [\![\lambda x.e : \alpha]\!] &= \exists\alpha_1\alpha_2.\mathsf{def}\ x : \alpha_1\ \mathsf{in}\ [\![e : \alpha_2]\!] \wedge \{\lambda x.e : \alpha_1 \mapsto \alpha_2\} \in \alpha \\ [\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \alpha]\!] &= \exists\alpha_1\alpha_2\alpha_3.\alpha_1 \subseteq \alpha_2 \wedge \alpha_3 \subseteq \alpha \wedge [\![e_1 : \alpha_1]\!] \wedge \mathsf{def}\ x : \alpha_2\ \mathsf{in}\ [\![e_2 : \alpha_3]\!] \end{aligned}$$

The constraint generation mapping is explained via notions of "information flow":



- Constraint solving is performed iteratively (solving the least fixed point):
  1. Solving $\alpha_1 \subseteq \alpha_2$: Add all values in α_1 to α_2.
  2. Solving $\alpha_1 \subseteq \alpha_2 \mapsto \alpha_3$: For each value $\lambda x.e : \alpha_4 \mapsto \alpha_5 \in \alpha_1$, solve $\alpha_2 \subseteq \alpha_4$ and $\alpha_5 \subseteq \alpha_3$. (*Think subtyping*).
- Safety property:

$$[\![e]\!]^\sharp = \text{solved flowset } \alpha \supseteq \text{set of values } e \text{ could evaluate to} = [\![e]\!]$$

- **Problem**: Analysis is imprecise. Function's flow variable are shared between all sites (*monovariant*):
- **Solutions**:
  - 1st-order CFA is *polyvariant* – 1 variable per call site.
  - *polymorphic* analysis – 1 flow variable per type.

2.2.2 Andersen's Point-to Analysis

- **Problem**: Find a more precise set of variables that a point could alias. Assumption: "all addresses are taken" with pointer in LVA and Avail is weak.
- **Other uses**:
  - Safe parallelization
  - Improvement of data-flow analysis (LVA, Avail)
- **Solution**: Define a *points to* relation for each variable – the set of location/variables that each variable aliases.
- Andersen's Analysis:
  - Simplified/imprecise
  - Intra-procedural (1 *pt* relation per procedure, as opposed to per statement).
  - Control-flow insensitive
  - No pointer arithmetic / function pointers
  - Conflates `struct` fields e.g. $x.f = e$ and $x.g = e$ are treated the same, equivalent to $x = e$.
- Pointer operations are of the form:

$$
\begin{aligned}
x &:= \mathsf{new}_\ell && \text{(Malloc)} \\
x &:= \mathsf{null} && \text{(Null)} \\
x &:= \&y && \text{(Address of)} \\
x &:= y && \text{(Alias)} \\
x &:= *y && \text{(Field access)} \\
*x &:= y && \text{(Field mutation)}
\end{aligned}
$$

- **Abstract values**: $V = \{x : x \in \mathsf{vars}\} \cup \{\mathsf{new}_\ell : \ell \in \mathsf{locs}\} \cup \{\mathsf{null}\}$. *Point-to* relation is a function $pt : \mathsf{vars} \to V$.

- **Atomic Constraints**:

$$A ::= pt(x) \subseteq pt(x) \mid \{v\} \subseteq pt(x) \mid pt(x) \subseteq pt(*x) \mid pt(*x) \subseteq pt(x).$$

Interpretation of $pt(x) \subseteq pt(*y)$ is semantically equivalent to $\forall z \in pt(y).pt(x) \subseteq pt(z)$ (vice versa for $pt(*x) \subseteq pt(y)$)

Definition 2.2.3. (**Constraint generation**) We have the following constraint generation rules for Andersen's analysis:

$$\begin{aligned}
[\![x := \&y]\!] &= \{y\} \subseteq pt(x) \\
[\![x := \mathsf{null}]\!] &= \{\ \mathsf{null}\ \} \subseteq pt(x) \\
[\![x := \mathsf{new}_\ell]\!] &= \{\ \mathsf{new}_\ell\ \} \subseteq pt(x) \\
[\![x := y]\!] &= pt(y) \subseteq pt(x) \\
[\![x := *y]\!] &= pt(*y) \subseteq pt(x) \\
[\![*x := y]\!] &= pt(y) \subseteq pt(*x)
\end{aligned}$$

- α is omitted from constraint gen. since existentials + binders are not required for the WHILE language.
- Constraint solving is performed in the same way as 0CFA.
- **Time complexity**: $O(n^3)$.
- **Other Approaches**:

  Steengaard's Algorithm Merges the abstract values of a and b is any pointer can reference both. Less precise than Andersen's, but approximately linear time complexity!

  Shape Analysis Models an abstract heap, with nodes and may / must point to edges between notes. Very precise! But abstract heap becomes very large in practice.

2.3 Inference-Based Analysis

- **Idea**: Define inductive judgements $\Gamma \vdash e : \phi$, read as: e has the property ϕ in the context Γ.

Definition 2.3.1. (**Inference-Based Analysis**) An inference-based analysis of a property Φ consists of an inductive relation $\Gamma \vdash e : \phi \in \Phi$, with a *safety property*:

$$\Gamma \vdash e : \phi \implies [\![e]\!] \in [\![\phi]\!]$$

where e is the *value* of e and $[\![\phi]\!]$ is the set of values with property $\phi \in \Phi$.

- For $\lambda^{\to}$, they often take the form:

$$\frac{x : \phi \in \Gamma}{\Gamma \vdash x : \phi}$$

$$\frac{\Gamma, x : \phi_1 \vdash e : \phi_2}{\Gamma \vdash \lambda x.e : \phi_1 \to \phi_2}$$

$$\frac{\Gamma \vdash e_1 : \phi_1 \to \phi_2 \qquad \Gamma \vdash e_2 : \phi_1}{\Gamma \vdash e_1\ e_2 : \phi_2}$$

- **Examples**: Type systems, odd-even analysis, effect systems.
- They are another *form* of abstract interpretation (and can easily be related to constraint-based analysis using judgements of the form $C \vdash e : \phi$ read as: under satisfiable assumptions C, e has the property ϕ).

2.3.1 Effect Systems

- **Idea**: Tracking side-effects (e.g. mutable state, IO, etc) using inference-based analysis.

Definition 2.3.2. ($\lambda^{\to}_{\mathbf{IO}}$) The simply typed lambda calculus with IO $\lambda^{\to}_{\mathrm{IO}}$ extends $\lambda^{\to}$ with following IO operations:

$$e ::= \ldots \mid \mathsf{with}\ x \leftarrow \xi\ \mathsf{in}\ e \mid e \to \xi; e$$

where:

– $\mathsf{with}\ x \leftarrow \xi\ \mathsf{in}\ e$ reads the contents of *channel* ξ into x, binding it in e.

– $e_1 \to \xi; e_2$ writes the value of e_1 into channel ξ, then evaluating the expression e_2, sequentially.

- Effects for an $\lambda^{\to}_{\text{IO}}$ expression are a set of read-write effects to various channels:

$$F \subseteq \{W_\xi, R_\xi : \xi \in \Xi\}$$

- Types for $\lambda^{\to}_{\text{IO}}$ are as usual, with the addition of *latent effects* to the function type:

$$\tau ::= \mathsf{int} \mid \tau \xrightarrow{F} \tau$$

- Inference rules, with *effect subtyping* (required for branches, etc):

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau \,\&\, \emptyset} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \,\&\, F}{\Gamma \vdash \lambda x.e : \tau_1 \xrightarrow{F} \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{F} \tau_2 \,\&\, F_1 \qquad \Gamma \vdash e_2 : \tau_1 \,\&\, F_2}{\Gamma \vdash e_1\ e_2 : \tau_2 \,\&\, F_1 \cup F_2 \cup F}$$

$$\frac{\Gamma \vdash e : \tau_1 \xrightarrow{F_1} \tau_2 \,\&\, F_0 \qquad F_1 \subseteq F_2}{\Gamma \vdash e : \tau_1 \xrightarrow{F_2} \tau_2 \,\&\, F_0}$$

- *Safety property*:

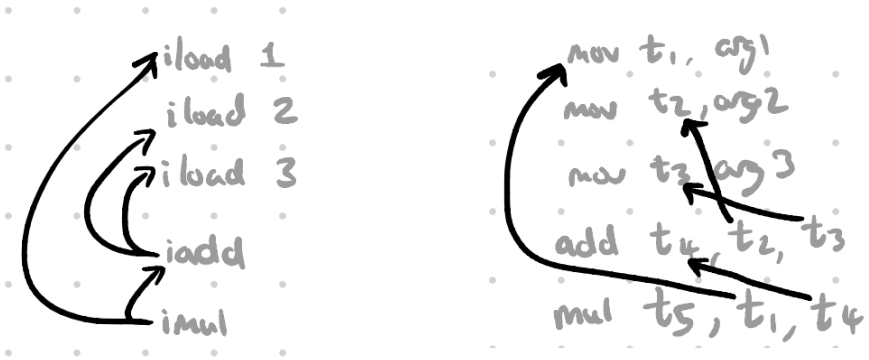
$$\vdash e : \tau \,\&\, F \implies \exists v \in [\![\tau]\!], f \subseteq F. (v, f) \in [\![e]\!]$$

- Additional structures may enrich the effect system:
  - Effects cannot be permuted (a list, instead of a set).
  - Additional combinators on effects for branching.
    e.g. $F_1 \,\&\, (F_2 \mid F_3)$ = effects of F_1 *and* the effects of F_2 *or* F_3.
    See IA semantics supervision work.

3 Low-Level Optimizations

3.1 Intermediate Representations

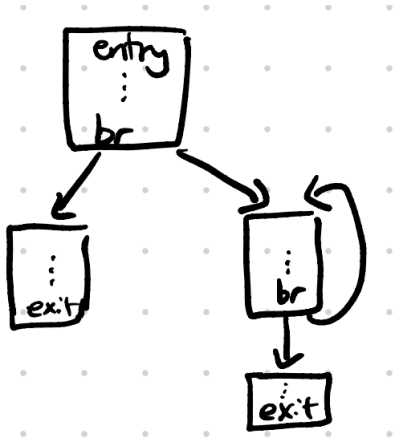
- **Problem**: Need a low-level IR representation of the program.
- **Stack machines**: Stack machine instructions are a poor representation:
  - Data flow is hard to analyze (as dependencies are implicit through the stack).
  - Rewriting is difficult to prove.
- **3-Address Code**: 3-address codes are much easier to reorder and perform analysis on them.
  - Instructions have explicit arguments, used to model data dependencies.
  - Not limited by hardware restrictions. e.g. unlimited registers.



Definition 3.1.1. (**3-Address Code**) The 3-address code consists of the following instructions ($i \in \mathcal{I}$):

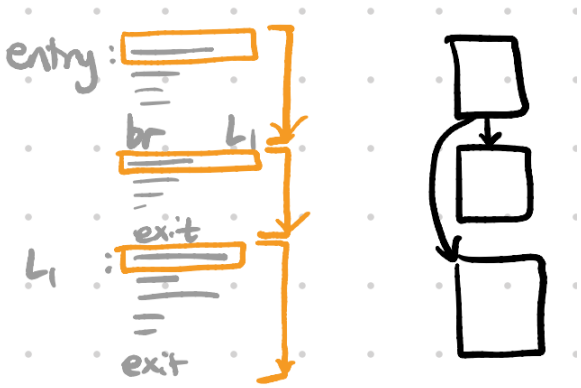
Operator	RISC-V Instruction	Effect
Load	`l`t `rd, x(rs1)`	`rf[rd] = data[rf[rs1] + x]`
Store	`s`t `rs1, x(rs2)`	`data[rf[rs2] + x] = rf[rs1]`
Load Reserved	`lr.`t `rd, rs1`	`rd[rd] = data[rf[rs1]]` and places a "reservation" on address `rf[rs1]`.
Store Conditional	`sc.`t `rd, rs1, rs2`	`data[rf[rs2]] = rf[rs1]`. `rf[rd] = 1` $\iff$ valid reservation on `rf[rs2]` address.
Shift Left	`sll, slli`	`<<`
Shift Right	`srl, srli`	`>>`
Shift Right Arithmetic	`sra, srai`	`>>` but uses sign bit (instead of 0)
Bitwise-And	`and, andi`	`&`
Bitwise-Or	`or, ori`	`|`
Bitwise-Xor	`xor, xori`	`^`
Add	`add, addi`	`+`
Subtract	`sub, subi`	`-`
Equal	`beq rs1, rs2, o`	`if (rs1 == rs2) PC += o;`
Not Equal	`bne rs1, rs2, o`	`if (rs1 != rs2) PC += o;`
Less Than	`blt rs1, rs2, o`	`if (rs1 < rs2) PC += o;`
Greater Than or Equal	`bge rs1, rs2, o`	`if (rs1 >= rs2) PC += o;`
Less Than (Unsigned)	`bltu rs1, rs2, o`	`if ((unsigned)rs1 < (unsigned)rs2) PC += o;`
Greater Than or Equal (Unsigned)	`bgeu rs1, rs2, o`	`if ((unsigned)rs1 >= (unsigned)rs2) PC += o;`
Jump to Register	`jr rs1`	`jalr x0, 0(rs1)`
Jump to Label	`call .L1`	`jal ra, .L1`
Return	`ret`	`jalr x0, 0(ra)`
No-op	`nop`	`add x0, x0, x0`
Move	`mv rd, rs1`	`addi rd, rs1, 0`
Branch if zero	`beqz rs1, .L1`	`beq rs1, x0, .L1`
Branch greater than	`bgt rs1, rs2, .L1`	`blt rs2, rs1, .L1`
Load immediate	`li rd, x`	Determined by assembler. Based on width of `x`

Definition 3.1.2. (**flowgraph**) A flowgraph G is a graph $G = (V, E)$ where vertices are instructions $V \subseteq \mathcal{I}$ and edges $E = \{(i_1, i_2) : i_1 \text{ may branch to } i_2\}$.



Definition 3.1.3. (**Basic Block**) A basic block b is a maximal sequence of instructions $i_1, \ldots, i_n$ with no internal control flow s.t:

$$\forall i \in \langle i_2, \ldots, i_n \rangle . |\mathsf{pred}(i)| = 1$$
$$\forall i \in \langle i_1, \ldots, i_{n-1} \rangle . |\mathsf{succ}(i)| = 1$$



- Basic blocks reduce the space and time requirements for analysis algorithms by calculating and storing data-flow and control-flow information per-block, compared to per-instruction in flowgraphs.

- Algorithm for computing a basic blocks:
  1. Find all instructions i that are *leaders*. A leader is:
     - The first instruction of the procedure
     - The target of any branch / call
     - Any instruction immediately following a branch instruction
  2. For each leaders i, the basic block b is defined as the instructions from i to the next leader.

3.1.1 Forms of Analysis

- Types of analysis:

  Local Local analysis *within* a basic block.

  Intra-procedural Analysis *between* basic blocks (in isolation). Techniques vary, but optimizations are usually split into *control flow analyses* and *data flow analyses*.

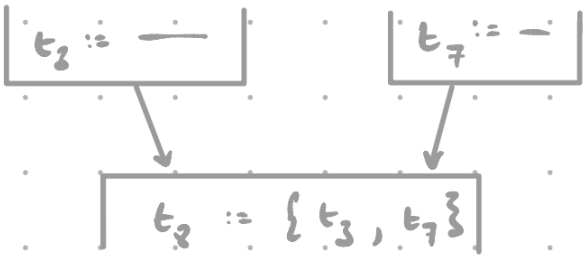
  Inter-procedural Analysis between procedures (the whole program). Examples include *inlining* or *unreachability analysis*. Very expensive in practice (many compilers do not perform WPO).

- **Note**: Many compilers only use a handful of optimizations:
  - Optimizations often have a low-payoff.
  - They are complex to implement (maintainability issues).
  - Their analyses too expensive during compilation (see Rust).
- **Control flow analysis**: Determine control flow by constructing flowgraph G.
- **Data flow analysis**: Determine the data flow dependencies (e.g. variable uses, expression evaluations, etc) *using* a flowgraph G.
- **Sensitivity**:
  - **Flow sensitive**: Considers the order or control flow of statements.
  - **Path sensitive**: Considers loop conditions in analysis e.g. `if` x `> 0 then ... else ...` assume $x > 0$ in `then` branch and $x \leq 0$ in `else` branch. (Hoare Logic)

3.1.2 Static Single Assignment Form

Definition 3.1.4. (**Static Single Assignment Form**) An IR is said to be in static single assignment form (SSA) if each variable/register in the IR is assigned once (statically, dynamically a variable could be assignment multiple times e.g. a loop), and every variable is defined before its usage.

- **Conversion to SSA**:
  1. Replace each assignment to t with a new variable t_i (ith version), and replacing each reference of the variable with the version that is live at that point.
  2. Control-flow divergence and reconciliation requires notion of a *superposition* of versions (or a ϕ-function). The superposition $\{t_{i_1}, \ldots, t_{i_k}\}$ denotes the resultant variable is t_{i_k} if we arrive from predecessor k.



- **Benefits of SSA**:
  - Reduces registers (reduces live ranges).
  - Improves data-flow optimizations (e.g. liveness).

3.2 Local Optimizations

- Performed by traversing a section of a basic blocks (known as a *peep-hole / window*), where the window is optimized to an equivalent set of instructions w/ better performance.
- **Algebraic Simplifications / Peephole optimizations**:

```
addi xn, xn, 0      ⇝   // addi xn, xn, 0 has no effect (x = x + 0)
...                     ...

mv xm, xn           ⇝   mv xm, xn
mv xn, xm               // moving contents of xm to xn has no effect
```

Advantages	Disadvantages
Many applicable optimizations	Limited scope due to *peephole*. Cannot deal w/ optimizations based on control flow / inter-procedural optimizations
Simple implementation. Pattern matching optimizations w/in peephole optimization w/ traversal algorithm across AST / linearized code.	
Combined with inter-procedural optimizations (e.g. inlining) yields extremely effective optimizer.	

3.2.1 Constant Folding

Definition 3.2.1. (**Constant Folding**) Constant folding is an optimization where constant expressions are evaluated at compile time.

- Optimizes running time. Compile time → Running time tradeoff.

Definition 3.2.2. (**Constant Propagation**) Constant propagation is the process of substituting the identifier x of a constant expressions e evaluated at compile time w/ it's value v.

```
int x = 5;
int y = x * 2;  ⇝  int z = a[10];
int z = a[y];
```

- Implementing using *reaching definition analysis* (binder dependencies).

Advantages	Disadvantages
Increases runtime performance	Decreases compiler performance (massive slowdown for large programs)
Unsafe optimizations may be applied. e.g. `0 * x` $\neq$ `0` (by IEEE)	
Simple implementation. Pattern matching on operators w/ operands tagged `Const`.	

3.3 Control-Flow Analyses

- **Goal**: determine certain control-flow properties using a CFG.
- **Note**: Analyses are often *conservative* (for safety reasons), since control-flow may be impossible to precisely determine in a static context (Halting problem).
- Control-flow analyses are often *not* structure-preserving, whereas Data-flow analyses are.

3.3.1 Unreachability Analysis

Definition 3.3.1. (**Unreachable Code**) An instruction i is said to be *unreachable* if it's node in the flowgraph G is unreachable from some root set R of entry nodes in G.

Definition 3.3.2. (**Dead Code**) An instruction i is said to be dead if it's computational effect is unused.

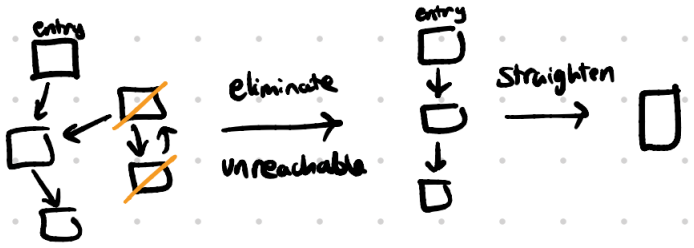
- Unreachability assumptions:
  - Assume *both* branches of a conditional branch are taken.
  - All loops *eventually* terminate.
  - Indirect jumps (`jr`) can go to any address that has previously been taken.

Unreachable Code	Dead Code
Control-flow property	Data-flow property
Wastes `.text` (program) space	Wastes CPU cycles at runtime
`return x;` `z := x + y;` ← Unreachable	`z := x + y;` ← Dead `return x;`

- **Analysis**: A simple DFS/BFS traversal from the root set R, marking visited nodes $n \in \mathsf{visited}(R)$, is sufficient to determine whether a node n is unreachable, as n is unreachable $\iff n \notin \mathsf{visited}(R)$.
- **Transformations**:

  Unreachable Code Elimination (Intra-procedural) Delete basic blocks b if b is unreachable from root set R.

  Optimizations / macro systems often introduce "obvious" unreachable code. For example: `if (true) ...` could result from constant folding.



  Unreachable Procedure Elimination (Inter-procedural) Delete functions / procedures not reachable from the main function (using a call graph).

3.3.2 Control-Flow Simplification

- Control-flow simplification is a form of non-structure-preserving peephole optimizations.

- **If Simplification**:

Original	Optimized

`// f has no side-effects`
`if f(x) then ();`
C
	`// if-statement had no`
	`// semantic effect`
	C

`// else branch has no effect` `if` B `then` C `else ()`	`if` B `then` C
`// then branch has no effect` `if` B `then () else` C	`if not` B `then` C

`// Constant condition`
`// (vice verse for false)`
`if true then` C_1 `else` C_2
	C_1

`// dual optimization for else`
`if` $\mathcal{C}_B[B]$
`then` $\mathcal{C}_C$`[if` B `then` C_1 `else` C_2`]`
`else` C_3
	`if` $\mathcal{C}_B[B]$
	`then` $\mathcal{C}_C[C_1]$
	`else` C_3

- **Loop Unrolling**: With `while` and (especially) `for` loops with small number of iterations (determined by a constant), unroll the loop:

```
for i = 0 to 3 do
  C
done
```
$\rightsquigarrow$
```
i := 0;
C;
i := i + 1;
C;
i := i + 1;
C;
i := i + 1;
C
```

3.4 Data-Flow Analyses

3.4.1 Live Variable Analysis

- **Liveness** $\approx x$ is used in the future.

Definition 3.4.1. (**Liveness**) A variable x is said to *semantically live* at instruction i iff

$$\exists \text{environments} \rho_1, \rho_2 . \rho_1 =_{\backslash x} \rho_2 \wedge [\![G \lfloor i]\!]_{\rho_1} \neq [\![G \lfloor i]\!]_{\rho_2}$$

where $G \lfloor i$ is the subgraph of G rooted at i.

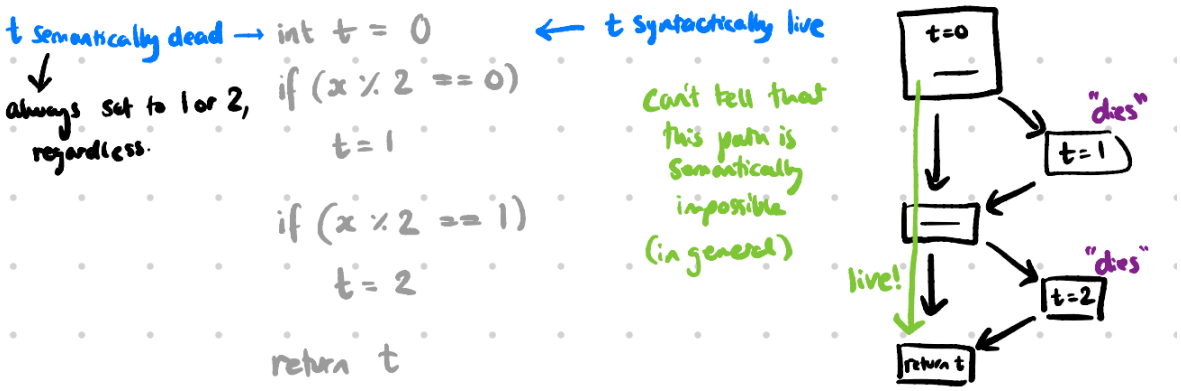
- Intuitively, x is live if the execution sequence starting at i observably depends on x.

Definition 3.4.2. (**Syntactic Liveness**) A variable x is *syntactically live* at instruction i iff there exists a path $p = i = i_1 \rightarrow \cdots \rightarrow i_k = i'$ s.t

$$\forall 1 < l < k . x \notin \mathsf{def}(i_l) \wedge x \in \mathsf{ref}(i')$$

- Syntactic liveness is decidable and an **over-approximation** of semantic liveness:

$$\mathsf{sem\text{-}live}(i) \subseteq \mathsf{syn\text{-}live}(i)$$



- **Analysis**:
  - **Backwards analysis** to propagate the uses/refs upwards to defs.
  - Data-flow equations:

$$\begin{aligned} \mathsf{in\text{-}live}(i) &= (\mathsf{out\text{-}live}(i) \setminus \mathsf{def}(i)) \cup \mathsf{ref}(i) \\ \mathsf{out\text{-}live}(i) &= \bigcup_{i' \in \mathsf{succ}(i)} \mathsf{in\text{-}live}(i') \end{aligned}$$

    (Re)-defining a variable "kills" it.
    Referencing the variable makes it live.
    Combined equation:

$$\mathsf{live}(i) = \mathsf{in\text{-}live}(i) = \left(\bigcup_{i' \in \mathsf{succ}(i)} \mathsf{live}(i') \setminus \mathsf{def}(i) \right) \cup \mathsf{ref}(i)$$

  - Algorithm: Iteratively computes the least fixed point.

```
for i = 1 to n do
  live.(i) <- ∅
done;
while (live changes) do
  for i = 1 to n do
    live.(i) <- (⋃_{i'∈succ(i)} live.(i') \ def(i)) ∪ ref(i)
  done
done
```

    Termination is guaranteed since live sets are bounded.
    Optimizations: Implement sets using *bit vectors*. $live.(i) = b$ where $b[x] = 1$ iff $x \in \mathsf{live}(i)$.
    Compute liveness at the basic block level as opposed to instruction level:

$$\mathsf{live}(b) = \left(\cdots \left(\left(\bigcup_{s \in \mathsf{succ}(b)} \mathsf{live}(s) \setminus \mathsf{def}(b_n) \right) \cup \mathsf{ref}(b_n) \right) \cdots \setminus \mathsf{def}(b_1) \right) \cup \mathsf{ref}(b_1)$$

- Considerations for pointers (addresses):

`*p = e`	$\mathsf{def} = \emptyset$ $\mathsf{ref} = \mathsf{ref}(e) \cup \{p\}$
`x = *p`	$\mathsf{def} = \{x\}$ $\mathsf{ref} = \texttt{all address taken variables} \cup \{p\}$

3.4.2 Available Expression Analysis

- **Available** $e \approx$ already computed e and no changes in free variables in e

Definition 3.4.3. (**Available**) An expression e is said to be *semantically available* at instruction i iff:

(i) e has previously been computed and not subsequently invalided (by changes in free variables of e),

(ii) on all *execution sequences* to i.

- Property (i) is decidable, but (ii) is undecidable $\implies$ *syntactic approximation*

Definition 3.4.4. (**Syntactically Available**) An expression e is said to be *syntactically available* at instruction i iff:

(i) e has previously been computed and not subsequently invalided (by changes in free variables of e),

(ii) on all *paths* in the flowgraph G to i.

$\forall$ path $p = i' \rightarrow^* i . \exists i_e \in p . i_e$ evaluates $e \wedge \forall i'' \in i_e \rightarrow i . \mathsf{fv}(e) \cap \mathsf{def}(i'') = \emptyset$

- **Safety**: underestimate availability:

$$\mathsf{syn\text{-}avail}(i) \subseteq \mathsf{sem\text{-}avail}(i)$$

- **Analysis**:
  - **Forward analysis** to propagate expressions to children with universe of expressions U (computable since finite # of expressions in program).
  - Intuitive data-flow equations:

$$\mathsf{in\text{-}avail}(i) = \begin{cases} \bigcap_{i' \in \mathsf{pred}(i)} \mathsf{out\text{-}avail}(i') & \text{if } \mathsf{pred}(i) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$
$$\mathsf{out\text{-}avail}(i) = (\mathsf{in\text{-}avail}(i) \cup \mathsf{gen}(i)) \setminus \mathsf{pred}(i)$$

    where:

$$\mathsf{gen}(x := e) = \{e\}$$
$$\mathsf{kill}(x := e) = \{e \in U : \mathsf{fv}(x) \cap \mathsf{def}(e) \neq \emptyset\}$$

  - Data-flow equations:

$$\mathsf{in\text{-}avail}(i) = \begin{cases} \bigcap_{i' \in \mathsf{pred}(i)} \mathsf{out\text{-}avail}(i') & \text{if } \mathsf{pred}(i) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$
$$\mathsf{out\text{-}avail}(i) = (\mathsf{in\text{-}avail}(i) \setminus \mathsf{kill}(i)) \cup \mathsf{gen}(i)$$

    where:

    * $e \in \mathsf{gen}(i)$ iff i computes e and does not invalidate it.
    * $e \in \mathsf{kill}(i)$ iff i may invalidate e ($\mathsf{fv}(e) \cap \mathsf{def}(i) \neq \emptyset$) and does not recompute it.

    Differences for compatability w/ LVA + re-ordering of instructions.

    Combined equation:

$$\mathsf{avail}(i) = \mathsf{in\text{-}avail}(i) = \begin{cases} \bigcap_{p \in \mathsf{pred}(i)} (\mathsf{avail}(p) \setminus \mathsf{kill}(p)) \cup \mathsf{gen}(p) & \text{if } \mathsf{pred}(i) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

– **Algorithm**:

```
avail.(1) <- ∅ (* entry/root has no available *)
for i = 2 to n do
  avail.(i) <- U (* available expressions removed by ⋂ *)
done;
while (live changes) do
  for i = 1 to n do
    avail.(i) <- ⋂_{p∈pred(i)}(avail.(p) \ kill(p)) ∪ gen(p)
  done
done
```

Optimizations:

* Bit vector: Suppose we have n expressions. `avail` is an array of n-bit numbers, where the ith bit indicates the availability of the ith expression.
* Basic blocks:

$$\mathsf{avail}(i) = \bigcap_{p \in \mathsf{pred}(i)} (\cdots (\mathsf{avail}(p) \setminus \mathsf{kill}(p_1)) \cup \mathsf{gen}(p_1) \cdots \setminus \mathsf{kill}(p_{k_p})) \cup \mathsf{gen}(p_{k_p})$$

- Considerations for pointers (addresses):

`*p = e`	kill = `all expressions w/ address taken variables` $\cup \{p\}$
`x =` $\mathcal{C}_e$`[*p]`	gen $= \emptyset$

3.4.3 Reachability Analysis

Definition 3.4.5. (**Reachable**) A definition of i is said to semantically reach i' in flowgraph G if all variables $x \in \mathsf{def}(i)$ have the same value on entry to i'.

Definition 3.4.6. (**Syntactic Reachability**) A definition of i is said to syntactically reach i' in flowgraph G if there exists a path $p = i = i_1 \rightarrow \cdots \rightarrow i_k = i'$ s.t

$$\forall x \in \mathsf{def}(i). \forall 1 < i < k. x \notin \mathsf{def}(i_k)$$

- **Safety**:

$$\mathsf{syn\text{-}reach}(i) \subseteq \mathsf{sem\text{-}reach}(i)$$

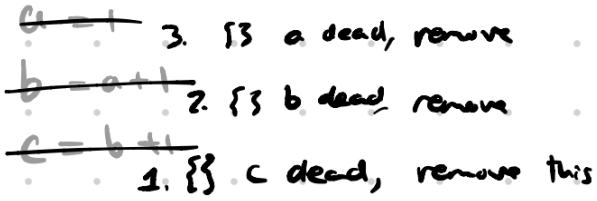
- **Analysis**:
  - **Forward analysis**
  - Data-flow equation:

$$\mathsf{reach}(i) = \bigcup_{p \in \mathsf{pred}(i)} (\mathsf{reach}(p) \setminus \mathsf{kill}(p)) \cup \mathsf{gen}(p)$$

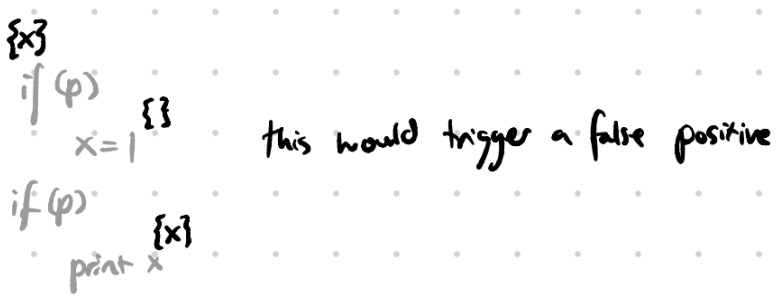
    See supervision work for more details.

3.4.4 Data-Flow Transformations and Anomalies

- **Dead Code Elimination**:
  - Compute liveness live.
  - **Transformation**: Remove instruction i if $x \notin \mathsf{live\text{-}out}(n)$ and $x \in \mathsf{def}(i)$.
  - **Safety**: If x is dead after assignment $\implies$ assignment was pointless (no observable side-effects).



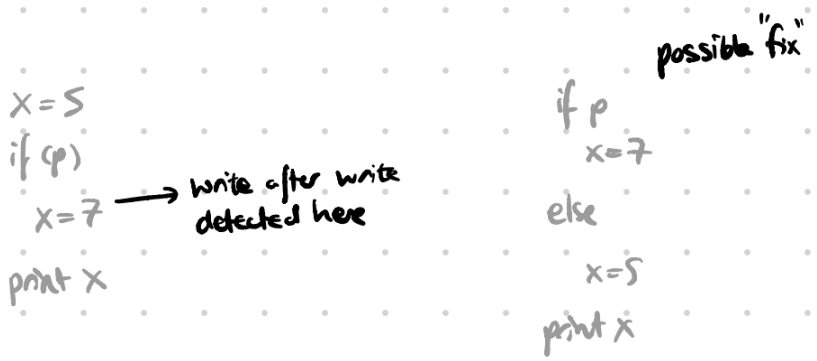
- **Uninitialized Variables**:
  - $\mathsf{live}(i_{\text{entry}})$ contains the set of *undefined* variables used in the procedure w/ entry instruction i_{entry}.
  - Compiler should emit a warning to the user
  - **Problem**: Due to overapproximation of LVA $\implies$ false positives.



- **Write-write Anomalies**:
  - **Problem**: Variable written too by not read before previous write $\implies$ previous write has no effect.
  - Inverse LVA solution using a *forward analysis* w/ data-flow equation:

$$\mathsf{wnr}(i) = \bigcup_{p \in \mathsf{pred}(i)} (\mathsf{wnr}(p) \setminus \mathsf{ref}(p)) \cap \mathsf{def}(p)$$

  - Compiler should emit a warning to the user



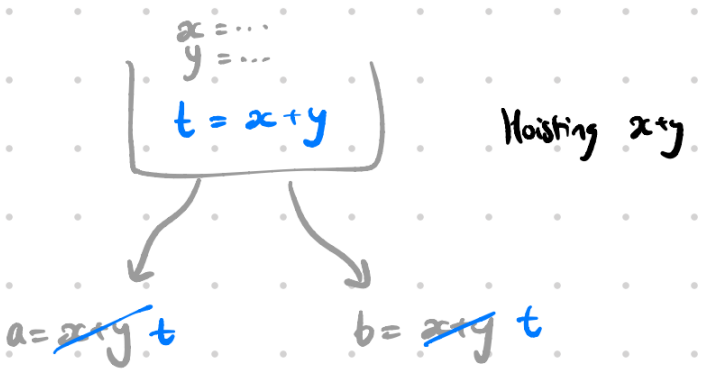
- **Common Subexpression Elimination**:
  - **Transformation**: If e is available at instruction i, replace e with temporary variable t and replace original evaluation of e (in context x := $\mathcal{C}[e]$) with:

$$t := e;$$
$$x := \mathcal{C}[t]$$

  - **Warning**: Optimization may be detrimental to later phases:
    * Added temporary $t \implies$ higher register pressure
    * Added copy instructions. Load from memory may be slower than a simple arithmetic expression.

- **Copy Propagation**:
  - Perform reachability analysis.
  - **Transformation**: Eliminate instructions of the form: `mv rd, rs1` at i, replacing `rd` with `rs1` at i' if `rs1` is reachable at i' from i.

- **Code Motion**:
  - Code Hoisting (Move up flowgraph) or Sinking (Move down flowgraph).
  - **Analysis**: Compute *very busy expressions*. A very busy expression $e \approx e$ will definitely by evaluated later in the program (a backwards version of avail).
  - **Transformation**: Replace later occurrences of e w/ temporary variable t and *hoist* e:



  - **Warning**: May increase register pressure.
  - **Loop invariant**: If ref(e) are reachable from *outside* the loop $\implies$ e doesn't dependent on loop state (e.g. counters), thus can be hoisted out of the loop.

- **Partial redundancy**:

- **Analysis**: Partially available expression analysis (change $\bigcap$ to $\bigcup$ in availability data-flow equations).
- An expression is *partially redundant* if it is computed more than once on *some* path p in flowgraph G.
- Same **transformation** as *common subexpression elimination*.
- **Benefit**: Combines common subexpression elimination and loop-invariant code motion into one optimization!

3.5 Inlining

Definition 3.5.1. (**Inlining**) Inline expression is an optimization technique by inlining the function body for a given call.

- Optimizes running time by eliminating overhead of call prologue and epilogue . Space $\rightarrow$ Time tradeoff by duplicating function bodies.
- **Heuristics** for Inline expansion:
  1. Expand function-calls that are executed frequently. Determines by static-analysis of loop-nest depth (or at runtime w/ JIT)
  2. Expand functions w/ very small bodies. Minimizing code explosion from inline expansion.
  3. Expand functions that are only called once (and aren't exported), then apply unreachable procedure elimination (deleting original function definition).

Advantages	Disadvantages
Eliminates the instructions required by the calling convention: the prologue and epilogue (and the `call`/`jump` required).	Inlining duplicates function body, increase in code size $\Longrightarrow$ may result in a *code explosion*.
Reduces register spillage from arguments	Increases *working set* (the set of pages the program requires access to at a given time). Increases page faults $\Longrightarrow$ thrashing (in serve cases)
Removes callee, caller boundary, allowing for more intra-procedural optimizations (e.g. peephole, etc)	Increases instruction cache miss rate (due to increased code size).
Eliminating function calls improves the temporal + spatial locality of instructions.	

4 Code Generation

4.1 Register Allocation

- **Problem**: Assigning virtual registers in an IR (e.g. normal form 3-address code, SSA) to a limited number of *physical* general purpose registers.

Definition 4.1.1. (**Clash Graph**) A clash graph $C = (V, E)$ of flowgraph G where $V \subseteq$ virtual registers and

$$(r, s) \in E \iff \exists \text{instruction } i \in V(G).r \in \mathsf{live}(i) \wedge s \in \mathsf{live}(i)$$

- Clash graphs model dependencies between simultaneously live variables/registers.

Definition 4.1.2. (**Coloring**) A coloring of a graph $G = (V, E)$ is a function $\chi : V \to \mathsf{color}$ satisfying:

$$\forall (v_1, v_2) \in E.\chi(v_1) \neq \chi(v_2)$$

- **Algorithm**:
  1. Compute flowgraph G and perform liveness analysis.
  2. Construct the clash graph C.
  3. Color the clash graph C, where colors correspond to physical registers.
- **Problem**: Graph coloring is an NP-complete problem.
- **Solution**: Heuristic algorithm:
  1. Select virtual register t from C with fewest clashes (smallest degree).

2. If #clashes < #physical registers:

   Then Push t onto stack.
   Remove t from C.

   Else Select a virtual register s to *spill* (using a heuristic).
   Remove s from C.

3. Repeat 1–2 until C is empty.
4. Pop each virtual register from stack. Assign a physical register (deterministically).

- **Time Complexity**: $O(V^2 + E)$.
- **Problem**: $O(V^2 + E)$ is slow (e.g for JIT).
- **Solution**: Use other algorithms. Example: *Linear scan* (see supervision work).

Definition 4.1.3. (**Spilling**) If all physical registers are in use, then register values are pushed (*spilled*) onto the stack, freeing the register.

- When spilling registers, must *reserve* registers for spilled values. If using n spilled values simultaneously $\implies$ n reserved registers.
- **Problem**: # of reserved registers (n) prior to allocation is unknown.
- **Solution**: If spillage detected, restart allocation algorithm w/ reserved registers.
- **Problem**: Determining optimal register to spill.
- **Solution**: Spilling allocation heuristic:
  - Compute # of times register is accessed (as a 'liveness factor'). Spill registers w/ fewest accesses.
  - Loop nesting $\implies$ 'liveness factor' = # accesses $\times$ loop depth.

4.1.1 Non-Orthogonal Instructions

Definition 4.1.4. (**Non-Orthogonal Instruction Sets**) A non-orthogonal instruction set consists of instructions where the addressing mode of instructions is dependent on the instruction type.

- For example: requiring ALU operations to use an accumulator register.
- To allocate registers for non-orthogonal instructions allocate virtual registers for the pre-allocated physical registers.
- When translating non-orthogonal instructions, add `mv` instructions to move register values into specific physical registers.
- **Problem**: Non-orthogonal instructions add many `mv` instructions.
- **Solution**: Preference graphs.

Definition 4.1.5. (**Preference Graph**) A preference graph P of flowgraph G is a graph $P = (V, E)$ where $V \subseteq$ virtual registers and

$$(r, s) \in E \iff \texttt{mv s, r} \in V(G)$$

- When allocating colors, we use a preference graph P to attempt that $\forall (r, s) \in V(P).\chi(r) = \chi(s)$.
- If adjacent registers in P have the same color $\implies$ the `mv` instruction becomes a `nop`.

4.1.2 Calling Conventions

Definition 4.1.6. (**Calling Convention**) A calling convention defines the architecture registers that are maintained by the *callee* and the *caller*.

- RISC-V: argument registers `a0-a7`, return address `ra`, result register `a0`.
- Registers may be maintained by allocating stack space to save registers.
- **Problem**: Some registers may be corrupted (caller saved) while executing a procedure

- **Solution**: Synthesize edges on clash graph between possibly corrupted registers and live registers at `call` instruction.
- **Problem**: Saving registers introduced many `mv` instructions.
- **Solution**: Preference graph.

4.2 Instruction Scheduling

- **Motivation**: Specific architectures may provide parallelism / speedup which the compiler should take advantage of.
- Architecture types:
  - Single cycle: Each instruction takes a single cycle to execute. No violations of the sequential programming model.
  - Pipelined: Instructions are pipelined (each stage taking a single cycle to execute). May introduce stalls (to deal w/ hazards).
  - Multi-instruction: MIMD, MISD, etc. (*not covered in this course*).
- Interlocked processors: Detect hazards and insert stalls.
- Non-interlocked processors: Compiler must insert `nop`s to prevent data / control hazards.
- See IB computer design notes on forwarding and load hoisting. *course covers compiler techniques for data hazards.*
- $\mathsf{read}(i)$ = addresses/registers read by instruction i, $\mathsf{write}(i)$ = addresses/registers written to by instruction i.
  Dependencies:
  - Read-after-write (RAW): $\mathsf{write}(i_1) \cap \mathsf{read}(i_2) \neq \emptyset$. i_1 writes before i_2 reads.
  - Write-after-read (WAR): $\mathsf{read}(i_1) \cap \mathsf{write}(i_2) \neq \emptyset$. i_1 reads before i_2 writes.
  - Write-after-write (WAW): $\mathsf{write}(i_1) \cap \mathsf{write}(i_2) \neq \emptyset$. i_1 and i_2 write to the same address/register.

- **Idea**: Within basic blocks, re-order instructions to minimize stalls subject to data dependencies.
- **Note**: If load instruction w/ *unknown address* e.g. `lw t2, 16(t0); sw t3, 4(t1)`, then add dependencies between loads and stores (since they *may* read-/write to the same address. e.g. when `t1 = t0 + 12`).

Definition 4.2.1. (**Data Dependence Graph**) For the basic block b, the data dependence graph G is the directed graph $G = (b, E)$ where

$(i_1, i_2) \in E \iff i_2$ depends on i_1 according to above dependencies

- **Note**: Any topological sort of data dependence graph is a *valid* schedule.
- **Problem**: Choosing schedule which *minimizes # of stalls* is NP-complete.
- **Solution**: Scheduling Heuristics. Choose a next instruction s.t:
  1. Doesn't conflict with the previous instruction
  2. Is most likely to conflict if first of a pair (e.g. `lw` is better than `add`, since `lw` is more likely to cause a hazard). *This heuristic adds load hoisting.*
  3. Is as far as possible from the last instruction in the data dependency graph. *Keeps rough order of instructions.*
- **Algorithm**:

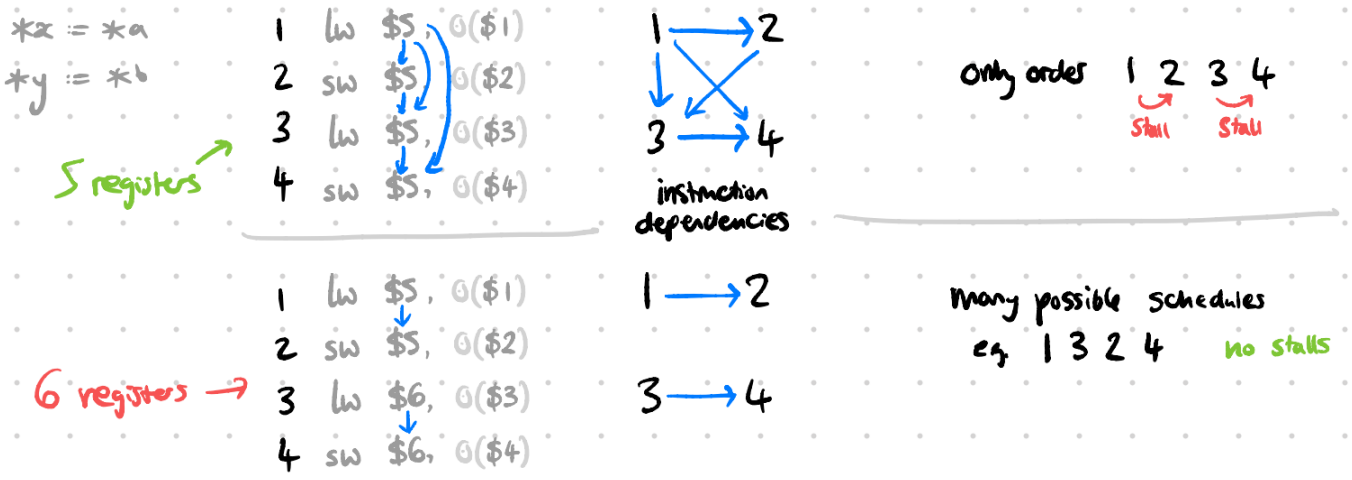
```
construct data dependence graph G of b;
candidates := sources(G); // instructions w/ no dependencies
while (not (Set.empty candidates)) do
  select i from candidates;
  if i satisfies 1-3 heuristics then
    emit i
  else
    (if non-interlocked processor then
       emit nop
     else
      select i from candidates s.t 2-3 heuristics are satisfied;
```

```
        emit i);
    remove emitted instruction from G;
    update candidates // sources(G) where G is updated
done
```

- **Time complexity**: $O(|b|^2)$.

4.2.1 Antagonism with Register Allocation

- **Problem**: Maximizing register utilization creates additional dependencies, which limits instruction scheduling (resulting in more stalls).
- **Example**:



- **Observation**: Allocating too aggressively $\implies$ limitations on instruction scheduling $\therefore$ more stalls.
- This is an example of the *phase order problem*. It has no solution, as predicting interaction of register allocation and instruction scheduling is very dependent on architecture + code structure.
- **Ad-hoc solution**: Limitations on instruction scheduling occur due to eager reuse or registers. One solution is to allocate registers in a round robin fashion, but selecting a register distinct from all others in a basic block that satisfies coloring constraints.

5 Decompilation

5.1 Control-flow Reconstruction

5.2 Type Reconstruction