Basic Blocks

A maximal sequence of instructions $n_1, \ldots, n_k$ with exactly one predecesor (except for n_1) and exactly one successor (except for n_k). They allow memoisation of data flow information.

Local scope: within basic blocks.

Global/intra-procedural scope: between basic blocks.

Inter-procedural scope: whole program.

Detect these by partitioning based on *leader instructions*: the first instruction, branch target instructions and the instructions immediately after a branch.

Unreachable Code

Deadness: data-flow property.

Unreachability: control-flow property.

Detected intra-procedurally by marking procedure entry node and successors as reachable. Inter-procedurally work on the call graph.

Straightening coalesces basic blocks which contain a redundant jump.

Live Variable Analysis

Semantic versus syntactic liveness: $semlive(n) \subseteq synlive(n)$.

LVA is a backwards data-flow analysis, calculated by iterated least fixed point method since effect monotonic and set of variables finite.

$$live(n) = \left(\left(\bigcup_{s \in succ(n)} live(s) \right) \backslash def(n) \right) \cup ref(n)$$

For safety we must overestimate ambiguous references (e.g. assume all address-taken variables are referenced) and underestimate ambiguous definitions (i.e. assume no variables are defined).

Available Expression Analysis

Semantic versus syntactic availability: $semavail(n) \supseteq synavail(n)$.

AVAIL is a forwards data-flow analysis, calculated by iterated greatest fixed point method since effect monotonic and set of variables finite.

$$avail(n) = \begin{cases} \bigcap_{p \in pred(n)} ((avail(p) \backslash kill(p)) \cup gen(p)) & pred(n) \neq \emptyset \\ \emptyset & pred(n) = \emptyset \end{cases}$$

A node *generates* an expression e if it must compute the value of e and does not subsequently redefine any of the variables occuring in e. A node *kills* an expression e if it may redefine some of the variables occuring in e and does not subsequently recompute the value of e.

For safety we must underestimate ambiguous generation and overestimate ambiguous killing (e.g. of address taken local variables).

Data-Flow Anomalies

An instruction is dead when the variable it assigns to is not live on exit from that instruction.

Variables that are live at the beginning of a program represent potentially unmatched usages of the variables.

Write-write anomalies occur when a variable may be written twice with no intervening read:

$$wnr(n) = \bigcup_{p \in pred(n)} ((wnr(p) \backslash ref(p)) \cup def(p))$$

And then watch out for variables that overlap when you perform the union operation.

Register Allocation

Clash graphs contain one node for each virtual register and an edge between those registers that are simultaneously live. Heuristic colouring of the clash graph proceeds as follows:

1. Choose a vertex with the fewest number of incident edges
2. If that vertex has fewer edges than there are colours, remove the vertex and its edges from the graph and push the vertex onto a stack
3. Otherwise choose a register to spill and remove that vertex and edges from the graph
4. Repeat from step 1 until the graph is empty
5. Pop each vertex from the stack and colour it in the most conservative way that avoids any existing neighbour colors

Uncolored registers remaining at the end are spilled to memory. In practice may need to restart register allocation with one or two fewer colors to provide temporary space for spilled values.

A preference graph contains information about which register pairs appear together in a MOV.

Clash graphs can contain edges between those virtuala registers which are potentially wiped by e.g. procedure calls and all virtual registers live at the corresponding instruction.

Non-orthogonal instructions are handled by using architecture specific virtual registers corresponding to hardware ones and generating instructions (with corresponding MOVs) that only operate on the appropriate registers.

Redundancy Elimination

CSE is enabled by AVAIL but may increase register pressure.

Copy propagation scans forward from instructions of the form $x{=}y$ replacing unmodified occurances of x by y.

Code hoisting reduces program size by moving duplicated expressions in two branches to before the branch, and relies on very busy expression (VBE) analysis.

Loop-invariant code motion depends on reaching definition analysis.

Partial redundancy elimination combines CSE with loop-invariant code motion: an expression is partially redundant when it is computed more than once on some paths through a flowgraph (e.g. at the top and bottom of a loop).

SSA

Instead of live range splitting on user variables, we ensure virtual registers are only assigned to once: SSA form. Use ϕ functions to deal with control-flow join points.

Strength Reduction

An optimisation that replaces expensive operations (e.g. multiplication) with cheap ones (e.g. addition). For example, this may occur in loops which index into arrays: we want to replace the multiplication with a repeated addition. In general it works as long as we have induction variable $i = i \oplus c$, $j = c_2 \oplus (c_1 \otimes i)$ and $x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$.

Abstract Interpretation / Strictness Analysis

Abstract domain $D^\#$, abstraction function $\alpha : D \to D^\#$, concretisation function $\gamma : D^\# \to D$ and $\gamma(\alpha(x)) \supseteq x$.

In strictness analysis, take $D = \mathbb{Z} \cup \{\bot\}$, $D^\# = \{0, 1\}$ where 0 indicates that a computation certainly will not terminate. This can be compacted with a Boolean representation.

To cope with recursive functions we take the least fixed point of the Boolean formulae, starting from functions that never halt. The effect is monotone because we cannot have negation.

Constraint Based Analysis

0CFA discovers which values may reach different places in the program.

Program points i are associated with flow variables α_i that contain the possible values to occur at any program point.

$$\begin{array}{ccc} c^a & \to & \alpha_a \supseteq \{c^a\} \\ (\lambda x^a.e^b)^c & \to & \alpha_c \supseteq \{(\lambda x^a.e^b)^c\} \\ x^a \text{ bound at } x^b & \to & \alpha_a \supseteq \alpha_b \\ (let \ \ldots^a = \ldots^b \ in \ \ldots^c)^d & \to & \alpha_d \supseteq \alpha_c, \alpha_a \supseteq \alpha_b \\ (\ldots^a \ \ldots^b)^c & \to & (\alpha_b \mapsto \alpha_c) \supseteq \alpha_a \end{array}$$

Where $(\gamma \mapsto \delta) \supseteq \beta$ says that whenever $\beta \supseteq \{(\lambda x^q.e^r)^p\}$ we have $\alpha_q \supseteq \gamma \wedge \delta \supseteq \alpha_r$.

We can improve it by using 1CFA where functions get their own flow variables for each call site.

Inference Based Analysis / Effect Systems

Specify judgements $\Gamma \vdash e : \phi$, typically structurally induced. Have safety condition of the form $(\emptyset \vdash e : t) \Rightarrow ([[e]] \in [[t]])$.

$e ::= x \mid \lambda x.e \mid e_1\, e_2 \mid \xi?x.e \mid \xi!e_1.e_2$

$t ::= int \mid t_1 \xrightarrow{F} t_2$

$$\frac{\Gamma[x : int] \vdash e : t, F}{\Gamma \vdash \xi?x.e : t, \{R_\xi\} \cup F}$$

$$\frac{\Gamma \vdash e_1 : int, F \quad \Gamma \vdash e_2 : t, F'}{\Gamma \vdash \xi!e_1.e_2 : t, F \cup \{W_\xi\} \cup F'}$$

$$\frac{}{\Gamma[x : t] \vdash x : t, \emptyset}$$

$$\frac{\Gamma[x : t] \vdash e : t', F}{\Gamma \vdash \lambda x.e : t \xrightarrow{F} t', \emptyset}$$

$$\frac{\Gamma \vdash e_1 : t \xrightarrow{F''} t', F \quad \Gamma \vdash e_2 : t, F'}{\Gamma \vdash e_1 e_2 : t', F \cup F' \cup F''}$$

$$\frac{\Gamma \vdash e : t \xrightarrow{F'} t', F \quad F' \subseteq F''}{\Gamma \vdash e : t \xrightarrow{F''} t', F}$$

Safety condition: $(\emptyset \vdash e : t, F) \Rightarrow (v \in [[t]] \wedge f \subseteq F$ where $(v, f) = [[e]])$

Instruction Scheduling

We want to avoid pipeline stalls by hoisting loads as far up as possible.

Must respect data dependencies: read after write, write after read, write after write. The instructions can be represented as a DAG with these dependencies forming the edges between them.

Static scheduling heuristics: every time we're emitting the next instruction, try and choose one which:

- Does not conflict with the previous instruction
- Is most likely to conflict with other instructions
- Is as far away as possible (in the DAG) from an instruction which can valildly be scheduled last

Emitting algorithm:

1. Initialise a candidate list to contain nodes of the DAG with no predecessors
2. While the list is non-empty:
   (a) If possible emit a candidate satisfying all three heuristics
   (b) Otherwise emit an instruction satisfying last two heuristics or a NOP (if using delay slots)
   (c) Remove the emitted instruction from the DAG and add instructions from the DAG which now have no predecessors to the list

This conflicts with register colouring because minimising use of registers linearises the DAG: solve this by allocating registers cyclically rather than conservatively (after respecting e.g. preference graph).

Decompilation

Usually allowed for interoprability purposes.

Extract a flowgraph from assembler instructions as usual.

A *dominator* of n is a node that control flow must pass through to reach n. An *immediate dominator* is the unique node that dominates n but doesn't dominate any other dominator of n. This can be used to construct a dominance tree. *Intervals* are regions of the graph where branches may go outwards but all branches into the region must go to the head of the interval. *Back edges* are those in the flow graph whose head dominates their tail.

To recover control flow, observe that back edges have associated loops. Intervals allow you to identify conditionals (single or double sided) in a straighforward fashion, and the constructs identified can be replaced in the graph with a more abstract node so the detection can be iteratively applied.

Type reconstruction can be done after converting to SSA form and then applying constraint based analysis based on the instructions that operate on a particular virtual register.