CompArch

Optimisation = Analysis + Transformation

Unreachable Code Elimination

1. Mark the entry node of each procedure as reachable.
2. Mark every **successor** of a marked node, repeat until no change.

`goto` may have multiple potential successor nodes, depending on semantics: jumping to a variable address might mean marking all nodes as reachable.

Unreachable Procedure Elimination

1. Mark `main` as reachable.
2. Mark every procedure called from a marked procedure as reachable, repeat until no change.

Function pointers (indirect calls) have the same safety issues as `goto` above. Can improve by only adding functions which are address-taken somewhere in the program, rather than all procedures.

Live Variable Analysis (LVA)

- A variable is **semantically** live if changing its value affects the **observable** behaviour of the program.
- A variable is **syntactically** live if there is some path through the flowgraph to a node in which the current value might be **used** (not necessarily actually hit at runtime).

Safety constitutes overestimation, that syntactically live variables are a superset of semantically live variables.

Backwards flow equations. $\text{def}(n)$ is the set of variable assigned to in the node, $\text{ref}(n)$ is the set of variables used in the node.

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

$$\text{out-live}(n) = \bigcup_{s \in \text{succ}(n)} \text{in-live}(s)$$
$$\text{in-live}(n) = \text{out-live}(n) \setminus \text{def}(n) \cup \text{ref}(n)$$

For safety with **pointers**, treat pointer dereferences as referencing any address-taken variable, and assignments to a pointer dereference as not defining any variables: $\text{ref}(*p) = \{p, x, y, ...\}$ and $\text{def}(*p = x) = \{\}$.

Dataflow Anomalies

If a variable is live at entry to a procedure, then we can report a warning that a variable has been used before being assigned to.

Write-write anomalies (writing to a variable without ever reading the old value) can be detected by doing the opposite of LVA: go forwards, and union with def instead of ref (then each set holds the variables not yet read from). Any instruction which adds an existing variable to the set can be given a write-write anomaly warning.

Register Allocation by Colouring

- Generate a graph with virtual registers as nodes and edges between virtual registers which are simultaneously live. Virtual registers represent architectural registers in the IL, but are renamed to physical registers at code generation.

- Choose a virtual register with the lowest clashes (edges).
  - If the number of clashes is lower than the number of colours/physical registers then colour it and remove it from the graph.
  - Otherwise, mark the register as being spilled to memory (store it in a set) and remove it from the graph.

Spilled registers need relevant `MOV` instructions to be generated.

A **preference graph** can be built up to track extra information about the register dependencies to affect node selection decisions (such as number of accesses, whether the use is in a loop body, programmer `register` hints, ...).

Spilling

We need a spare register to hold variables which were spilled to memory: we may need more if multiple spilled registers are used as operands to the same instruction.

If we find that we need to spill during register allocation, just restart the register allocation with one register reserved as 'temporary' for the spilled value. If we find another spill, reserve another register, and so on (up to the max number of operands per instruction).

Non-orthogonal instructions

Instructions that always write to a predetermined register (eg. 'multiply and store the result in `r0`') mess up the allocation, as do procedure calls (which use eg. registers `r\{0..4\}` as argument registers and `r\{5..10\}` as being preserved over function calls).

- Reserve virtual registers for all architectural registers that have special usage (the registers used for results/function arguments).
- When an instruction/procedure call requires an operand in an architectural register, generate a preceding `mov` instruction from the source register to the virtual register (node) which represents the architectural register.
- Generate a trailing `mov` instruction from the virtual register to the destination register for instructions which output to a specific register.

Remember that the virtual registers **are** the architectural registers, they represent a logical mapping rather than eg. the difference between architectural and physical registers from CompArch. Moving data to the virtual register is just the register-allocation-stage way of moving data to the architectural register: at code generation, we specifically map the reserved arch registers to the real arch registers.

Available Expressions (AVAIL)

Available Expressions are those expressions that have been syntactically computed prior to that node and haven't been invalidated.

Forwards flow equations:

$$\mathrm{avail}(n) = \begin{cases} \bigcap_{p \in \mathrm{pred}(n)} \left(\mathrm{avail}(p) \setminus \mathrm{kill}(p) \cup \mathrm{gen}(p) \right) & \text{, if } \mathrm{pred}(n) \neq \emptyset \\ \emptyset & \text{, if } \mathrm{pred}(n) = \emptyset \end{cases}$$

$\mathrm{gen}(n)$ gives the expressions computed at n, but **minus any expressions changed by assignments**: $\mathrm{gen}(\texttt{x = y + z}) = \{\texttt{y + z}\}$, but $\mathrm{gen}(\texttt{x = x + y}) = \emptyset$, as the new value of `x` invalidates the expression. $\mathrm{kill(n)}$ contains the variables assigned to in the node.

Common Subexpression Elimination

If an expression is computed both at n and on a node on all paths to n, then we can arrange for all the prior computations to store their results in a local variable so we don't need to evaluate the expression

again.

1. For each node n containing `x = a ⊕ b` with `a ⊕ b` available at n:
2. Create a temporary variable `t` and replace the assignment in n with `x = t`.
3. On each path to n, for the latest computation of `y = a ⊕ b`, add an instruction `t = y`.

Static Single Assignment Form (SSA)

It can be advantageous to use different registers for a variable throughout its scope if it gets reassigned to: otherwise we have a form of false dependency (we depend on the data but are limited by the name).

SSA uses a new variable for every assignment, which **minimises the live ranges of each variable**, so allows for **more efficient register allocation**.

When paths merge in the flowgraph we need to ensure that temporaries of the same variable which were assigned to on the branches have their values updated into one variable. Can use ϕ functions, which are usually just a theoretical concept: `x = ϕ(x1, x2)` means `x = x1` if control took a left branch and `x = x2` if control took a right branch. The **register allocation pass** can arrange for `x1` and `x2` to be assigned to the **same register**. On some architectures this might not always be possible, so the final resort is to implement an actual branch to update the variables properly.

Phase Order

Different optimisations have different impacts depending on their relative ordering: even preference choices in one pass can alter the effectiveness of subsequent passes. Code motion optimisations are hard to predict the impact of.

Algebraic Identities

Simple syntax rewriting, replacing eg. `x = 1 + 2` with `x = 3` or `[1,2] ++ [3,4]` with `[1,2,3,4]`.

Strength Reduction

Generally, switch expensive instructions for cheaper ones. Specifically for loops:

1. Find induction variables which are used as `i = i ⊕ c`.
2. Find another variable `j = (i ⊗ c1) ⊕ c2`.
3. Move the assigment `j = (i ⊗ c1) ⊕ c2` to above the loop.
4. Add an end-of-loop-body instruction `j = j ⊕ (c1 ⊗ c)`.

Can then patch up the loop header using the known relation between `i` and `j`.

Example:

```
int total = 0;
for (int x = 0; x < 100; x += 10)
    total += a * x + b;
```

```
for (int tot = b; tot < a * 90 + b; )
    tot += 10 * a;
```

Abstract Interpretation

Abstract to reach a more easily-computable representation that allows for extracting properties.

$\otimes$	$(-)$	(0)	$(+)$
$(-)$	$(+)$	(0)	$(-)$
(0)	(0)	(0)	(0)
$(+)$	$(-)$	(0)	$(+)$

$\oplus$	$(-)$	(0)	$(+)$	$(?)$
$(-)$	$(-)$	$(-)$	$(?)$	$(?)$
(0)	$(-)$	(0)	$(+)$	$(?)$
$(+)$	$(?)$	$(+)$	$(+)$	$(?)$
$(?)$	$(?)$	$(?)$	$(?)$	$(?)$

Abstract interpretations can give different results depending on the syntax, even if the semantics are the same: $(x+1)(x-3)$ has abstract interpretation $(?)$ when $x = (-)$, but $x^2 - 2x - 3$ has interpretation $(+)$.

Strictness Analysis

Use-case of abstract interpretation.

Associate standard and abstract interpretations with the builtin functions (a_i and $a_i^\sharp$), and deduce standard and abstract interpretations for the user-defined functions (f_i and $f_i^\sharp$).

Assume the domain of the functions D is augmented with a value $\perp$ to represent a non-terminating value: $D \cup \{\perp\}$.

A function a is **strict** in its ith parameter iff $\forall d_i.\ a(d_1, \ldots, \perp, \ldots, d_n) = \perp$. For example, $+$ is strict in both parameters as:

$$\perp + y = \perp$$
$$x + \perp = \perp$$
$$x + y = x +_{\mathbb{Z}} y$$

Define the space of abstract values to be $D = \{0, 1\}$ where **0 represents guaranteed looping** and **1 represents possible termination**. Safety here means that we **underestimate** the strictness of functions (we don't want to alter semantics).

We can define the abstract interpretations of the builtin functions from their given semantics:

$$a^\sharp(x_1, ..., x_n) = \begin{cases} 0 & \text{, if } \forall d_i \in D \text{ where } (x_i = 0 \implies d_i = \perp), a(d_1, ..., d_n) = \perp \\ 1 & \text{, otherwise} \end{cases}$$

The weird condition just means 'if $x_i = 0$ then we force the corresponding semantic argument d_i to be $\perp$'. Essentially just generate $a^\sharp$ by trying every 0/1 input value for the arguments and seeing if a is terminating or not, then find an expression that matches that. For example, $x +^\sharp y = x \wedge y$ and $\text{cond}^\sharp(p, x, y) = p \wedge (x \vee y)$.

Can generate strictness values for user-defined functions by composing their builtin components. We can't compute it from scratch as it's undecidable (determining whether a functions doesn't terminate): we have to use explicitly specified builtin functions instead.

In general, need a least-fixed-point iteration over all of the user-defined functions so that updates to one function's strictness predicate updates the predicates of functions that use it. The start value for each strictness predicate should be 0, we assume no functions terminate, then we refine down to more accurate predicates using our knowledge of what definitely terminates.

Example: `plus(x, y) = cond(eq(x, 0), y, plus(sub1(x), add1(y)))`.

Given that $\text{plus}^\sharp(x, y) = x \wedge y$, $\text{eq}^\sharp(x, y) = x \wedge y$, $\text{add1}^\sharp(x) = x$, $\text{sub1}^\sharp(x) = x$, $0^\sharp = 1$.

1. $\text{plus}^\sharp(x, y) = 0$ (initial assumption)
2. $\text{plus}^\sharp(x, y) = \text{cond}^\sharp(\text{eq}^\sharp(x, 0), y, \text{plus}^\sharp(\text{sub1}^\sharp(x), \text{add1}^\sharp(x))) = x \wedge (y \vee 0) = x \wedge y$
3. $\text{plus}^\sharp(x, y) = \text{cond}^\sharp(\text{eq}^\sharp(x, 0), y, \text{plus}^\sharp(\text{sub1}^\sharp(x), \text{add1}^\sharp(x))) = x \wedge (y \vee (x \wedge y)) = x \wedge y$
4. No change, so we terminate with $\text{plus}^\sharp(x, y) = x \wedge y$

The strictness optimisation just allows us to evaluate arguments before passing them as thunk arguments.

Constraint-Based Analysis

Generally, traverse the syntax tree emitting constraints that are solved later (eg. if x is constrained to be even, we can later infer that `x + 1` is odd).

Control-Flow Analysis

Aim to compute the **set of values** (constants and lambda-abstractions) that any expression in the program can take.

Language is $e ::= x \mid c \mid \lambda x.\, e \mid e_1\, e_2 \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2$.

Define a **program point** to be a label which uniquely identifies a syntactic expression:
$(\texttt{let}\ \texttt{x}^2\ \texttt{=}\ \texttt{1}^3\ \texttt{in}\ (\texttt{x}^4\ \texttt{+}\ \texttt{2}^5)^6)^1$.

Associate a **flow variable** (representing the set of possible values) with each program point $(\alpha_1, ...)/$
The space of possible **flow values** for the example above are $\{\texttt{1}^3, \texttt{2}^5, (\texttt{x}^4\ \texttt{+}\ \texttt{2}^5)^6\}$.

Safety consists of overestimating the possible values computed by the expressions.

- For a term x^i, generate $\alpha_i \supseteq \alpha_j$ where α_j is the flow variable for the expression bound to x^i (eg. by $(\lambda x^i.\ ...)\ e^j$ or $\texttt{let}\ x^i\ \texttt{=}\ e^j\ \texttt{in}\ ...$). Don't think this is actually necessary as α_i should be constrained by the bindings themselves.
- For a term c^i, generate $\alpha_i \supseteq \{c^i\}$.
- For a term $(\lambda x^j.\ e^k)^i$, generate $\alpha_i \supseteq \{(\lambda x^j.\ e^k)^i\}$.
- For a term $(e_1^j\ e_2^k)^i$, generate $(\alpha_k \to \alpha_i) \supseteq \alpha_j$.
- For a term $(\texttt{let}\ x^l\ \texttt{=}\ e^j\ \texttt{in}\ e^k)^i$, generate $\alpha_i \supseteq \alpha_k$ and $\alpha_l \supseteq \alpha_j$.
- For a term $(\texttt{if}\ e_1^j\ \texttt{then}\ e_2^k\ \texttt{else}\ e_3^l)^i$, generate $\alpha_i \supseteq \alpha_k$ and $\alpha_i \supseteq \alpha_k$.

$(\alpha_k \to \alpha_i) \supseteq \alpha_j$ is shorthand for 'whenever $(\lambda x^a.\ e^b)^c \in \alpha_j$ then $\alpha_a \supseteq \alpha_k$ and $\alpha_i \supseteq \alpha_b$'. This means that **any of the actual parameters become possible values of the formal parameters, and the possible values of the application are the possible values of the function bodies**.

This approach is monovariant (0-CFA), which imposes the inaccuracy that different invocations of a function all affect the same set of possible values, rather than just a call-site specific version. A polyvariant approach fixes this.

Inference-Based Program Analysis

Generally, use a judgement system like $\Gamma \vdash e : \phi$ to infer properties from the terms of the language. Type systems are a specific example, but can be generalised:

$$\Gamma = \{2 : \text{even}, (+) : (\text{even} \to \text{even} \to \text{even}) \wedge (\text{even} \to \text{odd} \to \text{odd}) \wedge ..., (\times) : ...\}$$
$$\Gamma \vdash (\lambda x.\ \lambda y.\ 2 \times x + y) : ...$$

Effect Systems

Specific example of inference-based program analysis, modelling communicating systems.

Model effects as W_C and R_C for write/reads to a channel C, use terms $C!e_1.\ e_2$ for writing e_1 and $C?x.\ e$ for reading x.

Judgements of the form $\Gamma \vdash e : t, F$, with immediate effects F. Latent effects are reresented by tags on function types, $t_1 \xrightarrow{F} t_2$.

$$\frac{x : t \in \Gamma}{\Gamma \vdash x : t, \emptyset} \text{ (var)} \qquad \frac{\Gamma, x : t_1 \vdash e : t_2, F}{\Gamma \vdash (\lambda x.\ e) : t_1 \xrightarrow{F} t_2, \emptyset} \text{ (abs)} \qquad \frac{\Gamma \vdash e_1 : t_1 \xrightarrow{F_l} t_2 \quad \Gamma \vdash e_2 : t_1, F_2}{\Gamma \vdash e_1\ e_2 : t_2, F_1 \cup F_2 \cup F_l} \text{ (app)}$$

$$\frac{\Gamma, x : \text{int} \vdash e : t, F}{\Gamma \vdash (C?x.\ e) : t, F \cup \{R_C\}} \text{ (read)} \qquad \frac{\Gamma \vdash e_1 : \text{int}, F_1 \quad \Gamma \vdash e_2 : t, F_2}{\Gamma \vdash (C!e_1.\ e_2) : t, F_1 \cup F_2 \cup \{W_C\}} \text{ (write)}$$

$$\frac{\Gamma \vdash e : t_1 \xrightarrow{F_l} t_2, F \quad F_l \subseteq F_l'}{\Gamma \vdash e : t_1 \xrightarrow{F_l'} t_2, F} \text{ (sub)}$$

The sub rule is necessary to patch-up the types when there's eg. a conditional expression where the two branches are functions with different latent effects.

Can change the representation of effects to carry more information: allow writing values of types other than int, or use lists instead of sets to capture ordering information.