COMPILER CONSTRUCTION · WEEK 9 · STATION III

Optimisation and Rice

Every optimiser is an approximation. Sound or complete, never both — and register allocation is NP-hard.

WHAT AN OPTIMISER CLAIMS

"This program means the same, and costs less." Both halves are about behaviour.

An optimisation replaces a program by another. The claim is that they are equivalent on every input, and that the second is cheaper. Equivalence of programs is a property of behaviour, so by Rice it is undecidable in general. Cost is measured on a machine, so it is a metric, and metrics are approximations too.

Hence every optimiser answers a decidable stand-in: not "is this code dead?" but "is there a path in the text to this code?"; not "is this value constant?" but "is it assigned once, from a literal, in this block?". Text questions that imply the behaviour question one way — sound — and miss cases the other way — incomplete.

WHY SELFIE DOES NOT

Not because it is hard to add. Because every optimisation is a place where the compiler's output stops being a transcription of the source, and the class is about the transcription. The exercises below show what would change.

TWO CLASSICS

Constant folding and dead code elimination.

```
// before
x = 6 * 7;
if (0) y = 1;
while (c != 0) c = c + 2;
z = 2; // never read again
// after folding and dead code
x = 42;
while (c != 0) c = c + 2;
```

Folding. `6 * 7` is a term with two literal factors: the compiler can evaluate it at compile time and emit one `addi`. Sound, because the machine's arithmetic and the compiler's are the same 64-bit wrap — which is a claim that has to be true of the compiler's own host.

Dead code. `if (0)` never runs; the text says so. `z = 2` is dead if nothing reads `z` afterwards — and "afterwards" is a path in the text, not an execution. The loop stays: whether it runs forever is not the compiler's to know, and removing a non-terminating loop changes behaviour.

Every one of these decisions is about the text, standing in for a fact about behaviour. The optimiser's contract is that the stand-in is sound; its quality is how often it fires.

REGISTERS

Register allocation is graph colouring, and graph colouring is NP-hard.

Selfie's stack of seven temporaries is the simplest allocator there is, and it fails at nesting depth eight. A real allocator asks: which values are *live* at the same time, and can two values that are never live together share a register?

Draw a node per value and an edge between values live simultaneously. Colouring the graph with k colours, one per register, so that no edge joins two nodes of one colour, is register allocation with k registers. Chaitin, 1981. And k-colouring is NP-complete.

Liveness itself — is this value read again? — is a property of behaviour. The allocator uses the text's paths as the proxy, so the graph is already an approximation before the colouring starts.

WHICH IS WHERE WEEK 11 BEGINS

Decidable, and unaffordable in the worst case. Compilers colour greedily, spill to memory when they fail, and get within a few percent of optimal on real code. Hardness in the worst case, structure in the typical one — the whole of station IV in one pass of the compiler.

TWO WAYS TO BE WRONG

Sound but incomplete, or complete but unsound. Choosing is the discipline.

SOUND, INCOMPLETE

Everything it says is true; it does not say everything. A type checker. A dead-code pass that only removes what the text proves unreachable. A model checker's "no" up to k. Compilers must be here: an unsound optimisation is a miscompilation.

COMPLETE, UNSOUND

It finds everything, and some of what it finds is not there. A linter that warns on every suspicious pattern. A fuzzer's crash that is a bug in the harness. Useful for humans, fatal for a compiler.

Rice forbids sound and complete together for any interesting property. Every tool in the workshop around selfie is a stated choice: rotor and bitme sound within a bound, buzzr complete about crashes it sees, the type checker sound about pointers. The next three weeks build the first of these.

ASSIGNMENT

`array-allocation`: arrays in the data segment and on the stack.

1. Specify. Extend the grammar so that a variable declaration may be `type identifier [ integer ]`, globally and locally.
2. Allocate. Globally: `n` words in the data segment, and the symbol table entry records the size. Locally: `n` words in the frame, so the prologue's `addi sp,sp,-locals` grows, and offsets from `s0` shift.
3. Access. Last week's `a[i]` now has two cases: the identifier is a pointer, or it is an array, whose address is its location rather than its value. The symbol table must say which. Decide the semantics, then implement.
4. `./grader/self.py array-allocation`, `self-compile`, and use a global array in selfie.c.

WATCH

Bounds are not checked: `a[n]` on an array of `n` words reads the neighbour. That is C, it is decided, and week 12 is the first week that can find such an access for you.