1 The Factor Compiler

If we could sort programming languages by the fuzzy notions we tend to have about how "high-level" they are, toward the high end we'd find dynamically-typed languages like Python, Ruby, and PHP—all of which are generally more interpreted than compiled. Despite being as high-level as these popular languages, Factor's implementation is driven by performance. Factor source is always compiled to native machine code using either its simple, non-optimizing compiler or (more typically) the optimizing compiler that performs several sorts of data and control flow analyses. In this section, we look at the general architecture of Factor's implementation, after which we place a particular emphasis on the transformations performed by the optimizing compiler.

Though there are projects for this

1.1 Low-level Optimizations

The low-level intermediate representation (IR) in `compiler.cfg` takes the more conventional form of a control flow graph (CFG). A CFG (not to be confused with "context-free grammar") is an arrangement of instructions into *basic blocks*: maximal sequences of "straight-line" code, where control does not transfer out of or into the middle of the block. Directed edges are added between blocks to represent control flow—either from a branching instruction to its target, or from the end of a basic block to the start of the next one. Construction of the low-level IR proceeds by analyzing the control flow of the high-level IR and converting the nodes of **??** into lower-level, more conventional instructions modeled after typical assembly code. There are over a hundred of these instructions, but many are simply different versions of the same operation. For instance, while instructions are generally called on *virtual registers* (represented in Factor simply by integers), there are *immediate* versions of instructions. The `##add` instruction, as an example, represents the sum of the contents of two registers, but `##add-imm` sums the contents of one register and an integer literal. Other instructions are inserted to make stack reads and writes explicit, as well as to balance the height. Below is a categorized list of all the instruction objects (each one is a subclass of the `insn` tuple).

cite

Is the complete list really necessary?

- Loading constants: `##load-integer`, `##load-reference`
- Optimized loading of constants, inserted by representation selection: `##load-tagged`, `##load-float`, `##load-double`, `##load-vector`
- Stack operations: `##peek`, `##replace`, `##replace-imm`, `##inc-d`, `##inc-r`
- Subroutine calls: `##call`, `##jump`, `##prologue`, `##epilogue`, `##return`
- Inhibiting tail-call optimization (TCO): `##no-tco`
- Jump tables: `##dispatch`
- Slot access: `##slot`, `##slot-imm`, `##set-slot`, `##set-slot-imm`
- Register transfers: `##copy`, `##tagged>integer`

- Integer arithmetic: `##add`, `##add-imm`, `##sub`, `##sub-imm`, `##mul`, `##mul-imm`, `##and`, `##and-imm`, `##or`, `##or-imm`, `##xor`, `##xor-imm`, `##shl`, `##shl-imm`, `##shr`, `##shr-imm`, `##sar`, `##sar-imm`, `##min`, `##max`, `##not`, `##neg`, `##log2`, `##bit-count`
- Float arithmetic: `##add-float`, `##sub-float`, `##mul-float`, `##div-float`, `##min-float`, `##max-float`, `##sqrt`
- Single/double float conversion: `##single>double-float`, `##double>single-float`
- Float/integer conversion: `##float>integer`, `##integer>float`
- SIMD operations: `##zero-vector`, `##fill-vector`, `##gather-vector-2`, `##gather-int-vector-2`, `##gather-vector-4`, `##gather-int-vector-4`, `##select-vector`, `##shuffle-vector`, `##shuffle-vector-halves-imm`, `##shuffle-vector-imm`, `##tail>head-vector`, `##merge-vector-head`, `##merge-vector-tail`, `##float-pack-vector`, `##signed-pack-vector`, `##unsigned-pack-vector`, `##unpack-vector-head`, `##unpack-vector-tail`, `##integer>float-vector`, `##float>integer-vector`, `##compare-vector`, `##test-vector`, `##test-vector-branch`, `##add-vector`, `##saturated-add-vector`, `##add-sub-vector`, `##sub-vector`, `##saturated-sub-vector`, `##mul-vector`, `##mul-high-vector`, `##mul-horizontal-add-vector`, `##saturated-mul-vector`, `##div-vector`, `##min-vector`, `##max-vector`, `##avg-vector`, `##dot-vector`, `##sad-vector`, `##horizontal-add-vector`, `##horizontal-sub-vector`, `##horizontal-shl-vector-imm`, `##horizontal-shr-vector-imm`, `##abs-vector`, `##sqrt-vector`, `##and-vector`, `##andn-vector`, `##or-vector`, `##xor-vector`, `##not-vector`, `##shl-vector-imm`, `##shr-vector-imm`, `##shl-vector`, `##shr-vector`
- Scalar/vector conversion: `##scalar>integer`, `##integer>scalar`, `##vector>scalar`, `##scalar>vector`
- Boxing and unboxing aliens: `##box-alien`, `##box-displaced-alien`, `##unbox-any-c-ptr`, `##unbox-alien`
- Zero-extending and sign-extending integers: `##convert-integer`
- Raw memory access: `##load-memory`, `##load-memory-imm`, `##store-memory`, `##store-memory-imm`
- Memory allocation: `##allot`, `##write-barrier`, `##write-barrier-imm`, `##alien-global`, `##vm-field`, `##set-vm-field`
- The foreign function interface (FFI): `##unbox`, `##unbox-long-long`, `##local-allot`, `##box`, `##box-long-long`, `##alien-invoke`, `##alien-indirect`, `##alien-assembly`, `##callback-inputs`, `##callback-outputs`
- Control flow: `##phi`, `##branch`
- Tagged conditionals: `##compare-branch`, `##compare-imm-branch`, `##compare`, `##compare-imm`

- Integer conditionals: `##compare-integer-branch`, `##compare-integer-imm-branch`, `##test-branch`, `##test-imm-branch`, `##compare-integer`, `##compare-integer-imm`, `##test`, `##test-imm`
- Float conditionals: `##compare-float-ordered-branch`, `##compare-float-unordered-branch`, `##compare-float-ordered`, `##compare-float-unordered`
- Overflowing arithmetic: `##fixnum-add`, `##fixnum-sub`, `##fixnum-mul`
- Garbage collector (GC) checks: `##save-context`, `##check-nursery-branch`, `##call-gc`
- Spills and reloads, inserted by the register allocator: `##spill`, `##reload`

```
: optimize-cfg ( cfg -- cfg' )
    optimize-tail-calls
    delete-useless-conditionals
    split-branches
    join-blocks
    normalize-height
    construct-ssa
    alias-analysis
    value-numbering
    copy-propagation
    eliminate-dead-code ;
```

Listing 1: Optimization passes on the low-level IR

By translating the high-level IR into instructions that manipulate registers directly, we reveal further redundancies that can be optimized away. The `optimize-cfg` word in Listing 1 shows the passes performed in doing this. The first word, `optimize-tail-calls`, performs tail call elimination on the CFG. *Tail calls* are those that occur within a procedure and whose results are immediately returned by that procedure. Instead of allocating a new call stack frame, we may convert tail calls into simple jumps, since afterwards the current procedure's call frame isn't really needed. In the case of recursive tail calls, we can convert special cases of recursion into loops in the CFG, so that we won't trigger call stack overflows. For instance, consider Figure 1 on the next page, which shows the effect of `optimize-tail-calls` on the following definition:

used in ??, not defined

```
: tail-call ( -- ) tail-call ;
```

Note the recursive call (trivially) occurs at the end of the definition, just before the return point. When translated to a CFG, this is a `##call` instruction, as seen in block 4 to the left of Figure 1 on the following page. This is also just before the final `##epilogue` and `##return` instructions in block 8, as blocks 5–7 are effectively empty (these excessive `##branch`es will be eliminated in a later pass). Because of this, rather than make a whole new subroutine call, we can convert it into a `##branch` back to the beginning of the word, as in the CFG to the right.

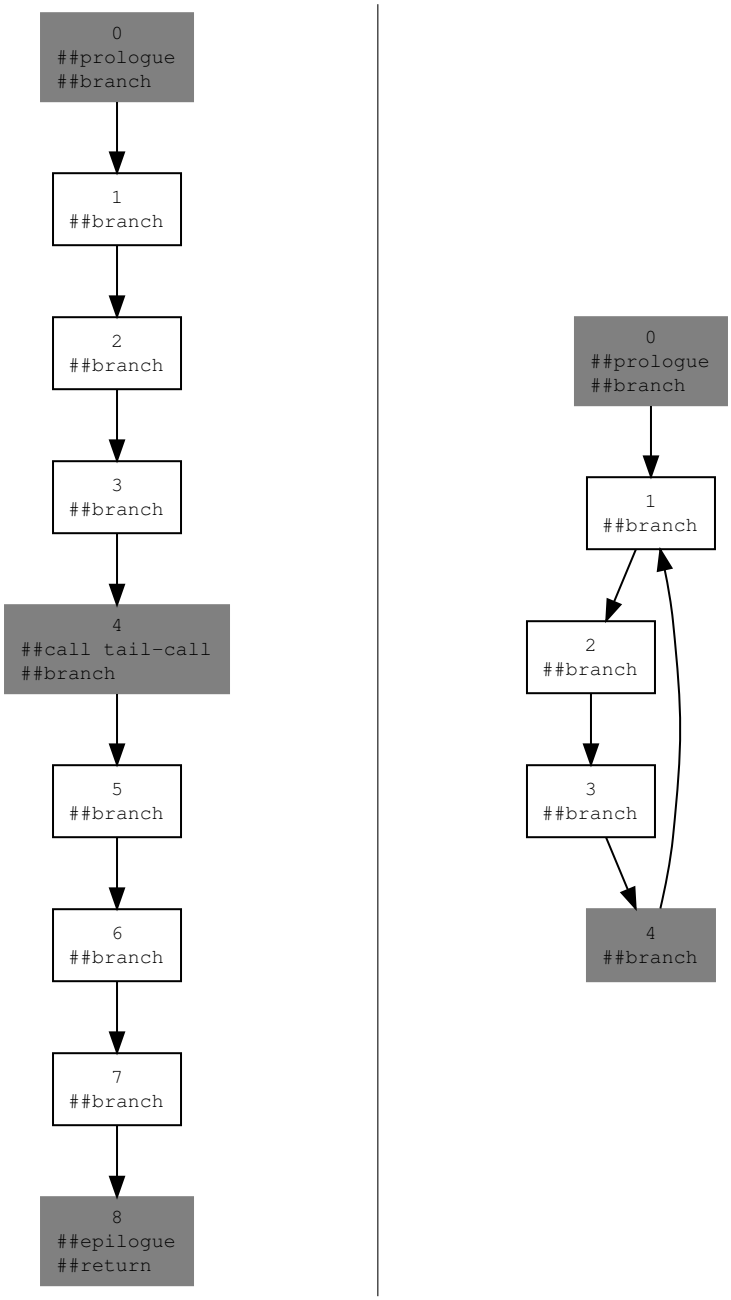

Figure 1: `tail-call` before and after `optimize-tail-calls`

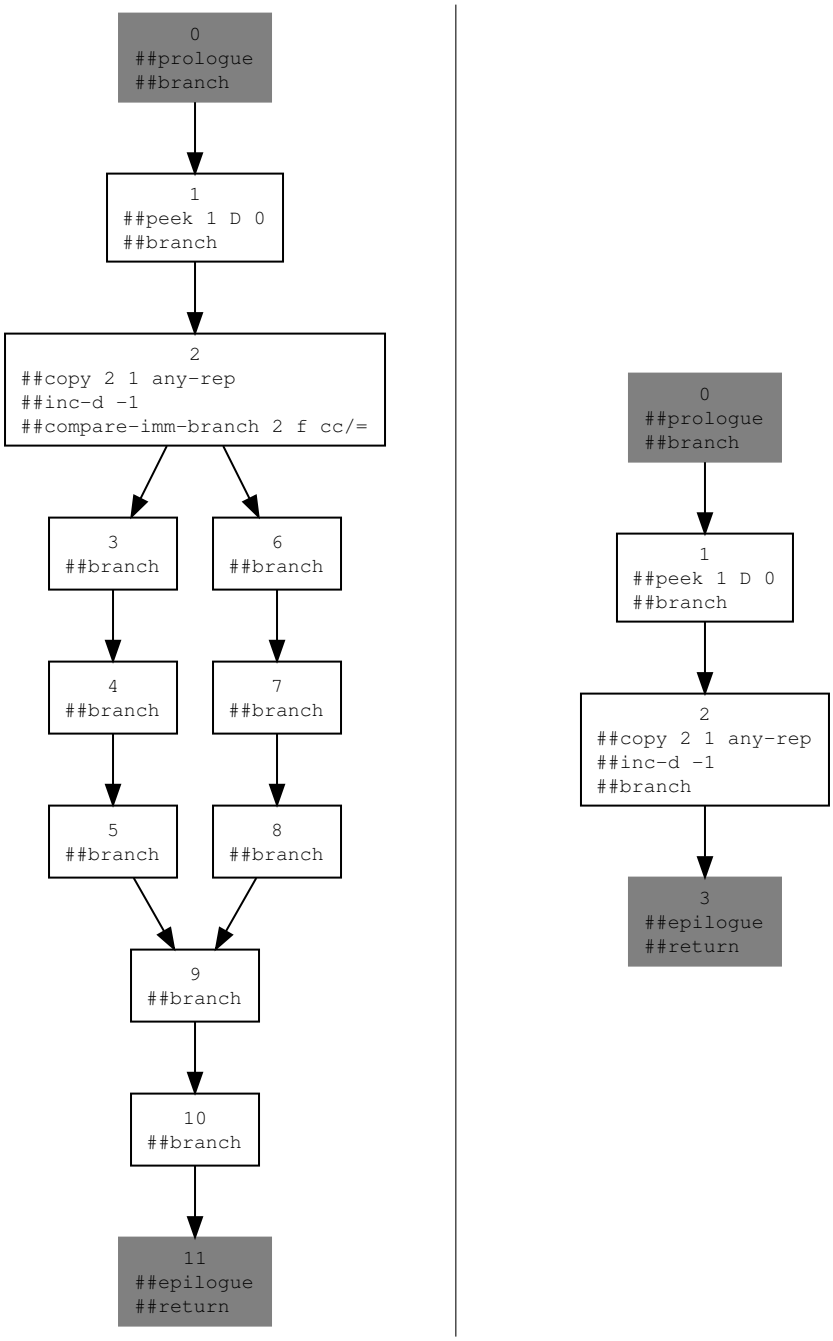

Figure 2: `[ ] [ ] if` before and after `delete-useless-conditionals`

The next pass in Listing 1 on page 3 is `delete-useless-conditionals`, which removes branches that go to the same basic block. This situation might occur as a result of optimizations performed in the high-level IR. To see it in action, Figure 2 on the preceding page shows the transformation on a purposefully useless conditional, `[ ] [ ] if`. Before removing the useless conditional, the CFG `##peek`s at the top of the data stack (`D 0`), storing the result in the virtual register `1`. This value is popped, so we decrement the stack height (`##inc-d -1`). Then, `##compare-imm-branch` in block 2 compares the value in the virtual register `2` (which is a copy of `1`, the top of the stack) to the immediate value `f` to see if it's not equal (signified by `cc/=`). However, both branches jump through several empty blocks and merge at the same destination. Thus, we can remove both branches and replace `##compare-imm-branch` with an unconditional `##branch` to the eventual destination. We see this on the right of Figure 2 on the previous page.

In order to expose more opportunities for optimization, `split-branches` will actually duplicate code. We use the fact that code immediately following a conditional will be executed along either branch. If it's sufficiently short, we copy it up into the branches individually. That is, we change `[ A ] [ B ] if C` into `[ A C ] [ B C ] if`, as long as `C` is small enough. Later analyses may then, for example, more readily eliminate one of the branches if it's never taken. Figure 3 on the following page shows what such a transformation looks like on a CFG. The example `[ 1 ] [ 2 ] if dup` is essentially changed into `[ 1 dup ] [ 2 dup ] if`, thus splitting the block with two predecessors (block 9) on the left.

The next pass, `join-blocks`, compacts the CFG by joining together blocks involved in only a single control flow edge. Mostly, this is to clean up the myriad of empty or short blocks introduced during construction, like sequences of a bunch of `##branch`es. Figure 4 on page 8 shows this pass on the CFG of `0 100 [ 1 fixnum+fast ] times`. `fixnum+fast` is a specialized version of `+` that suppresses overflow and type checks. We use it here to keep the CFG simple. We'll be using this particular code to illustrate all but one of the remaining optimization passes in Listing 1 on page 3, as it's a motivating example for the work in this thesis. The passes before `join-blocks` don't change the CFG seen on the left in Figure 4 on page 8, but we get rid of the useless `##branch` blocks in the CFG on the right.

Figure 5 on page 9 shows the result of applying `normalize-height` to the result of `join-blocks`. This phase combines and canonicalizes the instructions that track the stack height, like `##inc-d`. While the shuffling in this example isn't complex enough to be interesting, neither is this phase. It amounts to more cleanup: multiple height changes are combined into single ones at the beginnings of the basic blocks. In Figure 5 on page 9, this means that `##inc-d` is moved to the top of block 1, as compared to the right of Figure 4 on page 8.

In converting the high-level IR to the low-level, we actually lose the static single assignment (SSA) form of `compiler.tree`. Not only does the construction do this, but `split-branches` also copies basic blocks verbatim, so any value defined will have a duplicate definition site, violating the SSA property. `construct-ssa` recomputes a so-called *pruned* SSA form, wherein ϕ functions are inserted only if the variables are live after the insertion point. This cuts down on useless ϕ functions. Figure 6 on page 10 shows the reconstructed SSA form of the CFG from Figure 5 on page 9.

cite TDMSC and construction algorithm

The next pass, `alias-analysis`, doesn't change the CFG of `0 100 [ 1 fixnum+fast ] times`, so we won't have an accompanying figure. At a high level, `alias-analysis` is easy to understand: it eliminates redundant memory loads and stores by rewriting certain patterns of memory access.

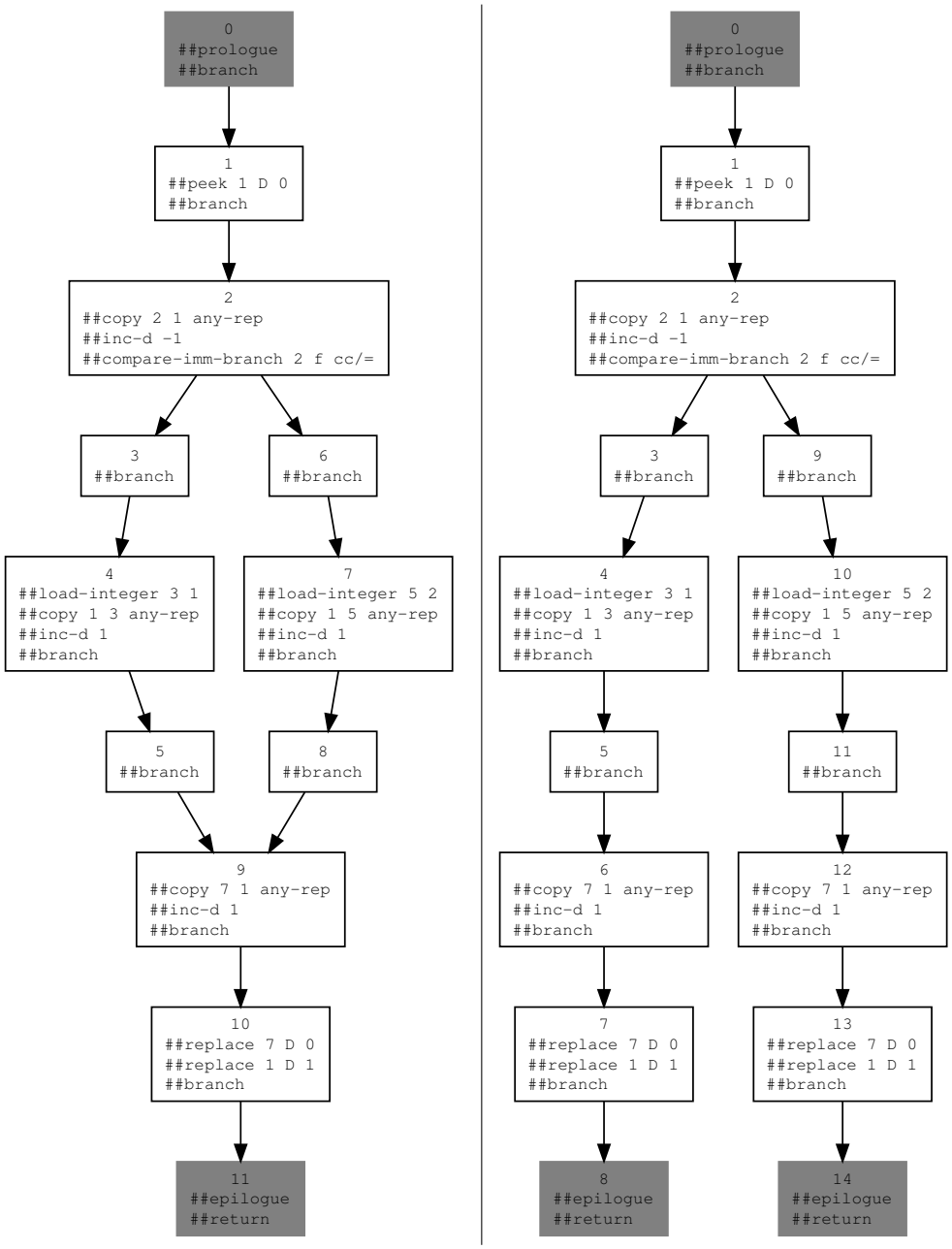

Figure 3: `[ 1 ] [ 2 ]` **`if dup`** before and after `split-branches`

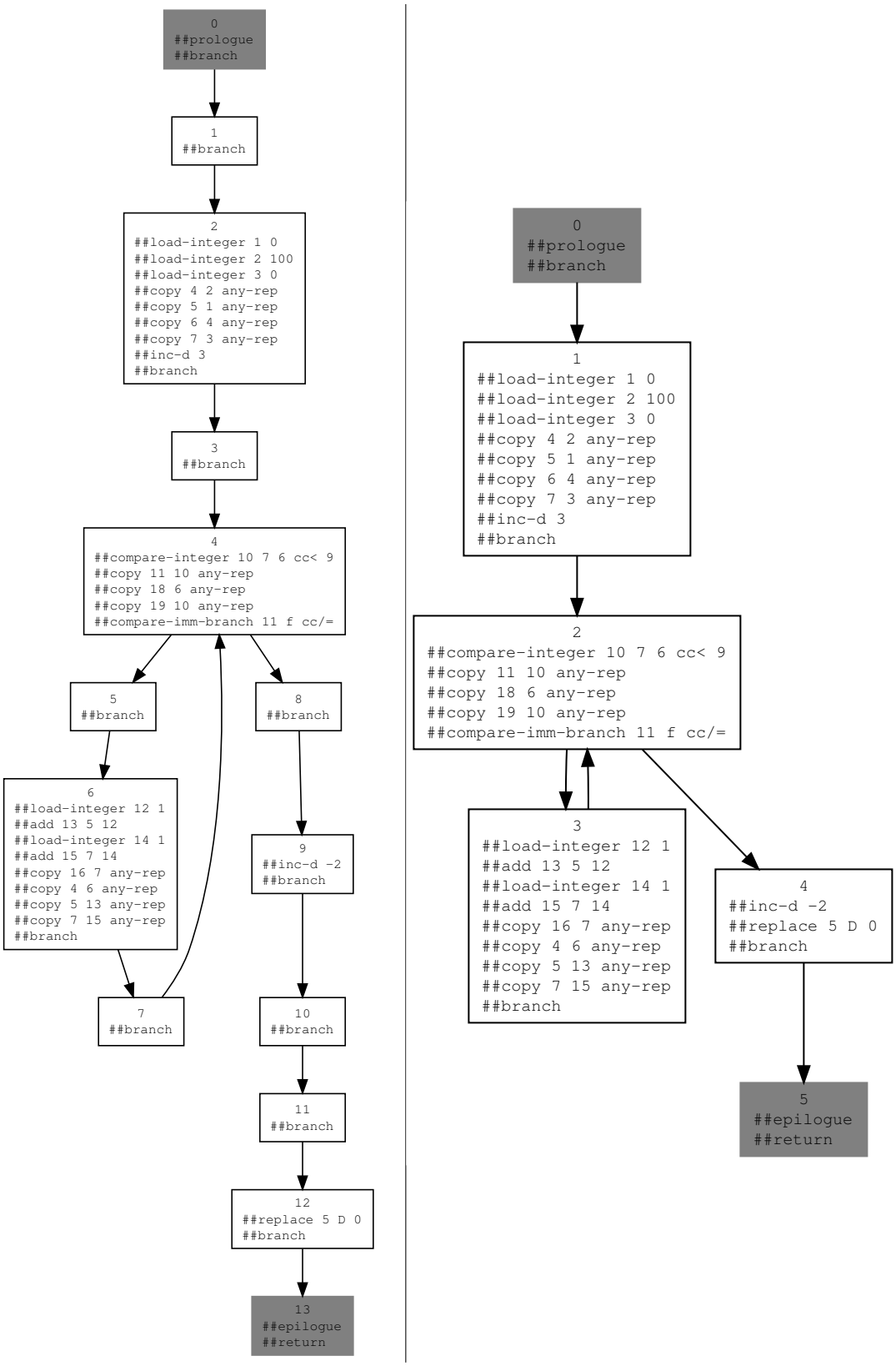

Figure 4: `0 100 [ 1 fixnum+fast ] times` before and after `join-blocks`

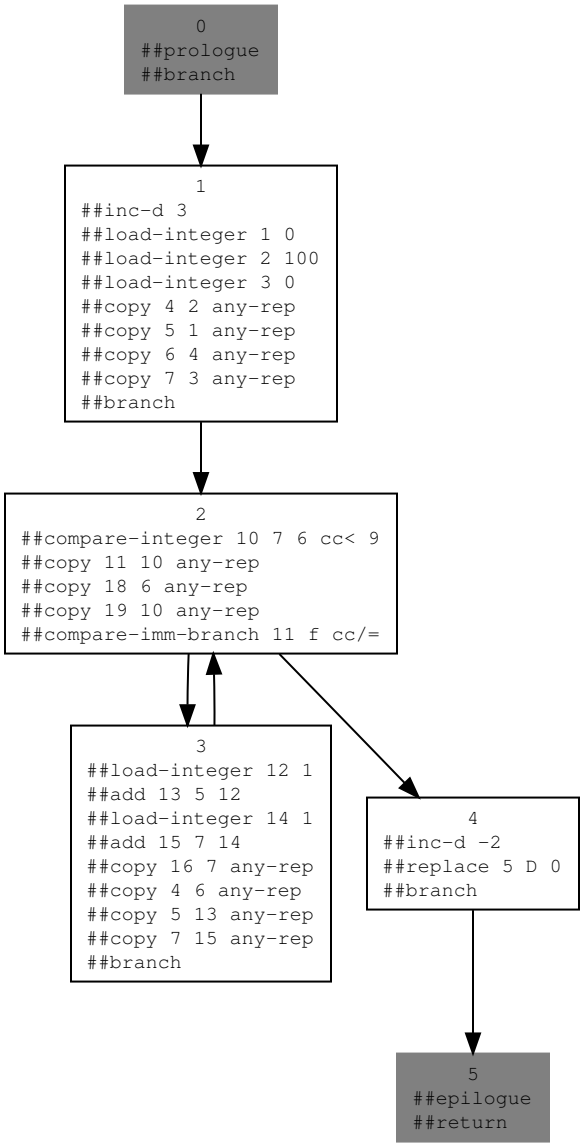

Figure 5: `0 100 [ 1 fixnum+fast ] times` after `normalize-height`

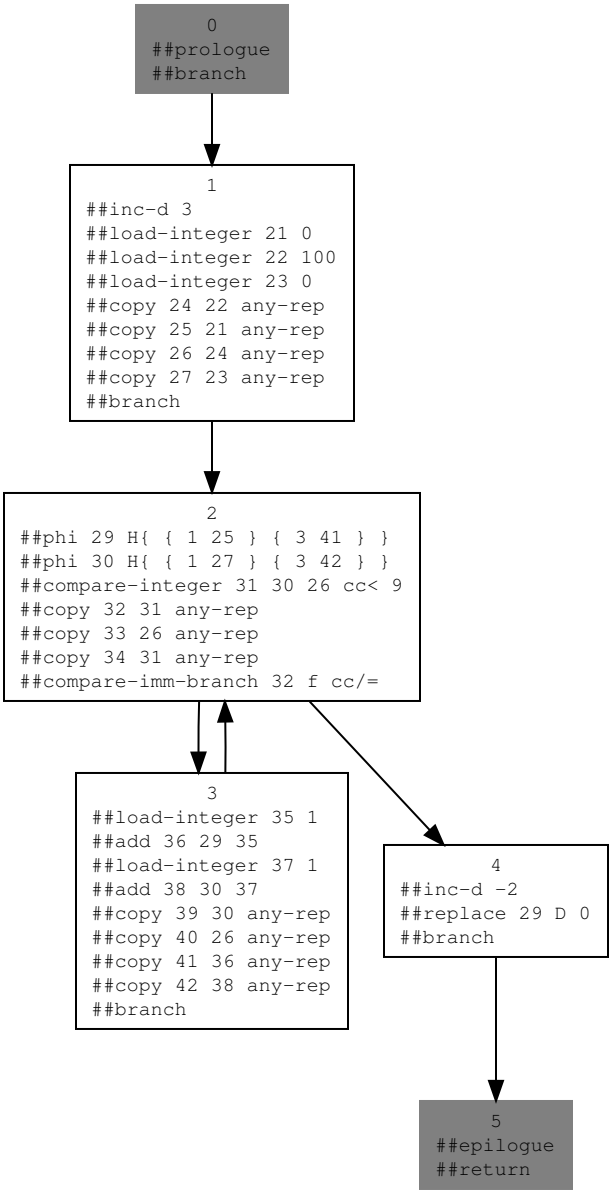

Figure 6: `0 100 [ 1 fixnum+fast ] times` after `construct-ssa`

If the same location is loaded after being stored, we convert the latter load into a `##copy` of the value we stored. Two reads of the same location with no intermittent write gets the second read turned into a `##copy`. Similarly, if we see two writes without a read in the middle, the first write can be removed.

`value-numbering` is the key focus of this thesis. It will be detailed in **??** on page ??. For now, it does to think of it as a combination of common subexpression elimination and constant folding. In Figure 7 on the following page, we see several changes:

- `##load-integer 23 0` in block 1 of Figure 6 on the previous page (which assigns the value `0` to the virtual register `23`) is redundant, so is replaced by `##copy 23 21`.
- In block 2, the last instruction `##compare-imm-branch 32 f cc/=` is the same as `##compare-integer-branch 30 26 cc<`. The source register (`32`) of the original is a `##copy` of `31`, which itself is computed by `##compare-integer 31 30 26 cc< 9`. So, the `##compare-imm-branch` is equivalent to a simple `##compare-integer-branch`, which doesn't use the temporary virtual register `9` and doesn't waste time comparing against the `f` object.
- The second operands in both `##add`s of block 3 are just constants stored by `##load-integer`s. So, these are turned into `##add-imm`s.
- Also, the second `##load-integer` in block 3 just loads `1` like the first instruction. Therefore, it's replaced by a `##copy`.

In **??**, we'll see how and why this pass fails to identify other equivalences.

Following `value-numbering`, `copy-propagation` performs a global pass that eliminates `##copy` instructions. Uses of the copies are replaced by the originals. So, in Figure 8 on page 13, we can see that all of the `##copy` instructions have been removed and, for instance, the use of the virtual register `25` in block 2 has been replaced by `21`, since `25` was a copy of it.

Next, dead code is removed by `eliminate-dead-code`. Figure 9 on page 14 shows that the `##compare-integer` in block 2 and the `##load-integer` in block 3 were removed, since they defined values that were never used.

The final pass in Listing 1 on page 3, `finalize-cfg`, itself consists of several more passes. We will not get into many details here, but at a high level, the most important passes figure out how virtual registers should map to machine registers. We first figure out when certain values can be unboxed. Then, instructions are reordered in order to reduce *register pressure*. That is, we try to schedule instructions around each other so that we don't need to store more values than we have machine registers. That way, we avoid *spilling* registers onto the heap, which wastes time. After leaving SSA form, we perform a *linear scan* register allocation, which replaces virtual registers with machine registers and inserts `##spill` and `##reload` instructions for the cases we can't avoid. Figure 10 on page 15 shows an example on an Intel x86 machine, which has enough registers that we needn't spill anything.

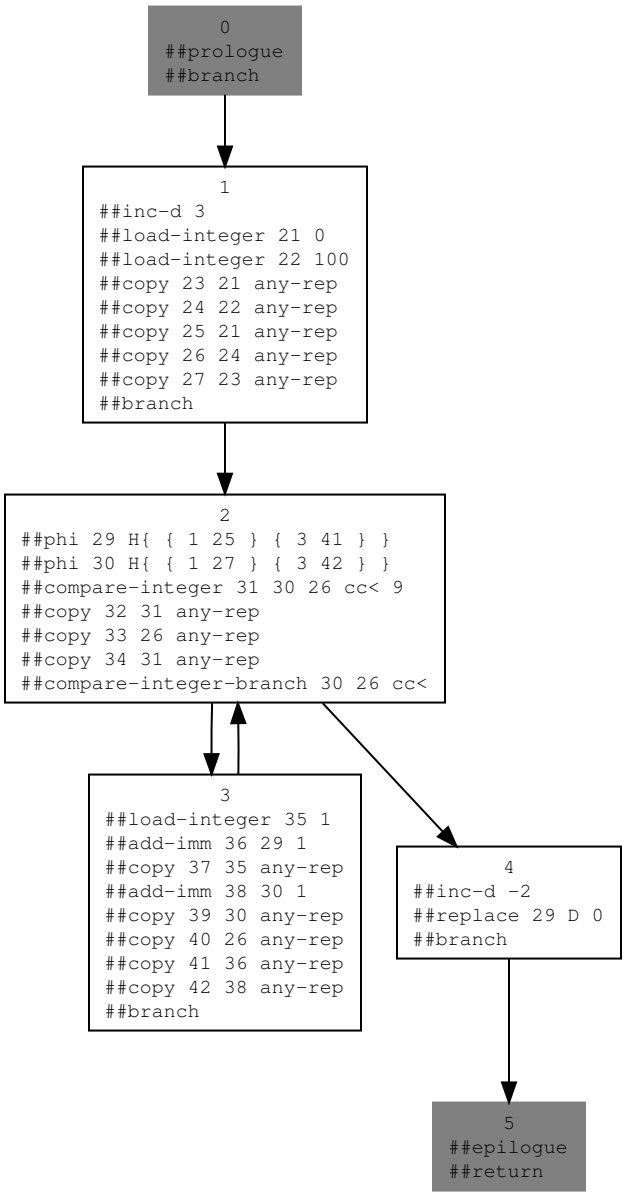

Figure 7: `0 100 [ 1 fixnum+fast ] times` after `value-numbering`

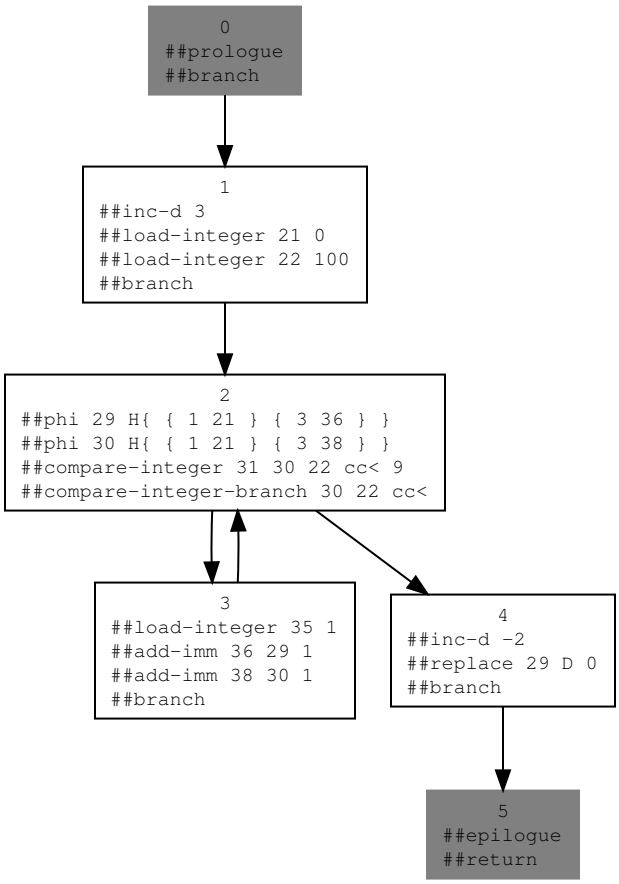

Figure 8: `0 100 [ 1 fixnum+fast ]` **`times`** after `copy-propagation`

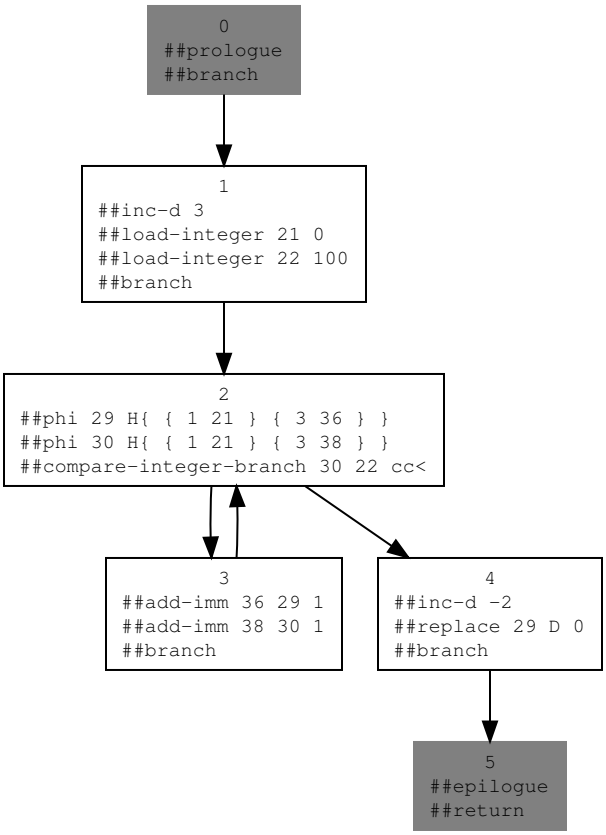

Figure 9: `0 100 [ 1 fixnum+fast ]` **`times`** after `eliminate-dead-code`

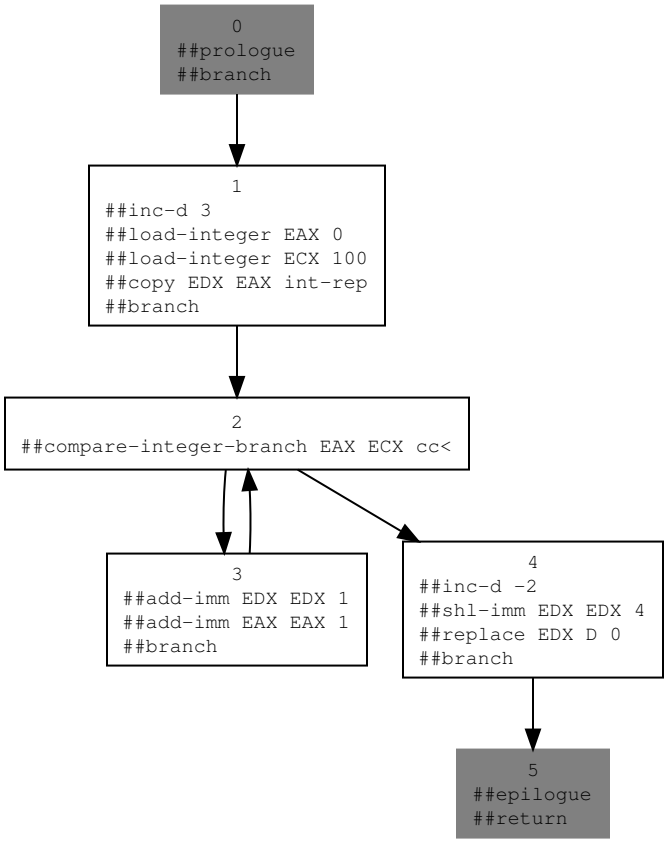

Figure 10: `0 100 [ 1 fixnum+fast ] times` after `finalize-cfg`