GLOBAL VALUE NUMBERING IN FACTOR

A Thesis

Presented to the

Faculty of

California State Polytechnic University, Pomona

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

In

Computer Science

By

Alex Vondrak

2011

SIGNATURE PAGE

THESIS:	GLOBAL VALUE NUMBERING IN FACTOR
AUTHOR:	Alex Vondrak
DATE SUBMITTED:	Summer 2011
	Computer Science
Dr. Craig Rich Thesis Committee Chair Computer Science	______________________________
Dr. Daisy Sang Computer Science	______________________________
Dr. Amar Raheja Computer Science	______________________________

Acknowledgments

To Lindsay—he is my rock

Abstract

Compilers translate code in one programming language into semantically equivalent code in another language—canonically from a high-level language to low-level machine primitives. Generally, the further removed a language's abstractions get from those of a computer, the harder it gets to compile code into an efficient representation. What isn't redundant in the source language may map to repetitive target instructions that waste time recomputing results. To combat this, compilers try to optimize away redundancies by looking for values that are provably equivalent when the program is run.

This thesis explores the theory and implementation of a particularly aggressive analysis called global value numbering in a particularly high-level language called Factor. Factor is a stack-based, dynamically-typed, object-oriented language born in late 2003. A baby among languages (now at version 0.94), its compiler craves all the optimizations it can get. By altering the existing local value numbering pass, redundancies can be identified and eliminated across entire programs, rather than isolated regions of code. This induces speedups as high as 45% across the majority of benchmarks. The results from these comparatively simple changes hold much promise for future improvements in making Factor programs more efficient.

Table of Contents

List of Figures

1 Introduction

Compilers translate programs written in a source language (e.g., Java) into semantically equivalent programs in some target language (e.g., assembly code). They let us make our source language arbitrarily abstract so we can write programs in ways that humans understand while letting the computer execute programs in ways that machines understand. In a perfect world, such translation would be straightforward. Reality, however, is unforgiving. Straightforward compilation results in clunky target code that performs a lot of redundant computations. To produce efficient code, we must rely on less-than-straightforward methods. Typical compilers go through a stage of *optimization*, whereby a number of semantics-preserving transformations are applied to an *intermediate representation* of the source code. These then (hopefully) produce a more efficient version of said representation. Optimizers tend to work in *phases*, applying specific transformations during any given phase.

Global value numbering (GVN) is such a phase performed by many highly-optimizing compilers. Its roots run deep through both the theoretical and the practical. Using the results of this analysis, the compiler can identify expressions in the source code that produce the same value—not just by lexical comparison (i.e., comparing variable names), but by proving equivalences between what's actually computed at runtime. These expressions can then be simplified by further algorithms for redundancy elimination. This is the very essence of most compiler optimizations: avoid redundant computation, giving us code that runs as quickly as possible while still following what the programmer originally wrote.

High-level, dynamic languages tend to suffer from efficiency issues. They're often interpreted rather than compiled, and perform no heavy optimization of the source code. However, the Factor language (`http://factorcode.org`) fills an intriguing design niche, as it's very high-level yet still fully compiled. It's still young, though, so its compiler craves all the improvements it can get. In particular, while the current Factor version (as of this writing, 0.94) has a *local* value numbering analysis, it is inferior to GVN in several significant ways.

In this thesis, we explore the implementation and use of GVN in improving the strength of optimizations in Factor. Because Factor is a young and relatively unknown language, Chapter 2 provides a short tutorial, laying a foundation for understanding the changes. Chapter 3 describes the overall architecture of the Factor compiler, highlighting where the exact contributions of this thesis fit in. Finally, Chapter 4 goes into detail about the existing and new value numbering passes, closing with a look at the results achieved and directions for future work.

In the unlikely event that you want to cite this thesis, you may use the following BibTeX entry:

```
@mastersthesis{vondrak:11,
  author = {Alex Vondrak},
  title  = {Global Value Numbering in Factor},
  school = {California Polytechnic State University, Pomona},
  month  = sep,
  year   = {2011},
}
```

2 Language Primer

Factor is a rather young language created by Slava Pestov in September 2003 [*Factor* 2010]. Its first incarnation was an embedded scripting language for a game that targeted the Java Virtual Machine (JVM). As such, its feature set was minimal. Factor has since evolved into a general-purpose programming language, gaining new features and redesigning old ones as necessary for larger programs. Today's implementation sports an extensive standard library and has moved away from the JVM in favor of native code generation. In this chapter, we cover the basic syntax and semantics of Factor for those unfamiliar with the language. This should be just enough to understand the later material in this thesis. More thorough documentation can be found via Factor's website, `http://factorcode.org`.

2.1 Stack-Based Languages

Like Reverse Polish Notation (RPN) calculators, Factor's evaluation model uses a global stack upon which operands are pushed before operators are called. This naturally facilitates postfix notation, in which operators are written after their operands. For example, instead of `1 + 2`, we write `1 2 +`. Figure 1 on the following page shows how `1 2 +` works conceptually:

- `1` is pushed onto the stack
- `2` is pushed onto the stack
- `+` is called, so two values are popped from the stack, added, and the result (`3`) is pushed back onto the stack

Other stack-based programming languages include Forth [American National Standards Institute and Computer and Business Equipment Manufacturers Association 1994], Cat [Diggins 2007], and PostScript [Adobe Systems Incorporated 1999].

The strength of this model is its simplicity. Evaluation essentially goes left to right: literals (like `1` and `2`) are pushed onto the stack, and operators (like `+`) perform some computation using values currently on the stack. This "flatness" makes parsing easier,

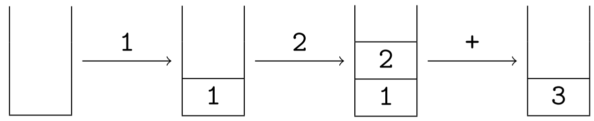

Figure 1: Visualizing stack-based calculation

since we don't need complex grammars with subtle ambiguities and precedence issues. Rather, we basically just scan left-to-right for tokens separated by whitespace. In the Forth tradition, functions are called *words* since they're made up of any contiguous non-whitespace characters. This also lends to the term *vocabulary* instead of "module" or "library". In Factor, the parser works as follows:

- If the current character is a double-quote (`"`), try to parse ahead for a string literal.
- Otherwise, scan ahead for a single token.
  - If the token is the name of a *parsing word*, that word is invoked with the parser's current state.
  - If the token is the name of an ordinary (i.e., non-parsing) word, that word is added to the parse tree.
  - Otherwise, try to parse the token as a numeric literal.

Parsing words serve as hooks into the parser, letting Factor users extend the syntax dynamically. For instance, instead of having special knowledge of comments built into the parser, the parsing word `!` scans forward for a newline and discards any characters read (adding nothing to the parse tree).

Similarly, there are parsing words for what might otherwise be hard-coded syntax for data structure literals. Many act as sided delimeters: the parsing word for the left-delimiter will parse ahead until it reaches the right-delimiter, using whatever was read in between to add objects to the data structure. For example, `{ 1 2 3 }` denotes an array of three numbers. Note the deliberate spaces in between the tokens, so that the delimeters are themselves distinct words. In `{␣1␣2␣3␣}` (with spaces as marked), the parsing word `{` parses objects until it reaches `}`, collecting the results into an array. The `{` word would not

```
V{ 1 2 3 }                        ! vector
B{ 1 2 3 }                        ! byte array
BV{ 1 2 3 }                       ! byte vector
HS{ 1 2 3 }                       ! hash set
H{ { key1 val1 } { key2 val2 } } ! hash table
```

Figure 2: Data structure literals in Factor

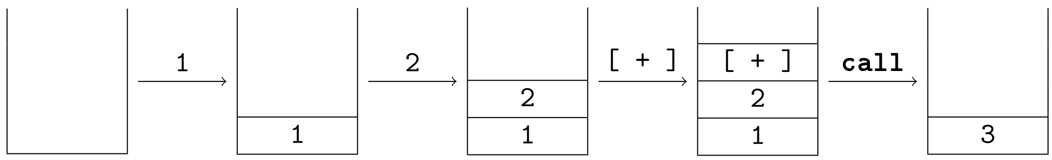

Figure 3: Quotations

be called if not for that space, whereas `{1␣2␣3}` parses as the word `{1`, the number `2`, and the word `3}`—not an array. Further, since the left-delimeter words parse recursively, such literals can be nested, contain comments, etc. Other literals include those in Figure 2.

A particularly important set of parsing words in Factor are the square brackets, `[` and `]`. Any code in between such brackets is collected up into a special sequence called a *quotation*. Essentially, it's a snippet of code whose execution is suppressed. The code inside a quotation can then be run with the **`call`** word. Quotations are like anonymous functions in other languages, but the stack model makes them conceptually simpler, since we don't have to worry about variable binding and the like. Consider a small example like

```
1 2 [ + ] call
```

You can think of **`call`** working by "erasing" the brackets around a quotation, so this example behaves just like `1 2 +`. Figure 3 shows its evaluation: instead of adding the numbers immediately, `+` is placed in a quotation, which is pushed to the stack. The quotation is then invoked by **`call`**, so `+` pops and adds the two numbers and pushes the result onto the stack. We'll show how quotations are used in Section 2.5 on page 16.

2.2 Stack Effects

Everything else about Factor follows from the stack-based structure outlined in Section 2.1. Consecutive words transform the stack in discrete steps, thereby shaping a result. In a way, words are functions from stacks to stacks—from "before" to "after"—and whitespace is effectively function composition. Even literals (numbers, strings, arrays, quotations, etc.) can be thought of as functions that take in a stack and return that stack with an extra element pushed onto it.

With this in mind, Factor requires that the number of elements on the stack (the *stack height*) is known at each point of the program in order to ensure consistency. To this end, every word is associated with a *stack effect* declaration using a notation implemented by parsing words. In general, a stack effect declaration has the form

```
( input1 input2 ... -- output1 output2 ... )
```

where the parsing word `(` scans forward for the special token `--` to separate the two sides of the declaration, and then for the `)` token to end the declaration. The names of the intermediate tokens don't technically matter—only how many of them there are. However, names should be meaningful for clarity's sake. The number of tokens on the left side of the declaration (before the `--`) indicates the minimum stack height expected before executing the word. Given exactly this number of inputs, the number of tokens on the right side is the stack height after executing the word.

For instance, the stack effect of the `+` word is `( x y -- z )`, as it pops two numbers off the stack and pushes one number (their sum) onto the stack. This could be written any number of ways, though. `( x x -- x )`, `( number1 number2 -- sum )`, and `( m n -- m+n )` are all equally valid. Further, while the stack effect `( junk x y -- junk z )` has the same relative height change, this declaration would be wrong, since it requires at least three inputs but `+` might legitimately be called on only two.

For the purposes of documentation, of course, the names in stack effects do matter. They correspond to elements of the stack from bottom-to-top. So, the rightmost value

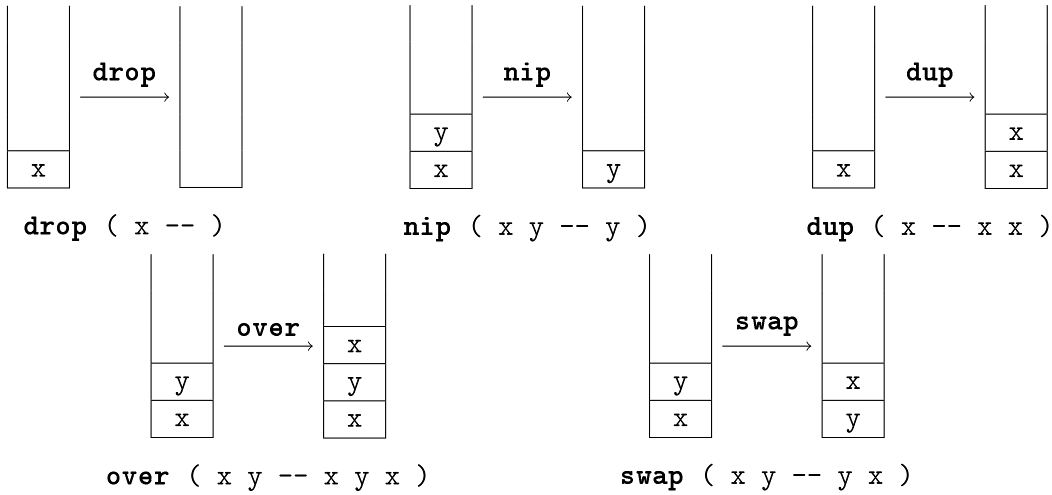

Figure 4: Stack shuffler words and their effects

on either side of the declaration names the top element of the stack. We can see this in Figure 4, which shows the effects of standard *stack shuffler* words. These words are used for basic data flow in Factor programs. For example, to discard the top element of the stack, we use the **`drop`** word, whose effect is simply `( x -- )`. To discard the element just below the top of the stack, we use **`nip`**, whose effect is `( x y -- y )`. This stack effect indicates that there are at least two elements on the stack before **`nip`** is called: the top element is `y`, and the next element is `x`. After calling the word, `x` is removed, leaving the original `y` still on top of the stack. Other shuffler words that remove data from the stack are **`2drop`** with the effect `( x y -- )`, **`3drop`** with the effect `( x y z -- )`, and **`2nip`** with the effect `( x y z -- z )`.

The next stack shufflers duplicate data. **`dup`** copies the top element of the stack, as indicated by its effect `( x -- x x )`. **`over`** has the effect `( x y -- x y x )`, which tells us that it expects at least two inputs: the top of the stack is `y`, and the next object is `x`. `x` is copied and pushed on top of the two original elements, sandwiching `y` between two `x`s. Other shuffler words that duplicate data on the stack are **`2dup`** with the effect `( x y -- x y x y )`, **`3dup`** with the effect `( x y z -- x y z x y z )`, **`2over`** with the effect `( x y z -- x y z x y )`, and **`pick`** with the effect `( x y z -- x y z x )`.

True to the name **`swap`**, the final shuffler in Figure 4 permutes the top two elements of the stack, reversing their order. The stack effect `( x y -- y x )` indicates as much. The left side denotes that two inputs are on the stack (the top is `y`, the next is `x`), and the right side shows the outputs are swapped (the top element is `x` and the next is `y`). Factor has other words that permute elements deeper into the stack. However, their use is discouraged because it's harder for the programmer to mentally keep track of more than a couple items on the stack. We'll see how more complex data flow patterns are handled in Section 2.5 on page 16.

2.3 Definitions

```
: hello-world ( -- )
    "Hello, world!" print ;
```

Figure 5: Hello World in Factor

Using the basic syntax of stack effect declarations described in Section 2.2, we can now understand how to define words. Most words are defined with the parsing word **`:`**, which scans for a name, a stack effect, and then any words up until the **`;`** token, which together become the body of the definition. Thus, the classic example in Figure 5 defines a word named `hello-world` which expects no inputs and pushes no outputs onto the stack. When called, this word will display the canonical greeting on standard output using the **`print`** word.

A slightly more interesting example is the `norm` word in Figure 6. This squares each of the top two numbers on the stack, adds them, then takes the square root of the sum. Figure 7 on the following page shows this in action. By defining a word to perform these steps, we can replace virtually any instance of **`dup`** `*` **`swap dup`** `* + sqrt` in a program simply with `norm`. This is a deceptively important point. Data flow is made explicit via

```
: norm ( x y -- norm )
    dup * swap dup * + sqrt ;
```

Figure 6: The Euclidean norm, $\sqrt{x^2 + y^2}$

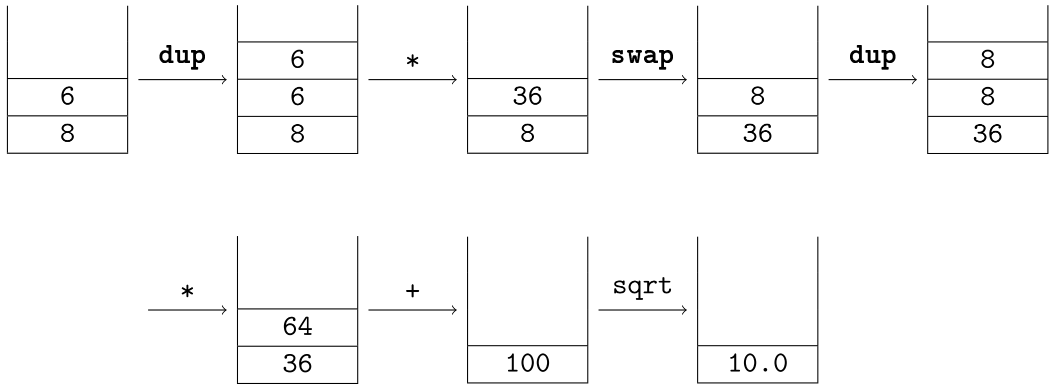

Figure 7: `norm` example

```
: ^2 ( n -- n^2 )
    dup * ;

: norm ( x y -- norm )
    ^2 swap ^2 + sqrt ;
```

Figure 8: `norm` refactored

stack manipulation rather than being hidden in variable assignments, so repetitive patterns become painfully evident. This makes identifying, extracting, and replacing redundant code easy. Often, you can just copy a repetitive sequence of words into its own definition verbatim. This emphasis on "factoring" your code is what gives Factor its name.

As a simple case in point, we see the subexpression `dup *` appears twice in the definition of `norm` in Figure 6 on the previous page. We can easily factor that out into a new word and substitute it for the old expressions, as in Figure 8. By contrast, programs in more traditional languages are laden with variables and syntactic noise that require more work to refactor: identifying free variables, pulling out the right functions without causing finicky syntax errors, calling a new function with the right variables, etc. Though Factor's stack-based paradigm is atypical, it is part of a design philosophy that aims to facilitate readable code focusing on short, reusable definitions.

Be that as it may, every once in awhile stack code gets too complicated to do away with more traditional notation. For these cases, Factor has a vocabulary called `locals`, which

```
:: norm ( x y -- norm )
    x x * :> x^2
    y y * :> y^2
    x^2 y^2 + sqrt ;
```

Figure 9: `norm` with local variables

introduces syntax for defining words that use named lexical variables. Defining words with `::` instead of `:` turns the stack effect declaration into a full-fledged parameter list. The inputs are assigned to their corresponding names in the effect, which are used throughout the body in lieu of stack manipulation. The outputs just mean the same thing as before (i.e., the right side of the effect doesn't declare any variables like the left does). We can also assign local variables in the body of the word by using the syntax `:> destination`, which assigns `destination` to the value on the top of the stack. Figure 9 shows a version of `norm` that uses these features, though they aren't really necessary here. Interestingly, `locals` is implemented entirely in high-level Factor code, using parsing words to convert the syntax into equivalent stack manipulations.

2.4 Object Orientation

You may have noticed that the examples in Section 2.3 did not use type declarations. While Factor is dynamically typed for the sake of simplicity, it does not do away with types altogether. In fact, Factor is object-oriented. However, its object system doesn't rely on classes possessing particular methods, as is common. Instead, it uses *generic words* with methods implemented for particular classes. To start, though, we must see how classes are defined.

Tuples

The central data type of Factor's object system is called a *tuple*, which is a class composed of named *slots*—like instance variables in other languages. Tuples are defined with the **`TUPLE:`** parsing word as shown in Figure 10 on the following page. A class name is

```
TUPLE: class
    slot-spec1 slot-spec2 slot-spec3 ... ;

TUPLE: subclass < superclass
    slot-spec1 slot-spec2 slot-spec3 ... ;
```

Figure 10: Basic tuple definition syntax

specified first; if it is followed by the `<` token and a superclass name, the tuple inherits the slots of the superclass. If no superclass is specified, the default is the **`tuple`** class. Any number of slot specifiers follow, and the definition is terminated by the **`;`** token.

Tuple definitions automatically generate several different words, most of which depend on how slots are specified. There are various ways to specify slots, but we use only two basic forms in later code examples. We can see both in the first tuple of Figure 11 on the next page, which defines an object to represent regular expressions. The first three slots have the form `{ name read-only }`, which specifies a slot named `name` that can't be modified once initialized, akin to a **`final`** variable in Java. The next two specifiers are simpler, being just the names of the slots. Such slots can be modified freely. The following words are automatically defined for the first tuple:

- The `regexp` *class word* acts like a literal representing the class. This gets used for instantiation and method definitions, which we'll see later.

- The `regexp?` *class predicate* is a word with the stack effect `( object -- ? )`. That is, it returns a boolean (either **`t`** or **`f`**, conventionally written in stack effects as a single question mark) indicating whether the top of the stack is an instance of the `regexp` class. This is like a class-specific variant of Java's **`instanceof`**.

- Each slot has an associated *reader* word with the stack effect `( object -- value )`. These are analogous to "getter" methods in other languages. Each one is named after the slot whose value is extracted, so this example defines `raw>>`, `parse-tree>>`, `options>>`, `dfa>>`, and `next-match>>`.

```
TUPLE: regexp
    { raw read-only }
    { parse-tree read-only }
    { options read-only }
    dfa next-match ;

TUPLE: reverse-regexp < regexp ;
```

Figure 11: Sample tuple definitions from Factor's `regexp` vocabulary

- Similarly, any slot that is not marked `read-only` has a corresponding *writer* word with the stack effect `( value object -- )`. These destructively write the value into the eponymous slot of the object. Here, only two are defined, named `dfa<<` and `next-match<<`.

- Extra *setter* words are defined in terms of writers. These will have the stack effect `( object value -- object' )`, leaving the modified instance on top of the stack. The first tuple in Figure 11 defines `>>dfa` and `>>next-match`, which are equivalent to **`over`** `dfa<<` and **`over`** `next-match<<`, respectively. The shuffler duplicates `object` and pushes it to the top of the stack. More accurately, it duplicates a reference to `object`, as Factor's data stack is actually a stack of pointers. That way, changes to the new top of the stack with `dfa<<` or `next-match<<` will be reflected in the original `object`, which is left over at the end.

- *Changer* words are also created with the stack effect `( object quot -- object' )`. Here, `change-dfa` and `change-next-match` are defined. The quotation is called on the slot's current value in `object`. The result of calling the quotation is then stored in the slot. For instance, incrementing an integer slot named `foo` could be done with `[ 1 + ] change-foo`.

The second tuple in Figure 11 also defines a class word and predicate. Since it inherits from `regexp`, `reverse-regexp` gets the same five slots. If we had any other slot specifiers in the definition, it would have those in addition to the slots of its parent class. The reader, writer, setter, and changer methods will work on instances of `reverse-regexp`, since

```
TUPLE: color ;

: <color> ( -- color )
    color new ;

TUPLE: rgb < color red green blue ;

: <rgb> ( r g b -- rgb )
    rgb boa ;
```

Figure 12: Tuple constructors

inheritance establishes an “is-a” relationship from subclass to superclass—any instance of `reverse-regexp` is also an instance of `regexp`, though the reverse is not necessarily true. That is, `regexp?` will return **`t`** on instances of `reverse-regexp`, but `reverse-regexp?` will only return **`t`** on instances of `regexp` that are also `reverse-regexp`s. By viewing a class as the set of all objects that respond positively to the class predicate, we may partially order classes with the subset relationship. This fact will be important later.

To construct an instance of a tuple, we can use either **`new`** or **`boa`**. **`new`** will not initialize any of the slots to a particular input value—all slots will default to Factor's canonical false value, **`f`**. For example, **`new`** is used in Figure 12 to define `<color>` (by convention, the constructor for `foo` is named `<foo>`). First, we push the class `color`, then just call **`new`**, leaving a new instance on the stack. Since this particular tuple has no slots, using **`new`** makes sense. We might also use it to initialize a class, then use setter words to only assign a particular subset of slots' values (as long as the slots aren't `read-only`).

However, we often want to initialize a tuple with values for each of its slots. For this, we have **`boa`**, which works similarly to **`new`**. This is used in the definition of `<rgb>` in Figure 12. The difference here is the additional inputs on the stack—one for each slot, in the order they're declared. That is, we're constructing the tuple **b**y **o**rder of **a**rguments, giving us the fun pun “**`boa`** constructor”. So, `1 2 3 <rgb>` will construct an `rgb` instance with the `red` slot set to `1`, the `green` slot set to `2`, and the `blue` slot set to `3`.

Generics and Methods

Unlike more common object systems, we do not define individual methods that "belong" to particular tuples. In Factor, for a given generic word you define a method that specializes on a class. When the generic word is called on an object, it selects the method most specific to the object's class. This is determined by the aforementioned partial ordering of classes by their inheritance relationships.

Generic words are declared with the syntax

```
GENERIC: word-name ( stack -- effect )
```

Words defined this way will then dispatch on the class of the top element of the stack (necessarily the rightmost input in the stack effect). To define a new method with which to control this dispatch, we use the syntax

```
M: class word-name definition... ;
```

Factor's `sets` vocabulary gives us an accessible example of a generic word. `set` is a *mixin* class, defined by the **`MIXIN:`** parsing word. That is, the `set` class is a union of other classes, and users may extend the members of this union with the **`INSTANCE:`** word. We can this in Figure 13 on the following page, which shows the standard members of the `set` mixin. Note that the **`USING:`** form specifies vocabularies being used (like Java's **`import`**) and **`IN:`** specifies the vocabulary in which the definitions appear (like Java's **`package`**). We can see here that instances of the `sequence`, `hash-set`, and `bit-set` classes are all instances of `set`, so will respond **`t`** to the predicate `set?`. Similarly, `sequence` is a mixin class with many more members, including `array`, `vector`, and `string`.

Figure 14 on the next page shows the `cardinality` generic from Factor's `sets` vocabulary, along with its methods. This generic word takes a `set` instance from the top of the stack and pushes the number of elements it contains. For instance, if the top element is a `bit-set`, we extract its `table` slot and invoke another word, `bit-count`, on that. But if the top element is **`f`** (the canonical false/empty value), we know the cardinality is 0.

```
USING: bit-sets hash-sets sequences ;
IN: sets

MIXIN: set
INSTANCE: sequence set
INSTANCE: hash-set set
INSTANCE: bit-set set
```

Figure 13: Set instances

```
IN: sets
GENERIC: cardinality ( set -- n )

USING: accessors bit-sets math.bitwise sets ;
M: bit-set cardinality table>> bit-count ;

USING: kernel sets ;
M: f cardinality drop 0 ;

USING: accessors assocs hash-sets sets ;
M: hash-set cardinality table>> assoc-size ;

USING: sequences sets ;
M: sequence cardinality length ;

USING: sequences sets ;
M: set cardinality members length ;
```

Figure 14: Set cardinality using Factor's object system

For any `sequence`, we may offshore the work to a different generic, **`length`**, defined in the `sequences` vocabulary. The final method gives a default behavior for any other `set` instance, which simply uses `members` to obtain an equivalent `sequence` of set members, then calls **`length`**.

We can see how the class ordering is used when `cardinality` selects the proper method for the object being dispatched upon. For instance, while no explicit method for `array` is defined, any instance of `array` is also an instance of `sequence`. In turn, every instance of `sequence` is also an instance of `set`. We have methods that dispatch on both `set` and `sequence`, but the latter is more specific, so that is the method invoked on an `array`. If we define our own class, `foo`, and declare it as an instance of `set` but not as an instance of `sequence`, then the `set` method of `cardinality` will be invoked. Sometimes resolving the precedence gets more complicated, but these edge-cases are beyond the scope of our discussion.

2.5 Combinators

Quotations, introduced in Section 2.1, form the basis of both control flow and data flow in Factor. Not only are they the equivalent of anonymous functions, but the stack model also makes them syntactically lightweight enough to serve as blocks akin to the code between curly braces in C or Java. Higher-order words that make use of quotations on the stack are called *combinators*. It's simple to express familiar conditional logic and loops using combinators, as we'll show first. In the presence of explicit data flow via stack operations, even more patterns arise that can be abstracted away. The last half of this section explores how we can use combinators to express otherwise convoluted stack-shuffling logic more succinctly.

Control Flow

The most primitive form of control flow in typical programming languages is, of course, the **`if`** statement, and the same holds true for Factor. The only difference is that Factor's **`if`** isn't syntactically significant—it's just another word, albeit implemented as a primitive. For the moment, it will do to think of **`if`** as having the stack effect `( ? true false -- )`. The third element from the top of the stack is a boolean condition, and it's followed by two quotations. The first quotation (`true`) is called if the condition is true, and the second quotation (`false`) is called if the condition is false. Specifically, **`f`** is a special object in Factor for falsity. It is a singleton object—the sole instance of the **`f`** class—and is the only false value in the entire language. Any other object is necessarily boolean true. For a canonical boolean, there is the **`t`** object, but its truth value exists only because it is not **`f`**.

```
5 even? [ "even" print ] [ "odd" print ] if

{ } empty? [ "empty" print ] [ "full" print ] if

100 [ "isn't f" print ] [ "is f" print ] if
```

Figure 15: Conditional evaluation in Factor

Basic **`if`** use is shown in Figure 15. The first example will print "odd", the second "empty", and the third "isn't f". All of them leave nothing on the stack.

However, the simplified stack effect for **`if`** is quite restrictive. Because the effect `( ? true false -- )` has no extra inputs and no outputs at all, it intuitively means that the `true` and `false` quotations both have the effect `( -- )`. We'd like to loosen this restriction, but per Section 2.2, Factor must know the stack height after the **`if`** call. We could give **`if`** the effect `( x ? true false -- y )` so that the two quotations could each have the stack effect `( x -- y )`. This would work for the `example1` word in Figure 16 on the next page, yet it's just as restrictive. For instance, the `example2` word would need **`if`** to have the effect `( x y ? true false -- z )`, since each branch has the effect `( x y -- z )`. Furthermore, the quotations might even have different effects, but still leave the overall stack height balanced. Only one item is left on the stack after a call to `example3` regardless, even though the two quotations have different stack effects: **`+`** has the effect `( x y -- z )`, while **`drop`** has the effect `( x -- )`.

In reality, there are infinitely many correct stack effects for **`if`**. Factor has a special notation for such *row-polymorphic* stack effects. If a token in a stack effect begins with two dots, like `..a` or `..b`, it is a *row variable*. If either side of a stack effect begins with a row variable, it represents any number of inputs or outputs. Thus, we could give **`if`** the stack effect

```
( ..a ? true false -- ..b )
```

to indicate that there may be any number of inputs below the condition on the stack, and that any number of outputs will be present after the call to **`if`**. Note that these numbers

```
: example1 ( x -- 0/x-1 )
    dup even? [ drop 0 ] [ 1 - ] if ;

: example2 ( x y -- x+y/x-y )
    2dup mod 0 = [ + ] [ - ] if ;

: example3 ( x y -- x+y/x )
    dup odd? [ + ] [ drop ] if ;
```

Figure 16: **`if`**'s stack effect varies

aren't necessarily equal, which is why we use distinct row variables (`..a` and `..b`) in this case. However, this still isn't quite enough to capture the stack height requirements. It doesn't communicate that `true` and `false` must affect the stack in the same ways, which has remained tacit to this point. For this, the notation `quot: ( stack -- effect )` gives quotations a nested stack effect. Using the same names for row variables in both the "inner" and "outer" stack effects will refer to the same number of inputs or outputs. Thus, our final (correct) stack effect for **`if`** is

```
( ..a ? true: ( ..a -- ..b ) false: ( ..a -- ..b ) -- ..b )
```

This tells us that the `true` quotation and the `false` quotation will each leave the stack at the same height as **`if`** does overall, and that neither expects any extra inputs.

Though **`if`** is necessarily a language primitive, other control flow constructs are defined in Factor itself. It's simple to write combinators for iteration and looping as recursive words that invoke quotations. Figure 17 on the following page showcases some common looping patterns. The most basic yet versatile word is **`each`**. Its stack effect is

```
( ... seq quot: ( ... x -- ... ) -- ... )
```

Each element `x` of the sequence `seq` will be passed to `quot`, which may use any of the underlying stack elements. Here, unlike **`if`**, we enforce that `quot`'s output stack height is exactly one less than the input. Otherwise, depending on the number of elements in `seq`, we might dig arbitrarily deep into the stack or flood it with a varying number of

```
{ "Lorem" "ipsum" "dolor" } [ print ] each

0 { 1 2 3 } [ + ] each

10 iota [ number>string print ] each

3 [ "Ho!" print ] times

[ t ] [ "Infinite loop!" print ] while

[ f ] [ "Executed once!" print ] do while
```

Figure 17: Loops in Factor

values. The first use of **each** in Figure 17 is balanced, as the quotation has the effect `( str -- )` and no additional items were on the stack to begin with (i.e., `...` stands in for 0 elements). Essentially, it's equivalent to *"Lorem"* **print** *"ipsum"* **print** *"dolor"* **print**. On the other hand, the quotation in the second example has the stack effect `( total n -- total+n )`. This is still balanced, since there is one additional item below the sequence on the stack (namely `0`), and one element is left by the end (the sum of the sequence elements). So, this example is the same as `0 1 + 2 + 3 +`.

Any instance of the extensive `sequence` mixin will work with **each**, making it very flexible. The third example in Figure 17 shows **iota**, which is used here to create a *virtual* sequence of integers from 0 to 9 (inclusive). No actual sequence is allocated, merely an object that behaves like a sequence. In Factor, it's common practice to use **iota** and **each** in favor of repetitive C-like **for** loops.

Of course, we sometimes don't need the induction variable in loops. That is, we just want to execute a body of code a certain number of times. For these cases, there's the **times** combinator, with the stack effect

```
( ... n quot: ( ... -- ... ) -- ... )
```

This is similar to **each**, except that `n` is a number (so we needn't use **iota**) and the quotation doesn't expect an extra argument (i.e., a sequence element). Therefore, the

```
{ 1 2 3 } [ 1 + ] map

{ 1 2 3 4 5 } [ even? ] filter

{ 1 2 3 } 0 [ + ] reduce
```

Figure 18: Higher-order functions in Factor

example in Figure 17 on the previous page is equivalent to *"Ho!"* **print** *"Ho!"* **print** *"Ho!"* **print**.

Naturally, Factor also has the **while** combinator, whose stack effect is

`( ..a pred: ( ..a -- ..b ? ) body: ( ..b -- ..a ) -- ..b )`

The row variables are a bit messy, but it works as you'd expect: the `pred` quotation is invoked on each iteration to determine whether `body` should be called. The **do** word is a handy modifier for **while** that simply executes the body of the loop once before leaving **while** to test the precondition as per usual. Thus, the last example in Figure 17 on the preceding page executes the body once, despite the condition being immediately false.

In the preceding combinators, quotations were used like blocks of code. But really, they're the same as anonymous functions from other languages. As such, Factor borrows classic tools from functional languages, like **map** and **filter**, as shown in Figure 18. **map** is like **each**, except that the quotation should produce a single output. Each such output is collected up into a new sequence of the same class as the input sequence. Here, the example produces `{ 2 3 4 }`. **filter** selects only those elements from the sequence for which the quotation returns a true value. Thus, the **filter** in Figure 18 outputs `{ 2 4 }`. Even **reduce** is in Factor, also known as a *left fold*. An initial element is iteratively updated by pushing a value from the sequence and invoking the quotation. In fact, **reduce** is defined as **swapd each**, where **swapd** is a shuffler word with the stack effect `( x y z -- y x z )`. Thus, the example in Figure 18 is the same as `0 { 1 2 3 } [ + ]` **each**, as in Figure 17 on the preceding page.

These are just some of the control flow combinators defined in Factor. Several variants exist that meld stack shuffling with control flow, or can be used to shorten common patterns such as empty false branches. An entire list is beyond the scope of our discussion, but the ones we've studied should give a solid view of what standard conditional execution, iteration, and looping looks like in a stack-based language.

Data Flow

While avoiding variables makes it easier to refactor code, keeping mental track of the stack can be taxing. If we need to manipulate more than the top few elements of the stack, code gets harder to read and write. Since the flow of data is made explicit via stack shufflers, we actually wind up with redundant patterns of data flow that we otherwise couldn't identify. In Factor, there are several combinators that clean up common stack-shuffling logic, making code easier to understand.

The first combinators we'll look at are **`dip`** and **`keep`**. These are used to preserve certain stack elements, do a computation, then restore said elements. For an uncompelling but illustrative example, suppose we have two values on the stack, but we want to increment the second element from the top. `without-dip1` in Figure 19 on the next page shows one strategy, where we shuffle the top element away with **`swap`**, perform the computation, then **`swap`** the top back to its original place. A cleaner way is to call **`dip`** on a quotation, which will execute that quotation just under the top of the stack, as in `with-dip1`. While the stack shuffling in `without-dip1` isn't terribly complicated, it doesn't convey our meaning very well. Shuffling the top element out of the way becomes increasingly difficult with more complex computations. In `without-dip2`, we want to call `-` on the two elements below the top. For lack of a more robust stack shuffler, we use **`swap`** followed by **`swapd`** to rearrange the stack so we can call `-`, then **`swap`** it back to the desired order. This is even less clear, plus **`swapd`** is actually a deprecated word in Factor, since its use starts making code harder to reason about. Alternatively, we could dream up a more complex stack shuffler with exactly the stack effect we wanted in this situation. But this solution doesn't scale: what if we had

```
: without-dip1 ( x y -- x+1 y )
    swap 1 + swap ;

: with-dip1 ( x y -- x+1 y )
    [ 1 + ] dip ;

: without-dip2 ( x y z -- x-y z )
    swap swapd - swap ;

: with-dip2 ( x y z -- x-y z )
    [ - ] dip ;

: without-keep1 ( x -- x+1 x )
    dup 1 + swap ;

: with-keep1 ( x -- x+1 x )
    [ 1 + ] keep ;

: without-keep2 ( x y -- x-y y )
    swap over - swap ;

: with-keep2 ( x y -- x-y y )
    [ - ] keep ;
```

Figure 19: Preserving combinators

to calculate something that required more inputs or produced more outputs? Clearly, **`dip`** provides a cleaner alternative in `with-dip2`.

`keep` provides a way to hold onto the top element of the stack, but still use it to perform a computation. In general, `[ ... ]` **`keep`** is equivalent to **`dup`** `[ ... ]` **`dip`**. Thus, the current top of the stack remains on top after the use of **`keep`**, but the quotation is still invoked with that value. In `with-keep1` in Figure 19, we want to increment the top, but stash the result below. Again, this logic isn't terribly complicated, though `with-keep1` does away with the shuffling. `without-keep2` shows a messier example where a simple **`dup`** will not save us, as we're using more than just the top element in the call to `-`. Rather, three of the four words in the definition are dedicated to rearranging the stack in just the right way, obscuring the call to `-` that we really want to focus on. On the other hand, `with-keep2` places the subtraction word front-and-center in its own quotation, while **`keep`**

does the work of retaining the top of the stack.

The next set of combinators apply multiple quotations to a single value. The most general form of these so-called *cleave* combinators is the word **`cleave`**, which takes an array of quotations as input, and calls each one in turn on the top element of the stack. So,

```
x { [ a ] [ b ] [ c ] } cleave
```

takes the top element, `x`, and applies each quotation to it: `a` is applied to `x`, then `b` to `x`, then `c` to `x`. Of course, for only a couple of quotations, wrapping them in an array literal becomes cumbersome. The word **`bi`** exists for the two-quotation case, and **`tri`** for three quotations. Cleave combinators are often used to extract multiple slots from a tuple. Figure 20 on the next page shows such a case in the `with-bi` word, which improves upon using just **`keep`** in the `without-bi` word. In general, a series of **`keep`**s like

```
[ a ] keep [ b ] keep c
```

is the same as

```
{ [ a ] [ b ] [ c ] } cleave
```

which is more readable. We can see this in action in the difference between `without-tri` and `with-tri` in Figure 20 on the following page. In cases where we need to apply multiple quotations to a set of values instead of just a single one, there are also the variants **`2cleave`** and **`3cleave`** (and the corresponding **`2bi`**, **`2tri`**, **`3bi`**, and **`3tri`**), which apply the quotations to the top two and three elements of the stack, respectively.

To apply multiple quotations to multiple values, Factor has *spread* combinators. Whereas cleave combinators abstract away repeated instances of **`keep`**, spread combinators replace nested calls to **`dip`**. The archetypical combinator, **`spread`**, takes an array of quotations, like **`cleave`**. However, instead of applying each one to the top element of the stack, each one corresponds to a separate element of the stack. Thus,

```
x y { [ a ] [ b ] } spread
```

```
TUPLE: coord x y ;

: without-bi ( coord -- norm )
    [ x>> sq ] keep y>> sq + sqrt ;

: with-bi ( coord -- norm )
    [ x>> sq ] [ y>> sq ] bi + sqrt ;

: without-tri ( x -- x+1 x+2 x+3 )
    [ 1 + ] keep [ 2 + ] keep 3 + ;

: with-tri ( x -- x+1 x+2 x+3 )
    [ 1 + ] [ 2 + ] [ 3 + ] tri ;
```

Figure 20: Cleave combinators

invokes `a` on `x` and `b` on `y`. Much like **`cleave`**, there are shorthand words for the two- and three-quotation cases. These are suffixed with asterisks to indicate the spread variants, so we have **`bi*`** and **`tri*`**. In Figure 21 on the next page, the `without-bi*` word shows the simple **`dip`** pattern that **`bi*`** encapsulates. Here, we're converting the string `str1` (the second element from the top) into uppercase and `str2` (the top element) to lowercase. In `with-bi*`, the `>upper` ("to uppercase") and `>lower` ("to lowercase") words are seen first, uninterrupted by an extra word. More compelling is the way that **`tri*`** replaces the **`dip`**s that can be seen in `without-tri*`. In comparison, `with-tri*` is less nested and easier to comprehend at first glance. While there are **`2bi*`** and **`2tri*`** variants that spread quotations to two values apiece on the stack, they are uncommon in practice.

Finally, *apply* combinators invoke a single quotation on multiple stack entries in turn. While there is a generalized word, it's more common to use the corresponding shorthands. Here, they are suffixed with at-signs, so **`bi@`** applies a quotation to each of the top two stack values, and **`tri@`** to each of the top three. This way, rather than duplicate code for each time we want to call a word, we need only specify it once. This is demonstrated clearly in Figure 22 on the following page. In `without-bi@`, we see that the quotation `[ sq ]` (for squaring numbers) appears twice for the call to **`bi*`**. In general, we can replace spread combinators whose quotations are all the same with a single quotation and an

```
: without-bi* ( str1 str2 -- str1' str2' )
    [ >upper ] dip >lower ;

: with-bi* ( str1 str2 -- str1' str2' )
    [ >upper ] [ >lower ] bi* ;

: without-tri* ( x y z -- x+1 y+2 z+3 )
    [ [ 1 + ] dip 2 + ] dip 3 + ;

: with-tri* ( x y z -- x+1 y+2 z+3 )
    [ 1 + ] [ 2 + ] [ 3 + ] tri* ;
```

Figure 21: Spread combinators

```
: without-bi@ ( x y -- norm )
    [ sq ] [ sq ] bi* + sqrt ;

: with-bi@ ( x y -- norm )
    [ sq ] bi@ + sqrt ;

: without-tri@ ( x y z -- x+1 y+1 z+1 )
    [ 1 + ] [ 1 + ] [ 1 + ] tri* ;

: with-tri* ( x y z -- x+1 y+1 z+1 )
    [ 1 + ] tri@ ;
```

Figure 22: Apply combinators

apply combinator. Thus, `with-bi@` cuts down on the duplicated `[ sq ]` in `without-bi@`. Similarly, we can replace the three repeated quotations passed to **`tri*`** in `without-tri@` with a single instance passed to **`tri@`** in `with-tri@`. Like other data flow combinators, we have the numbered variants. **`2bi@`** has the stack effect `( w x y z quot -- )`, where `quot` expects two inputs, and is thus applied to `w` and `x` first, then to `y` and `z`. Similarly, **`2tri@`** applies the quotation to the top six objects of the stack in groups of two. Like their spread counterparts, they are not used very much.

This concludes our overview of Factor. Various other features are hidden away in different vocabularies, most of which are understandable based on the basics we've covered. For the purposes of this thesis, it's not important to know every little detail about Factor. Future code snippets will always be accompanied by further explanation, though they should generally be readable given this short tutorial.

3 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 (though there has been compelling work on this front [e.g., Biggar 2009]). 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 chapter, 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.

3.1 Organization

At the lowest level, Factor is written atop a C++ virtual machine (VM) that is responsible for basic runtime services. This is where the non-optimizing base compiler is implemented. It's the base compiler's job to compile the simplest primitives: operations that push literals onto the data stack, **`call`**, **`if`**, **`dip`**, words that access tuple slots as laid out in memory, stack shufflers, math operators, functions to allocate/deallocate call stack frames, etc. The aim of the base compiler is to generate native machine code as fast as possible. To this end, these primitives correspond to their own stubs of assembly code. Different stubs are generated by Factor depending on the instruction set supported by the particular machine in use. Thus, the base compiler need only make a single pass over the source code, emitting these assembly instructions as it goes.

This compiled code is saved in an *image file*, which contains a complete snapshot of the current state of the Factor instance, similar to many Smalltalk and Lisp systems [Pestov, Ehrenberg, and Groff 2010]. As code is parsed and compiled, the image is updated, serving as a cache for compiled code. This modified image can be saved so that future Factor instances needn't recompile vocabularies that have already been loaded.

The VM also handles method dispatch and memory management. Method dispatch incorporates a *polymorphic inline cache* to speed up generic words. Each generic word's call site (i.e., every point in the code at which we invoke a generic) is associated with a state:

- In the *cold* state, the call site's instruction performs relatively expensive computations to find the right method for the class being dispatched upon, which we'd like to avoid. To speed up future calls at that site, a polymorphic inline cache stub is generated, thus transitioning it to the next state.
- In the *inline cache* state, a stub has been generated that caches the locations of methods for classes that have already been seen. This way, if a generic word at a particular call site is invoked often upon only a small number of classes (as is often in the case in loops, for example), we don't need to waste as much time doing method lookup. By default, if more than three different classes are dispatched upon, we transition to the next state.
- In the *megamorphic* state, the call instruction points to a larger cache that is allocated for the specific generic word (i.e., it is shared by all call sites). While not as efficient as an inline cache, this can still improve the performance of method dispatch.

To manage memory, the Factor VM uses a generational garbage collector (GC), which carves out sections of space on the heap for objects of different ages. Garbage in the oldest generation is collected with a mark-sweep-compact algorithm, while younger generations rely on a copying collector [Wilson 1992]. This way, the GC is specialized for large numbers of short-lived objects that will stay in the younger generations without being promoted to the older generation. In the oldest space, even compiled code can be compacted. This is to avoid heap fragmentation in applications that must call the compiler at runtime, such as Factor's interactive development environment [Pestov, Ehrenberg, and Groff 2010].

Values are referenced by tagged pointers, which use the three least significant bits of the pointer's address to store type information. This is possible because Factor aligns objects on an eight-byte boundary, so the three least significant bits of an address are always 0. These bits give us eight unique tags, but Factor has more than eight data types. So, to indicate that the type information is stored elsewhere, two tags are reserved. One is for VM types without their own tag, and the other is for tuples, each of which defines its own type. Sufficiently small integers (e.g., 29-bit integers on a 32-bit machine, since the other 3 bits are used for the type tag) are stored directly in the pointer, so they needn't be heap-allocated (i.e., "boxed"). Larger integers and floating point numbers are normally boxed, but the optimizing compiler may find ways to store floats in registers.

The VM is meant to be minimal, as Factor is mostly *self-hosting*. That is, the real workhorses of the language are written in Factor itself, including the standard libraries, parser, object system, and the optimizing compiler. It's possible for the compiler to be written in Factor because of the *bootstrapping* process that creates a new image from scratch. First, a minimal *boot image* is created from an existing *host* Factor instance. When the VM runs the boot image, it initiates the bootstrapping process. Using the host's parser, the base compiler will compile the core vocabularies necessary to load the optimizing compiler. Once the optimizing compiler can itself be compiled, it is used to recompile (and thus optimize) all of the words defined so far. With the basic vocabularies recompiled, any additional vocabularies can be loaded using the optimized compiler and saved into a new, working image.

Thus, while the Factor VM is important, it is a small part of the code base. Since the bootstrapping process allows the optimizing compiler (hereafter just "the compiler") to be written in the same high-level language it's compiling, we can avoid the fiddly low-level details of the C++ backend. This is more conducive to writing advanced compiler optimizations, which are often complicated enough without having a concise, dynamically-typed, garbage-collected language like Factor to help us.

3.2 High-level Optimizations

To manipulate source code abstractly, we must have at least one intermediate representation (IR)—a data structure representing the instructions. It's common to convert between several IRs during compilation, as each form offers different properties that facilitate particular analyses. The Factor compiler optimizes code in passes across two different IRs: first at a high-level using the `compiler.tree` vocabulary, then at a low-level with the `compiler.cfg` vocabulary. In this section, we look at the former IR and the optimizations performed on it.

Representation

The high-level IR arranges code into a vector of `node` objects, which may themselves have children consisting of vectors of nodes—a tree structure that lends to the name `compiler.tree`. This ordered sequence of nodes represents control flow in a way that's effectively simple, annotated stack code. Figure 23 on the next page shows the definitions of the tuples that represent the "instruction set" of this stack code. Each object inherits (directly or indirectly) from the `node` class, which itself inherits from **`identity-tuple`**. This is a tuple whose **`equal?`** method is defined to always return **`f`** so that no two instances are equivalent unless they are the same object in memory.

Notice that most nodes define some sort of `in-d` and `out-d` slots, which mark each of them with the input and output data stacks. This represents the flow of data through the program. Here, stack values are denoted simply by integers, giving each value a unique identifier. An `#introduce` instance is inserted wherever the next node requires stack values that have not yet been named. Thus, while `#introduce` has no `in-d`, its `out-d` introduces the necessary stack values. Similarly, `#return` is inserted at the end of the sequence to indicate the final state of the data stack with its `in-d` slot.

The most basic operations of a stack language are, of course, pushing literals and calling functions. The `#push` node thus has a `literal` slot and an `out-d` slot, giving a name to the single element it pushes to the data stack. `#call` is used for normal word invocations.

```
TUPLE: node < identity-tuple ;

TUPLE: #introduce < node out-d ;
TUPLE: #return < node in-d info ;

TUPLE: #push < node literal out-d ;
TUPLE: #call < node word in-d out-d body method class info ;

TUPLE: #renaming < node ;
TUPLE: #copy < #renaming in-d out-d ;
TUPLE: #shuffle < #renaming mapping in-d out-d in-r out-r ;

TUPLE: #declare < node declaration ;

TUPLE: #terminate < node in-d in-r ;

TUPLE: #branch < node in-d children live-branches ;
TUPLE: #if < #branch ;
TUPLE: #dispatch < #branch ;

TUPLE: #phi < node phi-in-d phi-info-d out-d terminated ;

TUPLE: #recursive < node in-d word label loop? child ;
TUPLE: #enter-recursive < node in-d out-d label info ;
TUPLE: #call-recursive < node label in-d out-d info ;
TUPLE: #return-recursive < #renaming in-d out-d label info ;

TUPLE: #alien-node < node params ;
TUPLE: #alien-invoke < #alien-node in-d out-d ;
TUPLE: #alien-indirect < #alien-node in-d out-d ;
TUPLE: #alien-assembly < #alien-node in-d out-d ;
TUPLE: #alien-callback < node params child ;
```

Figure 23: High-level IR nodes

```
V{
    T{ #push { literal 1 } { out-d { 6256273 } } }
    T{ #introduce { out-d { 6256274 } } }
    T{ #call
        { word + }
        { in-d V{ 6256274 6256273 } }
        { out-d { 6256275 } }
    }
    T{ #return { in-d V{ 6256275 } } }
}
```

Figure 24: `[ 1 + ] build-tree`

The `in-d` and `out-d` slots effectively serve as the stack effect declaration. In later analyses, data about the word's definition may be stored across the `body`, `method`, `class`, and `info` slots.

The word `build-tree` takes a Factor quotation and constructs the equivalent high-level IR form. In Figure 24, we see the output of the simple example `[ 1 + ] build-tree`. Note that `T{ class { slot1 value1 } { slot2 value2 } ... }` is the syntax for tuple literals. The first node is a `#push` for the `1` literal, which is named "`6256273`". Since `+` needs two input values, an `#introduce` pushes a new "phantom" value. `+` gets turned into a `#call` instance. The sum is pushed to the data stack, so the `out-d` slot is a singleton that names this value. Finally, `#return` indicates the end of the routine, its `in-d` indicating the value left on the stack (the sum pushed by `#call`).

The next tuples in Figure 23 on the previous page reassign existing values on the stack to fresh identifiers. The `#renaming` superclass has the two subclasses `#copy` and `#shuffle`. The former represents the bijection from elements of `in-d` to elements of `out-d` in the same position; corresponding values are copies of each other. Stack shufflers are translated to more general `#shuffle` nodes with `mapping` slots that dictate how the new values in `out-d` correspond to the input values in `in-d`. For instance, Figure 25 on the following page shows how **`swap`** takes in the values `6256132` and `6256133` and outputs `6256134` and `6256135`, where the first output is mapped to the second input and the second output is mapped to the first input. The `in-r` and `out-r` slots of `#shuffle` correspond to the *retain* stack,

```
V{
    T{ #introduce { out-d { 6256132 6256133 } } }
    T{ #shuffle
        { mapping { { 6256134 6256133 } { 6256135 6256132 } } }
        { in-d V{ 6256132 6256133 } }
        { out-d V{ 6256134 6256135 } }
    }
    T{ #return { in-d V{ 6256134 6256135 } } }
}
```

Figure 25: `[ swap ] build-tree`

```
V{
    T{ #introduce { out-d { 6256069 } } }
    T{ #declare { declaration { { 6256069 fixnum } } } }
    T{ #return { in-d V{ 6256069 } } }
}
```

Figure 26: `[ { fixnum } declare ] build-tree`

which is an implementation detail beyond the scope of this discussion.

`#declare` is a miscellaneous node used for the **`declare`** primitive. It simply annotates type information to stack values, as in Figure 26. `#terminate` is another one-off node, but a much more interesting one. While Factor normally requires a balanced stack, sometimes we purposefully want to throw an error. `#terminate` is introduced where the program halts prematurely. When checking the stack height, it gets to be treated specially so that *terminated* stack effects unify with any other effect. That way, branches will still be balanced even if one of them unconditionally throws an error. Figure 27 on the following page shows `#terminate` being introduced by the **`throw`** word.

Next, Figure 23 on page 31 defines nodes for branching based off the superclass `#branch`. The `children` slot contains vectors of nodes representing different branches. `live-branches` is filled in during later analyses to indicate which branches might be executed so "dead" ones that are never taken may be removed. Mostly, we'll see `#if`, which will have two elements in its `children` slot representing the true and false branches. On the other hand, `#dispatch` has an arbitrary number of children. It corresponds to the

```
V{
    T{ #push { literal "Error!" } { out-d { 6256051 } } }
    T{ #call
        { word throw }
        { in-d V{ 6256051 } }
        { out-d { } }
    }
    T{ #terminate { in-d V{ } } { in-r V{ } } }
    T{ #return { in-d V{ } } }
}
```

Figure 27: `[ "Error!" throw ] build-tree`

dispatch primitive, which is an implementation detail of the generic word system used to speed up method dispatch.

You may have noted the emphasis on introducing new values, instead of reassigning old ones. Even `#shuffle`s output unique identifiers, letting their values be determined by the `mapping`. The reason for this is that `compiler.tree` uses static single assignment (SSA) form, wherein every variable is defined by exactly one statement. This simplifies the properties of variables, which helps optimizations perform faster and with better results [Cytron et al. 1991]. By giving unique names to the targets of each assignment, the SSA property is guaranteed. However, `#branch`es introduce ambiguity: after, say, an `#if`, what will the `out-d` be? It depends on which branch is taken. To remedy this problem, after any `#branch` node, Factor will place a `#phi` node—the classical SSA "phony function", ϕ. While it doesn't perform any literal computation, conceptually ϕ selects between its inputs, choosing the "correct" argument depending on control flow. This can then be assigned to a unique value, preserving the SSA property. In Factor, a `#phi` node's arguments are represented by the `phi-in-d` slot, which is a sequence of sequences. Each element corresponds to the `out-d` of the respective child of the preceding `#branch` node. The `#phi`'s `out-d` gives unique names to the output values.

For example, the `#phi` in Figure 28 on the following page will select between the `6256248` return value of the first child or the `6256249` output of the second. Either way, we can refer to the result as `6256250` afterwards. The `terminated` slot of the `#phi` tells us if there was

```
V{
    T{ #introduce { out-d { 6256247 } } }
    T{ #if
        { in-d { 6256247 } }
        { children
            {
                V{
                    T{ #push
                        { literal 1 }
                        { out-d { 6256248 } }
                    }
                }
                V{
                    T{ #push
                        { literal 2 }
                        { out-d { 6256249 } }
                    }
                }
            }
        }
    }
    T{ #phi
        { phi-in-d { { 6256248 } { 6256249 } } }
        { out-d { 6256250 } }
        { terminated V{ f f } }
    }
    T{ #return { in-d V{ 6256250 } } }
}
```

Figure 28: `[ [ 1 ] [ 2 ] if ] build-tree`

a `#terminate` in any of the branches.

The `#recursive` node encapsulates *inline recursive* words. In Factor, words may be annotated with simple compiler declarations, which guide optimizations. If we follow a standard colon definition with the **`inline`** word, we're saying that its definition can be spliced into the call site, rather than generating code to jump to a subroutine. Inline words that call themselves must additionally be declared **`recursive`**. For example, we could write `: foo ( -- ) foo ; inline recursive`. The nodes `#enter-recursive`, `#call-recursive`, and `#return-recursive` denote different stages of the recursion—the beginning, recursive call, and end, respectively. They carry around a lot of metadata about the nature of the recursion, but it doesn't serve our purposes to get into the details. Similarly, we gloss over the final nodes of Figure 23 on page 31, which correspond to Factor's foreign function interface (FFI) vocabulary, called `alien`. At a high level, `#alien-node`, `#alien-invoke`, `#alien-indirect`, `#alien-assembly`, and `#alien-callback` are used to make calls to C libraries from within Factor.

Optimizations

Now that we're familiar with the structure of the high-level IR, we can turn our attention to optimization. Figure 29 on the following page shows the passes performed on a sequence of nodes by the word `optimize-tree`. Before optimization can begin, we must gather some information and clean up some oddities in the output of `build-tree`. `analyze-recursive` is called first to identify and mark loops in the tree. Effectively, this means we detect loops introduced by `#recursive` nodes. Future passes can then use this information for data flow analysis. Afterwards, `normalize` makes the tree more consistent by doing two things:

- All `#introduce` nodes are removed and replaced by a single `#introduce` at the beginning of the whole program. This way, further passes needn't handle `#introduce` nodes.

- As constructed, the `in-d` of a `#call-recursive` will be the entire stack at the time of the call. This assumption happens because we don't know how many inputs it needs until the `#return-recursive` is processed, because of row polymorphism (refer to Section 2.5). So, here we figure out exactly what stack entries are needed, and trim the `in-d` and `out-d` of each `#call-recursive` accordingly.

Once these passes have cleaned up the tree, `propagate` performs probably the most extensive analysis of all the phases. In short, it performs an extended version of sparse conditional constant propagation (SCCP) [Wegman and Zadeck 1991]. The traditional data flow analysis combines global copy propagation, constant propagation, and constant folding in a *flow-sensitive* way. That is, it will propagate information from branches that it knows

```
: optimize-tree ( nodes -- nodes' )
  [
      analyze-recursive
      normalize
      propagate
      cleanup
      dup run-escape-analysis? [
          escape-analysis
          unbox-tuples
      ] when
      apply-identities
      compute-def-use
      remove-dead-code
      ?check
      compute-def-use
      optimize-modular-arithmetic
      finalize
  ] with-scope ;
```

Figure 29: Optimization passes on the high-level IR

are definitely taken (e.g., because `#if` is always given a true input). Instead of using the typical single-level (numeric) constant value lattice, Factor uses a lattice augmented by information about numeric value ranges, array lengths/bounds, and classes (which form a partial order, as described briefly in Section 2.4). Additionally, the pass may inline certain calls if enough information is present. As values are refined, they propagate to other values that depend on them. For example, by refining the range of possible values a particular numeric variable can have, we might discover that, say, it's small enough to fit in a **`fixnum`** rather than a **`bignum`**. Then a math operator called on it may be inlined to a more specific version. Or, if the interval has zero length, we may replace the value with a constant, which contributes to constant folding.

`propagate` iterates through the nodes collecting all of this data until reaching a stable point where inferences can no longer be drawn. Technically, this information doesn't alter the tree at all; we merely store it so that speculative decisions may be realized later. The next word in Figure 29, `cleanup`, does just this by inlining words, folding constants, removing overflow checks, deleting unreachable branches, and flattening inline-recursive

```
TUPLE: data-struct
    { a read-only }
    { b read-only } ;

: escaping-via-#return ( -- data-struct )
    1 2 data-struct boa ;

: escaping-via-#call ( -- )
    1 2 data-struct boa pprint ;

: non-escaping ( -- a+b )
    1 2 data-struct boa [ a>> ] [ b>> ] bi + ;
```

Figure 30: Escaping vs. non-escaping tuple allocations

words that don't actually wind up calling themselves (e.g., because the calls got constant-folded).

The next major pass is `escape-analysis`, whose information is used for the actual transformation `unbox-tuples`. This discovers tuples that *escape* by getting passed outside of a word. For instance, the inputs to `#return` obviously escape, as they are passed to the world outside of the word in question. Similarly, inputs to the `#call` of another word escape. So, though the tuples in `escaping-via-#return` and `escaping-via-#call` in Figure 30 both escape, we can see the one in `non-escaping` does not. Because it does not escape to another location that may potentially use it, the last allocation is unnecessary. By identifying this, `unbox-tuples` can then rewrite the code to avoid allocating a `data-struct` altogether, instead manipulating the slots' values directly. Note that this only happens for immutable tuples, all of whose slots are `read-only`. Otherwise, we would need to perform more advanced pointer analyses to discover aliases.

`apply-identities` follows to simplify words with known identity elements. If, say, one argument to `+` is `0`, we can simply return the other argument. This converts the `#call` to `+` into a simple `#shuffle`. These identities are defined for most arithmetic words.

Another few simple passes come next in Figure 29 on the previous page. True to its name, `compute-def-use` computes where SSA values are defined and used. Values that are never used are eliminated by `remove-dead-code`. `?check` conditionally performs some consistency checks on the tree, mostly to make sure that no errors were introduced in the stack flow. If a global variable isn't toggled on, this part is skipped. We run `compute-def-use` again to update the information after altering the tree with dead code elimination.

Finally, `optimize-modular-arithmetic` performs a form of strength-reduction on arithmetic words that only use the low-order bits of their inputs/results, which may also remove unnecessary overflow checks. `finalize` cleans up a few random miscellaneous bits of the tree (removing empty shufflers, deleting `#copy` nodes, etc.) in preparation for lower-level optimizations.

3.3 Low-level Optimizations

Representation

The low-level 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 [Aho et al. 2007]. Construction of the low-level IR proceeds by analyzing the control flow of the high-level IR and converting the nodes of Section 3.2 into lower-level instructions typical of 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.

For posterity, below is a categorized list of all the instruction objects (each one is a subclass of the `insn` tuple). It's not imperative to know each of these instructions. Their names generally indicate their purposes clearly enough. If need be, though, you can refer back to this list.

- 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`
- Single instruction, multiple data (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 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`
- GC checks: `##save-context`, `##check-nursery-branch`, `##call-gc`
- Spills and reloads, inserted by the register allocator: `##spill`, `##reload`

Optimizations

By translating the high-level IR into instructions that manipulate registers directly, we reveal (and sometimes introduce) further redundancies that can be optimized away. The `optimize-cfg` word in Figure 31 on the following page 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 32 on page 44, which shows the effect of `optimize-tail-calls` on the following definition:

```
: 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

```
: 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 ;
```

Figure 31: Optimization passes on the low-level IR

the left of Figure 32 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.

The next pass in Figure 31 is `delete-useless-conditionals`, which removes branches that go to the same basic block. This situation might occur as a result of optimizations performed on the high-level IR. To see it in action, Figure 33 on page 45 shows the transformation on a purposefully useless conditional, `[ ] [ ] if`. On the left, 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 in block 2 using `##inc-d -1`. Then, `##compare-imm-branch` 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/=`)—that is, to see if we should take the "then" branch or the "else" one. However, both branches jump through several empty blocks and merge at the same destination. Thus, we can remove them and replace `##compare-imm-branch` with an unconditional `##branch` to the eventual destination. We see this on the right of Figure 33 on page 45.

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

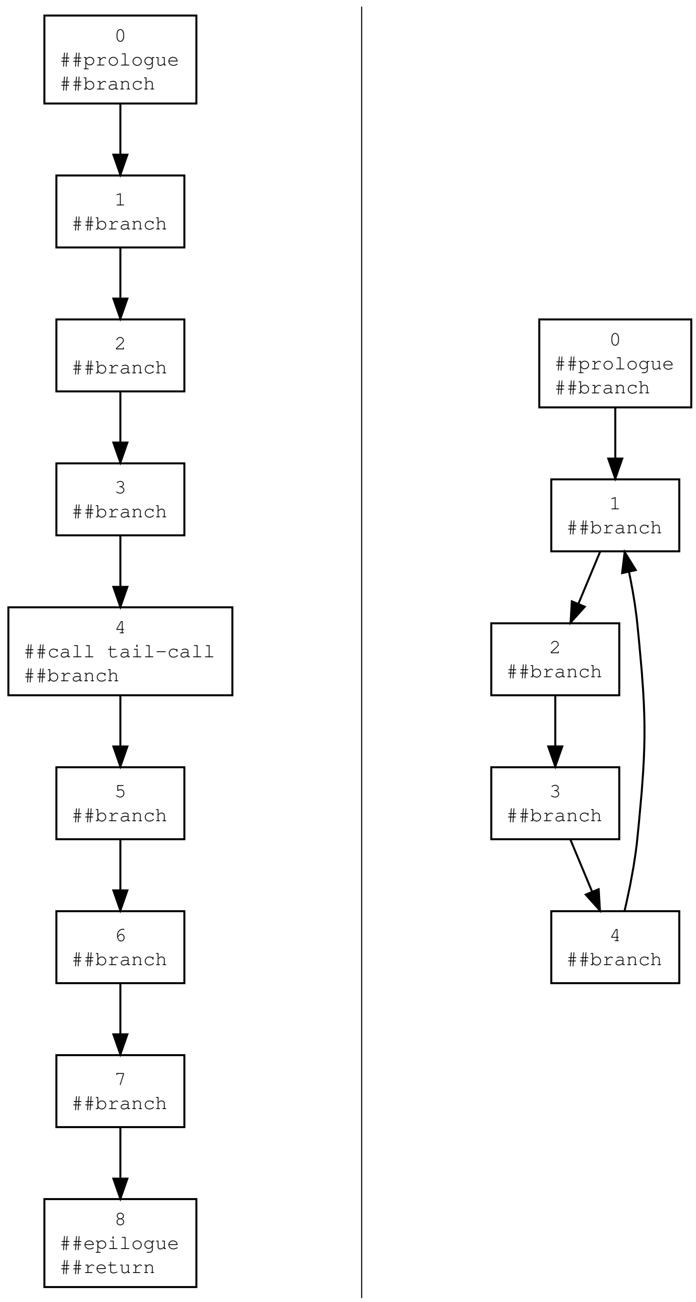

Figure 32: `tail-call` before and after `optimize-tail-calls`

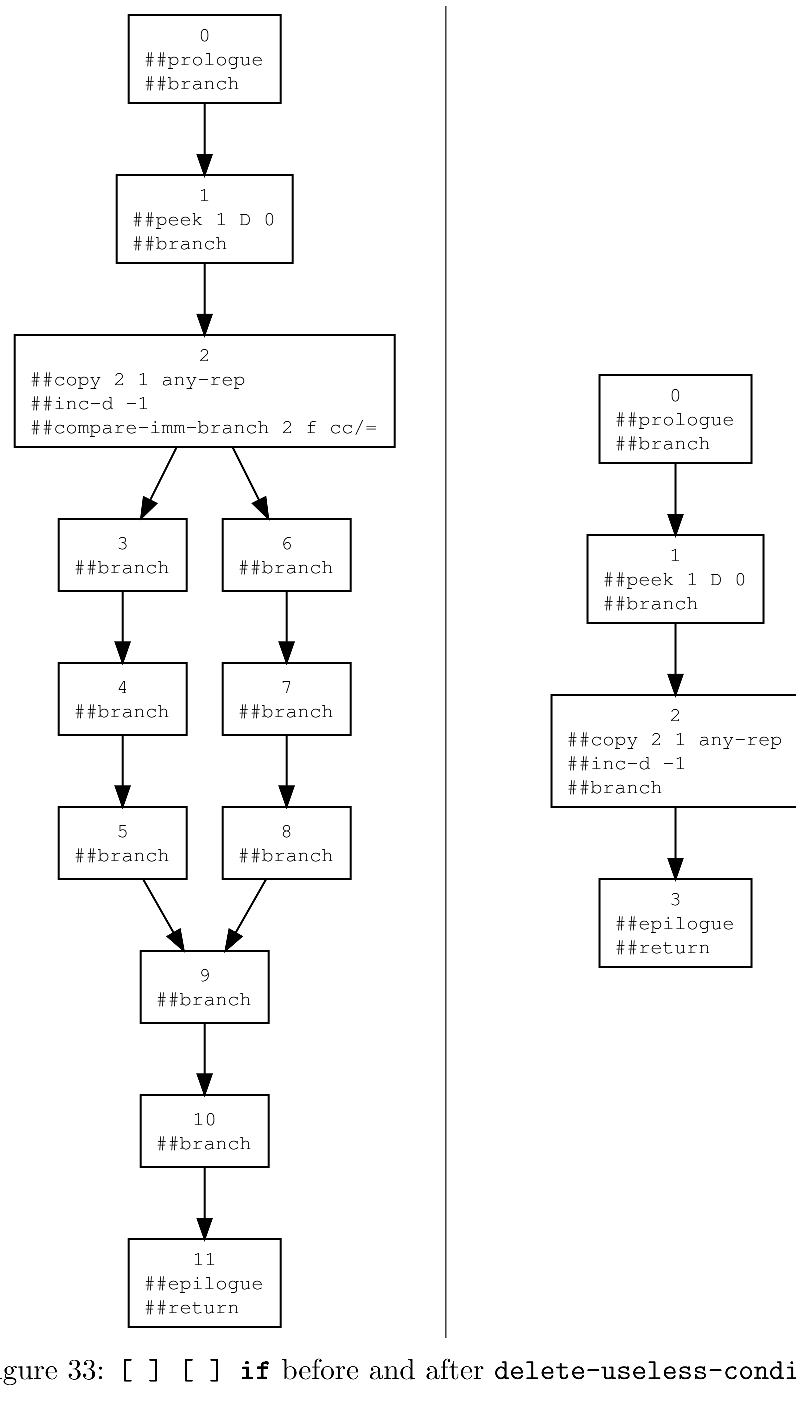

Figure 33: `[ ] [ ] if` before and after `delete-useless-conditionals`

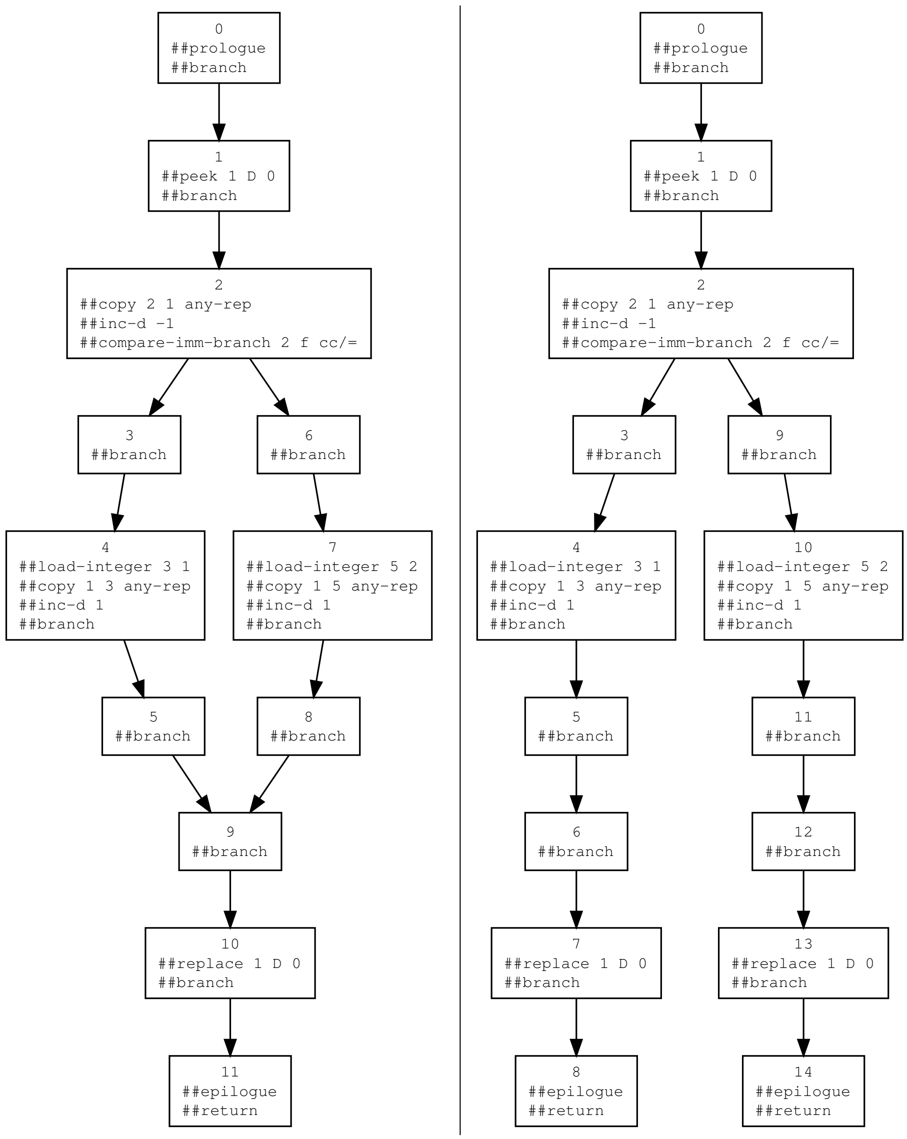

Figure 34: `[ 1 ] [ 2 ] if dup` before and after `split-branches`

executed along both branches regardless. 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 34 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 35 on the next page shows this pass on the CFG of `0 100 [ 1 fixnum+fast ] times`, which increments the top of the stack 100 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 Figure 31 on page 43, as it's a motivating example for the work in this thesis. None of the passes before `join-blocks` change the CFG seen on the left in Figure 35 on the next page, but we get rid of the useless `##branch` blocks, giving us the CFG on the right.

Figure 36 on page 49 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 cleaning: multiple height changes are combined into single ones at the beginnings of the basic blocks. In Figure 36 on page 49, this means that `##inc-d` is moved to the top of block 1, as compared to the right of Figure 35 on the following page.

In converting the high-level IR to the low-level, we actually lose the 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 [Briggs et al. 1998; Das and Ramakrishna 2005]. Figure 37 on page 50 shows the reconstructed SSA form of the CFG from Figure 36 on page 49, where `##phi`s are inserted at the start of block 2. Not only is there one for the element we're incrementing with `fixnum+fast`, but `times` introduces an induction variable to track how many iterations have been done. As block 2 is the entry for the loop, it either gets the values of these variables from the beginning (block 1) for from the actual body of the loop (block 3). Hence, we need `##phi`s to distinguish the values.

The next pass, `alias-analysis`, doesn't change the CFG we're currently working with, so we won't have an accompanying figure. At a high level, `alias-analysis` is easy to

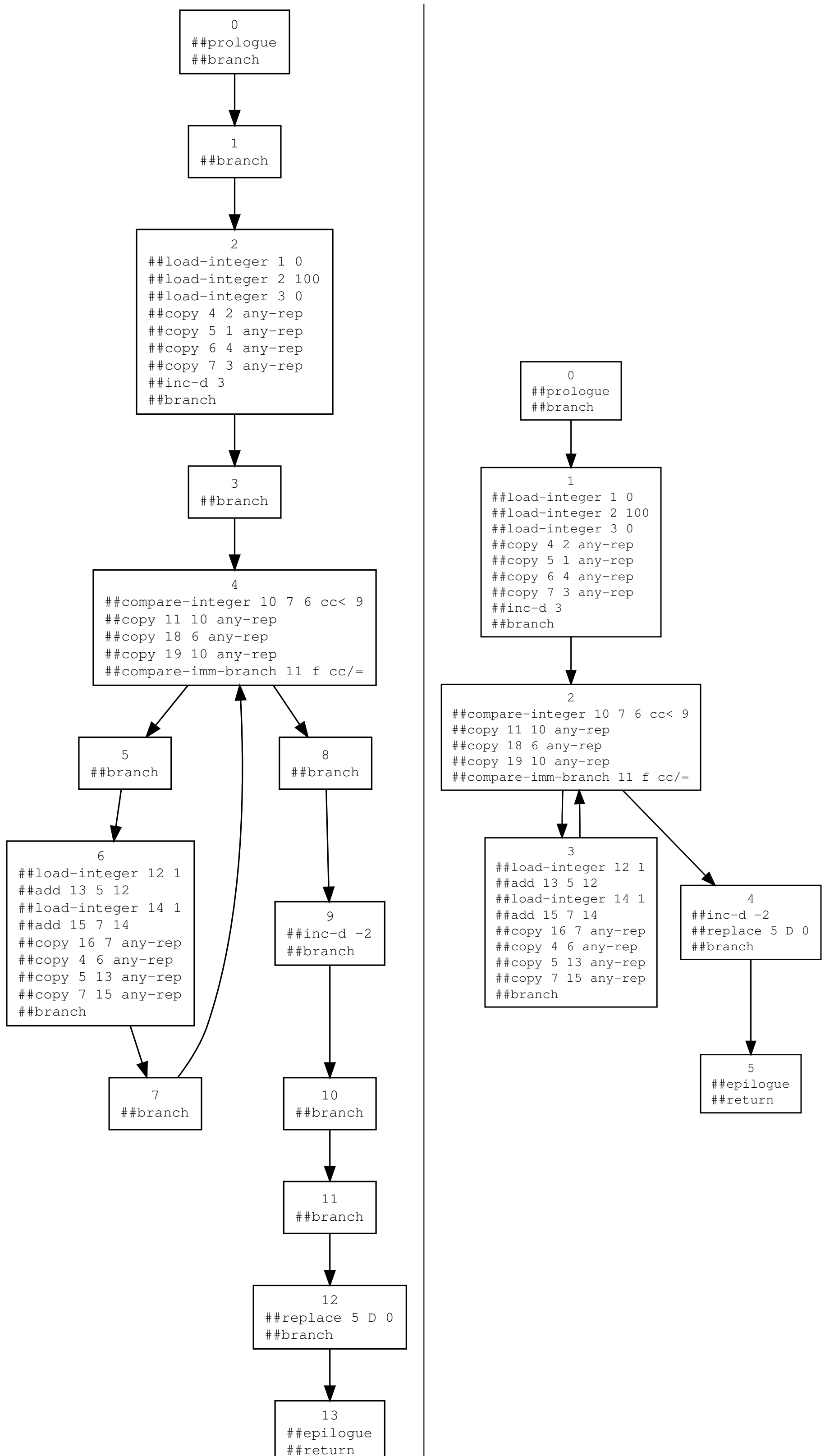

Figure 35: `0 100 [ 1 fixnum+fast ] times` before and after `join-blocks`

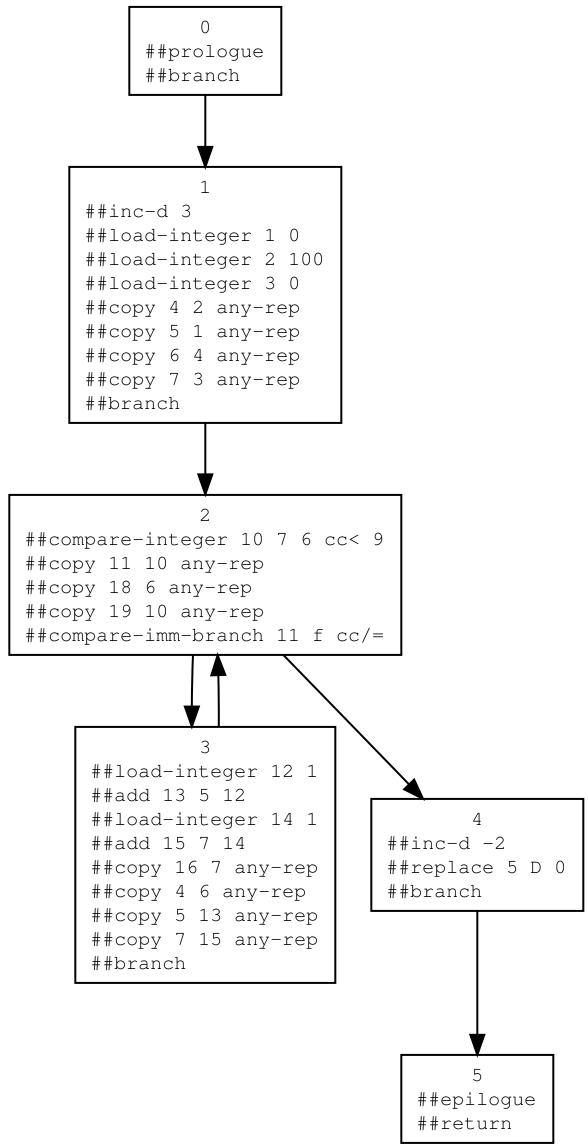

Figure 36: `0 100 [ 1 fixnum+fast ] times` after `normalize-height`

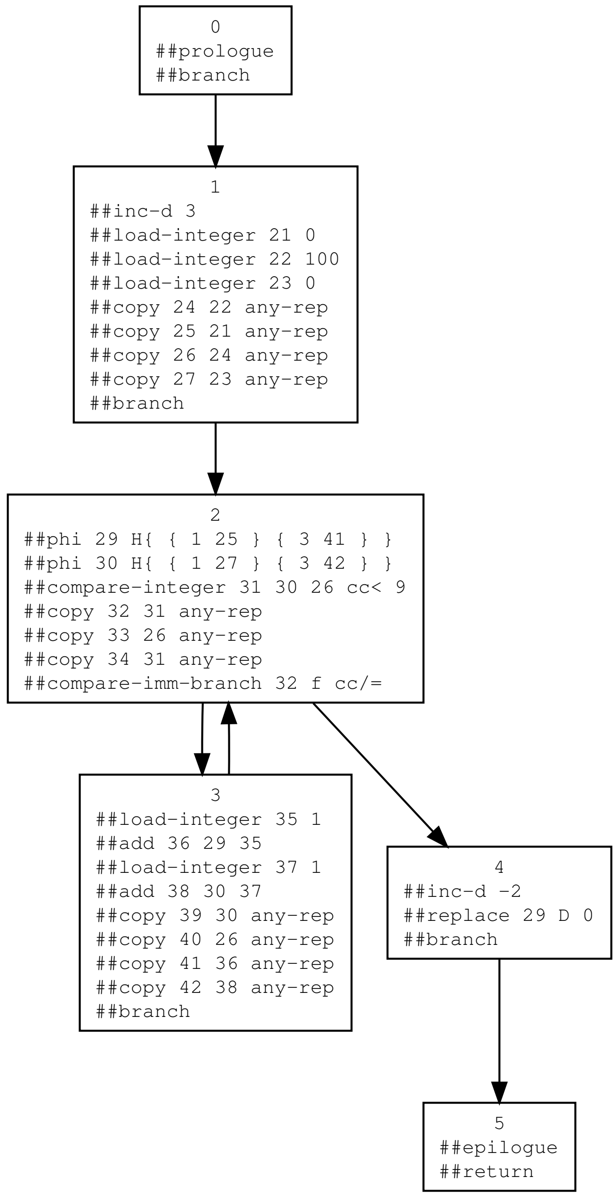

Figure 37: `0 100 [ 1 fixnum+fast ] times` after `construct-ssa`

understand: it eliminates redundant memory loads and stores by rewriting certain patterns of memory access. If the same location is loaded immediately after being stored, we convert the load into a `##copy` of the value we wrote. 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 Chapter 4. For now, it does to think of it as a combination of common subexpression elimination and constant folding. In Figure 38 on the next page, we see several changes from this pass:

- `##load-integer 23 0` in block 1 of Figure 37 on the preceding page (which assigns the value `0` to the virtual register `23`) is redundant, so is replaced by `##copy 23 21`.
- The last instruction in block 2 is redundant. The original

  ```
  ##compare-imm-branch 32 f cc/=
  ```

  intuitively means "if $\textsc{reg}[\texttt{32}] \neq \texttt{f}$, go to …". Its replacement,

  ```
  ##compare-integer-branch 30 26 cc<
  ```

  means "if $\textsc{reg}[\texttt{30}] < \textsc{reg}[\texttt{26}]$, go to…". This is because the source register (`32`) of the original is a `##copy` of `31`, which itself is computed by

  ```
  ##compare-integer 31 30 26 cc< 9
  ```

  In pseudo-code, this is like $\textsc{reg}[\texttt{31}] \leftarrow (\textsc{reg}[\texttt{30}] < \textsc{reg}[\texttt{26}])$, where `9` is a temporary virtual register used for calculation. So, the `##compare-imm-branch` is equivalent to a simple `##compare-integer-branch`, because the former is saying "if $(\textsc{reg}[\texttt{30}] < \textsc{reg}[\texttt{26}]) \neq \texttt{f}$, go to…", while the latter simply says "if $\textsc{reg}[\texttt{30}] < \textsc{reg}[\texttt{26}]$, go to…". Converting it means that the calculation of `31` is unnecessary, so may be pruned away later.
- 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.

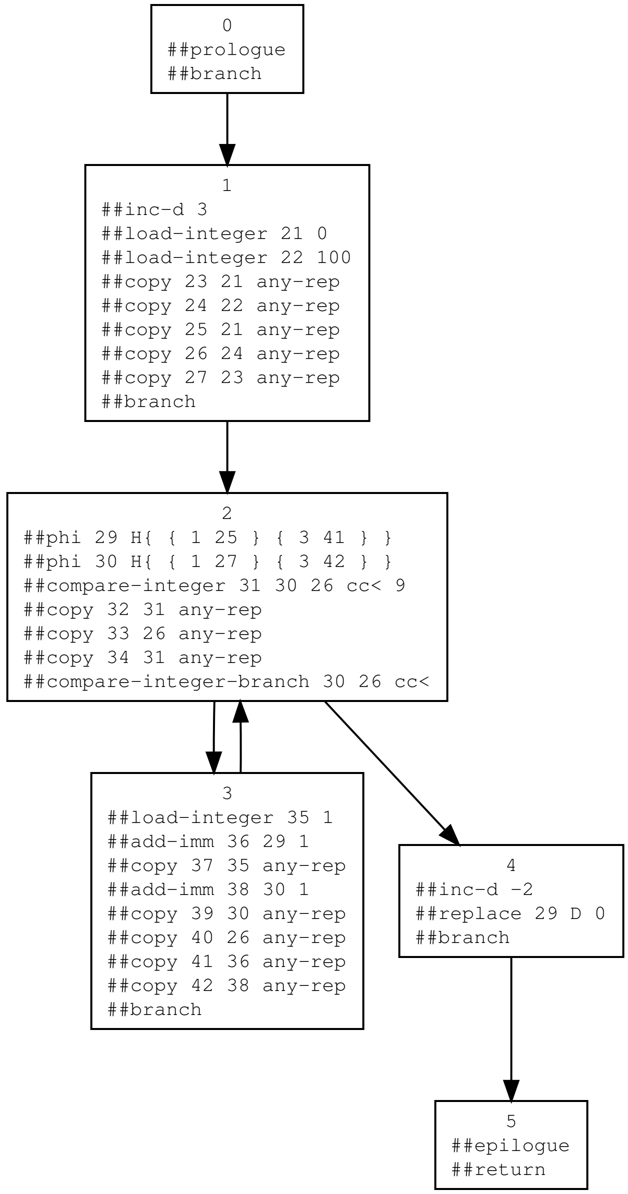

Figure 38: `0 100 [ 1 fixnum+fast ] times` after `value-numbering`

- Also, the second `##load-integer` in block 3 just loads `1` like the first instruction in the block. Therefore, it's replaced by a `##copy`.

In Chapter 4, 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 39 on the next page, we can see that all of the `##copy` instructions have been removed. Also, for

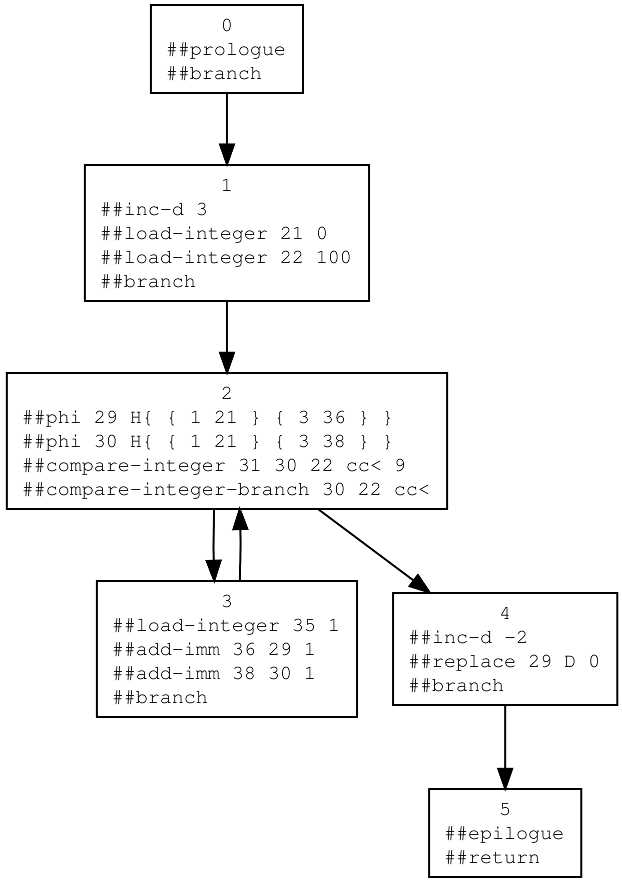

Figure 39: `0 100 [ 1 fixnum+fast ] times` after `copy-propagation`

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 40 on the following page 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 Figure 31 on page 43, `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. First, we 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 on slower memory accesses. After removing

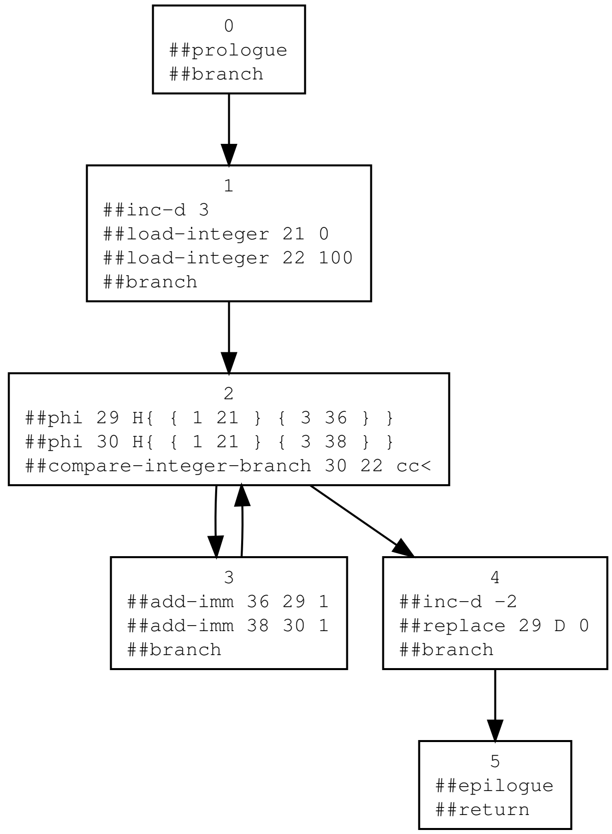

Figure 40: `0 100 [ 1 fixnum+fast ]` **`times`** after `eliminate-dead-code`

`##phi`s and 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 41 on the following page shows the output on an Intel x86 machine, which has enough registers that we needn't spill anything.

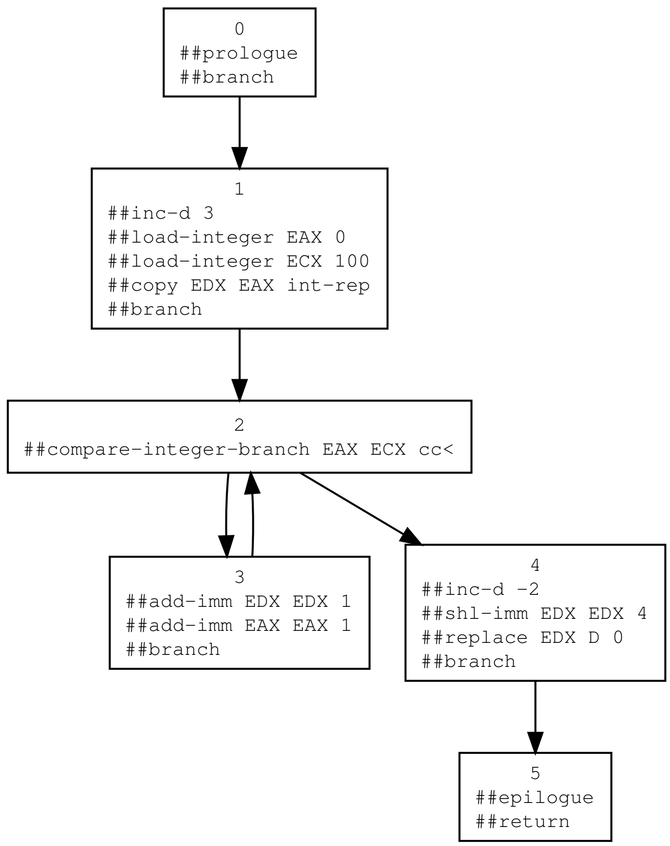

Figure 41: `0 100 [ 1 fixnum+fast ]` **`times`** after `finalize-cfg`

4 Value Numbering

At a very basic level, most optimization techniques revolve around avoiding redundant or unnecessary computation. Thus, it's vital that we discover which values in a program are equal. That way, we can simplify the code that wastes machine cycles repeatedly calculating the same values. Classic optimization phases like constant/copy propagation, common subexpression elimination, loop-invariant code motion, induction variable elimination, and others discussed in the de facto treatise, "The Dragon Book" [Aho et al. 2007], perform this sort of redundancy elimination based on information about the equality of expressions.

In general, the problem of determining whether two expressions in a program are equivalent is undecidable. Therefore, we seek a *conservative* solution that doesn't necessarily identify all equivalences, but is nevertheless correct about any equivalences it does identify. Solving this equivalence problem is the work of *value numbering* algorithms. These assign every value in the program a number such that two values have the same value number if and only if the compiler can prove they will be equal at runtime.

Value numbering has a long history in literature and practice, spanning many techniques. In Section 3.3 we saw the `value-numbering` word, which is actually based on some of the earliest—and least effective—methods of value numbering. Section 4.1 describes the way Factor's current algorithm works, highlighting its shortcomings to motivate the main work of this thesis, which is covered in Sections 4.2 and 4.3. We finish the chapter by analyzing the results of these changes and reviewing the literature for further enhancements that can be made to this optimization pass.

4.1 Local Value Numbering

Tracing the exact origins of value numbering is difficult. It's thought to have originally been invented in the 1960s by Balke [Simpson 1996]. The earliest tangible reference to value numbering (at least, the earliest point where discussions in the literature seem to start) appears in an oft-cited but unpublished work of Cocke and Schwartz [1970]. The technique is relatively simple, but not as powerful as other methods for reasons described hereafter.

The algorithm considers a single basic block. For each instruction (from top to bottom) in the block, we essentially let the value number of the assignment target be a hash of the operator and the value numbers of the operands. That is, we hash the *expression* being computed by an instruction. Thus, assuming a proper hash function, two expressions are *congruent* if they have the same operators and their operands are congruent. This is our approximation of runtime equivalence. It's important that the hash is based on the value numbers of the statement's operands, not just the operands as they appear in code (i.e., *lexical* equivalence). Any information about congruence is propagated through the value numbers. We'll have discovered any such equivalences among the operands before computing the value number of the assignment target. This is because every value in a basic block is either defined before it's used, or else defined at some point in a predecessor, which we don't care about when only considering one basic block.

This is the first shortcoming of the algorithm. It is *local*, focusing on only one basic block at a time. Any definitions outside the boundaries of the basic block won't be reused, even if they reach the block. This severely limits the scope of the redundancies we can discover. We could improve upon this by considering the algorithm across an entire loop-free CFG in any *topological order*. In such an ordering, a basic block B comes before any other block B' to which it has an edge. Thus, any "outside" variables that instructions in B' rely on must have come from B or earlier, which will have already been computed in a traversal of such an ordering. However, CFGs usually contain cycles or loops (at least interesting ones do), which make such an ordering impossible. We could still pick a topological order that ignores back-edges, but we may encounter operands whose values flow along those back-edges, so haven't been processed yet. We'll address this issue later.

In Factor, expressions are basically instructions (the `insn` objects discussed in Section 3.3) that have had their destination registers stripped. Instructions can be converted to expressions with the `>expr` word defined in the `compiler.cfg.value-numbering.expressions` vocabulary. For instance, an `##add` instruction with the destination register `1` and source registers `2` and `3` will be converted into an array of three elements:

- The `##add` class word, indicating the expression is derived from an `##add` instruction.
- The value number of the virtual register `2`.
- The value number of the virtual register `3`.

Some instructions are not *referentially transparent*, meaning they can't be replaced with the value they compute without changing the program's behavior. For example, `##call` and `##branch` cannot reasonably be converted into expressions. In these cases, `>expr` merely returns a unique value.

The hashing of expressions takes place in the so-called *expression graph* implemented in the vocabulary shown in Figure 42 on the next page. This consists of three global hash tables that relate virtual registers, value numbers, instructions, and expressions. Since virtual registers are just integers, we actually use them as value numbers, too. `vregs>vns` maps virtual registers to their value numbers. If a virtual register is mapped to itself in this table, its definition is the canonical instruction that we use to compute the value. This instruction is stored in the `vns>insns` table. Finally, the most important mapping is `exprs>vns`. True to its name, it uses expressions as keys, which (of course) are implicitly hashed. Thus, we can use this table to determine the equivalence of expressions.

Other definitions in Figure 42 on the following page manipulate expressions and the graph. The global variable `input-expr-counter` is used in the generation of unique expressions discussed earlier. `init-value-graph` initializes this and all the tables. `set-vn` establishes a mapping from a virtual register to a value number in `vregs>vns`. `vn>insn` gives terse access to the `vns>insns` table. `vreg>insn` uses `vregs>vns` and `vns>insns` to get the canonical instruction that defines a given virtual register. Finally, `vreg>vn` looks up the value of a key in the `vregs>vns` table. Importantly, if the key is not yet present in the table, it is automatically mapped to itself—it's assumed that the virtual register does not correspond to a redundant instruction.

This is the second shortcoming of the algorithm. It must make a *pessimistic* assumption about congruences. That is, it starts by assuming that every expression has a unique value number, then tries to prove that there are some values which are actually congruent. This

```
! Copyright (C) 2008, 2010 Slava Pestov.
! See http://factorcode.org/license.txt for BSD license.
USING: accessors kernel math namespaces assocs ;
IN: compiler.cfg.value-numbering.graph

SYMBOL: input-expr-counter

! assoc mapping vregs to value numbers
! this is the identity on canonical representatives
SYMBOL: vregs>vns

! assoc mapping expressions to value numbers
SYMBOL: exprs>vns

! assoc mapping value numbers to instructions
SYMBOL: vns>insns

: vn>insn ( vn -- insn ) vns>insns get at ;

: vreg>vn ( vreg -- vn ) vregs>vns get [ ] cache ;

: set-vn ( vn vreg -- ) vregs>vns get set-at ;

: vreg>insn ( vreg -- insn ) vreg>vn vn>insn ;

: init-value-graph ( -- )
    0 input-expr-counter set
    H{ } clone vregs>vns set
    H{ } clone exprs>vns set
    H{ } clone vns>insns set ;
```

Figure 42: The `compiler.cfg.value-numbering.graph` vocabulary

```
: value-numbering-step ( insns -- insns' )
    init-value-graph
    [ process-instruction ] map flatten ;

: value-numbering ( cfg -- cfg )
    dup [ value-numbering-step ] simple-optimization

    cfg-changed predecessors-changed ;
```

Figure 43: Main words from `compiler.cfg.value-numbering`

fails to discover congruences for values that flow along back-edges, whether we consider a single basic block or an entire topological ordering.

One the other hand, an advantage of this local value numbering algorithm is its simplicity. It makes a single pass over all the instructions, identifying and replacing redundancies *online* (i.e., rewriting as it goes). It's straightforward to write, and even to extend. At every step, the currently known value numbers will be sound, so we can use this information for copy/constant propagation, constant folding, and common subexpression elimination.

To see how Factor accomplishes these extensions, we'll take a look at the `compiler.cfg.value-numbering` vocabulary. Figure 43 shows the main words that start the optimization pass. The `value-numbering-step` word is called on the sequence of instructions that comprise each basic block. It starts the expression graph from a blank slate with `init-value-graph`, then **`map`**s the word `process-instruction` on each of them. This is a generic word that we'll study momentarily; it returns either a single `insn` object or a sequence of them in the case that an instruction is replaced by several others. Then, the work of `value-numbering` is to just call `value-numbering-step` on each basic block, which is done with a combinator called `simple-optimization`. The words `cfg-changed` and `predecessors-changed` alter some internal state of the CFG that has been potentially invalidated by some transformations performed by `process-instruction`.

The methods of `process-instruction` are shown in Figure 44 on the following page. Skipping the first three normal words for the moment, the default behavior for dispatching on an `insn` is to invoke yet another generic word, `rewrite`. This word will return either

```
GENERIC: process-instruction ( insn -- insn' )

: redundant-instruction ( insn vn -- insn' )
    [ dst>> ] dip [ swap set-vn ] [ <copy> ] 2bi ;

:: useful-instruction ( insn expr -- insn' )
    insn dst>> :> vn
    vn vn vregs>vns get set-at
    vn expr exprs>vns get set-at
    insn vn vns>insns get set-at
    insn ;

: check-redundancy ( insn -- insn' )
    dup >expr dup exprs>vns get at
    [ redundant-instruction ] [ useful-instruction ] ?if ;

M: insn process-instruction
    dup rewrite [ process-instruction ] [ ] ?if ;

M: foldable-insn process-instruction
    dup rewrite
    [ process-instruction ]
    [ dup defs-vregs length 1 = [ check-redundancy ] when ] ?if ;

M: ##copy process-instruction
    dup [ src>> vreg>vn ] [ dst>> ] bi set-vn ;

M: array process-instruction
    [ process-instruction ] map ;
```

Figure 44: The workhorse of `compiler.cfg.value-numbering`

a replacement `insn` (or sequence thereof) or `f`, indicating that no change has taken place. Thus, by recursively calling `process-instruction`, we can do more specialized processing on this rewritten replacement (e.g., dispatching on `insn` again, which applies `rewrite` once more). If the instruction can't be simplified further, we simply return it. (Note that `[ X ] [ Y ] ?if` is the same as `dup [ nip X ] [ drop Y ] if`.)

For instances of `foldable-insn` (a subclass of `insn` for those that can be converted to useful expressions with `>expr`), we similarly invoke `rewrite` recursively until no more rewriting occurs. When that happens, rather than just return the instruction, we invoke `check-redundancy`—though only if the instruction defines exactly one virtual register, which will be stored in a slot named `dst`. `check-redundancy` checks if the expression being computed by the instruction is already a key of the `exprs>vns` table. If it is, the instruction is redundant, and we call `redundant-instruction`; otherwise, we call `useful-instruction`. The former uses `set-vn` to give the instruction's `dst` virtual register the same value number as the equivalent expression. Since value numbers are actually virtual registers, we may also use these as the source and destination registers in a new `##copy` instruction, which is then returned. On the other hand, `useful-instruction` saves the instruction's information in the expression graph by setting the appropriate values in `vregs>vns`, `exprs>vns`, and `vns>insns`.

The `##copy` method of `process-instruction` cannot do anything to simplify the instruction, but will set the value number of the destination register to that of the source. By calling `vreg>vn` on the source register, we make sure to call `set-vn` between the destination and the canonical value number of the source.

Finally, the `array` method is used for the purposes of recursion, in the case that `rewrite` returns a sequence of replacement instructions. The correct action is, of course, to descend into this new sequence of instructions with `process-instruction`.

Underlying all of the redundancy elimination is the `rewrite` generic word. It has too many methods to look at the source code in-depth here, but it's instructive to give an overview of the transformations. These methods actually make up the bulk of the `compiler.cfg.value-numbering` code. They're spread across various sub-vocabularies. `compiler.cfg.value-numbering.rewrite` defines the generic itself, along with a handful of utilities. The method for the most general instruction class, `insn`, is defined to unconditionally return `f`, meaning no rewriting is performed by default. That way, we need only define `rewrite` methods for more specific instruction classes to get specialized behavior.

`compiler.cfg.value-numbering.alien` contains methods that simplify nodes related to Factor's FFI. Most involve fusing together the results of intermediate arithmetic. The instructions that access raw memory (namely `##load-memory`, `##load-memory-imm`, `##store-memory`, and `##store-memory-imm`) tend to have arguments for address arithmetic. Each has slots for a `base` register containing an address and a literal `offset` from it. But if `base` is defined by an `##add-imm` instruction, we can just update the `offset`, incrementing it by the literal operand of the `##add-imm`. Then, `base` will just be changed to the register operand of the `##add-imm`. This removes the memory instruction's need for the `##add-imm`, increasing the chances that the latter will become dead code to be removed later. Unlike the `-imm` variants, `##load-memory` and `##store-memory` also take a `displacement` register, which works like a non-immediate `offset`. Therefore, `##add`s can be similarly fused into `##load-memory-imm` and `##store-memory-imm` by transforming them into `##load-memory` and `##store-memory` instructions with the `##add`'s operand as the `displacement`. A few other similar transformations are also done, including rewrites for `##box-displaced-alien`s and `##unbox-any-c-ptr`s.

`compiler.cfg.value-numbering.comparisons` defines methods for the various branching and comparison instructions (which simply store booleans in registers, rather than branching upon them). The major optimizations performed are as follows:

- If possible, instructions are converted to more specific forms. For example, non-immediate instructions (e.g., `##compare`) may be turned into their `-imm` counterparts (e.g., `##compare-imm`) if one of their source registers corresponds to a literal value. `##compare-integer-imm` is also converted to `##test` if the architecture supports it. This corresponds to a special instruction in x86 that performs a bitwise AND for its side effects on particular flags, discarding the actual result. This can be more efficient when using the AND result as a boolean.

- If both inputs to a comparison or branch are literals, we may constant-fold the instruction. In the case of comparisons, this means converting it into a `##load-reference` of the proper boolean. In branches, this modifies the CFG so that the path which isn't taken is removed completely.

- Like a novice programmer writing `if (some_boolean != false) { ... }` in Java, the compiler may generate redundant boolean comparisons that need cleaning up.

That is, the intermediate boolean values are eliminated when the result of a comparison is used by another comparison, collapsing the whole thing into a single instruction.

`compiler.cfg.value-numbering.folding` defines some auxiliary words for constant-folding arithmetic. `unary-constant-fold` and `binary-constant-fold` perform the actual operation on the one or two constant inputs provided. These words are used in `compiler.cfg.value-numbering.math`, which predictably simplifies math via standard rules. Arithmetic identities are rewritten. Conceptually, $x + 0$ becomes just x, for instance. If self-inverting instructions (namely `##neg` for numerical negation and `##not` for boolean negation) are called on registers that themselves correspond to the same instruction, we can safely rewrite them into `##copy` instructions—intuitively turning a double-negative into a positive. Non-immediate instructions are converted to their `-imm` forms if possible. When both operands are constant, the expression is folded. The most interesting math optimizations use the associative and distributive laws. *Reassociation* conceptually converts $(x \otimes y) \otimes z$ into $x \otimes (y \otimes z)$ when both y and z are constants and $\otimes$ is associative. So, for example,

```
##add-imm 1 X Y
##add-imm 2 1 Z
```

is converted into just

```
##add-imm 2 X (Y+Z)
```

where `X` is a virtual register, and `Y` and `Z` are constants. *Distribution* converts $(x \oplus y) \otimes z$ into $(x \otimes z) \oplus (y \otimes z)$, where y and z are constants, $\oplus$ corresponds to addition or subtraction, and $\otimes$ to multiplication or left bitwise shifts. Therefore,

```
##add-imm 1 X Y
##mul-imm 2 1 Z
```

is converted into

```
##mul-imm 3 X Y
##add-imm 2 3 (Y*Z)
```

Notice that a new intermediate virtual register, `3`, had to be created. However, if the product of `Y` and `Z` can be computed at compile-time and fits in an immediate operand, then we save cycles by using `##mul-imm` on smaller numbers (multiplying the contents of `X` by the constant `Y`, rather than their sum by `Z`).

The last few methods of `rewrite` provide some obvious simplifications. `compiler.cfg.value-numbering.simd` performs some limited constant-folding for vector operations. `compiler.cfg.value-numbering.slots` propagates `##add-imm` address calculation to `##slot`, `##set-slot`, and `##write-barrier` instructions in a manner similar to `compiler.cfg.value-numbering.alien`. Finally, `compiler.cfg.value-numbering.misc` provides a single method to rewrite `##replace` into `##replace-imm` if possible.

To finish the discussion of local value numbering and Factor's particular implementation, we'll examine the example from Figure 38 on page 52 in depth. For convenience, the before/after snapshot of the CFG is reproduced in Figure 45 on the following page.

`value-numbering-step` begins at block 1, where `process-instruction` is **`map`**ped across the instructions. `##inc-d 3` does not have a `rewrite` method, so remains untouched; it is also not a `foldable-insn`, so it is simply returned. While `##load-integer 21 0` doesn't have a `rewrite` method, it is a `foldable-insn`, so `process-instruction` calls `check-redundancy`. At this point, the expression graph is empty. Calling `>expr` converts this instruction into an `integer-expr` object representing `0`. `useful-instruction` leaves the tables as follows:

```
! vregs>vns
H{ { 21 21 } }


! exprs>vns
H{ { T{ integer-expr { value 0 } } 21 } }
```

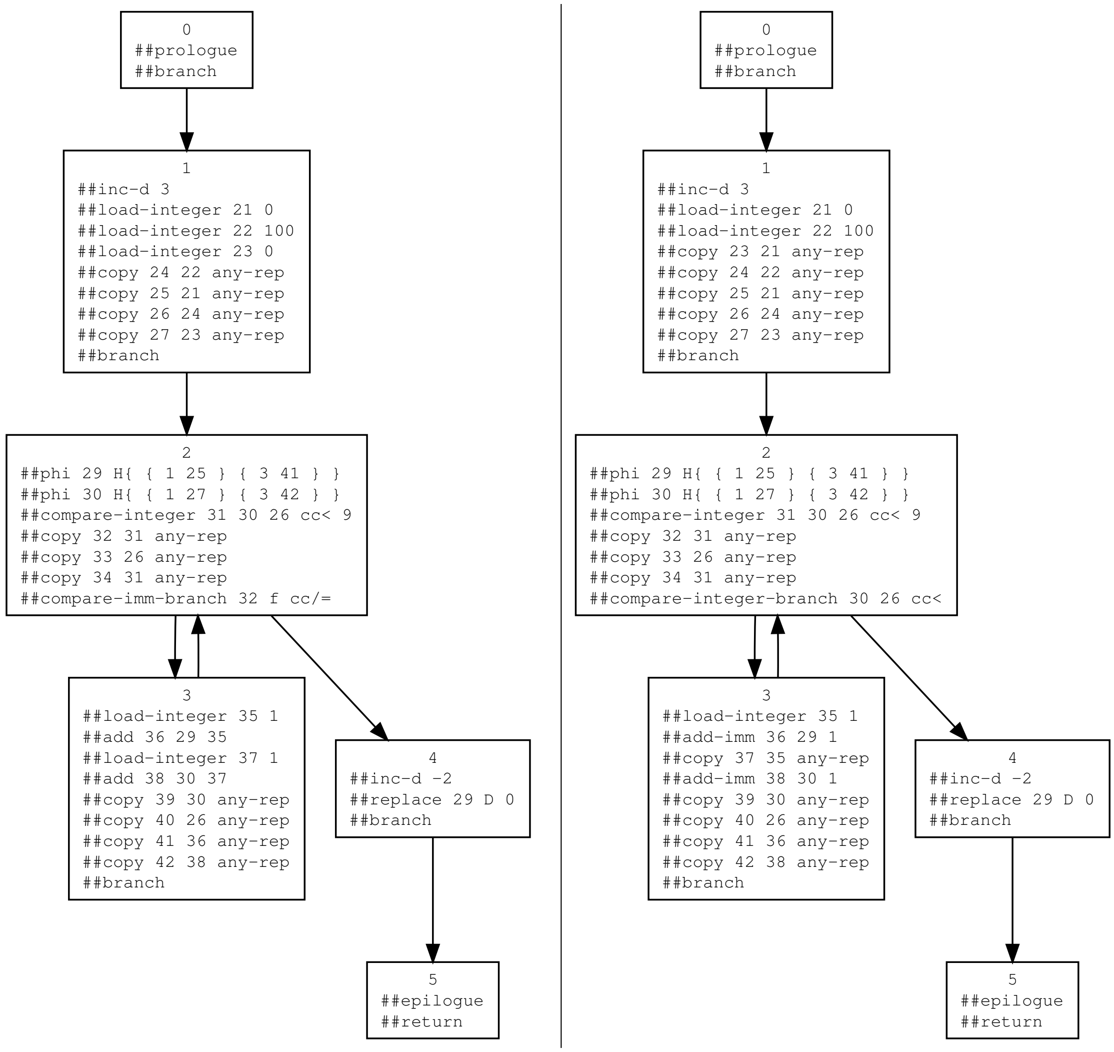

Figure 45: `0 100 [ 1 fixnum+fast ] times` before and after `value-numbering`

```
! vns>insns
H{
    { 21 T{ ##load-integer { dst 21 } { val 0 } { insn# 1 } } }
}
```

The next instruction in block 1, `##load-integer 22 100`, behaves similarly, leaving:

```
! vregs>vns
H{ { 21 21 } { 22 22 } }
```

```
! exprs>vns
H{
    { T{ integer-expr { value 0 } } 21 }
    { T{ integer-expr { value 100 } } 22 }
}


! vns>insns
H{
    { 21 T{ ##load-integer { dst 21 } { val 0 } { insn# 1 } } }
    {
        22
        T{ ##load-integer { dst 22 } { val 100 } { insn# 2 } }
    }
}
```

The following instruction is `##load-integer 23 0`. In calling `check-redundancy`, we discover that the integer expression for `0` is already in `exprs>vns`, so this is turned into a `##copy`, and the value number is noted. The remaining instructions in block 1 (aside from `##branch`) are all instances of `##copy`. `process-instruction` thus only sets their value numbers in the `vregs>vns` table, leaving us with the following at the end of block 1:

```
! vregs>vns
H{
    { 21 21 }
    { 22 22 }
    { 23 21 }
    { 24 22 }
    { 25 21 }
    { 26 22 }
    { 27 21 }
```

```
}


! exprs>vns
H{
    { T{ integer-expr { value 0 } } 21 }
    { T{ integer-expr { value 100 } } 22 }
}


! vns>insns
H{
    { 21 T{ ##load-integer { dst 21 } { val 0 } { insn# 1 } } }
    {
        22
        T{ ##load-integer { dst 22 } { val 100 } { insn# 2 } }
    }
}
```

Next, block 2 in Figure 45 on page 66 is processed. The tables are all reset, so even though block 1 happens to dominate block 2, none of its definitions are known to `value-numbering`. The `##phi`s are ignored, as no important methods dispatch upon them. In trying to rewrite the `##compare-integer`, we call `vreg>vn` on the operands. Since they aren't in the `vregs>vns` table yet, they are assumed to be unique values. This assumption is pessimistic—we'd rather the values be the same, so we can remove redundancy. It happens to be correct here, though, as `26` corresponds to the integer `100`, while `30` is an induction variable of the loop. However, `##compare-integer` cannot be rewritten into an immediate form, since our focus is local to the basic block, so we don't know that the virtual register `26` has the value `100`. The `##copy` instructions are processed as usual, and `##compare-imm-branch 32 f cc/=` is rewritten into a `##compare-integer-branch`, as the virtual register `32` has the same value (through the copies) as the `##compare-integer` result. This is a case of simplifying the

```
if (some_boolean != false) { ... }
```

pattern, and the definition of the register `31` becomes dead code after `rewrite` finishes with this last instruction. The expression graph is populated as follows by the end of block 2:

```
! vregs>vns
H{
    { 32 31 }
    { 33 26 }
    { 34 31 }
    { 26 26 }
    { 30 30 }
    { 31 31 }
}


! exprs>vns
H{ { { ##compare-integer 30 26 cc< } 31 } }


! vns>insns
H{
    {
        31
        T{ ##compare-integer
            { dst 31 }
            { src1 30 }
            { src2 22 }
            { cc cc< }
            { temp 9 }
            { insn# 2 }
```

```
        }
    }
}
```

Once again, the tables are reset and we proceed to block 3. The first instruction, `##load-integer 35 1`, is entered into the expression graph. Since `35` is an operand of `##add 36 29 35`, `rewrite` changes this instruction into an `##add-imm`, as we know the constant value of the operand. The next `##load-integer` gets turned into a `##copy`, like in block 1, and the next `##add` is similarly changed to `##add-imm`. The copies do little but set more value numbers. As `process-instruction` calls `vreg>vn` on their sources, we'll insert entries into `vregs>vns` for virtual registers defined outside of the block, like `26`. This leaves us with the following tables:

```
! vregs>vns
H{
    { 35 35 }
    { 36 36 }
    { 37 35 }
    { 38 38 }
    { 39 30 }
    { 40 26 }
    { 41 36 }
    { 26 26 }
    { 42 38 }
    { 29 29 }
    { 30 30 }
}

! exprs>vns
H{
```

```
    { { ##add-imm 30 1 } 38 }
    { { ##add-imm 29 1 } 36 }
    { T{ integer-expr { value 1 } } 35 }
}

! vns>insns
H{
    { 36 T{ ##add-imm { dst 36 } { src1 29 } { src2 1 } } }
    { 38 T{ ##add-imm { dst 38 } { src1 30 } { src2 1 } } }
    { 35 T{ ##load-integer { dst 35 } { val 1 } { insn# 0 } } }
}
```

The fourth invocation of `value-numbering-step` does not do anything interesting, as the `##replace` cannot be changed into a `##replace-imm`.

In summary, we managed to replace redundancies within basic blocks online by maintaining some simple hash tables. After copy propagation and dead code elimination, the CFG gets finalized to the one shown in Figure 46 on the following page. Because the value numbering algorithm was local, the `##compare-integer-branch` in block 2 could not be simplified to a `##compare-integer-imm-branch`, and we instead have to waste a register on the integer `100`. But it's important to note that even considering a topological ordering of the CFG wouldn't have worked, as we'd have to ignore back-edges. The `##phi`s that used to be in block 2 had inputs that flowed along the back-edge, and our pessimistic assumption would have to classify these values as distinct. One is for the counter introduced by **`times`**, and the other is from the top value of the stack being incremented by `fixnum+fast`. In this case, however, these induction variables are actually equal: both start at `0` and are incremented by `1` on each loop iteration. In terms of the CFG in Figure 46 on the next page, the `EAX` and `EDX` registers are equivalent. Yet the combination of the pessimism and locality of the algorithm keep us from discovering this.

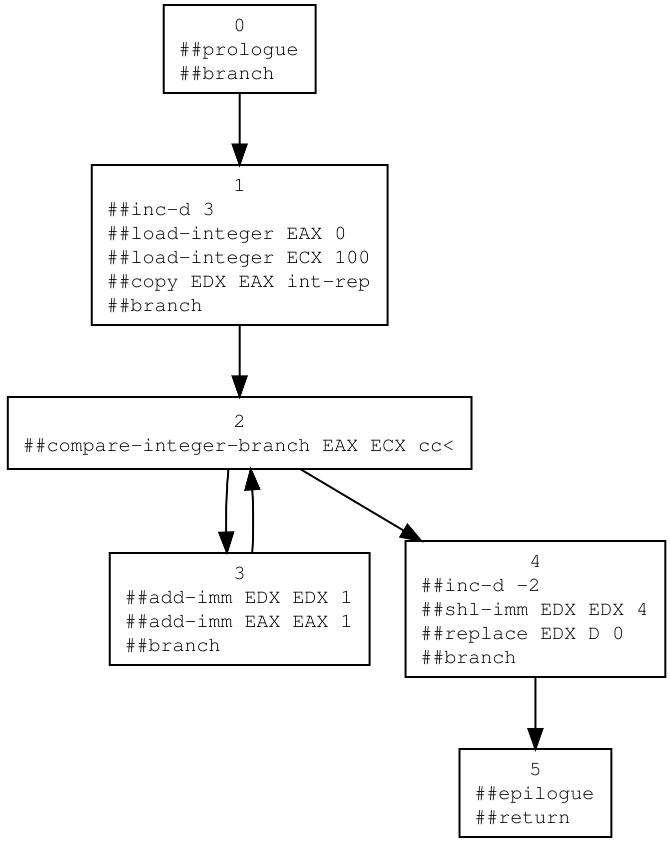

Figure 46: The final representation for `0 100 [ 1 fixnum+fast ] times`

4.2 Global Value Numbering

Background

Answering the challenges of Cocke and Schwartz, the researchers Alpern, Wegman, and Zadeck [1988] described what would be the de facto value numbering algorithm for several years, and rightly so. It was a properly *global* value numbering algorithm, working across an entire CFG instead of on single basic blocks. Their paper was important in another very relevant way: it is the first published reference to SSA form [VanDrunen 2004], including an algorithm for its construction.

Though we could try to extend the scope of Factor's local value numbering, it is still inherently pessimistic. The algorithm of Alpern, Wegman, and Zadeck, which is commonly referred to simply as AWZ, uses a modification of a minimization algorithm for finite state automata [Hopcroft 1971]. It works on an *optimistic* assumption by first assuming every value has the same value number, then trying to prove that values are actually different. It does this by treating value numbers as *congruence classes* that partition the set of virtual registers. If two values are in the same class, then they are congruent, where congruence is the same as in Section 4.1.

Such a partition is not unique, in general. For instance, a trivial one places each value in its own congruence class. So, we look for the *maximal fixed point*—the solution that has the most congruent values and therefore the fewest congruence classes. We must start with a congruence class for each operation so that, say, all values computed by `##add`s are grouped together, those computed by `##mul`s are in the same class, etc. We must then iteratively look at our collection of classes, separating them when we discover incongruent values. For an SSA variable in class P, we look at its defining expression. If an operand at position i belongs to class Q, then the i^{th} operand of every other value in P should also be in Q. Otherwise, P must be *split* by removing those variables whose i^{th} operands are not in class Q and placing them in a new congruence class. We keep splitting classes until the partitioning stabilizes.

The optimistic assumption may seem dangerous. Is it possible that we're "overoptimistic"? That two values assumed to be congruent and not proven incongruent might actually be inequivalent when the program is run? The AWZ paper dedicates a section to proving that two congruent variables are equivalent at a point p in the program if their definitions dominate p. The proof is a bit quirky, but reasonable. They develop a dynamic notion of dominance in a running program which implies static dominance in the code, then show that congruence implies runtime equality (though equivalence does not imply congruence).

AWZ made the need for global value numbering (GVN) algorithms apparent. However, finite state automata minimization makes for a more complicated algorithm than hash-based value numbering. A naïve implementation can be quadratic, although careful data structure and procedure design can make it $O(n \log n)$. Furthermore, it's resistant to the same improvements we easily added to the local value numbering. To even consider the commutativity of operations requires changes in operand position tracking and splitting—the heart of the algorithm. It is generally limited by what the programmer writes down: deeper congruences due to, say, algebraic identities can't be discovered.

Research progress in the decades following AWZ, with various people attempting to find better GVN algorithms. It came to light that performing an optimization using the information from GVN may introduce more congruences. If we can somehow perform the two analyses simultaneously, they'll produce better results. This generalizes to interdependent compiler optimizations at large, as elucidated in Click's dissertation [1995], which describes a method for formalizing and combining separate optimizations that make optimistic assumptions (whatever they happen to be for each particular analysis). He uses this to merge GVN with *conditional constant propagation*, which itself is a combination of constant propagation and unreachable code elimination (pretty much like the `propagate` pass from Section 3.2). Furthermore, GVN was extended to handle algebraic identities, propagate constants, and fold redundant ϕs. Unfortunately, the straightforward algorithm for this is $O(n^2)$, while the $O(n \log n)$ version presented is not just complicated, but can also miss some congruences between ϕ-functions [Click 1995; Simpson 1996].

Hot on the heels of this work, Simpson's [1996] dissertation provides probably the most exhaustive treatment of GVN algorithms. He presents several extensions, such as incorporating hash-based local value numbering into SSA construction, handling commutativity in AWZ, and performing redundant store elimination. He builds off of the two classical algorithms independently, which underlines their inherent differences and limitations. In general, hash-based value numbering is easy to extend without greatly impacting the runtime complexity, as is the case in Factor's implementation.

Drawing from this experience, Simpson's hallmark algorithm combines the best of both worlds by taking the hash-based algorithm which is easy to understand, implement, and extend, and making it global. Dubbed the "reverse postorder (RPO) algorithm", it simply applies hash-based value numbering iteratively over the CFG until we reach the same fixed point computed by AWZ. (The fact that it computes the exact same fixed point is proven fairly straightforwardly in the dissertation.) It could technically traverse the CFG in any topological order, but Simpson defaults to reverse postorder.

Because it is based off the hashing algorithm, we get the benefits essentially for free. The same simplifications can be performed, but with the added knowledge of global congruences. Since the majority of Factor's value numbering code is dedicated to the `rewrite` generic, it makes sense to reuse as much of that code as possible. Therefore, this thesis details the conversion of Factor's local algorithm to the RPO algorithm.

Implementation

The most fundamental change is to the expression graph. Referring to Figure 47 on the following page, we see most of the same code as in Figure 42 on page 59, with changes indicated by arrows ($\longrightarrow$). Two more global variables have been added, namely `changed?` and `final-iteration?`. The former is what we use to guide the fixed-point iteration. As long as value numbers are changing, we keep iterating. An important side effect of this is that we can no longer perform `rewrite` online, since the transformations we make aren't guaranteed to be sound on any iteration except the final one. This makes the RPO algorithm work *offline*, first discovering redundancies, then eliminating them in a separate pass. When this elimination pass starts, we'll set `final-iteration?` to **`t`**.

A key change is in the `vreg>vn` word, which now makes an optimistic assumption about previously unseen values. Given a new virtual register that wasn't in the `vregs>vns` table, the old version would map the register to itself, making the value its own canonical representative. However, if this version tries to look up a key that does not exist in the hash table, it will simply return **`f`** (which Factor will do by default with the **`at`** word). Therefore, every value in the CFG starts off with the same value "number", **`f`**. By the end of the GVN pass, there should be no value left that hasn't been put in the `vregs>vns` table, as we'll have processed every definition.

To keep track of whether `vregs>vns` changes, we simply need to alter `set-vn`. Here, we use **`maybe-set-at`**, a utility from the `assocs` vocabulary. This works like **`set-at`**, establishing a mapping in the hash table. In addition, it returns a boolean indicating

```
! Copyright (C) 2008, 2010 Slava Pestov, 2011 Alex Vondrak.
! See http://factorcode.org/license.txt for BSD license.
USING: accessors kernel math namespaces assocs ;
IN: compiler.cfg.gvn.graph

SYMBOL: input-expr-counter

! assoc mapping vregs to value numbers
! this is the identity on canonical representatives
SYMBOL: vregs>vns

! assoc mapping expressions to value numbers
SYMBOL: exprs>vns

! assoc mapping value numbers to instructions
SYMBOL: vns>insns

! boolean to track whether vregs>vns changes
SYMBOL: changed?

! boolean to track when it's safe to alter the CFG in a rewrite
! method (i.e., after vregs>vns stops changing)
SYMBOL: final-iteration?

: vn>insn ( vn -- insn ) vns>insns get at ;

: vreg>vn ( vreg -- vn ) vregs>vns get at ;

: set-vn ( vn vreg -- )
    vregs>vns get maybe-set-at [ changed? on ] when ;

: vreg>insn ( vreg -- insn ) vreg>vn vn>insn ;

: clear-exprs ( -- )
    exprs>vns get clear-assoc
    vns>insns get clear-assoc ;

: init-value-graph ( -- )
    0 input-expr-counter set
    H{ } clone vregs>vns set
    H{ } clone exprs>vns set
    H{ } clone vns>insns set ;
```

Figure 47: The `compiler.cfg.gvn.graph` vocabulary

change: if a new key has been added to the table, we return **`t`**. Otherwise, we return **`t`** only in the case where an old key is mapped to a new value. If an old key is mapped to the same value that's already in the table, **`maybe-set-at`** returns **`f`**. Therefore, when `vregs>vns` does change, we set `changed?` to **`t`** (which is what the **`on`** word does).

Finally, we define a new utility word, `clear-exprs`, which resets the `exprs>vns` and `vns>insns` tables. Unlike the local value numbering phase, we don't reset the entire expression graph. The only reason optimism works is that we keep trying to disprove our foolhardy assumptions about hte value numbers established by `vregs>vns`. At first, every value belongs in one congruence class by being mapped to **`f`**. We make a pass over the CFG to disprove whatever we can about this. If we've introduced new congruence classes (new values in the `vregs>vns` hash), we do another iteration. But each time, we use the congruence classes discovered from the previous iteration. At the start of each new pass, the expressions and instructions in `exprs>vns` and `vns>insns` are invalidated—their results are based on old information. So, these are erased on each iteration. Much like AWZ, we keep splitting classes until they can't be split anymore.

This logic is captured in Figure 48 on the following page. Rather than reset the tables when we start processing each basic block in `value-numbering-step` like before, we call `clear-exprs` on each iteration over the CFG in `value-numbering-iteration`. Note that `value-numbering-step` no longer returns the changed instructions, as we aren't replacing them online. Instead of `simple-optimization`, `value-numbering-iteration` uses `simple-analysis`, which only expects global state to change—no instructions are updated in the block. Much to our advantage, `simple-analysis` already traverses the CFG in reverse postorder, so we needn't worry about traversal order. The top-level word `determine-value-numbers` ties this all together by calling `value-numbering-iteration` until we can get through it with `changed?` remaining false.

Note that the work of `value-numbering-step` is further divided into two words, `simplify` and `value-number`. These combine to do much the same work as `process-instruction` in Figure 44 on page 61. `simplify` makes the repeated calls to `rewrite` until the instruction cannot be simplified further. Its definition is given in whole

```
: value-numbering-step ( insns -- )
    [ simplify value-number ] each ;

: value-numbering-iteration ( cfg -- )
    clear-exprs [ value-numbering-step ] simple-analysis ;

: determine-value-numbers ( cfg -- )
    final-iteration? off
    init-value-graph
    '[
        changed? off
        _ value-numbering-iteration
        changed? get
    ] loop ;
```

Figure 48: Main logic in `compiler.cfg.gvn`

```
GENERIC: simplify ( insn -- insn' )

M: insn simplify dup rewrite [ simplify ] [ ] ?if ;
M: array simplify [ simplify ] map ;
M: ##copy simplify ;
```

Figure 49: Iterated rewriting in `compiler.cfg.gvn`

in Figure 49. We then pass the simplified instruction to `value-number`, which is defined in Figure 50 on the next page. This also has a similar structure to `process-instruction`. The main difference is that instructions are no longer returned (again, they aren't altered in place). So, the `array` method uses **`each`** instead of **`map`** to recurse into the results of `rewrite`.

A subtle change is necessary with the `alien-call-insn` and `##callback-inputs` methods. Whereas `process-instruction` merely skipped over certain instructions that could not be rewritten, here we don't have that luxury. We need to be careful to `set-vn` every virtual register that gets defined by any instruction. While making a pessimistic assumption, it didn't matter if we did this: any unseen value would be presumed important by `vreg>vn`. However, with the optimistic assumption, `vreg>vn` will give the impression that unseen values are all the same by returning **`f`**. Therefore, we simply record the virtual registers that

```
GENERIC: value-number ( insn -- )

M: array value-number [ value-number ] each ;

: record-defs ( insn -- ) defs-vregs [ dup set-vn ] each ;

M: alien-call-insn value-number record-defs ;
M: ##callback-inputs value-number record-defs ;

M: ##copy value-number [ src>> vreg>vn ] [ dst>> ] bi set-vn ;

: redundant-instruction ( insn vn -- )
    swap dst>> set-vn ;

:: useful-instruction ( insn expr -- )
    insn dst>> :> vn
    vn vn set-vn
    vn expr exprs>vns get set-at
    insn vn vns>insns get set-at ;

: check-redundancy ( insn -- )
    dup >expr dup exprs>vns get at
    [ redundant-instruction ] [ useful-instruction ] ?if ;

M: ##phi value-number
    dup inputs>> values [ vreg>vn ] map sift
    dup all-equal? [
        [ drop ] [ first redundant-instruction ] if-empty
    ] [ drop check-redundancy ] if ;

M: insn value-number
    dup defs-vregs length 1 = [ check-redundancy ] [ drop ] if ;
```

Figure 50: Assigning value numbers in `compiler.cfg.gvn`

may be defined in any instructions. Specifically, `alien-call-insn` and `##callback-inputs` are classes that correspond to FFI instructions.

The `##copy` method uses `set-vn` the same way as before. `redundant-instruction`, `useful-instruction`, and `check-redundancy` are also largely the same. These have just been tweaked to not return instructions.

The `##phi` method in Figure 50 on the preceding page represents a major change. Before, `##phi`s were left uninterpreted. Congruences between induction variables that flowed along back-edges would not be identifiable. But now, by checking for redundant `##phi`s, we may reduce them to copies. Each `##phi` object has an `inputs` slot, which is a hash table from basic block to the virtual register that flows from that block. Thus, there is one input for each predecessor. The **`values`** of the hash will be the virtual registers that might be selected for the `dst` value. We look up the value numbers of these, removing all instances of **`f`** with the **`sift`** word. If all of the inputs are congruent, we can call `redundant-instruction`, setting the value number of the `##phi`'s `dst` to the value number of its first input (without loss of generality). The `all-equal?` word will return **`t`** if the sequence is empty (as it's vacuously true), so we must make sure guard the call to **`first`**, since it may be a runtime error. If the sequence is empty, we needn't note the redundancy, as the `##phi`'s `dst` will already have the optimistic value number **`f`** anyway. Otherwise, we call `check-redundancy`. The purpose of this is to identify `##phi`s that are equal to each other. Even if its inputs are incongruent, a `##phi` might still represent a copy of another induction variable. So that `check-redundancy` works, we also define a `>expr` method in `compiler.cfg.gvn.expressions`, as seen in Figure 51 on the next page. Here, the expression is an array consisting of the `##phi` class word, the current basic block's number, and the inputs' value numbers. We include the basic block number because only `##phi`s within the same block can be considered equivalent to each other.

The final method in Figure 50 on the preceding page defines the default behavior for `value-number`, which calls `check-redundancy` on the simplified instruction if it defines a single virtual register. Note that we separate the `alien-call-insn` and `##callback-inputs` logic from this, since they happen to define a variable number of

```
M: ##phi >expr
    inputs>> values [ vreg>vn ] map
    basic-block get number>> prefix
    \ ##phi prefix ;
```

Figure 51: ϕ expressions in `compiler.cfg.gvn.expressions`

registers. A particular instance may define only one register, but still won't have a `dst` slot. To avoid calling `dst>>` and triggering an error in `useful-instruction`, we needed separate methods for the FFI classes.

With these changes, we can globally identify value numbers, including equivalences that arise from simplifying instructions (even though no replacements are actually done yet). To illustrate this, consider again the example `0 100 [ 1 fixnum+fast ] times`, reproduced in Figure 52 on the next page. As the expression graph changes frequently in this new algorithm, instead of showing the literal hash tables we'll show the congruence classes they induce. To avoid confusion with virtual registers, value numbers will be written in brackets, like $\langle\mathtt{n}\rangle$. Then, a line like

$$\langle\mathtt{n}\rangle = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\} \qquad (\mathtt{expr})$$

means that `vregs>vns` has mappings from `x`, `y`, and `z` to `n`. Additionally, `exprs>vns` has a mapping from `expr` to `n` (taking the liberty to elide some Factor syntax from `expr`, such as the curly braces around array literals). Because the instruction is little more than the expression with an assignment destination, they serve no real point in understanding the value numbering process. So, `vns>insns` entries will be elided, as will any mappings to `f`, as they're understood to be implicit when a key is absent from the table.

`determine-value-numbers` starts the first iteration, which of course starts at basic block 1. `##inc-d` is a no-op, but the first two `##load-integer`s are established as useful instructions. `##load-integer 23 0` is recognized as redundant, since at this point we know that `21` has the value `0`. The `##copy` instructions all pile on value number mappings, leaving

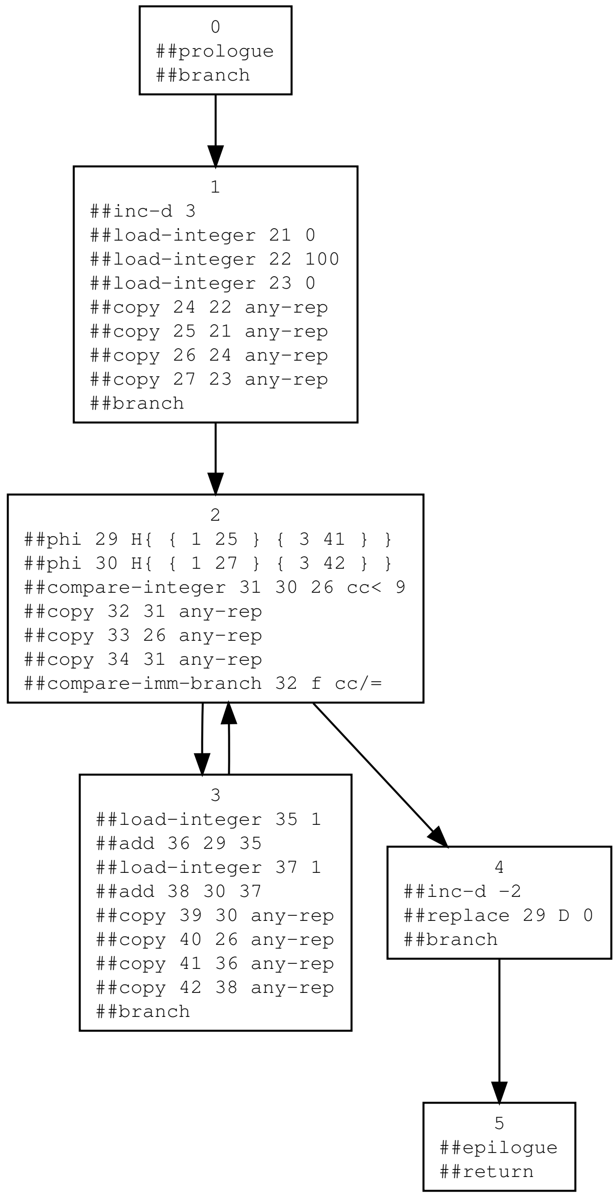

Figure 52: `0 100 [ 1 fixnum+fast ] times` before the new value numbering pass

us with the following classes:

$$\langle \mathtt{21} \rangle = \{\mathtt{21}, \mathtt{23}, \mathtt{25}, \mathtt{27}\} \quad (\mathtt{0})$$

$$\langle \mathtt{22} \rangle = \{\mathtt{22}, \mathtt{24}, \mathtt{26}\} \quad (\mathtt{100})$$

At iteration 1, basic block 2, the first `##phi` has inputs `25` (from block 1) and `41` (from block 3). The former has the value number $\langle \mathtt{21} \rangle$, while the latter is still at **`f`**. We treat this value number much like a $\top$ element, unifying it with the other input to give us the assumption that `29` will be a copy of `25`. Thus, it gets the same value number. A similar choice happens for the second `##phi`. The instruction `##compare-integer 31 30 26 cc< 9` is an interesting case. Due to our optimistic assumptions thus far, we believe `30` is carrying the value `0`, and that `26` is set to `100`. Thus, this instruction gets constant-folded by `simplify` into `##load-reference 31 t`. The CFG isn't changed, but the expression graph reflects this belief. Later, this assumption will be invalidated. The following copies are processed as usual, with the distinct difference here that `##copy 33 26 any-rep` has the global knowledge of the value number of `26`. Because the `##compare-integer` was constant-folded, so is the `##compare-imm-branch`—and to the same value, no less. This leaves us with:

$$\langle \mathtt{21} \rangle = \{\mathtt{21}, \mathtt{23}, \mathtt{25}, \mathtt{27}, \mathtt{29}, \mathtt{30}\} \quad (\mathtt{0})$$

$$\langle \mathtt{22} \rangle = \{\mathtt{22}, \mathtt{24}, \mathtt{26}, \mathtt{33}\} \quad (\mathtt{100})$$

$$\langle \mathtt{31} \rangle = \{\mathtt{31}, \mathtt{32}, \mathtt{34}\} \quad (\mathbf{t})$$

Block 3 of iteration 1 gives the `##load-integer`s' destinations the same value number, corresponding to the integer `1`. Because optimism makes the algorithm think that `29` and `30` correspond to the integer `0`, the `##add`s are constant-folded. This leaves us with:

$$\langle \mathtt{21} \rangle = \{\mathtt{21}, \mathtt{23}, \mathtt{25}, \mathtt{27}, \mathtt{29}, \mathtt{30}, \mathtt{39}\} \quad (\mathtt{0})$$

$$\langle \mathtt{22} \rangle = \{\mathtt{22}, \mathtt{24}, \mathtt{26}, \mathtt{33}, \mathtt{40}\} \quad (\mathtt{100})$$

$$\langle \mathtt{31} \rangle = \{\mathtt{31}, \mathtt{32}, \mathtt{34}\} \quad (\mathbf{t})$$

$$\langle \mathtt{35} \rangle = \{\mathtt{35}, \mathtt{36}, \mathtt{37}, \mathtt{38}, \mathtt{41}, \mathtt{42}\} \quad (\mathtt{1})$$

While block 4 is visited in each iteration, it doesn't define any registers, so doesn't affect the state of value numbering. Therefore, the above is the state left at the end of the first iteration.

Since `vregs>vns` clearly changed, iteration 2 commences by clearing the expressions (signified below by dashes), though the value numbers remain. Block 1 doesn't change from iteration 1, giving us:

$$\langle \texttt{21} \rangle = \{\texttt{21}, \texttt{23}, \texttt{25}, \texttt{27}, \texttt{29}, \texttt{30}, \texttt{39}\} \qquad (\texttt{0})$$

$$\langle \texttt{22} \rangle = \{\texttt{22}, \texttt{24}, \texttt{26}, \texttt{33}, \texttt{40}\} \qquad (\texttt{100})$$

$$\langle \texttt{31} \rangle = \{\texttt{31}, \texttt{32}, \texttt{34}\} \qquad (\text{—})$$

$$\langle \texttt{35} \rangle = \{\texttt{35}, \texttt{36}, \texttt{37}, \texttt{38}, \texttt{41}, \texttt{42}\} \qquad (\text{—})$$

Now that we're in iteration 2, the inputs to the `##phi`s of block 2 have been processed once before. For instance, in the first `##phi` we still believe that `25` corresponds to the integer `0` (which is incidentally correct), but now that `41` has the value number $\langle \texttt{35} \rangle$, we think it corresponds to the integer `1`. While this is incorrect, it does break the congruence between the inputs, making the first `##phi` a useful instruction. The second `##phi`, however, still looks like a copy of the first. Even so, this is sufficiently different that the following `##compare-integer` cannot be constant-folded like before. However, it can still be converted to a `##compare-integer-imm`, as one of its operands corresponds to an integer. The redundant `##compare-imm-branch` gets rewritten to the same expression as the `##compare-integer`, so winds up getting the same value number. This gives us:

$$\langle \texttt{21} \rangle = \{\texttt{21}, \texttt{23}, \texttt{25}, \texttt{27}, \texttt{39}\} \qquad (\texttt{0})$$

$$\langle \texttt{22} \rangle = \{\texttt{22}, \texttt{24}, \texttt{26}, \texttt{33}, \texttt{40}\} \qquad (\texttt{100})$$

$$\langle \texttt{29} \rangle = \{\texttt{29}, \texttt{30}\} \qquad (\texttt{\#\#phi 2 21 35})$$

$$\langle \texttt{31} \rangle = \{\texttt{31}, \texttt{32}, \texttt{34}\} \qquad (\texttt{\#\#compare-integer-imm 29 100 cc<})$$

$$\langle \texttt{35} \rangle = \{\texttt{35}, \texttt{36}, \texttt{37}, \texttt{38}, \texttt{41}, \texttt{42}\} \qquad (\text{—})$$

Block 3 of iteration 2 also changes, since the `##add`s can't be constant-folded like before due to our new discovery about the `##phi`s. However, the first one can still be converted to an ##add-imm, and the second is marked the same as the first. This leaves the following value numbers:

$$\langle \texttt{21} \rangle = \{\texttt{21}, \texttt{23}, \texttt{25}, \texttt{27}\} \qquad (\texttt{0})$$
$$\langle \texttt{22} \rangle = \{\texttt{22}, \texttt{24}, \texttt{26}, \texttt{33}, \texttt{40}\} \qquad (\texttt{100})$$
$$\langle \texttt{29} \rangle = \{\texttt{29}, \texttt{30}, \texttt{39}\} \qquad (\texttt{\#\#phi 2 21 35})$$
$$\langle \texttt{31} \rangle = \{\texttt{31}, \texttt{32}, \texttt{34}\} \qquad (\texttt{\#\#compare-integer-imm 29 100 cc<})$$
$$\langle \texttt{35} \rangle = \{\texttt{35}, \texttt{37}\} \qquad (\texttt{1})$$
$$\langle \texttt{36} \rangle = \{\texttt{36}, \texttt{38}, \texttt{41}, \texttt{42}\} \qquad (\texttt{\#\#add-imm 29 1})$$

Since the value numbers changed, we start iteration 3. The expressions are cleared, and block 1 once again does not change anything. The first ##phi in block 2 still gets classified as useful, so no value numbers change. The major difference, though, is that the previous iteration's value numbers for registers in block 3 update the expression we have for the ##phi. Whereas before we thought it was choosing between ⟨21⟩ (the integer 0) and ⟨35⟩ (the integer 1), the ##add wasn't constant-folded in the previous iteration. Therefore, the virtual register 41 now corresponds to the result of the ##add with the value number ⟨36⟩. We still can't disprove that the second ##phi is different (because it, in fact, isn't). So, we're left with the following after iteration 3 finishes with block 2:

$$\langle \texttt{21} \rangle = \{\texttt{21}, \texttt{23}, \texttt{25}, \texttt{27}\} \qquad (\texttt{0})$$
$$\langle \texttt{22} \rangle = \{\texttt{22}, \texttt{24}, \texttt{26}, \texttt{33}, \texttt{40}\} \qquad (\texttt{100})$$
$$\langle \texttt{29} \rangle = \{\texttt{29}, \texttt{30}, \texttt{39}\} \qquad (\texttt{\#\#phi 2 21 36})$$
$$\langle \texttt{31} \rangle = \{\texttt{31}, \texttt{32}, \texttt{34}\} \qquad (\texttt{\#\#compare-integer-imm 29 100 cc<})$$
$$\langle \texttt{35} \rangle = \{\texttt{35}, \texttt{37}\} \qquad (\text{—})$$
$$\langle \texttt{36} \rangle = \{\texttt{36}, \texttt{38}, \texttt{41}, \texttt{42}\} \qquad (\text{—})$$

Blocks 3 and 4 do not produce any more changes, so GVN has stabilized after 3 iterations, with our final congruence classes being:

$$\langle 21\rangle = \{21, 23, 25, 27\} \qquad \texttt{(0)}$$

$$\langle 22\rangle = \{22, 24, 26, 33, 40\} \qquad \texttt{(100)}$$

$$\langle 29\rangle = \{29, 30, 39\} \qquad \texttt{(\#\#phi 2 21 36)}$$

$$\langle 31\rangle = \{31, 32, 34\} \qquad \texttt{(\#\#compare-integer-imm 29 100 cc<)}$$

$$\langle 35\rangle = \{35, 37\} \qquad \texttt{(1)}$$

$$\langle 36\rangle = \{36, 38, 41, 42\} \qquad \texttt{(\#\#add-imm 29 1)}$$

4.3 Redundancy Elimination

Now that we've identified congruences across the entire CFG, we must eliminate any redundancies found. Since value numbering is now offline, this entails another pass. However, replacing instructions is more subtle with global value numbers than it is with local ones. Because values come from all over the CFG, we must consider if a definition is *available* at the point where we want to use it.

Figures 53 and 54 on pages 87–88 show the difference. In the former, we can see the CFG before value numbering for the code `[ 10 ] [ 20 ] if 10 20 30`. The two extra integers being pushed at the end don't really illustrate the point; they're just there to avoid branch splitting (see Section 3.3). In block 4, there's a `##load-integer 27 10`, which loads the value `10`. In globally numbering values, we associate the `##load-integer 22 10` in block 2 with the value `10` first, making it the canonical representative. However, we can't replace the instruction in block 4 with `##copy 27 22`, because control flow doesn't necessarily go through block 2, so the virtual register `22` might not even be defined. However, in Figure 54 on page 88, we see the CFG for the code `10 swap [ 10 ] [ 20 ] if 10 20 30`. In this case, the first definition of the value `10` comes from block 1, which dominates block 4. So, the definition is available, and we can replace the `##load-integer` in block 4 with a `##copy`.

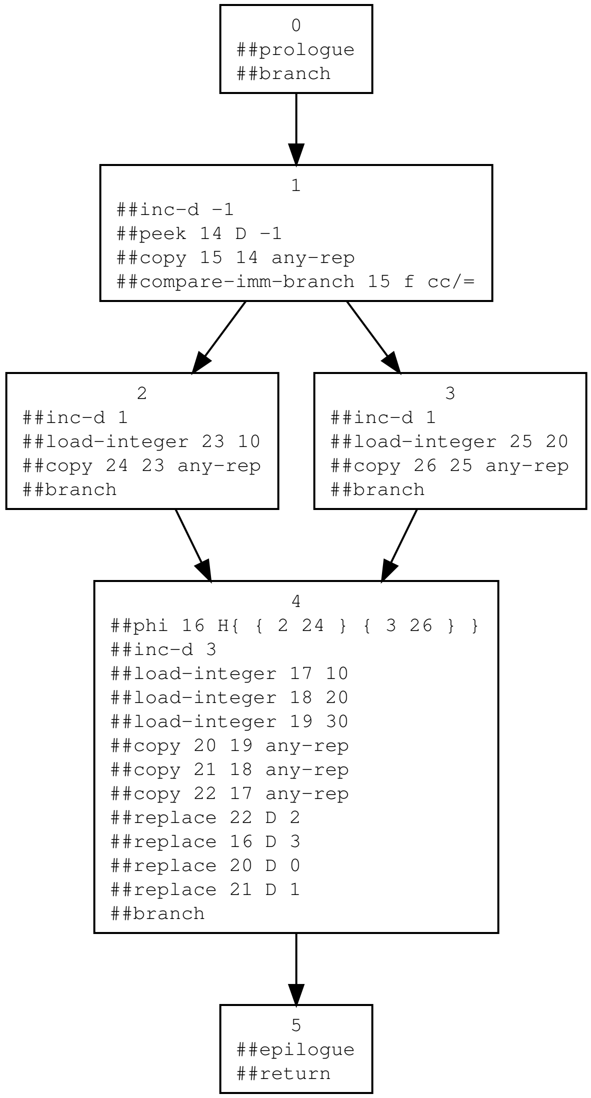

Figure 53: `10` is not available in block 4

There are several ways to decide if we can use a definition at a certain point. For instance, we could use dominator information, so that a definition in a basic block B can be used by any other block dominated by B [Simpson 1996]. However, here we'll use a data flow analysis called *available expression analysis*, since it is readily implemented. Mercifully, Factor has a vocabulary that automatically defines data flow analyses with little more than a single line of code.

Figure 55 on page 90 shows the vocabulary that defines the available expression analysis.

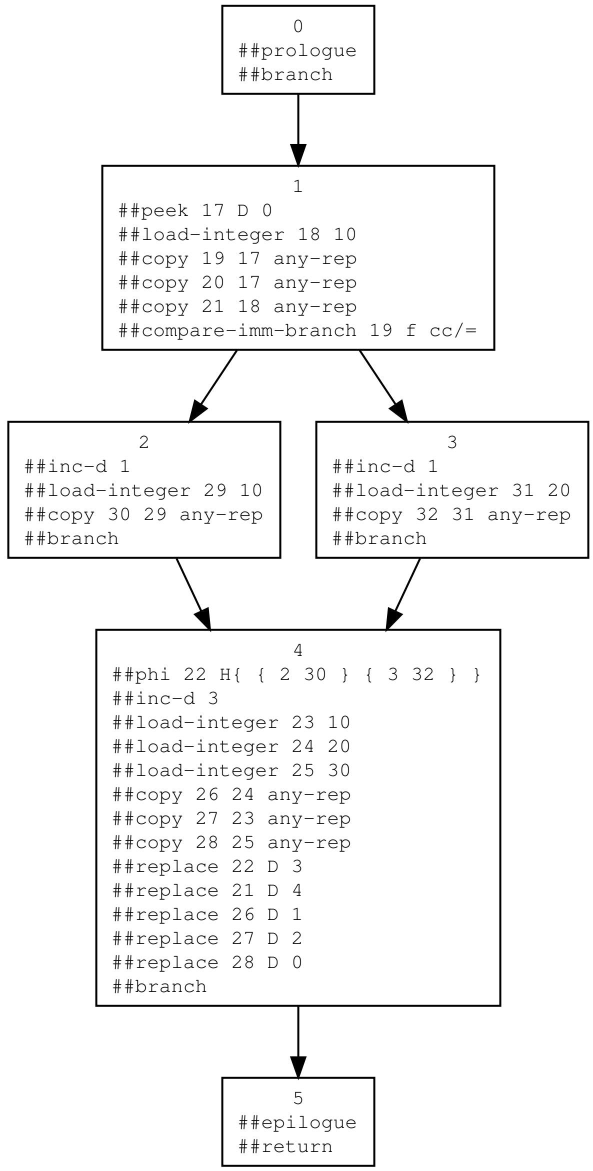

Figure 54: `10` is available in block 4

It is a forward analysis [see Aho et al. 2007] based on the flow equations below:

$$\texttt{avail-in}_i = \begin{cases} \varnothing & \text{if } i = 0 \\ \bigcap_{j \in \text{pred}(i)} \texttt{avail-out}_j & \text{if } i > 0 \end{cases}$$

$$\texttt{avail-out}_i = \texttt{avail-in}_i \cup \texttt{defined}_i$$

Here, the subscripts indicate the basic block number. $\texttt{defined}_i$ denotes the result of the `defined` word from Figure 55 on the following page. This returns the set of virtual registers defined in a basic block. Since we use virtual registers as value numbers, this is the same as giving us all the value numbers produced by a basic block. "Killed" definitions are impossible by the SSA property, so we needn't track redefinitions of virtual registers, as in other data flow analyses. Using set intersection as the confluence operator means that the $\texttt{avail-in}_i$ set will contain those values which are available on all paths from the start of the CFG to block i.

Using Factor's `compiler.cfg.dataflow-analysis` vocabulary, the implementation takes all of two lines of code. The `FORWARD-ANALYSIS: avail` line automatically defines several objects, variables, words, and methods that don't warrant full detail here. One we're immediately concerned with is the `transfer-set` generic, which dispatches upon the particular type of analysis being performed and is invoked on the proper in-set and basic block. There is no default implementation, as it is the chief difference between analyses. So, the next line uses `defined` and **`assoc-union`** to calculate the result of the data flow equation. Other pieces we'll see used are the top-level `compute-avail-sets` word that actually performs the analysis, the `avail-ins` hash table that maps basic blocks to their in-sets, and the `avail-in` word that is shorthand for looking up a basic block's in-set.

We want to use the results of this analysis in the `rewrite` methods so that they will only perform correct and meaningful rewrites. However, we also want to use `rewrite` in the `determine-value-numbers` pass, where we don't care about availability. In fact, we want to ignore availability altogether in that pass, so that we can discover as many congruences as possible. In order to separate these concerns, we need to have two modes for checking availability. Figure 55 on the next page defines the `available?` word to do just this. It will

```
! Copyright (C) 2011 Alex Vondrak.
! See http://factorcode.org/license.txt for BSD license.
USING: accessors assocs hashtables kernel namespaces sequences
sets
compiler.cfg
compiler.cfg.dataflow-analysis
compiler.cfg.def-use
compiler.cfg.gvn.graph
compiler.cfg.predecessors
compiler.cfg.rpo ;
FROM: namespaces => set ;
IN: compiler.cfg.gvn.avail

: defined ( bb -- vregs )
    instructions>> [ defs-vregs ] map concat unique ;

FORWARD-ANALYSIS: avail

M: avail-analysis transfer-set drop defined assoc-union ;

: available? ( vn -- ? )
    final-iteration? get [
        basic-block get avail-in key?
    ] [ drop t ] if ;

: available-uses? ( insn -- ? )
    uses-vregs [ available? ] all? ;

: with-available-uses? ( quot -- ? )
    keep swap [ available-uses? ] [ drop f ] if ; inline

: make-available ( vreg -- )
    basic-block get avail-ins get [ dupd clone ?set-at ] change-at ;
```

Figure 55: The `compiler.cfg.gvn.avail` vocabulary

```
! Before
: fold-branch ( ? -- insn )
    0 1 ?
    basic-block get [ nth 1vector ] change-successors drop
    \ ##branch new-insn ;

! After
: fold-branch ( ? -- insn )
    final-iteration? get [
        0 1 ?
        basic-block get [ nth 1vector ] change-successors drop
    ] [ drop ] if
    \ ##branch new-insn ;
```

Figure 56: Branch folding before and after

only check the actual availability if `final-iteration?` is true, otherwise defaulting to **`t`**. Therefore, during the value numbering phase, everything is considered available. We further define the utilities `available-uses?` and `with-available-uses?`. The former checks if all an instruction's uses are available, and the latter does this only if another quotation first returns a true value. That way, we can guard instruction predicates with a test for available uses, like `[ ##add-imm? ] with-available-uses?`.

Finding all the instances where `rewrite` needed to be altered was subtle. Since the old `value-numbering` was an online optimization, it didn't need to worry about modifying an instruction in memory. But by doing the fixed-point iteration, we cannot permit `rewrite` to destructively modify any object until the final iteration. An obvious instance was in `compiler.cfg.value-numbering.comparisons` with the word `fold-branch`, responsible for modifying the CFG to remove an untaken branch. We definitely would not want the branch removed while doing the fixed-point iteration, because the transformation is not necessarily sound. So, we can protect it with a check for `final-iteration?`, as in Figure 56.

More typical instances of the problems that occurred were in words like `self-inverse` from `compiler.cfg.value-numbering.math` (refer to Figure 57 on the next page). The idea is essentially to change

```
: self-inverse ( insn -- insn' )
    [ dst>> ] [ src>> vreg>insn src>> ] bi <copy> ;

! Before
M: ##neg rewrite
    {
        { [ dup src>> vreg>insn ##neg? ] [ self-inverse ] }
        { [ dup unary-constant-fold? ] [ unary-constant-fold ] }
        [ drop f ]
    } cond ;

! After
: self-inverse? ( insn quot -- ? )
    [ src>> vreg>insn ] dip with-available-uses? ; inline

M: ##neg rewrite
    {
        { [ dup [ ##neg? ] self-inverse? ] [ self-inverse ] }
        { [ dup unary-constant-fold? ] [ unary-constant-fold ] }
        [ drop f ]
    } cond ;
```

Figure 57: Rewriting words that are their own inverses

```
##neg 1 2
##neg 3 1
```

into

```
##neg 1 2
##copy 3 2 any-rep
```

since `##neg` undoes itself. But `rewrite` only has knowledge of one instruction at a time, so it looks up the redundant `##neg`'s source register in the `vns>insns` table to see if it's computed by another `##neg` instruction. For straight-line code this is alright, but the input to the originating `##neg` (in the example, the virtual register `2`) isn't necessarily available. So, we have to use `with-available-uses?` to make sure the virtual registers used by the result of a `vreg>insn` can themselves be utilized before we rewrite anything.

An even subtler issue that led to infinite loops occured in simplifcations like the arithmetic distribution in `compiler.cfg.value-numbering.math`. The problem is that the `rewrite` method would generate instructions that assigned to entirely brand new registers. These, of course, would invariably get value numbered, triggering a change in the `vregs>vns` table. A new iteration would begin, and (since it gets called on the same instructions as the previous iteration) `rewrite` would generate new virtual registers all over again. Therefore, the `vregs>vns` table would never stop changing. As a stop-gap, distribution had to be disabled altogether until the final iteration.

Armed with the correct rewrite rules, availability information, and global value numbers, we can perform global common subexpression elimination (GCSE). The logic in the `gcse` generic in Figure 58 on the following page is similar to `process-instruction` from Figure 44 on page 61 and `value-number` from Figure 50 on page 79. Unlike `value-number`, we do return an instruction (or sequence thereof) representing the replacement. Thus, the `array` method of `gcse` uses **`map`** instead of **`each`**, to hold onto the resulting sequence when recursing into several instructions.

`defs-available` is similar to `record-defs` from Figure 50 on page 79, except that value numbers have already stabilized, so we don't call `set-vn`. Instead, we use the `make-available` word, which was the last one defined in Figure 55 on page 90. We have to ensure that after processing an instruction, any register it defines is available to future instructions in the same block, thus enabling rewrites. So, we add the virtual register to that block's `avail-in` (which acts like a set, even though it's implemented by a hash table by Factor's data flow analysis framework). `alien-call-insns`, `##callback-inputs` instructions, and instances of `##copy` don't get rewritten any further, so we simply note that their definitions are available and move on.

The `?eliminate` word transforms an instruction into a `##copy` of the canonical value number that computes it. If the value number isn't available, we don't do anything but post-process with `defs-available`. If it is, a `##copy` is produced and its destination is made available. Thus, `eliminate-redundancy` works like `check-redundancy` from Figure 50 on page 79. We look up the expression computed by the instruction in the `exprs>vns` table. If it's there, we call `?eliminate`, but otherwise we leave the instruction alone and make its definitions available.

```
GENERIC: gcse ( insn -- insn' )

M: array gcse [ gcse ] map ;

: defs-available ( insn -- insn )
    dup defs-vregs [ make-available ] each ;

M: alien-call-insn gcse defs-available ;
M: ##callback-inputs gcse defs-available ;
M: ##copy gcse defs-available ;

: ?eliminate ( insn vn -- insn' )
    dup available? [
        [ dst>> dup make-available ] dip <copy>
    ] [ drop defs-available ] if ;

: eliminate-redundancy ( insn -- insn' )
    dup >expr exprs>vns get at
    [ ?eliminate ] [ defs-available ] if* ;

M: ##phi gcse
    dup inputs>> values [ vreg>vn ] map sift
    dup all-equal? [
        [ first ?eliminate ] unless-empty
    ] [ drop eliminate-redundancy ] if ;

M: insn gcse
    dup defs-vregs length 1 = [ eliminate-redundancy ] when ;

: gcse-step ( insns -- insns' )
    [ simplify gcse ] map flatten ;

: eliminate-common-subexpressions ( cfg -- )
    final-iteration? on
    dup compute-avail-sets
    [ gcse-step ] simple-optimization ;
```

Figure 58: Global common subexpression elimination in `compiler.cfg.gvn`

```
: value-numbering ( cfg -- cfg )
    needs-predecessors
    dup determine-value-numbers
    dup eliminate-common-subexpressions

    cfg-changed predecessors-changed ;
```

Figure 59: New global `value-numbering` word in `compiler.cfg.gvn`

The rest of the logic mirrors that of `value-number`. If the inputs to a `##phi` are all congruent, we'll call `?eliminate` to transform it into a `##copy` of its first input (without loss of generality). Otherwise, we check for equivalent `##phi`s with `eliminate-redundancy`. Finally, the `insn` method will default to calling `eliminate-redundancy` on instructions that define only one value, much like how `value-number` worked.

The main word that performs the pass is `eliminate-common-subexpressions`. `final-iteration?` is turned on (set to **`t`**), and we make sure to compute the `avail-in` sets needed to make `available?` work. Then, using `simple-optimization`, we iterate over each basic block. For each instruction, we first use `simplify` (refer to Figure 49 on page 78), then call `gcse` on the rewritten instruction. Thus, `rewrite` does the work of simplifying instructions, then `gcse` cleans up redundant ones by converting them into `##copy` instructions if possible. The new `value-numbering` word can be seen in Figure 59.

Consider for the last time the example `0 100 [ 1 fixnum+fast ]` **`times`**. Again, we have the CFG in Figure 60 on the following page. Making a final pass with `eliminate-common-subexpressions` gives us the CFG in Figure 61 on page 97. Compared to the CFG after the old `value-numbering` word was called (see Figure 38 on page 52), we have identified several more redundancies:

- The second `##phi` in block 2 has been turned into a `##copy` of the first.
- The `##compare-integer` of block 2 has been simplified to a `##compare-integer-imm`, since its operand `26` is both available and known to correspond to the integer value `100`.

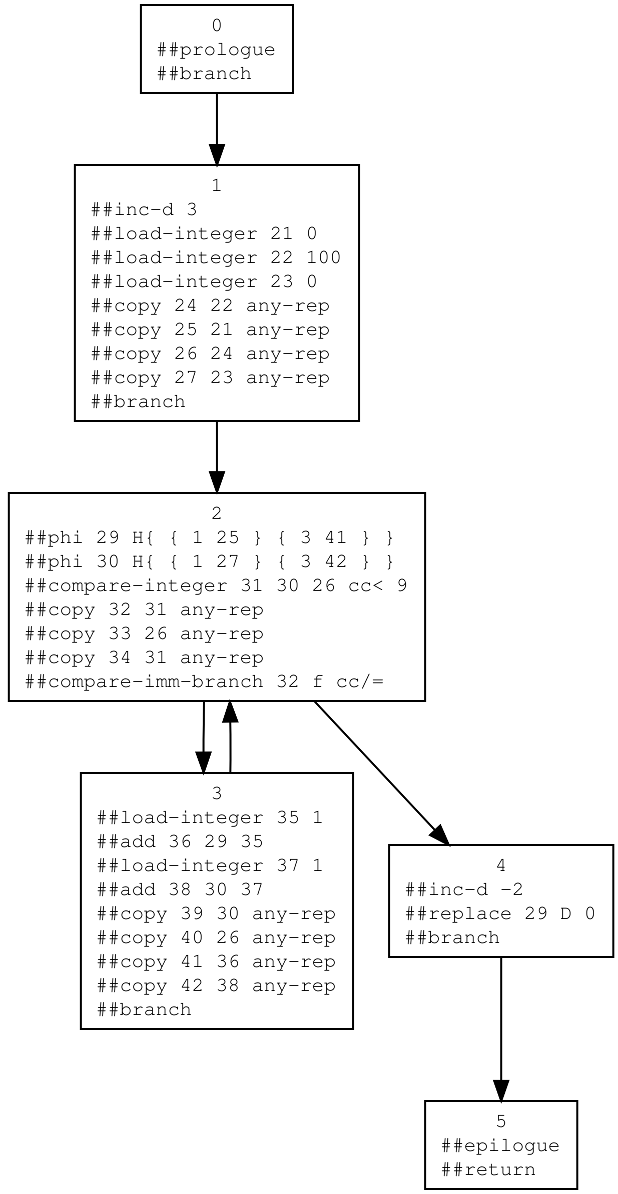

Figure 60: `0 100 [ 1 fixnum+fast ] times` before the new `value-numbering`

- Similarly, we've managed to convert the `##compare-integer-branch` at the end of block 2 into a `##compare-integer-imm-branch`.
- Because the `##phi`s have been recognized as copies (i.e., the induction variables are congruent), the second `##add` in block 3 is turned into a `##copy` of the first (which itself is still turned into an `##add-imm`).

Afterwards, the `copy-propagation` pass cleans up all of these newly identified copies,

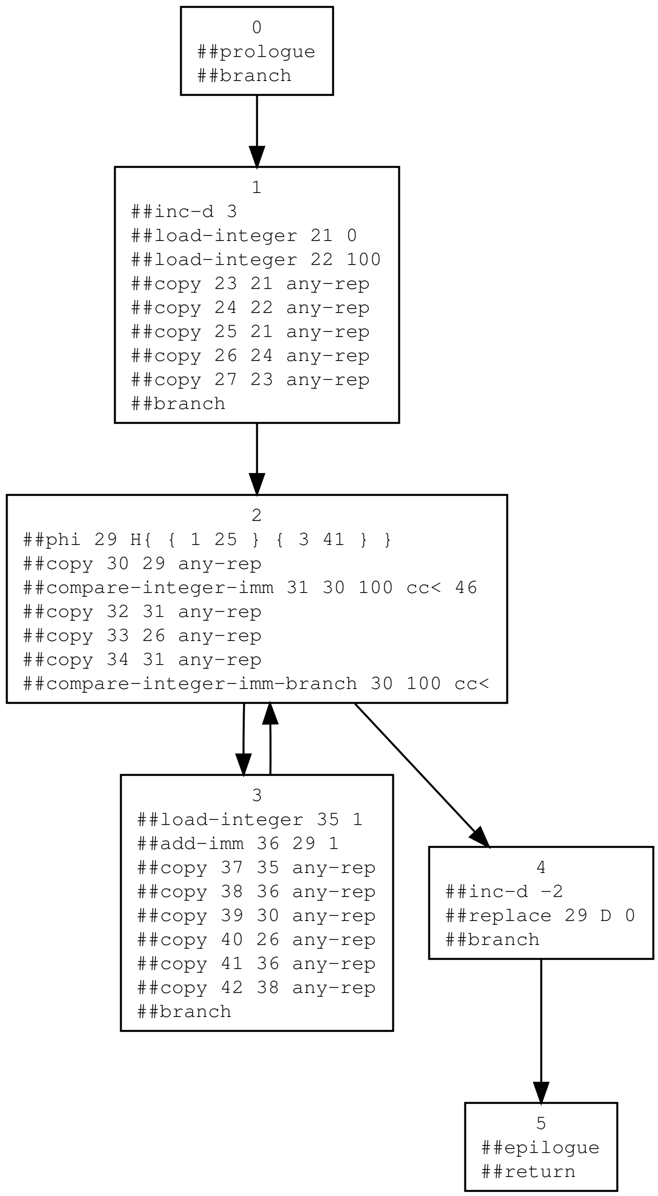

Figure 61: `0 100 [ 1 fixnum+fast ] times` after the new `value-numbering`

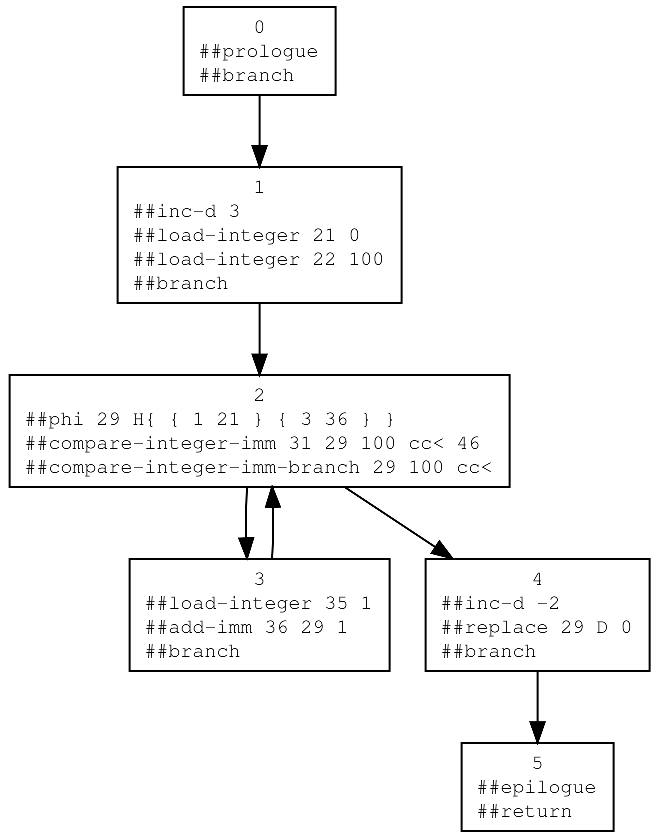

Figure 62: `0 100 [ 1 fixnum+fast ] times` after `copy-propagation`

as seen in Figure 62. `eliminate-dead-code` now gets rid of more instructions than before, such as the second `##load-integer` in block 1, since it has been propagated to the `-imm` instructions in block 2. See Figure 63 on the following page. At last, after `finalize-cfg` in Figure 64 on page 100, we see a loop that uses a single register—down from the three in Figure 41 on page 55.

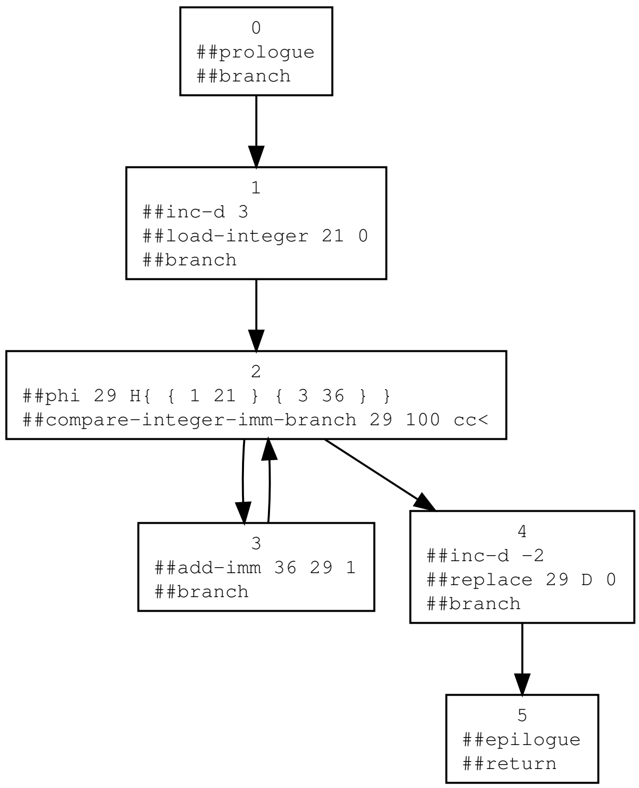

Figure 63: `0 100 [ 1 fixnum+fast ] times` after `eliminate-dead-code`

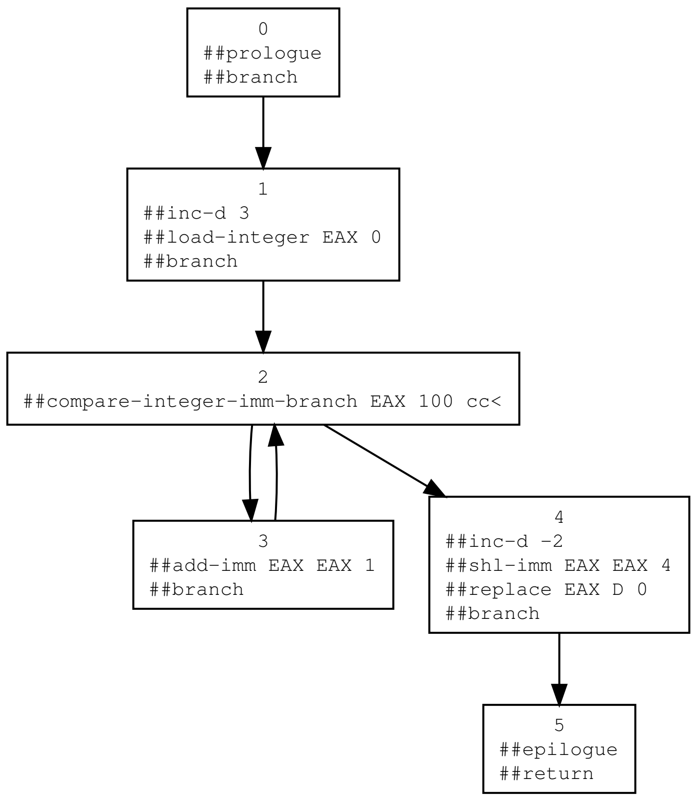

Figure 64: `0 100 [ 1 fixnum+fast ] times` after `finalize-cfg`

4.4 Results

The goal of improving the optimization in Factor is, of course, to reduce the average running time of programs, and to do so without changing their semantics. Short of formal verification, the latter requirement makes it necessary to thoroughly test any code that gets compiled with the new pass enabled. To this end, we'll employ Factor's extensive unit test coverage. While some vocabularies will have more tests than others, the total number of unit tests is quite large. By compiling each vocabulary and running their tests, we're indirectly testing the compiler: if tests that used to pass no longer do, then the new pass is changing the semantics of the code somehow. Though passing all tests does not guarantee the algorithm is correct, it does let us know that no known regressions have been introduced. Happily, with the new `value-numbering` phase enabled, all the same tests pass as before in a call to `test-all` from a freshly bootstrapped image.

The efficacy of the changes, on the other hand, must be measured relative to old benchmarks. Again, Factor has its bases covered, with a suite of 80 benchmarks run by the `benchmark` vocabulary. Each benchmark is run 5 times, and the garbage collector is run before each iteration. The minimum time from these runs is then used as the benchmark result. The informal data below comes from two separate runs on my own personal computer of the `benchmarks` word, which invokes all the `benchmark` sub-vocabularies. The "before" time used the local value numbering, while "after" times had `value-numbering` replaced with the GVN pass. The "change" is measured by the formula

$$\frac{\text{before} - \text{after}}{\text{before}} \times 100$$

to indicate the relative running times. Negative values in this column are good, as that means the running time has decreased.

Benchmark	Before (seconds)	After (seconds)	Change (%)
`3d-matrix-scalar`	3.705816738	3.046126696	−17.80
`3d-matrix-vector`	0.161298778	0.089539887	−44.49
`backtrack`	4.280001561	2.358672591	−44.89

Benchmark	Before (seconds)	After (seconds)	Change (%)
`base64`	5.127831493	2.853612485	−44.35
`beust1`	7.531546384	4.604929188	−38.86
`beust2`	20.308680548	12.843534349	−36.76
`binary-search`	3.729776895	2.349520427	−37.01
`binary-trees`	9.403166818	6.518867479	−30.67
`bootstrap1`	32.472196349	30.887877896	−4.88
`chameneos-redux`	2.923900422	2.041007328	−30.20
`continuations`	0.273525202	0.200695972	−26.63
`crc32`	0.010623653	0.005282642	−50.27
`dawes`	1.588111926	1.027176578	−35.32
`dispatch1`	7.640720326	5.106558985	−33.17
`dispatch2`	5.221652668	3.984754032	−23.69
`dispatch3`	9.710520454	6.203527737	−36.12
`dispatch4`	8.224931156	4.098265543	−50.17
`dispatch5`	4.74357434	3.478219608	−26.68
`e-decimals`	3.903754723	2.646958072	−32.19
`e-ratios`	4.774454589	3.658075473	−23.38
`empty-loop-0`	0.251816164	0.199189271	−20.90
`empty-loop-1`	1.039242509	0.857588545	−17.48
`empty-loop-2`	0.472215346	0.387974286	−17.84
`euler150`	37.785852299	27.05450689	−28.40
`fannkuch`	9.627490235	6.8970571	−28.36
`fasta`	7.25292282	5.640517069	−22.23
`fib1`	0.179389215	0.164933805	−8.06
`fib2`	0.205853157	0.138174211	−32.88
`fib3`	0.785036151	0.539739186	−31.25
`fib4`	0.391805799	0.260370111	−33.55
`fib5`	1.508625224	1.002724851	−33.53

Benchmark	Before (seconds)	After (seconds)	Change (%)
`fib6`	19.202504502	13.146010511	−31.54
`gc0`	7.360087104	5.508594031	−25.16
`gc1`	0.418173431	0.281497214	−32.68
`gc2`	25.611210221	19.716168704	−23.02
`gc3`	2.757943071	2.210785891	−19.84
`hashtables`	8.068216942	7.997106348	−0.88
`heaps`	4.360368411	4.32169158	−0.89
`iteration`	7.875561986	6.277891729	−20.29
`javascript`	17.881224721	12.74204052	−28.74
`knucleotide`	5.490420772	3.5704101	−34.97
`mandel`	0.251711276	0.198695557	−21.06
`matrix-exponential-scalar`	16.451432774	12.017000042	−26.95
`matrix-exponential-simd`	0.681684747	0.536850343	−21.25
`md5`	10.40516678	9.198666403	−11.60
`mt`	33.91981743	29.961085146	−11.67
`nbody`	9.203478441	6.795154145	−26.17
`nbody-simd`	0.845814208	0.854773096	+1.06
`nested-empty-loop-1`	0.097090973	0.068475608	−29.47
`nested-empty-loop-2`	0.893126911	0.861327078	−3.56
`nsieve`	1.086110659	1.137648699	+4.75
`nsieve-bits`	2.707271763	2.815509077	+4.00
`nsieve-bytes`	0.785041878	1.211421146	+54.31
`partial-sums`	3.762171661	4.130144177	+9.78
`pidigits`	2.182877913	2.195385034	+0.57
`random`	5.66540782	5.71913683	+0.95
`raytracer`	5.047070171	4.39514879	−12.92
`raytracer-simd`	1.072588515	0.980927338	−8.55
`recursive`	2.703509403	2.529087637	−6.45

Benchmark	Before (seconds)	After (seconds)	Change (%)
`regex-dna`	2.208584014	1.808859571	−18.10
`reverse-complement`	2.801163847	2.353254665	−15.99
`ring`	1.822206473	1.62482491	−10.83
`sfmt`	2.675838657	2.463367198	−7.94
`sha1`	11.964973943	11.142380303	−6.88
`simd-1`	1.857778672	1.703206011	−8.32
`sockets`	10.636346636	10.516448454	−1.13
`sort`	0.695635429	0.581855635	−16.36
`spectral-norm`	3.433630383	2.960833789	−13.77
`spectral-norm-simd`	2.743240011	3.237017655	+18.00
`stack`	1.580016742	2.004478602	+26.86
`struct-arrays`	2.180774222	2.421915609	+11.06
`sum-file`	0.883097981	0.957151577	+8.39
`terrain-generation`	1.611800222	1.887047663	+17.08
`tuple-arrays`	0.262747557	0.329399609	+25.37
`typecheck1`	1.750223408	1.674592158	−4.32
`typecheck2`	1.674738245	1.553203741	−7.26
`typecheck3`	1.891206648	1.735390184	−8.24
`ui-panes`	0.305595039	0.29985214	−1.88
`xml`	3.013709363	2.722223892	−9.67
`yuv-to-rgb`	0.398174487	0.318891664	−19.91

These informal results are promising: the mean speedup was −16.35% (median −18.97%), and of 80 benchmarks, only 13 showed any increase in running time. The mean speedup among those that ran faster was −22.24% (median −22.23%). Of the 13 that ran slower, even fewer showed significant increases in running time. Duplicated below for convenience are the slower benchmarks, sorted in decreasing order of the percent change. We can see the last five or six benchmarks exhibited negligible differences—not only is the relative change tiny, but the absolute difference in running times is less than 0.1 seconds. (The `tuple-arrays` results also show a similar absolute change, but it is relatively much larger.)

Benchmark	Before (seconds)	After (seconds)	Change (%)
`nsieve-bytes`	0.785041878	1.211421146	+54.31
`stack`	1.580016742	2.004478602	+26.86
`tuple-arrays`	0.262747557	0.329399609	+25.37
`spectral-norm-simd`	2.743240011	3.237017655	+18.00
`terrain-generation`	1.611800222	1.887047663	+17.08
`struct-arrays`	2.180774222	2.421915609	+11.06
`partial-sums`	3.762171661	4.130144177	+9.78
`sum-file`	0.883097981	0.957151577	+8.39
`nsieve`	1.086110659	1.137648699	+4.75
`nsieve-bits`	2.707271763	2.815509077	+4.00
`nbody-simd`	0.845814208	0.854773096	+1.06
`random`	5.66540782	5.71913683	+0.95
`pidigits`	2.182877913	2.195385034	+0.57

Overall, even transitioning to a relatively simple GVN algorithm amounts to a positive change in Factor's compiler. More redundancies are eliminated, resulting in speedier programs. Judging by unit tests, the implementation is at least as sound as the previous local value numbering, as all the same tests have passed.

4.5 Future Work

The GVN code presented in this thesis can be improved in various specific ways. Furthermore, the literature on GVN is extensive, and there are more overarching algorithmic strategies that have yet to be explored in the Factor code base. In this section, we review some of these options for possible directions that Factor's compiler can take from here.

As it stands, the new pass could be smarter. For instance, it does not consider the commutativity of certain operations. This would be straightforward to solve by making `>expr` sort the operands of commutative instructions' expressions, thereby placing arguments in a canonical order. This would increase the number of congruences discovered between `##add`s, `##mul`s, and even `##phi`s. Also, the `copy-propagation` pass is remarkably similar to the new `value-numbering`—in fact, it could be removed altogether. All it does is collect global information about congruences as established by `##copy` instructions (by a similar fixed-point iteration), then replace the virtual registers of instructions with the original value (i.e., the one not established by a `##copy`). This allows `copy-propagation` to remove all `##copy` instructions. But the information calculated by `value-numbering` is a superset of this copy-equivalence data, so it should be easy to do global copy propagation in the GVN phase and save compilation time on the redundant fixed-point iteration.

There remains an open question about the GVN implementation's use of availability, too. As it stands, it's rather strict: if the canonical value number for an expression is not directly available, `rewrite` gives up on reusing that value. However, we need not look just at the canonical leader (the first virtual register in the whole program to compute the particular expression). `rewrite` could change an instruction to reuse any available member of the congruence class. It remains to be seen when and if such rewriting would be useful or desirable.

Existing literature also gives plenty of material for a better implementation. We can make the existing RPO algorithm more efficient in practice by observing that the only times we need to iterate are where there are cyclic dependencies between values in the CFG. For instance, the example from Section 4.3 only has cyclic dependencies in the induction variables: the `##phi`s are defined by uses of virtual registers that are themselves defined by uses of the `##phi` targets. The RPO algorithm degenerates into the hash-based local algorithm of Section 4.1 on straight-line code. Thus, a more efficient algorithm will only iterate over the cycles between definitions instead of over the whole CFG.

Conceptually, we build a *value graph* (also known as *SSA graph* [Simpson 1996]) whose nodes represent definitions and directed edges represent uses. Since it just codifies def-use information, we needn't build an actual graph data structure. Using an algorithm due to Tarjan [1972], each strongly connected component (SCC) of the value graph is iterated upon, while single nodes are processed only once. The SCC algorithm is more efficient than the RPO algorithm in practice, but the principles are the same. This gives us a comparatively simple, easily-extended GVN algorithm with complexity $O(nd)$, where n is the number of vertices in the value graph (i.e., the number of values we're numbering) and d is the *loop connectedness* (the maximum number of back edges on any acyclic path) of the value graph. Though d can theoretically be $O(n)$, in practice it seems to be bounded by a small constant. In Simpson's experiments, the maximum value of d was 4.

A more thorough overhaul could incorporate further rewriting of the instructions. Gargi [2002] proposes a *predicated* value numbering algorithm that combines

- optimistic value numbering, thus emulating Simpson's RPO and SCC algorithms;
- constant folding, algebraic simplification, and unreachable code elimination, thus emulating Click's strongest combination [Click 1995];
- global reassociation, thus performing the work already done in Factor;
- predicate inference, which can infer the values of comparisons dominated by some related conditional;
- value inference, which can infer the values of variables dominated by some related predicate (similar to range propagation, as seen in Section 3.2); and
- ϕ-predication, which (if possible) associates each input of a ϕ with the predicate that controls the path that leads to the selection of that argument, thus letting us find flow-sensitive congruences.

This combination is given in a *sparse* formulation, which makes it efficient enough to apply all of these optimizations. Essentially, when optimistic assumptions are invalidated (which, of course, happens as we iterate until reaching the fixed point), instead of recalculating every result (as in the RPO algorithm), we only recalculate the values that may yet change from this new information (as in the SCC algorithm).

Any portion of Gargi's algorithm may be selectively disabled, thus letting us tweak it for specific compile time vs. code quality trade-offs we might have. It promotes a fairly good separation of concerns in the algorithm, too, letting the pseudocode be presented piecemeal for each optimization. Gargi presents compelling examples of predicated value numbering's strength, and its addition to Factor could prove very worthwhile.

Historically, the development of GVN algorithms has forked along two concurrent bloodlines. Those we've studied thus far reflect the more "practical" line, which was largely implementation-driven and less formal than the other algorithms. But those focused on formal reasoning have recently become much more viable, and the wealth of ideas from them are worthwhile.

For those acquainted with chapter 9 of Aho et al. 2007, (The Dragon Book), this work will seem familiar, as it's rooted in the results of Kildall [1973] and Cousot and Cousot [1977], upon which the chapter is based. The former was a precursor to GVN, in that it described an algorithm for common subexpression elimination that partitioned expressions into congruence classes. However, its method was phrased in terms of *lattices*, which are algebraic structures that we can reason about formally. This is as in The Dragon Book: a lattice is a partially-ordered set for which any two elements have a unique *least upper bound* (or *join*) and *greatest lower bound* (or *meet*). By defining meet and join operators on a partially-ordered set of abstract values, we can represent many sorts of analyses of our programs.

Cousot and Cousot [1977] formalize the salient properties of such interpretation over lattices in a framework dubbed *abstract interpretation.* To understand it intuitively, consider some arithmetic expression like (-5×14). Our first inclination is probably to interpret it with respect to numeric values, but we can understand it in several different contexts. Let's use signs ($+$, $-$, and $\pm$) as our abstract domain and consider the operators to be defined by the rules of signs. Figure 65 on the next page shows a lattice we can use for this. Using a version of $\times$ cast in the context of this lattice, we can interpret (-5×14) as

$$-5 \times 14 \quad \rightarrow \quad (-) \times (+) \quad \rightarrow \quad (-)$$

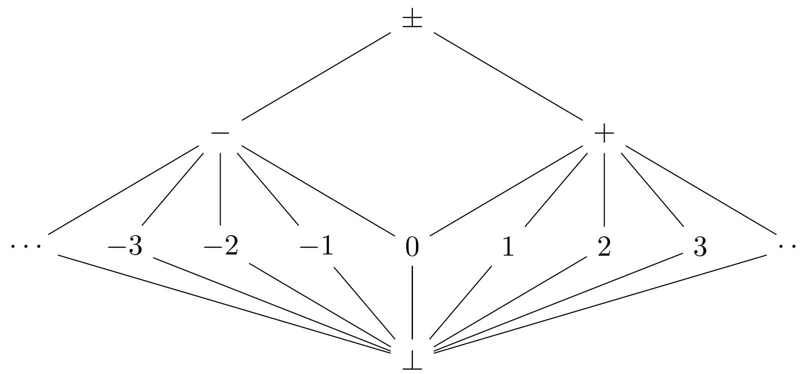

Figure 65: Abstract interpretation over signs

proving that (-5×14) is negative. Using this framework, the results are correct, but only useful within the confines of what we define. For instance, we can interpret $(-5 + 14)$ as

$$-5 + 14 \quad \rightarrow \quad (-) + (+) \quad \rightarrow \quad (\pm)$$

proving very little—the result is either positive or negative.

Despite the inherent limitations, we find the results useful as approximations of more complex properties. For example, we used congruence to approximate runtime equivalence. Only a year before AWZ was published, Steffen [1987] showed that Kildall's approach could be framed as abstract interpretation over *Herbrand equivalences*—that is, equivalences where operators are uninterpreted. This is actually the same notion of congruence we had from before: expressions are equivalent if their operators and operands are equivalent, irrespective of the result of applying the operator.

The primary strength of the abstract interpretation approaches are that they are *complete*; intuitively, there is no loss of information at each step of abstract interpretation. However, this "loss of information" is relative to the information encompassed by the abstract domains [Giacobazzi, Ranzato, and Scozzari 2000]. While we can find all Herbrand equivalences, we aren't guaranteed to find equivalences induced by interpreting operators, which was effectively the work done by combining optimizations (e.g., constant folding is the interpretation of certain operators upon constant operands). So, while complete, these approaches vary in *preciseness*. Most of the work in the abstract interpretation of GVN did little to study the results of interpreting the same operators we saw before, but note it's a promising direction for future research.

The cost of this completeness has traditionally been exponential time complexities. There have been several attempts to remedy this. Rüthing, Knoop, and Steffen [1999] note AWZ is incomplete, since it treats ϕ functions as uninterpreted, so fails to discover congruences between ϕs and ordinary expressions. Their attempt to improve upon it alternately applies AWZ and the normalization rules

$$\begin{aligned} \phi(a \otimes b, c \otimes d) &\quad\to\quad \phi(a, c) \otimes \phi(b, d) \\ \phi(x, x) &\quad\to\quad x \end{aligned}$$

until the partitioning reaches a fixed point. However, this is $O(n^2 \log n)$ in the expected case—$O(n^4 \log n)$ in the worst case—and it turned out to be incomplete not just in the presence of cycles [Rüthing, Knoop, and Steffen 1999] but also in certain acyclic code [Gulwani and Necula 2007].

Later, Gulwani and Necula [2004] furthered the quest for an efficient, complete GVN algorithm in a novel way by using *randomized interpretation* (which is what it sounds like). The paper even explored various interpretations—specifically of linear combinations, bitwise operators, memory loads/stores, and integer division—that could make results more precise. But it was still $O(n^4)$ and ran a small chance of making incorrect inferences due to its randomized nature. For compilers, this isn't really acceptable, though such a scheme could be used in things like program verification tools [Nie and Cheng 2007].

From their trip back to the drawing board, Gulwani and Necula returned with a polynomial time algorithm for GVN that is complete for all Herbrand equivalences among terms of a limited size [Gulwani and Necula 2007]. Choosing a size bound equal to the size of the entire program is clearly sufficient. Note, however, that this is specifically for Herbrand equivalences; they do not show their results for any interpreted operators, but note it's an important area for exploration. Adding to this, Nie and Cheng [2007] present a similar algorithm, except based on SSA form. Both wind up using the same size restrictions to guarantee complexity. Both also use an additional special-purpose data structure to represent the set of Herbrand equivalences and to perform abstract evaluations over them, which adds a conceptual load to the algorithms and might make them more difficult to implement. However, unlike most other abstract interpretation-based algorithms, Nie and Cheng's is demonstrably practical, as the authors implemented it for the GNU Compiler Collection (GCC). In their experiments, the size restriction turned out to be unnecessary for avoiding the exponential case, showing that the main bottleneck in complete GVN algorithms is typically their poor data structure choices.

Clearly, there is much room for future exploration. Even without crossing the boundaries of GVN into the scope of other compiler optimizations, we can eliminate all sorts of redundancies. The literature has a wealth of algorithms that all warrant experimentation. With varying degrees of aggressiveness, there are several opportunities to make Factor a more efficient high-level language.

References

Adobe Systems Incorporated. 1999. *PostScript Language Reference*. Third edition. Addison-Wesley. ISBN: 0-201-37922-8. URL: `http://partners.adobe.com/public/developer/en/ps/PLRM.pdf` (visited on August 15, 2011).

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. 2007. *Compilers: Principles, Techniques, & Tools*. Second edition. Addison-Wesley.

Alpern, B., Wegman, M. N., and Zadeck, F. K. 1988. "Detecting Equality of Variables in Programs". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '88. ACM, pages 1–11. ISBN: 0-89791-252-7. URL: `http://doi.acm.org/10.1145/73560.73561`.

American National Standards Institute and Computer and Business Equipment Manufacturers Association. 1994. *American National Standard for information systems: programming languages: Forth: ANSI/X3.215-1994*. American National Standards Institute.

Biggar, P. 2009. "Design and Implementation of an Ahead-of-Time Compiler for PHP". PhD thesis. Trinity College, Dublin. URL: `http://www.paulbiggar.com/research/wip-thesis.pdf` (visited on August 15, 2011).

Briggs, P., Cooper, K. D., Harvey, T. J., and Simpson, L. T. 1998. "Practical Improvements to the Construction and Destruction of Static Single Assignment Form". In: *Software—Practice & Experience* 28 (8), pages 859–881. ISSN: 0038-0644. URL: `http://portal.acm.org/citation.cfm?id=295545.295551`.

Click, C. N. 1995. "Combining Analyses, Combining Optimizations". PhD thesis. Houston, TX, USA: Rice University.

Cocke, J. and Schwartz, J. T. 1970. *Programming Languages and Their Compilers: Preliminary Notes*. Technical report. Courant Institute of Mathematical Sciences, New York Univeristy.

Cousot, P. and Cousot, R. 1977. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of Programming

Languages. POPL '77. ACM, pages 238–252. URL: http://doi.acm.org/10.1145/512950.512973.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. 1991. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". In: *ACM Transactions Programming Languages and Systems* 13 (4), pages 451–490. ISSN: 0164-0925. URL: http://www.eecs.umich.edu/~mahlke/583w03/reading/cytron_toplas_91.pdf (visited on August 15, 2011).

Das, D. and Ramakrishna, U. 2005. "A Practical and Fast Iterative Algorithm for phi-function Computation using DJ Graphs". In: *ACM Transactions Programming Languages and Systems* 27 (3), pages 426–440. ISSN: 0164-0925. URL: http://doi.acm.org/10.1145/1065887.1065890.

Diggins, C. 2007. *The Cat Programming Language*. URL: http://cat-language.com/ (visited on August 15, 2011).

Factor. 2010. From Factor's documentation wiki. URL: http://concatenative.org/wiki/view/Factor (visited on August 15, 2011).

Gargi, K. 2002. "A Sparse Algorithm for Predicated Global Value Numbering". In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. PLDI '02. ACM, pages 45–56. ISBN: 1-58113-463-0. URL: http://doi.acm.org/10.1145/512529.512536.

Giacobazzi, R., Ranzato, F., and Scozzari, F. 2000. "Making Abstract Interpretations Complete". In: *Journal of the ACM* 47 (2), pages 361–416. ISSN: 0004-5411. URL: http://doi.acm.org/10.1145/333979.333989.

Gulwani, S. and Necula, G. C. 2004. "Global Value Numbering Using Random Interpretation". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '04. ACM, pages 342–352. ISBN: 1-58113-729-X. URL: http://doi.acm.org/10.1145/964001.964030.

Gulwani, S. and Necula, G. C. 2007. "A Polynomial-Time Algorithm for Global Value Numbering". In: *Science of Computer Programming* 64 (1), pages 97–114. ISSN: 0167-6423. URL: http://portal.acm.org/citation.cfm?id=1222241.1222599.

Hopcroft, J. E. 1971. *An n log n Algorithm for Minimizing States in a Finite Automaton*. Technical report. Stanford, CA, USA: Stanford University.

Kildall, G. A. 1973. “A Unified Approach to Global Program Optimization”. In: *Proceedings of the 1st ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’73. ACM, pages 194–206. URL: http://doi.acm.org/10.1145/512927.512945.

Nie, J.-T. and Cheng, X. 2007. “An Efficient SSA-Based Algorithm for Complete Global Value Numbering”. In: *Proceedings of the 5th Asian Conference on Programming Languages and Systems*. APLAS ’07. Springer-Verlag, pages 319–334. ISBN: 3-540-76636-7, 978-3-540-76636-0. URL: http://portal.acm.org/citation.cfm?id=1784774.1784806.

Pestov, S., Ehrenberg, D., and Groff, J. 2010. “Factor: A Dynamic Stack-based Programming Language”. In: *Proceedings of the 6th Symposium on Dynamic languages*. DLS ’10. ACM, pages 43–58. ISBN: 978-1-4503-0405-4. URL: http://doi.acm.org/10.1145/1869631.1869637.

Rüthing, O., Knoop, J., and Steffen, B. 1999. “Detecting Equalities of Variables: Combining Efficiency with Precision”. In: *Proceedings of the 6th International Symposium on Static Analysis*. SAS ’99. Springer-Verlag, pages 232–247. ISBN: 3-540-66459-9. URL: http://portal.acm.org/citation.cfm?id=647168.718137.

Simpson, L. T. 1996. “Value-Driven Redundancy Elimination”. PhD thesis. Houston, TX, USA: Rice University.

Steffen, B. 1987. “Optimal Run Time Optimization Proved By A New Look at Abstract Interpretations”. In: *The International Joint Conference on Theory and Practice of Software Development on TAPSOFT ’87*. Springer-Verlag, pages 52–68. ISBN: 0-387-17660-8. URL: http://portal.acm.org/citation.cfm?id=29580.29585.

Tarjan, R. 1972. “Depth-First Search and Linear Graph Algorithms”. In: *SIAM Journal on Computing* 1.2, pages 146–160.

VanDrunen, T. J. 2004. “Partial Redundancy Elimination for Global Value Numbering”. PhD thesis. West Lafayette, IN, USA: Purdue University. ISBN: 0-496-15310-2.

Wegman, M. N. and Zadeck, F. K. 1991. “Constant Propagation with Conditional Branches”. In: *ACM Transactions on Programming Languages and Systems* 13 (2), pages 181–210. ISSN: 0164-0925. URL: http://doi.acm.org/10.1145/103135.103136.

Wilson, P. R. 1992. “Uniprocessor Garbage Collection Techniques”. In: *Proceedings of the International Workshop on Memory Management*. IWMM ’92. Springer-Verlag, pages 1–42. ISBN: 3-540-55940-X. URL: http://portal.acm.org/citation.cfm?id=645648.664824.