## 1 The Factor Compiler

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

Though there are projects for this

### 1.1 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. 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 are already contained in the image.

cite?

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 is associated with a state:

- In the *cold* state, the call site's instruction computes the right method for the class being dispatched upon, which is the operation we're trying to avoid. As it does this, 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. 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.

cite?

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 since Factor has more than eight data types, two tags are reserved to indicate that the type information is stored elsewhere. One is for VM types without their own tag, and the other is for user-defined tuples, each of which has 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. Larger integers and floating point numbers are boxed, but the optimizing compiler may unbox them 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.