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 machine instructions as it goes.

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, we don't need to waste as much time doing method lookup. By default, if more than 3 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

cite?

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.

The Factor implementation is structured into a virtual ma- chine (VM) written in C++ and a core library written in Factor. The VM provides essential runtime services, such as garbage collection, method dispatch, and a base compiler. The rest is implemented in Factor.

The VM loads an image le containing a memory snap- shot, as in many Smalltalk and Lisp systems. The source parser manipulates the code in the image as new denitions are read in from source les. The source parser is written in Factor and can be extended from user code (Section 2.3.1). The image can be saved, and effectively acts as a cache for compiled code. Values are referenced using tagged pointers [29]. Small integers are stored directly inside a pointers payload. Large integers and oating point numbers are boxed in the heap; however, compiler optimizations can in many cases elimi- nate this boxing and store oating point temporaries in regis- ters. Specialized data structures are also provided for storing packed binary data without boxing (Section 2.4).

The optimizing compiler is writ- ten in Factor and is used to compile most code. Factor is partially self-hosting and there is a bootstrap process, similar to Steel Bank Common Lisp [38]. An im- age generation tool is run from an existing Factor instance to produce a new bootstrap image containing the parser, ob- ject system, and core libraries. The Factor VM is then run with the bootstrap image, which loads a minimal set of li- braries which get compiled with the base compiler. The op- timizing compiler is then loaded, and the base libraries are recompiled with the optimizing compiler. With the optimiz- ing compiler now available, additional libraries and tools are loaded and compiled, including Factors GUI develop- ment environment. Once this process completes, the image is saved, resulting in a full development image.

The Factor implementation is structured into a virtual ma- chine (VM) written in C++ and a core library written in Factor. The VM provides essential runtime services, such as garbage collection, method dispatch, and a base compiler. The rest is implemented in Factor.

The VM loads an image le containing a memory snap- shot, as in many Smalltalk and Lisp systems. The source parser manipulates the code in the image as new denitions are read in from source les. The source parser is written in Factor and can be extended from user code (Section 2.3.1). The image can be saved, and effectively acts as a cache for compiled code. Values are referenced using tagged pointers [29]. Small integers are stored directly inside a pointers payload. Large integers and oating point numbers are boxed in the heap; however, compiler optimizations can in many cases elimi- nate this boxing and store oating point temporaries in regis- ters. Specialized data structures are also provided for storing packed binary data without boxing (Section 2.4).
Factor uses a generational garbage collection strategy to optimize workloads which create large numbers of short- lived objects. The oldest generation is managed using a mark-sweep-compact algorithm, with younger generations managed by a copying collector [46]. Even compiled code is subject to compaction, in order to reduce heap fragmentation in applications which invoke the compiler at runtime, such as the development environment. To support early binding, the garbage collector must modify compiled code and the callstack to point to newly relocated code. Run-time method dispatch is handled with polymorphic inline caches [32]. Every dynamic call site starts out in an uninitialized cold state. If there are up to three unique receiver types, a polymorphic inline cache is generated for the call site. After more than three cache misses, the call site transitions into a megamorphic call with a cache shared by all call sites. All source code is compiled into machine code by one of two compilers, called the base compiler and optimizing compiler. The base compiler

is a context threading compiler implemented in C++ as part of the VM, and is mainly used for bootstrapping purposes. The optimizing compiler is writ- ten in Factor and is used to compile most code. Factor is partially self-hosting and there is a bootstrap process, similar to Steel Bank Common Lisp [38]. An im- age generation tool is run from an existing Factor instance to produce a new bootstrap image containing the parser, ob- ject system, and core libraries. The Factor VM is then run with the bootstrap image, which loads a minimal set of li- braries which get compiled with the base compiler. The op- timizing compiler is then loaded, and the base libraries are recompiled with the optimizing compiler. With the optimiz- ing compiler now available, additional libraries and tools are loaded and compiled, including Factors GUI develop- ment environment. Once this process completes, the image is saved, resulting in a full development image.

The primary design considerations of the base compiler are fast compilation speed and low implementation complexity. As a result, the base compiler generates context-threaded code with inlining for simple primitives [3], performing a single pass over the input quotation. The base compiler generates code using a set of machine code templates for basic operations such as creating and tearing down a stack frame, pushing a literal on the stack, making a subroutine call, and so on. These machine code templates are generated by Factor code during the bootstrap process. This allows the base and optimizing compilers to share a single assembler backend written in Factor.

The optimizing compiler is structured as a series of passes operating on two intermediate representations (IRs), referred to as high-level IR and low-level IR. High-level IR represents control ow in a similar manner to a block-structured programming language. Low-level IR represents control ow with a control ow graph of basic blocks. Both intermediate forms make use of single static assignment (SSA) form to improve the accuracy and efciency of analysis [12].

or, y'know, like annotated stack code

High-level IR is constructed by the stack effect checker. Macro expansion and quotation inlining is performed by the stack checker online while high-level IR is being con- structed. The front end does not deal with local variables, as these have already been eliminated.

When static type information is available, Factors compiler can eliminate runtime method dispatch and allocation of in- termediate objects, generating code specialized to the under- lying data structures. This resembles previous work in soft typing [10]. Factor provides several mechanisms to facilitate static type propagation:

- Functions can be annotated as inline, causing the compiler to replace calls to the function with the function body.
- Functions can be hinted, causing the compiler to gener- ate multiple specialized versions of the function, each assuming different input types, with dispatch at the en- try point to choose the best-tting specialization for the given inputs.
- Methods on generic functions propagate the type infor- mation for their dispatched-on inputs.
- Functions can be declared with static input and output types using the typed library.

The three major optimizations performed on high-level IR are sparse conditional constant propagation (SCCP [45]), escape analysis with scalar replacement, and overow check elimination using modular arithmetic properties. The major features of our SCCP implementation are an extended value lattice, rewrite rules, and ow sensitivity. Our SCCP implementation augments the standard single- level constant lattice with information about object types, numeric intervals, array lengths and tuple slot types. Type transfer functions are permitted to replace nodes in the IR with in- line expansions. Type functions are dened on many of the core language words. SCCP is used to statically dispatch generic word calls by inlining a specic method body at the call site. This inlining generates new type information and new opportunities for constant folding, simplication and further inlining. In par- ticular, generic arithmetic operations which require dynamic dispatch in the general case can be lowered to simpler opera- tions as type information is discovered. Overow checks can be removed from integer operations using numeric interval information. The analysis can represent ow-sensitive type information. Additionally, calls to closures which combina- tor inlining cannot eliminate are eliminated when enough in- formation is available [16]. An escape analysis pass is used to discover object alloca- tions which are not stored on the heap or returned from the current function. Scalar replacement is performed on such allocations, converting tuple slots into SSA values. The modular arithmetic optimization pass identies in- teger expressions in which the nal result is taken to be modulo a power of two and removes unnecessary overow checks from any intermediate addition and multiplication operations. This novel optimization is global and can operate over loops.

Low-level IR is built from high-level IR by analyzing control ow and making stack reads and writes explicit. During this construction phase and a subsequent branch splitting phase, the SSA structure of high-level IR is lost. SSA form is recon- structed using the SSA construction algorithm described in [8], with the minor variation that we construct pruned SSA form rather than semi-pruned SSA, by rst computing live- ness. To avoid computing iterated dominance frontiers, we use the TDMSC algorithm from [13]. The main optimizations performed on low-level IR are local dead store and redundant load elimination, local value numbering, global copy propagation, representation selec- tion, and instruction scheduling. The local value numbering pass eliminates common sub- expressions and folds expressions with constant operands [9]. Following value numbering and copy propagation, a representation selection pass decides when to unbox oating point and SIMD values. A form of instruction scheduling intended to reduce register pressure is performed on low-level IR as the last step before leaving SSA form [39]. We use the second-chance binpacking variation of the lin- ear scan register allocation algorithm [43, 47]. Our variant does not take nodes into account, so SSA form is destruc- ted rst by eliminating nodes while simultaneously per- forming copy coalescing, using the method described in [6].