Factor: A Dynamic Stack-based Programming Language

Slava Pestov
Stack Effects LLC
slava@factorcode.org

Daniel Ehrenberg
Carleton College
ehrenbed@carleton.edu

Joe Groff
Durian Software
joe@duriansoftware.com

Abstract

Factor is a new dynamic object-oriented programming language. It began as an embedded scripting language and evolved to a mature application development language. The language has a simple execution model and is based on the manipulation of data on a stack. An advanced metaprogramming system provides means for easily extending the language. Thus, Factor allows programmers to use the right features for their problem domain. The Factor implementation is self-hosting, featuring an interactive development environment and an optimizing compiler. In this paper, the language and its implementation are presented.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.3.4 [*Programming Languages*]: Processors — Compilers, Optimization

General Terms Languages, Design, Performance

Keywords Factor, dynamic languages, stack-based languages

1. Introduction

Factor is a dynamic stack-based programming language. It was originally conceived as an experiment to create a stack-based language practical for modern programming tasks. It was inspired by earlier stack-based languages like Forth [33] and Joy [44]. The stack-based model allows for concise and flexible syntax with a high degree of factoring and code reuse. Driven by the needs of its users, Factor gradually evolved from this base into a dynamic, object-oriented programming language. Although there is little truly novel to the Factor language or implementation, Factor's combination of the stack-based paradigm with functional, object-oriented, and low-level programming features alongside its high-performance implementation and interactive development environment make it notable.

Factor programs look very different from programs in most other programming languages. At the most basic level, function calls and arithmetic use postfix syntax, rather than prefix or infix as in most programming languages. Factor provides local variables, but they are used in only a small minority of procedures because its language features allow most code to be comfortably written in a point-free style.

Factor is an object-oriented language with an object system centered around CLOS-inspired generic functions in place of traditional message passing. To make Factor suitable for development of larger applications, it has a robust module system. Factor's metaprogramming system allows for arbitrary extension of syntax and for compile-time computation. Factor allows the clean integration of high-level and low-level code with extensive support for calling libraries in other languages and for efficient manipulation of binary data.

Factor has an advanced, high-performance implementation. We believe good support for interactive development is invaluable, and for this reason Factor allows programmers to test and reload code as it runs. Our ahead-of-time optimizing compiler and efficient runtime system can remove much of the overhead of high-level language features. Together, these features make Factor a useful language for writing both quick scripts and large programs in a high-level way.

Factor is an open-source project and the product of many contributors. It is available for free download from `http://factorcode.org`.

This paper contributes the following:

- Abstractions and checking for managing the flow of data in stack-based languages (Section 2.1)
- A CLOS- and Dylan-inspired object system, featuring generic functions built upon a metaobject protocol and a flexible type system (Section 2.2)
- An expressive and easy-to-use system for staged metaprogramming (Section 2.3)
- The design of a foreign function interface and low-level capabilities in a dynamic language (Section 2.4)

```
"data.txt" utf8 file-lines
10 head
```

Figure 1: Retrieving the first ten lines of a file

- The design of an effective optimizing compiler for a dynamic language (Section 3)
- A case study in the evolution of a dynamic language (Secton 4)

Much of Factor's features and implementation are not radically different from previous dynamic languages. It is the combination of these features into a useful system that make Factor notable.

2. Language Design

Factor combines features from existing languages with new innovations. We focus here on the prominent unique aspects of Factor's design: our contributions to the stack-based language paradigm, the module and object systems, tools for metaprogramming, and low-level binary data manipulation support.

2.1 Stack-based Programming Language

In Factor, as in Forth and Joy [44], function parameters are passed by pushing them on an operand stack prior to performing the function call. We introduce two original contributions: a set of combinators which replace the stack shuffling words found in other stack languages, and a syntax for partial application.

2.1.1 Postfix Syntax

In stack-based languages, a function call is written as a single token, which leads to the term "word" being used in place of "function", in the Forth tradition. This contrasts with mainstream programming languages, in which function call syntax combines the function name with a list of parameters. Symbols that traditionally behave as operators with infix syntax in other languages, such as `+`, `-`, `*`, and `/`, are normal postfix words in Factor and receive no special syntactic treatment. Languages which use an operand stack in this manner have been called *concatenative*, because they have the property that programs are created by "concatenating" (or "composing") smaller programs. In this way, words can be seen as functions which take and return a stack [30].

Literals in a stack language, such as `"data.txt"` and `10` in Figure 1, can be thought of as functions that push themselves on the stack, making the values available to subsequent word calls which pop them off the stack. Some words, such as `utf8`, can also behave as literal symbols that push themselves on the stack. `file-lines` consumes two objects from the stack, the filename string `"data.txt"` and the encoding symbol `utf8`, and pushes back an array of strings, the contents of the specified file broken into lines of text.

On the second line, the standard library word `head` pops two objects from the stack, the array of lines of text resulting from `file-lines` and the integer `10`, and pushes a new array containing a fixed number of elements from the beginning of the input array. The final result of the example is an array containing the first ten lines of text from the file `"data.txt"` as strings.

The two lines of code in Figure 1 can be understood either as separate programs or concatenated to form a single program. The latter case has the effect of first running the first program and then the second. Note that in Factor source code, newlines between words are interpreted the same as spaces.

2.1.2 Higher-order Programming

Factor supports higher-order functions (functions which take functions as arguments). In the Joy tradition, we refer to higher-order functions as *combinators* and to anonymous functions as *quotations*. Quotation literals are pushed on the stack by surrounding a series of tokens with `[` and `]`, which delays the evaluation of the surrounded code and stores it in a quotation object. Combinators are invoked like any other word, with quotation objects as parameters. In Factor, all control flow is expressed using combinators, including common branching and looping constructs usually given special syntax in other languages. Some examples:

- `if` is a combinator taking three inputs from the stack: a boolean value, a "then" branch quotation, and an "else" branch quotation. For example,
  `2 even? [ "OK" ] [ "Cosmic rays detected" ] if`
  tests whether the value 2 is even, placing the string `"OK"` on the stack if so or `"Cosmic rays detected"` if not.
- The `each` standard library word implements what other languages call a "for-each" loop. It iterates in order over the elements of an array (or other sequence, see 2.2.2), invoking a quotation with each element as the input parameter on each iteration. For example,
  `{ "veni" "vidi" "vici" } [ print ] each`
  will print the strings `"veni"`, `"vidi"`, and `"vici"` to the console in order. (The { and } delimiters create a literal array object in the same manner `[` and `]` create a quotation.)
- `reduce` is a variation of `each` that accumulates a value between iterations. From the top of the stack, it takes an array, an initial accumulator value, and a quotation. On each iteration, it passes to the provided quotation the next item in the array along with the accumulated value thus far. For example,
  `{ 1 2 3 4 } 0 [ + ] reduce`
  will sum the numbers 1, 2, 3, and 4, and
  `{ 1 2 3 4 } 1 [ * ] reduce`
  will multiply them.

```
[ "#" head? not ] filter
[ string>number ] map
0 [ + ] reduce
```

Figure 2: Summing the numerical value of array elements not beginning with #

- map iterates over its input array like each but additionally collects the output value from each invocation of the quotation into a new array. For example,
`{ "veni" "vidi" "vici" } [ reverse ] map`
will make a reversed copy of every element in an array, collecting them into a new array `{ "inev" "idiv" "iciv" }`.
- filter also invokes a quotation over every element of an array but collects only the elements for which the quotation returns true into a new array, leaving them unchanged from the input array. For example,
`{ 1 4 9 16 } [ even? ] filter`
will create a new array containing the elements of the input array that are even: `{ 4 16 }`.

The code snippet in Figure 2 shows how a series of operations involving these combinators can be composed. As input, it expects an array on the stack containing string elements. The first line uses the head? word, which takes two strings off the stack and outputs a boolean indicating whether the first one begins with the second; for example, `"# comment" "#" head?` would output true whereas `"17" "#" head?` would output false. The snippet uses head? as a predicate quotation for filter (with the predicate negated by the not word) to remove strings beginning with # from the input array. This filtered array then acts as input to the second line, where the map applies the string>number word to each remaining array element, converting the elements from strings into numbers. Finally, on the third line, the reduce combinator adds the resulting numeric values together.

Code written in this style, in which a single input value is gradually transformed and reshaped in distinct steps into a result, is known as *pipeline code*, named due to the resemblance to the use of pipes in Unix shells. Pipeline code is expressed very naturally in Factor; given several words, say a, b, c, each taking a single object from the stack and pushing a single result, the code that applies each word to the result of the previous word is simply written as

```
a b c
```

In a non-concatenative language like Python, the following would be used:

```
c(b(a(x)))
```

Compared to Factor, the above code has more syntactic nesting, and the order of tokens is backwards from the order of evaluation. Functional languages often have operators for function composition inspired by traditional mathematical notation that help simplify the expression of pipeline code, such as Haskell's . operator:

```
c . b . a
```

The composition operator improves on the nesting and syntactic overhead required by the non-concatenative approach, but when compared to Factor, the approach still requires additional operators and leaves the order of operations reading in reverse order.

```
: tail-factorial ( accumulator n -- n! )
    dup 0 =
    [ drop ]
    [ [ * ] [ 1 - ] bi tail-factorial ]
    if ;

: factorial ( n -- n! )
    1 swap (factorial) ;
```

Figure 3: Tail-recursive factorial in Factor

Fundamentally, all control flow in Factor is achieved either by branching via the aforementioned if combinator or by recursion. if considers all values to be true except for the f word, which represents false. Looping combinators such as each combine branching with tail recursion to express loops. Factor guarantees tail call optimization. Figure 3 shows an example of conditionals and recursion to implement the factorial operation. The figure also introduces the : syntax for defining new words and the `( -- )` syntax for stack effects, and features the stack shuffle words dup, drop, and swap and the dataflow combinator bi. These features will be described in the following subsections.

2.1.3 Stack Effects

Factor provides a formal notation for describing the inputs and outputs of a word called *stack effect notation*, which is used both in Factor documentation and in Factor source code. For example, the stack effect of the tail-factorial word in Figure 3is written as follows, with the special token -- separating inputs from outputs:

```
( accumulator n -- n! )
```

This indicates that the word takes two inputs off the stack, given the names accumulator and n, and leaves one new result n!. The names are for documentation and do not affect the meaning of the program. The factorial word's stack effect `( n -- n! )` likewise indicates that it takes one input off the stack and outputs a new value. New words such as tail-factorial and factorial are defined with the : token, which is followed by the name of the new word, the stack effect, and the definition, with the definition terminated by the ; token. Every word definition must have a stack effect declared.

Factor stack effect notation is similar to conventional Forth stack effect notation. However, in Forth, the contents of a stack effect are mere comments skipped over by the parser, and their notation is a matter of programmer convention. By contrast, Factor provides a standard stack effect syntax and enforces the declared stack effects of words, as in StrongForth and Cat. In Factor, with very few exceptions, words must pop a fixed number of inputs off the stack, and push a fixed number of outputs onto the stack. Row-polymorphic combinators, described below, and macros, described in 2.3.2, are the only two exceptions to this rule. Stack effects are checked and enforced by a kind of type system in the language implementation, known as the *stack checker*.

Stack languages with higher-order functions and static stack checking encounter a unique type of polymorphism: a higher order function may sometimes accept quotation parameters with different stack effects, which affect the overall stack effect of the combinator. For example, the `each` combinator as described in 2.1.2 would have a stack effect of `( seq quot -- )` and take a quotation of effect `( elt -- )`. However, on every application of `each`, the `elt` input is presented above *the rest of the stack*, and the quotation can read and replace additional stack values below the element in a way that maintains the stack balance. `each` can thus take a quotation with effect `( x elt -- x' )`, `( x y elt -- x' y' )`, and so on. This allows code using looping combinators such as `each` to pass values between iterations. The overall stack effect of `each` in these cases becomes `( x seq quot -- x' )`, `( x y seq quot -- x' y' )`, and so on. This type of polymorphism is referred to as *row polymorphism*, due to influence from Cat's row-polymorphic type system[14].

Factor has a special notation for row-polymorphic stack effects. If the list of inputs or outputs begins with a token of the form `..a`, where `a` is any character, that token represents a *row variable* comprising multiple inputs or outputs. Quotation inputs can also be given stack effects in the form `name: ( inputs -- outputs )`, and row variables in those nested stack effects will be unified with row variables of the same name in the outer stack effect or other nested stack effects. The full row-polymorphic stack effect of `each` is thus:

```
( ..a seq quot ( ..a elt -- ..a ) -- ..a )
```

2.1.4 The Stack Checker

Factor's stack checker performs an abstract interpretation of the input program, simulating each word's effect on the stack. When conditional branches are encountered, both branches are evaluated and unified; a unification failure indicates that the two branches leave with inconsistent stack heights, which is a compile-time error.

The stack checker must be able to handle row polymorphism. The current approach is to inline calls to all words which call quotations of indeterminate stack effect so that the quotations that they call can be inlined. The inlining of words is driven by declarations. The stack checker tracks the flow of constants in a program in a simple, pessimistic way, and can inline quotations with this mechanism. If it finds a quotation that it cannot inline, but it must, then it rejects the program. This inlining is not just an optimization; some code is invalid without inlining.

Of course, not all programs can be written this way. In some cases, the invoked quotation may not be known until runtime. For this, there are two "escape hatches." One mechanism is to declare the stack effect of the quotation at its call site. The stack effect will be checked at runtime. In this case, the quotation does not need to be inlined. Quotations are normally called with the word `call`, and an effect can be specified with the syntax `call( -- )`. Alternatively, a quotation can be called with a user-supplied datastack, using the `with-datastack` word. This is useful for implementing a read-eval-print loop (Section 3.1).

We believe that a mechanism for checking the stack depth of programs is a necessary tool for concatenative programming languages. In concatenative languages without static stack checking, incorrect use of the stack can lead to hard-to-discover bugs. In the absence of static checking, if a caller expects a word to have a certain effect on the height of the stack, and to only change a particular number of items on the stack, then the programmer writing the calling code must write unit tests covering every possible code path in the callee word. While good test coverage is a desirable property to have in any software system, it can be difficult achieving full test coverage in practice.

2.1.5 Dataflow Combinators

Many stack languages provide a set of words for re-arranging operands at the top of the stack. These words can be used to glue other words together. A typical set of shuffle words is provided as part of Factor:

- `drop ( x -- )`
- `dup ( x -- x x )`
- `over ( x y -- x y x )`
- `swap ( x y -- y x )`

By convention, the stack effects of shuffle words use names in the stack effect inputs and outputs to indicate how the objects are reordered on the stack; for example, `swap` exchanges the positions of the top two stack elements, as indicated by its effect `( x y -- y x )`.

One downside of shuffle words is that understanding long expressions requires the reader to maintain and manipulate a mental model of the stack, making code with complex dataflow hard to write and to understand. This is a common objection programmers have to programming in stack-based languages. Factor provides an alternative facility, inspired by Joy[44], consisting of three fundamental kinds of combina-

```
TUPLE: edge face vertex opposite-edge next-edge ;
...
[ vertex>> ] [ opposite-edge>> vertex>> ] bi
```

Figure 4: Cleave combinators being used to call multiple slot accessors on a single object on the stack

```
: check-string ( obj -- obj )
    dup string? [ "Not a string" throw ] unless ;

: register-service ( name realm constructor -- )
    [ check-string ] [ ] [ call( -- value ) ] tri*
    ... ;
```

Figure 5: Spread combinators being used to change objects at the top of the stack

tor that encapsulate common, easy-to-understand dataflow patterns.

- `cleave` takes as input a single value along with an array of quotations and calls each quotation in turn on that value.
  `5 { [ 1 + ] [ 2 - ] } cleave`
  $\rightarrow$ `6 3`
- `spread` takes series of objects on the stack along with an array of an equal number of quotations and calls each quotation on the corresponding object.
  `"A" "b" { [ >lower ] [ >upper ] } spread`
  $\rightarrow$ `"a" "B"`
- `napply` takes a series of objects on the stack together with a single quotation and an integer as input and calls each quotation with the value. The number of values is determined by the integer.
  `"A" "B" [ >lower ] 2 napply`
  $\rightarrow$ `"a" "b"`

Shorthand forms are also available for binary and ternary cases. They are not strictly necessary, but they avoid the slight verbosity of the additional tokens { and } used to delimit arrays. They follow a naming scheme where the shorthand cleave combinators taking two and three quotations are named `bi` and `tri`, spread combinators are named `bi*` and `tri*`, and apply combinators are named `bi@` and `tri@`. In Figure 3, the `tail-factorial` word uses `bi` to reuse its `n` value, first multiplying it into the `accumulator` input and then adding `1` to it before tail-calling itself to start a new iteration.

A canonical use case for `cleave` is to extract objects from tuple slots as in Figure 4. `spread` is frequently used to preprocess or make assertions about inputs to words. In Figure 5, `name` must be a string, and `constructor` is replaced with a single object it produces when called.

```
:: frustum-matrix4 ( xy-dim near far -- matrix )
    xy-dim first2 :> ( x y )
    near x / :> xf
    near y / :> yf
    near far + near far - / :> zf
    2 near far * * near far - / :> wf

    {
        { xf  0.0  0.0 0.0 }
        { 0.0 yf   0.0 0.0 }
        { 0.0 0.0  zf  wf  }
        { 0.0 0.0 -1.0 0.0 }
    } ;
```

Figure 6: Constructing a perspective projection matrix, using local variables

2.1.6 Pictured Partial Application Syntax

We propose a syntax for the construction of point-free closures in a stack-based language. In Factor, quotations do not close over an environment of values; pushing a quotation on the operand stack does not allocate any memory and quotations are effectively literals in the source code. Factor has an additional facility for constructing new quotations from values on the stack; this resembles lexical closures in applicative languages. A "quotation with holes" can be written by prefixing it with `'[`, and using `_` to refer to values which are to be filled in from the stack when the quotation is pushed. The following two lines are equivalent:

```
5 '[ _ + ]
[ 5 + ]
```

2.1.7 Lexically Scoped Variables

With a few operations to shuffle the top of the stack, as well as the previously-mentioned dataflow combinators, the stack can be used for all dataflow and there is no need for local variables. In practice, some code is easier to express with local variables, so Factor includes support for local variables and lexical closures.

A word with named input parameters can be declared with the `::` token in place of `:`. Whereas normally, the names of input parameters in the stack effect declaration have no meaning, a word with named parameters makes those names available in the lexical scope of the word's definition. Within the scope of a `::` definition, additional lexical variables can be bound using the `:>` operator, which binds either a single value from the datastack to a name, or multiple stack values to a list of names surrounded by parentheses. Literal array and tuple syntax can include lexical variable names and construct data structures from lexical variable values.

Numerical formulas often exhibit non-trivial dataflow and benefit in readability and ease of implementation from using local variables. For example, Figure 6 constructs a perspective projection matrix for three-dimensional rendering. This would be somewhat awkward with purely stack-based code.

We have found lexical variables useful only in rare cases where there is no obvious solution to a problem in terms of dataflow combinators and the stack. Out of 38,088 word and method definitions in the source code of Factor and its development environment at the time of this writing, 310 were defined with named parameters. Despite their low rate of use, we consider lexical variables, and in particular lexically-scoped closures, a useful extension of the concatenative paradigm.

2.2 Organizing Programs

Whereas many languages, notably Java, combine their module system and type system, Factor separates the two concepts to maximize flexibity and modularity of code. *Vocabularies* provide modularity, source code organization, and namespaces. Independent of source code organization, *classes* and *generic words* organize the data types and operations of a program at run time.

2.2.1 Vocabularies

Factor code is organized in a system of nested modules called *vocabularies*. Like Java packages [27], Factor vocabularies have an on-disk directory structure corresponding to their module structure. A vocabulary contains zero or more definitions. The most common definitions are *word definitions*.

Every source file must explicitly specify all vocabularies it uses; only word names defined in these vocabularies will be in scope when the file is parsed. Any vocabulary dependencies which have not been loaded are loaded automatically.

Factor's vocabulary system does not provide language-enforced privacy. There is a syntactic mechanism to mark words as private, which places them in a separate sub-vocabulary. External code can still access this private vocabulary by explicitly specifying it. The choice not to enforce real privacy was done in order to maximize flexibility and interactivity.

2.2.2 Object System

Factor is a purely object-oriented programming language in the same sense as Smalltalk or Ruby: Every value is an object with an intrinsic type that participates in dynamic dispatch, and basic operations like array access, arithmetic and instance variable lookup are done through dynamically dispatched method calls. However, unlike Smalltalk or Ruby, Factor does not specially distinguish a receiver object for method calls. As in CLOS, there is no object or class that "owns" a method. Instead, special words called *generic words* have multiple implementations, called *methods*, based on the classes of their arguments. Generic words offer more flexibility than traditional message passing:

```
TUPLE: circle radius ;
TUPLE: rectangle length width ;
GENERIC: area ( shape -- area )
M: circle area
    radius>> dup * pi * ;
M: rectangle area
    [ length>> ] [ width>> ] bi * ;
```

Figure 7: Shapes and their area

- Methods on a generic word may be defined in the same file as a class or in a different file. This allows new generic words to dispatch on existing classes. It is also possible to define new classes with methods on existing generic words defined in the file where the class is defined.
- More complicated kinds of classes are possible. Predicate classes [17] fit into this model very easily.
- Multiple dispatch is natural to express. Though the core Factor object system does not yet implement multiple dispatch, it is available in an external library.

Factor's object system is implemented in Factor and can be extended through a meta-object protocol. Factor has three types of classes: primitive classes, tuple classes, and derived classes. *Primitive classes* are used for objects like strings, numbers, and words. These cannot be subclassed. *Tuple classes* are records with instance variables and single inheritance. They form a hierarchy rooted at the class `tuple`. Figure 7 shows a simple use of tuple classes to model shapes and a generic word to calculate their area.

Primitive classes and tuple classes both use method calls to access instance variables. For an instance variable called `foo`, the generic word to read the variable is called `foo>>`, and the generic word to write it is called `>>foo`.

Classes in Factor are not just record definitions; they are abstract sets of objects. *Derived classes* offer a way to specify new classes in terms of existing ones. A *predicate class* is a subclass of another class consisting of instances satisfying a predicate. A *union class* consists of the union of a list of classes, and an *intersection class* consists of the intersection of a list of classes.

A particular case of union classes is *mixins*. A mixin is an extensible union class. Mixins are used to share behavior between an extensible set of classes. If a method is defined on a mixin, then the definition is available to any class which chooses to add itself to the mixin. One particular use of mixins is to mark a set of classes which all implement methods on a set of generic words. Though Factor has no fixed construction for an interface as in Java, an informal *protocol* consisting of a set of generic words combined with a mixin to mark implementors is idiomatically used for the same purpose.

In Factor's standard library, compile-time metaprogramming is used to define several new features in the object system. This allows the core object system to remain simple while giving users access to advanced features.

The `delegate` library implements the Proxy Pattern [26]. The programmer can define a protocol and declare a class to delegate to another class using a piece of code to look up the delegate. The library will generate methods for each generic word in the protocol to perform the delegation. This reduces the amount of boilerplate code in the program. There are also libraries for the terse declaration of algebraic datatypes, for a limited form of multiple inheritance, and for multiple dispatch methods.

The Factor standard library uses the object system heavily. Arrays, vectors, strings and other types of sequences are abstracted as a set of generic words and a mixin class, together comprising the sequence protocol. There are similar protocols for associative mappings and for sets. One use case of these protocols is to make *virtual sequences* (or virtual associative mappings or virtual sets): objects which satisfy the sequence protocol but do not actually physically store their elements. One example of a virtual sequence is a range of integers with a given start, end, and step. Below is a definition of the factorial function using a range, which is much easier to read than the definition in Figure 3. The `product` word uses the sequence protocol to multiply together the elements of any object implementing the protocol.

```
: factorial ( n -- n! )
    [1,b] product ;
```

2.3 Ahead-of-time Metaprogramming

The philosophy of Factor's metaprogramming and reflection facilities is that users should be able to extend the language with the same mechanisms used to implement the language itself. This maximizes expressiveness while minimizing code duplication between the language implementation and its metaprogramming API.

Factor's syntax is entirely defined using *parsing words* written in Factor itself, and users can add their own parsing words to extend Factor's syntax. Additionally, Factor provides *macros*, which are used like ordinary words but perform partial evaluation on their first few parameters. *Functors* allow generic programming, and can be used to create classes or vocabularies parameterized by a list of arguments.

These three features allow for an alternative model of metaprogramming from that of C++ [41] or of scripting languages like Ruby [42]. Factor offers high runtime performance using a static compiler while maintaining flexibility. Like Ruby, this feature uses ordinary Factor code, rather than a restricted special language like C++ templates. Unlike Ruby, metaprogramming takes place explicitly before compilation, allowing an ahead-of-time compiler to be effective in optimizing the code, as in C++. We have not found cases where we wanted to use runtime metaprogramming rather than Factor's approach.

```
TUPLE: product id quantity price ;

: make-product-tag ( product -- xml )
    [ id>> ] [ quantity>> ] [ price>> ] tri
    [XML
        <product
            id=<->
            quantity=<->
            price=<->
        />
    XML] ;

: make-product-dump ( products -- xml )
    [ make-product-tag ] map
    <XML
        <products><-></products>
    XML> ;
```

Figure 8: Dumping a sequence of products as XML

It is common to use parsing words, macros and functors in conjunction. A parsing word might trigger the invocation of a functor, which in turn might expand into code containing macros. For example, the `SPECIALIZED-ARRAY:` syntax invokes a functor to create a *specialized array* type, a data structure designed to contain binary data in a specified packed format, similar to C++'s templated `std::vector` data structure.

2.3.1 Parsing Words

Factor's syntax is based on the Forth programming language. A program is a stream of whitespace-separated tokens. Some of these tokens are simple literals, like numbers or strings. Some tokens are words called at runtime. And some tokens are words run during parsing, called *parsing words*.

Parsing words can perform arbitrary computation, and usually make use of the parser API to read tokens from the source file and the word definition API to define new words. One use for parsing words is to create compound literals. For example, { is a parsing word which scans until the next matched } and creates an array consisting of the objects in between the brackets.

Parsing words are also used to implement definitions. The parsing word : defines a new word, by reading the stack effect and word body, and then storing the definition in the current vocabulary.

In the Factor standard library, the `<XML` parsing word creates a literal document in the eXtensible Markup Language (XML) [20], with special syntax for objects to be spliced into the document. The similar `[XML` parsing word creates an XML fragment which can be embedded in a larger document. XML literals have become a popular feature in new

```
SYNTAX: $[
    parse-quotation call( -- value ) suffix! ;
```

Figure 9: The parsing word `$[` allows arbitrary computation in-line at parse-time

programming languages such as Scala [19] and as additions to existing programming languages such as E4X [15]. In contrast to those languages, we were able to implement XML literals purely as a library feature. Figure 8 demonstrates a word which takes a sequence of `product` tuples, generating an XML document listing quantities and prices. Note that within an XML fragment, `<->` is used to take an argument from the stack, in a manner similar to the pictured partial application syntax discussed in Section 2.1.6.

As another example of Factor's metaprogramming capability, local variables are also implemented as a user-level library. The implementation converts code using locals to purely stack-based code.

As an example of creating a parsing word, Figure 9 shows how to create a parsing word to do arbitrary computation at parse-time. The word `parse-quotation` invokes the parser to return the Factor code between the current location and the matching `]`. The word `suffix!` is used to push the parsed value onto the parser's accumulator. Because parsing words always take and return an accumulator, the stack effect is implied and unnecessary.

2.3.2 Macros

Macros in Factor are special words which take some of their input parameters as compile-time constants. Based on these parameters, the macro is evaluated at compile-time, returning a quotation that replaces the macro call site. This quotation may take further parameters from the run-time stack.

One example of a macro is `cond`[1], used to provide a convenient syntax for if-else-if chains. As an argument, `cond` takes an array of pairs of quotations, in which the first quotation of each pair is the condition and the second is the corresponding outcome. An example is shown in Figure 10. The `cond` macro expands into a series of nested calls to the `if` combinator at compile time. Macro expansion is performed in the stack checker using the same constant propagation mechanism as quotation inlining (Section 2.1.4). When a macro invocation is encountered, the macro body is called at compile time with the constant inputs that it requires. Calling a macro with values that are not known to be constant is a compile-time error.

This integration with the stack checker gives Factor macros more flexibility than traditional Lisp macros [28]. Rather than requiring macro parameters to be literals immediately present in the syntax, they are only required to be constants as known by the stack checker. A combinator can call a macro with only some parameters immediately supplied, as long as the combinator is declared `inline` and usages of the combinator supply the necessary compile-time parameters. A simple example is a composition of the `length` word with the `case` combinator. The `case` combinator takes a sequence of pairs, where the first element in each pair is a value, and the second is a quotation to be called if the top of the stack at run time is equal to the value. We can define the `length-case` combinator, which takes a sequence and a sequence of pairs, dispatching on the length of the sequence:

```
: length-case ( seq cases -- )
    over length swap case ; inline
```

[1] For bootstrapping reasons, `cond` is not implemented like other macros, but it is conceptually the same.

```
"libssl" {
    { [ os winnt? ] [ "ssleay32.dll" ] }
    { [ os macosx? ] [ "libssl.dylib" ] }
    { [ os unix? ] [ "libssl.so" ] }
} cond cdecl add-library
```

Figure 10: Determining the name of the OpenSSL library to load based on the user's current platform

2.3.3 Functors

Although parsing words will already let you generate arbitrary code at compile-time, it can be inconvenient to use the word definition API repeatedly for similar definitions. The `functors` library provides syntactic sugar for this, in a manner that resembles C++ templates but allows for arbitrary computation in Factor. Functor syntax also resembles the *quasiquote* syntax of Common Lisp [28]. One major usage of functors is for the aforementioned specialized arrays of binary types.

Functors are useful in implementing *cords*, virtual sequence objects that present two underlying sequences as a single concatenated sequence. The functor in Figure 11 allows a type `T-cord` to be defined for a sequence type `T`. This allows for additional compiler optimization when using cords of `T` sequences compared to the non-specialized `generic-cord` implementation. The functor defines the `T-cord` type with a name derived from the name of the input type `T`, and declares the new type as a member of the `cord` mixin, allowing it to share method definitions with `generic-cord` and other specialized cord types. This functor is used to efficiently emulate 256-bit SIMD vector types on platforms with only 128-bit hardware vectors.

2.4 Low-level Features

Factor includes many tools for systems programming that allow for both high-efficiency specialized and high-level object-oriented patterns of usage. A foreign function interface provides access to procedures written in other languages

```
FUNCTOR: define-specialized-cord ( T -- )

T-cord DEFINES-CLASS ${T}-cord

WHERE

TUPLE: T-cord
    { head T read-only }
    { tail T read-only } ; final
INSTANCE: T-cord cord

;FUNCTOR
```

Figure 11: Creating a specialized `T-cord` virtual sequence for a type T using a functor

as if they were written in Factor. Binary data can be represented and manipulated efficiently. A new abstraction called destructors provides for the automatic disposal of resources.

2.4.1 Foreign Function Interface

Factor has a foreign function interface (FFI) for calling libraries written in other programming languages. Factor's FFI is inspired by Common Lisp's CFFI [4]. The FFI can call functions written in C, Fortran and Objective C. Additional libraries exist to communicate with Lua, Javascript, and C++. To call a C function, the type of the fuction merely has to be declared, as below:

```
FUNCTION: SSL* SSL_new ( SSL_CTX* ctx ) ;
```

When calling foreign functions with dynamically-typed values, Factor automatically wraps and unwraps binary types when used as parameters or return values: simple integer and floating-point types convert automatically between boxed Factor representations and native binary representations. Binary data types, described in the next section, are unwrapped when used as arguments and allocated when returned from foreign functions. This automatic handling eliminates a class of bugs present in usages of other languages' FFI.

2.4.2 Binary Data Support

Factor provides extensive support for binary data, providing optimized data structures that can be manipulated as ordinary Factor objects. These binary data types are useful for both communicating with foreign functions and for use in pure Factor code, for greater performance.

Factor's library includes three main kinds of objects for aggregating and manipulating binary data.

- **Structs** Structured binary containers that provide slot accessors like Factor's tuple objects and are declared like tuples.
- **SIMD vectors** 128-bit hardware vector types represented and manipulated as constant-length sequences.

```
SPECIALIZED-ARRAYS: uchar float ;

TYPED: float>8bit-image (
    in: float-array
    --
    out: uchar-array )
    [ 0.0 1.0 clamp 255.0 * >integer ]
    uchar-array{ } map-as ;
```

Figure 12: A word definition with type annotations for input and output parameters

- **Specialized arrays** Packed arrays of a specified native type compatible with the library's sequences protocol.

Factor provides a fundamental set of binary types that mirror the basic C types. From these primitive types, structs, specialized arrays, and vectors can be constructed. New struct types extend this binary type system, allowing arrays of structs or structs containing structs to be instantiated. These objects all provide interfaces compatible with standard Factor sequences and tuples, so binary data objects can be used in generic code and manipulated with standard Factor idioms.

Due to the optimizing compiler (Section 3.4), manipulation of these data structures can approach the performance of C. The compiler provides primitives for loading, storing, and operating on native integer, floating-point, and vector types. When dynamic Factor objects of these types are not needed, the compiler can operate on them unboxed, keeping the values in machine registers. Code using these data structures can be written at a high level, operating on sequence and tuple-like objects, which the compiler transforms into C-like direct manipulation of binary data.

For example, the `float>8bit-image` word given in Figure 12 uses Factor's standard generic `clamp`, `*`, and `>integer` words along with the `map-as` sequence combinator to convert an array of floating-point image components with values ranging from 0.0 to 1.0 into eight-bit unsigned integer components ranging from 0 to 255. With the help of type annotations on just the word's input (Section 3.4.2), Factor generates efficient code without unnecessary overhead.

2.4.3 Scoped Resource Management

A common problem in garbage-collected languages is that, although memory management is handled automatically, there is no provision for automatically releasing external resources such as file handles or nework connections. As a result, code for working with external resources is still susceptible to resource leaks, resource exhaustion, and premature deallocation. Some languages support *finalizers*, which cause garbage collection of an object to run a user-supplied

```
: perform-operation ( in out -- ) ... ;

[
    "in.txt" binary <file-reader> &dispose
    "out.txt" binary <file-writer> &dispose
    perform-operation
] with-destructors
```

Figure 13: Destructors example

hook which freeing associated external resources. However, finalizers are inappropriate for some resource cleanup due to their nondeterminism–for example, the aforementioned file handles and network connections are limited system-wide resources and should be deterministically released independent of their in-memory object representations. Alternative deterministic mechanisms for resource management are detailed in Section 5

Factor's *destructors* library provides a mechanism for easy deterministic resource management. Any object with associated external resources can implement a method on the `dispose` generic word to release its resources. The `with-destructors` combinator creates a new dynamic scope and runs a supplied quotation. The quotation can register disposable objects by calling one of two words, `&dispose` or `|dispose`. The former word always disposes its parameter when the `with-destructors` form is exited, whereas the latter only disposes if the supplied quotation raises an exception. For example, Figure 13 opens two files and performs an operation on them, ensuring that both files are properly closed.

3. Implementation

Factor has an advanced high-performance implementation. The language is always compiled, using either a simple or optimizing compiler. Generic dispatch is optimized both by attempting to statically select a method and through polymorphic inline caches. Memory allocation is also optimized through a combination of static and dynamic techniques, using compiler optimizations to minimize allocation together with generational garbage collection to manage the cases that cannot be eliminated.

3.1 The Interactive Environment

Factor is accompanied by an interactive environment based around a read-eval-print loop. The environment is built on top of a GUI toolkit implemented in Factor. Graphical controls are rendered via OpenGL, and issues such as clipboard support are handled by an abstraction layer with backends for Cocoa, Windows, and X11. Developer tools provided include a documentation and vocabulary browser, an object inspector, a single stepper and a tool for browsing errors.

When developing a Factor program, it is useful to test different versions of the program in the interactive environment. After changes to source files are made on disk, vocabularies can be reloaded, updating word definitions in the current image. The word `refresh-all` is used to reload all files that have changed compared to the currently loaded version.

Most dynamic languages allow code reloading by processing definitions in a source file as mutating the dictionary of definitions. Whenever a definition is used, it is looked up at runtime in the dictionary. There are two problems with this approach:

- The late binding creates overhead at each use of a definition, requiring name lookup or extra indirection. Late binding also hinders optimizations such as inlining.
- Stale definitions remain in memory when definitions are subsequently removed from a source file, and the source file is reloaded. This potentially triggers name clashes, allows space leaks, and causes other problems.

In Factor, the parser associates definitions with source files, and if a changed source file is reloaded, any definitions which are no longer in the source file are removed from the running image. The optimizing compiler coordinates with the incremental linker capability provided by the VM to reconcile static optimizations with on-the-fly source code changes.

When compiling word bodies, the optimizing compiler makes assumptions about the class hierarchy, object layouts, and methods defined on generic words. These assumptions are recorded as *dependencies* and stored in an inverted index. When one or more words or classes are redefined inside a development session, this dependency information is used to calculate a minimal set of words which require recompilation.

After a word is redefined, the segment of the heap containing compiled code is traversed to update the callers of the word. This allows Factor to use early binding while maintaining the illusion of late binding.

Tuples use an array-based layout while remaining compatible with redefinition, giving the illusion of a more flexible layout. This is achieved by performing a full garbage collection when a tuple class is redefined, allocating different amounts of space for tuples based on what fields have been added or removed.

3.2 Architecture

The Factor implementation is structured into a virtual machine (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 file containing a memory snapshot, as in many Smalltalk and Lisp systems. The source parser manipulates the code in the image as new definitions are read in from source files. 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 pointer's payload. Large integers and floating point numbers are boxed in the heap; however, compiler optimizations can in many cases eliminate this boxing and store floating point temporaries in registers. 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 written 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 image generation tool is run from an existing Factor instance to produce a new bootstrap image containing the parser, object system, and core libraries. The Factor VM is then run with the bootstrap image, which loads a minimal set of libraries which get compiled with the base compiler. The optimizing compiler is then loaded, and the base libraries are recompiled with the optimizing compiler. With the optimizing compiler now available, additional libraries and tools are loaded and compiled, including Factor's GUI development environment. Once this process completes, the image is saved, resulting in a full development image.

3.3 Base Compiler

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.

3.4 Optimizing Compiler

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 flow in a similar manner to a block-structured programming language. Low-level IR represents control flow with a control flow graph of basic blocks. Both intermediate forms make use of single static assignment (SSA) form to improve the accuracy and efficiency of analysis [12].

3.4.1 Front End

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 constructed. The front end does not deal with local variables, as these have already been eliminated.

3.4.2 Soft Typing

When static type information is available, Factor's compiler can eliminate runtime method dispatch and allocation of intermediate objects, generating code specialized to the underlying 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 generate multiple specialized versions of the function, each assuming different input types, with dispatch at the entry point to choose the best-fitting specialization for the given inputs.
- Methods on generic functions propagate the type information for their dispatched-on inputs.
- Functions can be declared with static input and output types using the `typed` library.

3.4.3 High-level Optimizations

The three major optimizations performed on high-level IR are sparse conditional constant propagation (SCCP [45]), escape analysis with scalar replacement, and overflow check elimination using modular arithmetic properties.

The major features of our SCCP implementation are an extended value lattice, rewrite rules, and flow 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 inline expansions. Type functions are defined on many of the core language words.

SCCP is used to statically dispatch generic word calls by inlining a specific method body at the call site. This inlining generates new type information and new opportunities for constant folding, simplification and further inlining. In particular, generic arithmetic operations which require dynamic dispatch in the general case can be lowered to simpler operations as type information is discovered. Overflow checks can be removed from integer operations using numeric interval information. The analysis can represent flow-sensitive type information. Additionally, calls to closures which combinator inlining cannot eliminate are eliminated when enough information is available [16].

An escape analysis pass is used to discover object allocations 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 identifies integer expressions in which the final result is taken to be modulo a power of two and removes unnecessary overflow checks from any intermediate addition and multiplication operations. This novel optimization is global and can operate over loops.

3.4.4 Low-level Optimizations

Low-level IR is built from high-level IR by analyzing control flow 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 reconstructed using the SSA construction algorithm described in [8], with the minor variation that we construct pruned SSA form rather than semi-pruned SSA, by first computing liveness. 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 selection, and instruction scheduling.

The local value numbering pass eliminates common subexpressions and folds expressions with constant operands [9]. Following value numbering and copy propagation, a representation selection pass decides when to unbox floating 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 linear scan register allocation algorithm [43, 47]. Our variant does not take ϕ nodes into account, so SSA form is destructed first by eliminating ϕ nodes while simultaneously performing copy coalescing, using the method described in [6].

3.5 Evaluation

We compared the performance of the current Factor implementation with four other dynamic language implementations:

- CPython 3.1.2[2], the primary Python implementation.
- SBCL 1.0.38[3], a Common Lisp implementation.
- LuaJIT 2.0.0beta4[4], a Lua implementation.
- V8 (SVN revision 4752)[5], a JavaScript implementation.

To measure performance, we used seven benchmark programs from the Computer Language Benchmark Game [25]. Benchmarks were run on an Apple MacBook Pro equipped with a 2.4 GHz Intel Core 2 Duo processor and 4GB of RAM. All language implementations were built as 64-bit binaries. The JavaScript, Lua and Python benchmarks were run as scripts from the command line, and the Factor and Common Lisp benchmarks were pre-compiled into standalone images.[6]

The results are shown in Figure 14[7]. The benchmarks demonstrate that Factor's performance is competitive with other state-of-the-art dynamic language implementations. Factor's relatively good performance can be explained by various aspects of its implementation.

- The ahead-of-time optimizing Factor compiler (Section 3.4), in these cases, is able to eliminate the same overhead of dynamic language features that the V8 and LuaJIT JITs do at runtime.
- `binarytrees` is heavy in allocation, so Factor's array-based layout for tuples helps performance (Section 3.1).
- The Factor compiler can generate code using SIMD instructions (Section 2.4), improving performance on `nbody`.
- Native code generation makes Factor, like LuaJIT, SBCL and V8, significantly faster than Python on all but three benchmarks, where most of Python's time is spent in library functions written in C.

4. Evolution

The Factor language and implementation has evolved significantly over time. The first implementation of Factor was hosted on the Java Virtual Machine and used as a scripting language within a larger Java program. As a result, the first iteration of the language was rather minimal, with no direct

[2] `http://www.python.org`

[3] `http://www.sbcl.org`

[4] `http://luajit.org`

[5] `http://code.google.com/p/v8/`

[6] More details about the test setup can be found online at `http://factor-language.blogspot.com/2010_05_01_archive.html`.

[7] The Languge Benchmark Game lacks a Lua implementation of `regexdna`.

	Factor	LuaJIT	SBCL	V8	Python
`binarytrees`	1.764	6.295	1.349	2.119	19.88
`fasta`	2.597	1.689	2.105	3.948	35.23
`knucleotide`	1.820	0.573	0.766	1.876	1.805
`nbody`	0.393	0.604	0.402	4.569	37.08
`regexdna`	0.990	——	0.973	0.166	0.874
`revcomp`	2.377	1.764	2.955	3.884	1.669
`spectralnorm`	1.377	1.358	2.229	12.22	104.6

Figure 14: The time in seconds taken to run seven benchmarks on five language implementations

support for user-defined types, generic functions, local variables, or automatic inclusion of vocabulary dependencies.

As the focus of the language shifted from embedded scripting to application development, new features were added and existing features were redesigned to better support larger codebases. Rather than design language features upfront, new features have been added incrementally as needed for the compiler and standard library. Language changes can usually be implemented as user-level libraries and move into the core of the language only at the point where it is deemed useful to use the feature in the implementation of the language itself. This type of evolution is only possible because of Factor's extensive metaprogramming capabilities.

For example, hashtables and structures from cons cells were originally used in place of objects. To address the problems that this created in writing larger programs, a library for generic words and user-defined types was created. This was later moved into the core of the language, and the standard library now uses object-oriented techniques extensively.

The stack checker was initially optional and significant bodies of code did not pass the stack checker. Later, with the addition of `call( -- )`, it became practical to require all code to pass the stack checker. This change immediately led to the discovery and repair of numerous infrequent bugs in the standard library.

We moved away from the JVM as a host platform due to a lack of support for tail-call optimization, continuations, and certain forms of dynamic dispatch. The switch to a native implementation improved performance significantly. Compiler optimizations and virtual machine improvements have been added to address performance bottlenecks in running programs. As more advanced optimizations have been added, the time it takes to compile the Factor development environment has remained roughly constant, following [24].

5. Related Work

Others have approached the problem of eliminating stack effect bugs (Section 2.1.4) in terms of adding a full-fledged static type system to a concatenative language. StrongForth [2] adds a relatively simple system, and Cat [14] adds a more detailed type system including support for row polymorphism. The design of our stack checker is similar to the Java Virtual Machine's bytecode verifier pass, and the invariants imposed on Factor code are similar to those of the Java Virtual Machine specification [37].

Other languages have syntax for creating anonymous functions, as in Section 2.1.6. For example Clojure supports syntax like `#(+ 1 %)` short for `(fn [x] (+ x 1))` [31]. Here, the role of `%` is the opposite of `_` in Factor, representing the argument rather than the retained value.

Factor's object system does not distinguish a receiver in method calls. Other similar object systems include CLOS [5], Cecil [11] and Dylan [40]. CLOS, like Factor, allows the object system to be extended through a meta-object protocol [35], whereas Cecil is more restrictive.

Parsing words (Section 2.3.1), are similar to Forth's immediate words. One major difference between Forth and Factor is that in Forth, control flow is implemented with immediate words such as `IF` and `THEN`; in Factor, control flow is done with combinators. A second major difference is that whereas the Forth parser has two modes, *compile* and *interpret*, in Factor there is effectively no interpret mode; the parser always compiles code into quotations. Even code entered at the top-level is first compiled into a quotation, and then the quotation is immediately called and discarded. Eliminating the two modes from the Forth parser eliminates the need for so-called *state-smart* words [18].

Other languages provide mechanisms for resource management, but we believe these to be more difficult to use than Factor's mechanism (Section 2.4.3). In C++, *Resource Acquisition is Initialization* (RAII) is the technique of using constructors to acquire resources and destructors to dispose of them. A stack-allocated object is used to wrap the external resource handle; the object's destructor runs deterministically at the end of its scope and deallocates the resource when the object leaves scope. Common Lisp popularized the `with-` idiom: libraries implement scoped resource management by enclosing the scope of the allocated resource in a higher-order function such as `with-open-file`, which encapsulates acquiring and releasing the external resource. This approach doesn't scale very well if many resources need to be acquired and released in the same piece of code, due to the resulting nesting of `with-` functions. The C# [34] and Python languages offer special syntax for scoped external resources, C#'s `using` keyword and Python's `with` statement, that provide the same functionality as the Common Lisp idiom, but as a built-in language feature.

Factor's FFI (Section 2.4.1) is related to several other languages' FFIs. Common Lisp's CFFI [4] is the most similar and provided a basis for our work. Python Ctypes [21] provides a more dynamic interface, without the need to predeclare functions in a DLL before calling them. Newspeak Aliens [7] allow similar capabilities to CFFI but with the addition of an explicit object representing the capability to call a set of foreign functions, rather than an explicit global one. A common alternative to foreign function interfaces in dynamic languages is to create a plug-in for the VM making new primitives for functions in a foreign library [22], a process which SWIG automates [1]. However, such a system would not be compatible with Factor's interactive environment.

Other dynamic languages have provided limited support for manipulating packed binary data. Languages that offer binary data support often do not provide as good support as ordinary language constructs. This is either through a high-overhead extension library like Python's Struct [23] or OCaml's Bigarray [36], or as a limited extension of their FFI facilities geared more toward interfacing with native libraries than toward high-performance data manipulation. By contrast, Factor's support for binary data is much stronger (Section 2.4.2).

6. Conclusion

We have demonstrated Factor, a new dynamic stack-based object-oriented programming language. Factor incorporates features from many different previous languages systems into a new product combining their advantages. Factor has a very flexible object system and metaprogramming model, allowing the best coding style to be used for the job. It combines tools for dealing with bits and foreign function calls with high-level programming tools in the same language, offering the best of both worlds. Its advanced optimizing compiler makes it realistic to implement high-performance programs with few or no type declarations. Taken all together, these features make Factor a system that allows complex, high-performance programs to be constructed rapidly and easily.

7. Acknowledgments

We thank Eduardo Cavazos for the first implementations of dataflow combinators and pictured partial application syntax in Factor. We would also like to thank Doug Coleman for his technical and financial support of the Factor project, as well as the 50 other contributors to the Factor project. Additional thanks to the many people who reviewed this paper and helped us prepare it for publication.

References

[1] Dave Beazley and et al. Simplified wrapper and interface generator. `http://www.swig.org/`, 2010.

[2] Stephan Becher. StrongForth homepage. `http://home.vrweb.de/stephan.becher/forth/`, 2008.

[3] Marc Berndl, Benjamin Vitale, Mathew Zaleski, and Angela Demke Brown. Context Threading: A Flexible and Efficient Dispatch Technique for Virtual Machine Interpreters. In *In CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 15–26. IEEE Computer Society, 2005.

[4] James Bielman and Luís Oliveira. CFFI – The Common Foreign Function Interface. `http://common-lisp.net/project/cffi/`, 2010.

[5] Daniel G. Bobrow and Gregor Kiczales. The Common Lisp Object System metaobject kernel: a status report. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 309–315, New York, NY, USA, 1988. ACM.

[6] Benoit Boissinot, Alain Darte, Fabrice Rastello, Benoit Dupont de Dinechin, and Christophe Guillon. Revisiting Out-of-SSA Translation for Correctness, Code Quality and Efficiency. In *CGO '09: Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 114–125, Washington, DC, USA, 2009. IEEE Computer Society.

[7] Gilad Bracha, Peter Ahe, Vassili Bykov, Yaron Kashai, and Eliot Miranda. The newspeak programming platform. `http://bracha.org/newspeak.pdf`, 2008.

[8] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Softw. Pract. Exper.*, 28(8):859–881, 1998.

[9] Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. Value numbering. *Softw. Pract. Exper.*, 27(6):701–724, 1997.

[10] Robert Cartwright and Mike Fagan. Soft typing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292, New York, NY, USA, 1991. ACM.

[11] Craig Chambers. The Cecil language: Specification and rationale. Technical report, University of Washington, 1993.

[12] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13:451–490, 1991.

[13] Dibyendu Das and U. Ramakrishna. A practical and fast iterative algorithm for ϕ-function computation using DJ graphs. *ACM Trans. Program. Lang. Syst.*, 27(3):426–440, 2005.

[14] Christopher Diggins. The Cat Programming Language. `http://www.cat-language.com/`, 2007.

[15] ECMA. ECMAScript for XML (E4X) Specification, 2005.

[16] Daniel Ehrenberg. Closure elimination as constant propagation. Programming Language Design and Implementation, Student Research Contest, 2010.

[17] Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate Dispatching: A Unified Theory of Dispatch. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 186–211, London, UK, 1998. Springer-Verlag.

[18] M. Anton Ertl. State-smartness — Why it is Evil and How to Exorcise it. In *EuroForth '98*, 1998.

[19] Martin Odersky et. al. The Scala Language Specification. Technical report, EPFL Lausanne, Switzerland, 2004.

[20] Tim Bray et. al. Extensible Markup Language (XML) 1.0 (fifth edition). World Wide Web Consortium, 2008.

[21] Python Software Foundation. ctypes – A foreign function library for Python. `http://docs.python.org/library/ctypes.html`, 2010.

[22] Python Software Foundation. Python/C API Reference Manual. `http://docs.python.org/library/ctypes.html`, 2010.

[23] Python Software Foundation. struct – Interpret strings as packed binary data. `http://docs.python.org/library/struct.html`, 2010.

[24] Michael Franz. Compiler Optimizations Should Pay for Themselves. In P. Schulthess, editor, *Advances in Modular Languages: Proceedings of the Joint Modular Languages Conference*, 1994.

[25] Brent Fulgham. Computer Language Benchmarks Game. `http://shootout.alioth.debian.org/`, 2010.

[26] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[27] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification*. Addison Wesley, 3rd edition, 2005.

[28] Paul Graham. *On Lisp*. Prentice Hall, 1993.

[29] David Gudeman. Representing Type Information in Dynamically Typed Languages, 1993.

[30] Dominikus Herzberg and Tim Reichert. Concatenative Programming: An Overlooked Paradigm in Functional Programming. In *Proceedings of ICSOFT 2009*, 2009.

[31] Rich Hickey. Clojure. `http://clojure.org/`, 2010.

[32] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *ECOOP '91: Proceedings of the European Conference on Object-Oriented Programming*, pages 21–38, London, UK, 1991. Springer-Verlag.

[33] American National Standards Institute. X3.215-1994, Programming Language Forth, 1996.

[34] Ecma International. *Standard ECMA-334: C# Language Specification*. 4 edition, 2006.

[35] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. The Art of the Metaobject Protocol, 1991.

[36] Xavier Leroy. The Objective Caml system – Documentation and user's manual. `http://caml.inria.fr/pub/docs/manual-ocaml/`.

[37] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[38] Christophe Rhodes. SBCL: A Sanely-Bootstrappable Common Lisp. In *Self-Sustaining Systems: First Workshop, S3 2008 Potsdam, Germany, May 15-16, 2008 Revised Selected Papers*, pages 74–86, Berlin, Heidelberg, 2008. Springer-Verlag.

[39] Vivek Sarkar, Mauricio J. Serrano, and Barbara B. Simons. Register-sensitive selection, duplication, and sequencing of instructions. In *ICS '01: Proceedings of the 15th international conference on Supercomputing*, pages 277–288, New York, NY, USA, 2001. ACM.

[40] Andrew Shalit. *The Dylan reference manual: the definitive guide to the new object-oriented dynamic language*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.

[41] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2000.

[42] Dave Thomas, Chad Fowler, and Andy Hunt. Programming Ruby: The Pragmatic Programmers' Guide, 2004.

[43] Omri Traub. Quality and Speed in Linear-Scan Register Allocation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 142–151. ACM Press, 1998.

[44] Manfred von Thun. Rationale for Joy, a functional language, 2000.

[45] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13:291–299, 1991.

[46] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In *IWMM '92: Proceedings of the International Workshop on Memory Management*, pages 1–42, London, UK, 1992. Springer-Verlag.

[47] Christian Wimmer. Linear Scan Register Allocation for the Java HotSpot^{TM}Client Compiler. Master's thesis, Institute for System Software, Johannes Kepler University Linz, 2004.