Background Report: Factor as a Practical Language for Software Applications

Reggie Gillett
27133123
w6k8
reggie.gillett@gmail.com

Graham St-Laurent
23310121
i5l8
gstlaurent@gmail.com

Lynsey Haynes
12686119
a4h8
lynseyahaynes@gmail.com

Brittany Roesch
22015119
r1d8
brroesch@gmail.com

Gord Minaker
34615112
u4s8
gordonminaker@hotmail.com

ABSTRACT

Factor is a concatenative and object-oriented programming language. This paper explores some of the unique features of contatenative programming languages and Factor in particular. We investigate features of Factor necessary for the implementation of an interactive game, and discuss the implications of their use.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages

Keywords

Factor, stack-based languages, concatenative languages

1. INTRODUCTION

Our discussion centers around an exploration of Factor as a programming language. The impetus of our exploration will be an evaluation of Factor's elements and features in the context of developing a computer game. We will focus on what makes Factor unique and highlight aspects with particular value to the creation of a substantial programming project.

2. FEATURES OF FACTOR

2.1 Factor as a Concatenative Language

2.1.1 Stack Paradigm

A stack-based language uses an operand stack to pass function arguments and return function values.[1] Literals, like numbers and strings, can be thought of as functions with no arguments that push themselves onto the stack. Operators and functions are applied to values already pushed onto the stack. The stack gets used in replace of, and possibly in addition to, named variables. Thus function calls are referred to as "words", since they are not followed by their arguments.

Postfix Syntax

Factor uses a postfix syntax, with operands being written first, followed by their operator or word. The order of operations is not held, rather values and operators are evaluated in the order in which they come. The combination of an operand stack and postfix syntax can make algebraic expressions look complicated and unintuitive.[2] For example, the relatively simple expression:

```
f(x, y, z) = y2 + x2 - |y|
```

could be written as:

```
f = drop dup dup * swap abs rot3 dup * swap - +
```

in a postfix language such as Factor. Despite its initially steep learning curve, postfix syntax combined with a stack base are easy for a computer to compile and generally have good performance. And once accustomed to it, according to Factor creator Slava Pestov, it becomes quite readable. In his words (where he malaprops the bicycle simile):

> Learning a stack language is like learning to ride a bicycle: it takes a bit of practice and you might graze your knees a couple of times, but once you get the hang of it, it becomes second nature.[3]

Dataflow Combinators

The stack manages the data flow of the program.[4] Since all operators work on the items at the top of the stack, dataflow combinators can rearrange or remove elements on the stack as the programmer needs. They can also be used to perform an operation while keeping its input intact, or apply a single operator or sets of operators to multiple values.[5] For example, the dip combinators will ignore a specified number of values on the top of the stack to invoke an operator further down on the stack:

```
dip2 (x quot -- x )
```

This hides the first element of the stack temporarily to apply the second element. The most prevalent dataflow combinators in Factor are dup (duplicates a stack element), drop (pops a stack element) , and swap (exchanges the first and second elements).[6] Dataflow combinators would no doubt be essential to a game built in Factor.

2.1.2 *Lexical Scope*

In a lexically-scoped language, the value of an identifier can be completely determined by looking at the program text, meaning the analysis can be done statically. Lexical resolution is determined at compile time and is also known as early binding. Most modern languages are lexically scoped, as developers and maintenance programmers have an easier time understanding lexically scoped languages, and compiler developers have an easier time writing efficient compilers. There are still a few dynamically scoped languages in common usage though; PostScript, the programming language which runs on printers, is perhaps the most commonly used of them.[7] Factor initially did not provide support for lexical scoping, but it has since been implemented as default. Adopting lexical scope enables two major changes in the programming language. It makes it easy to construct function closures and to construct objects with mutable state.[8]

Quotations

In Factor, a quotation is enclosed within square brackets and contains a sequence of literals and words. The following example demonstrates that the quotation is passed as the single object onto the stack.

```
[ "hello" . ]

--- Data stack:
[ "hello" . ]
```

But quotations do not keep track of the environment. In Pestov's words:

> Quotations do not close over any lexical environment; they are entirely self-contained, and their evaluation semantics only depend on their elements, not any state from the time they were constructed. So quotations are anonymous functions but not closures.[9]

Lexical Closures

To define a word which uses lexically scope variables, you use `::` to define a word which internally performs all the necessary rewriting to make closure conversion and lexical scoping work.[10] (Normal function definition uses single semicolon `:`, instead.) You follow the word name with a list of locals enclosed in pipe ('|') characters and use these locals anywhere in the body of the word. 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. The following is an example word using locals:

```
:: add-test | x y z | x y + z + ;
1 2 3 add-test .
  6
```

Also within the scope of a :: definition, additional lexical variables can be bound using the :> operator, which binds either a single value from the data stack to a name, or multiple stack values to a list of names surrounded by parentheses.[11] For example, if you were to provide the following code within a word body,

```
1 2 + :> x
```

the name 'x' would be bound to the value 3 and is in scope within the body anywhere after the initial binding.

All locals are stored on the retain stack, which is separate from the data stack. This means that internally it forms all of the inputs of a lambda into an array, which is then moved to the retain stack. Named parameter accesses are rewritten into the code which moves this parameter array to the data stack, pulls out the right element and moves it back to the retain stack [12].

In a stack-based language, lexical variables, and in particular lexically-scoped closures are a useful extension of the concatenative paradigm. They can particularly become helpful when there is no obvious solution to a problem in terms of re-arranging operands at the top of the stack [13].

2.1.3 *Point Free Style*

As stated above, 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. Point-free style is utilized by all known concatenative languages and is a programming paradigm in which function definitions do not identify the arguments (or "points") on which they operate. Instead the definitions merely compose other functions, among which are combinators that manipulate the arguments.[14]

For example, consider this sequence of operations in a hypothetical applicative language (that closely resembles Python):

```
def example(x):
   y = foo(x)
   z = bar(y)
   w = baz(z)
   return w
```

Another way of writing that emphasizes the function composition:

```
def example(x):
    return baz(bar(foo(x)))
```

Notice how `baz` is the last thing that function to get called, but it occurs first—i.e., it is the leftmost. Now let's see how the same series of functions would look in Haskell using point-free style and function composition:

```
example = baz . bar . foo
```

Notice the lack of parameters (the `.` is Haskell's way of signifying composition), as well as the fact that the lastmost function to be applied is the first one to appear. This can intuitively be understood as saying "`example` is the composition of `baz`, `bar`, and `foo`." In a concatenative language like Factor, however, this same sequence would look something like this:

```
: example   foo bar baz ;
```

Once again, no parameter appears so it is the function composition that is emphasized. But in this case, the functions appear in the order that they are called, so the intuitive reading is simply "`example` is a `baz`, a `bar`, and a foo", which is a subtly different perspective. We find that this postfix perspective emphasizes the process, whereas the other, prefix perspective emphasizes the product as a whole.

Regardless of the perspective, however, there are clear benefits to implementing code in point-free style. One advantage is that it allows the programmer to be succinct. It also allows the user to think of what the code is doing with the image of *composing functions* rather than imagining the process of arguments flowing through.

3. FEATURES REQUIRED FOR EFFECTIVE SOFTWARE DESIGN

3.1 Control Flow Structures

Central to the programming of any game are control flow structures. Loops and conditional expressions are integral to the construction of any game, simple or complex. For example, in virtually all games there exists code to interpret user input. Typically this consists of a single or series of if statements which execute particular actions based on a key pressed by the user.[15] For even the simplest games this is fundamental to their implementation. Take for an example a simple number guessing game, at some point the program will have to evaluate if the number entered by the user matches the number to be guessed. This control pattern is often included in a *game loop*, another essential piece of most games rooted in control flow.[16]. The *game loop* executes continuously processing user input, updating the game state and rendering the game on each turn.[17]

All control flow structures in Factor are expressed using higher-order functions, functions which take functions as arguments. In the tradition of other stack-based languages such as Joy, these are referred to as *combinators*. Combinators operate like any other word in Factor, taking their arguments from the stack, however unlike other words, these arguments are quotations rather than literals.[18] The previously-mentioned example of a number guessing game gives the example illustrated below, which illustrates the use of the if combinator:

```
: guessed-number ( n -- )
   secret-number  =
   [ "You guessed it!" print ]
   [ "Nope, that's not it" print ]
   if ;
```

It assumes our secret number is a constant defined elsewhere. We can see that if takes 3 arguments, a boolean and two quotations. Any object in the place of the boolean other than the special object f evaluates to true.[19]

All control Flow in Factor is achieved using the if combinator and recursion.[20]. Factor contains several looping combinators in its standard library which give the effect of iteration (`each`, `filter`, `map`, etc). All of these use recursion or branching and recursion to achieve their effects.[21]

3.2 User Interface

Any game will require the user to interact with the machine, so there will always be some sort of user interface. Factor provides a vocabulary implementing a cross-platform, object-based Graphical User Interface,[22] so using Factor for our game should be relatively painless.

In Factor's User Interface vocabulary, graphical control elements are of the type `gadget`, with the outermost gadget—the window itself—being subtype of gadget called a `world`. Ultimately, the world is simply the topmost node in a tree of gadgets. The UI is implemented using OpenGL, and Factor does provide a vocabulary with bindings to OpenGL, so full-featured 3D graphics are possible, as well as more basic 2D graphics. Although the gadget class is open-ended and there are many developer options for the UI, it does come with some basic types such as the `label`, `button`, or `editor`. Depending on time, we may try to implement some graphics, but simply providing a text area to read from, and an area that includes text output may be all we get to do.

Consequently, since Factor also provides built-in words that access *standard output* and *standard input*, instead of taking a GUI route, we may alternatively end up creating a text-based console application.

3.3 Unit Testing

In the development of any non-trivial software application, unit testing is a necessary first step to ensure error-free code. A programming language which lacks a unit-testing framework should be quickly forgotten. Factor has a fully adequate unit-testing framework that so that testing can begin quickly without installing or configuring external frameworks.

By simply typing "<*vocabulary-name*> `scaffold-tests`" into the listener, a test file is created. Within the test file, tests are typically written in the following format:

```
[<expected-output>] [ <input> ] unit-test
```

Tests can be run at any time from the listener by typing "<*vocabulary-name*> `test`". Failing tests are able to be debugged using the built-in error list tool.[23]

3.4 Object Orientedness

Factor is an object oriented programming language whose object system is unique from other common languages. In Factor, every value is treated as an object where an object can belong to an instance of one or more classes.[24] Each object holds a type value along with components and their instances are stored within words. Here we will consider the definition, instantiation, and modification of objects in Factor.

3.4.1 Object Class Definitions: Tuples & Slots

Classes in Factor are defined in terms of Tuples which hold objects belonging to the class in 'Slots'. Like other OO languages, classes can inherit from specified superclasses and the "final" designation is used for classes which should not be further subclassed. A typical class definition in Factor is written in the format:

```
TUPLE: <class-name> <slot1> <slot2> ... ;
```

As an example:

```
TUPLE: student sid name gpa ;
```

Defines a new class named 'student' which has fields, termed "slots", for the values of sid, name, and gpa. Unlike other object systems, methods cannot belong to classes in Factor. [25]

3.4.2 *Object modification - Methods*

Similar to the Common Lisp Object System (CLOS), methods in Factor cannot belong to classes. Rather, a structure termed a "generic word" defines zero or more methods all with the intention of performing the same functionality, normally using a different implementation for each specified input class type. When a generic word is referenced in the program, it will execute the method whose signature is most appropriate for the input to the call.[26]

4. SUMMARY

While the unique syntax of Factor might serve as initially as a deterrent to adopting the language for more ambitious software applications, if one persists its many benefits quickly become evident. Lexical scope gives Factor flexibility that can be lacking in other stack-based languages. Its point-free style allows for succinct code which highlights function composition. It is also worth noting that its syntax, while challenging at first, is easy for the computer to compile.

Through our investigations it has become evident that Factor is a language suited to not only creating a game, but to any software application. It possesses all the capabilities of any other functional language with a full compliment of libraries to extend its already rich feature set.[27] As soon as we becomes acclimatized to Factor's more idiosyncratic qualities, we expect to receive a euphoric joy while programming with it. One might describe it as quirky but powerful. It is with confidence that the authors of this paper agree that Factor is suited to the creation of any substantial program.

Notes

[1]Pestov, S., Ehrenberg, D., Groff, J.:*Factor: a dynamic stack-based programming language.* In: DLS 2010 Proceedings of the 6th Symposium on Dynamic Languages (2010)

[2]The Big Mud Puddle: Why Concatenative Programming Matters <http://evincarofautumn.blogspot.ca/2012/02/why-concatenative-programming-matters.html>

[3]http://docs.factorcode.org/content/article-cookbook-philosophy.html

[4]Factor/FAQ <http://concatenative.org/wiki/view/Factor/FAQ>

[5]Dataflow combinators - Factor Documentation (Dataflow combinators - Factor Documentation) <http://docs.factorcode.org/content/article-dataflow-combinators.html>

[6]Factor: a practical stack language: Prevalence of shuffle words and dataflow combinators <http://factor-language.blogspot.ca/2008/12/prevalence-of-shuffle-words-and.html>

[7]What is lexical scoping? (Fabulous adventures in coding) <http://ericlippert.com/2013/05/20/what-is-lexical-scoping/>

[8]Gentleman, R., & Ihaka, R. Lexical Scope and Statistical Computing. Retrieved from <http://cran.r-project.org/doc/misc/lexical.tex>

[9]Factor: a practical stack language: How Factor implements closures <http://factor-language.blogspot.ca/2010/01/how-factor-implements-closures.html>

[10]Pestov, S., Ehrenberg, D., Groff, J.: *Factor: a dynamic stack-based programming language.* In: DLS 2010 Proceedings of the 6th Symposium on Dynamic Languages (2010)

[11]Dataflow combinators - Factor Documentation (Dataflow combinators - Factor Documentation) <http://docs.factorcode.org/content/article-dataflow-combinators.html>

[12]Factor: a practical stack language: How Factor implements closures <http://factor-language.blogspot.ca/2007/03/adding-named-parameters-to-factor.html>

[13]Pestov, S., Ehrenberg, D., Groff, J.: *Factor: a dynamic stack-based programming language.* In: DLS 2010 Proceedings of the 6th Symposium on Dynamic Languages (2010)

[14]Tacit programming (Wikipedia) <http://en.wikipedia.org/wiki/Tacit_programming>

[15]Game Programming Patterns / Design Patterns Revisited <http://gameprogrammingpatterns.com/command.html>

[16]Game Programming Patterns / Design Patterns Revisited <http://gameprogrammingpatterns.com/command.html>

[17]Game Programming Patterns / Design Patterns Revisited <http://gameprogrammingpatterns.com/game-loop.html>

[18] Pestov, S., Ehrenberg, D., Groff, J.: *Factor: a dynamic stack-based programming language.* In: DLS 2010 Proceedings of the 6th Symposium on Dynamic Languages (2010)

[19]Control flow cookbook - Factor Documentation (Control flow cookbook - Factor Documentation) <http://docs.factorcode.org/content/article-cookbook-combinators.html>

[20]Pestov, S., Ehrenberg, D., Groff, J.: *Factor: a dynamic stack-based programming language.* In: DLS 2010 Proceedings of the 6th Symposium on Dynamic Languages (2010)

[21]Factor: a practical stack language: Prevalence of shuffle words and dataflow combinators <http://factor-language.blogspot.ca/2008/12/prevalence-of-shuffle-words-and.html>

[22]http://docs.factorcode.org/content/article-ui.html

[23]Unit testing - Factor Documentation (Unit testing - Factor Documentation) <http://docs.factorcode.org/content/article-tools.test.html>

[24]Factor - Progopedia <http://progopedia.com/language/factor/>

[25]Control flow cookbook - Factor Documentation (Control flow cookbook - Factor Documentation) <http://docs.factorcode.org/content/article-cookbook-combinators.html>

[26]Double, C. (2013, April 22). Factor Articles - Work in Progress. Retrieved from <http://bluishcoder.co.nz/factor-articles.pdf>

[27]Libraries - Factor Documentation (Libraries - Factor Documentation) <http://docs.factorcode.org/content/article-handbook-library-reference.html>