CONCATENATIVE PROGRAMMING
An Overlooked Paradigm in Functional Programming

Dominikus Herzberg
Department of Software Engineering, Heilbronn University
Max-Planck-Str. 39, 74081 Heilbronn, Germany

Tim Reichert
School of Computing, Engineering & Information Sciences, Northumbria University
Pandon Building, Camden Street, Newcastle Upon Tyne, U.K.

Keywords: Language-oriented programming, Functional programming, Concatenative languages.

Abstract: Based on the state of our ongoing research into Language-Driven Software Development (LDSD) and Language-Oriented Programming (LOP) we argue that the yet relatively unknown paradigm of concatenative programming is valuable for fundamental software engineering research and might prove to be a suitable foundation for future programming. To be sound, we formally introduce Concat, our research prototype of a purely functional concatenative language. The simplicity of Concat is contrasted by its expressiveness and a richness of inspiring approaches. Concatenative languages contribute a fresh and different sight on functional programming, which might help tackle challenges in LDSD/LOP from a new viewpoint.

1 INTRODUCTION

One of our main themes of research is Language-Driven Software Development (LDSD) and Language-Oriented Programming (LOP). It is about how the creation and use of languages might help us in building and engineering complex software systems. As a matter of fact, LDSD/LOP is a growing field of interest as is manifested by the research on Domain Specific Languages (DSLs), Model-Driven Development (MDD), generative software development and software factories, to name just a few areas.

In order to experiment with language layers and domain specific specializations and to test our conceptions and hypotheses, we made use of languages which are regarded as flexible and easily adaptable. Among these languages were Lisp/Scheme, Prolog and Smalltalk. Their interactive nature and their late-binding features due to dynamic typing turned out to be helpful in the setting of a "laboratory situation" for language experimentation. Still, some experiments turned out to fail e.g. applying extreme refactoring or attempting to uncover hidden design intentions in code. We felt having something in our way; a problem we could neither clearly pinpoint nor sketch a solution for. There was something "wrong" with the languages we used.

When we made contact with so-called concatenative programming languages things began to fall into place. As a result, we developed our own concatenative language called Concat. We benefited a lot from using the concatenative paradigm and still do; our work on Concat is research in progress. Concat is a language that is "as simple as possible, but no simpler" – to paraphrase a quote attributed to Albert Einstein – but still useful and practical.

Our claim is that concatenative languages are (a) ideally suited for language experimentation and (b) worth to be applied in software engineering because of its unique features.

The features that characterize and distinguish Concat in particular and concatenative languages in general are:

- Concat is a *functional language* (no explicit states) with static types and type inference. A concatenative language can also be dynamically typed and work without type inference; some variants are also functionally impure

Herzberg D. and Reichert T. (2009).
CONCATENATIVE PROGRAMMING - An Overlooked Paradigm in Functional Programming.
In *Proceedings of the 4th International Conference on Software and Data Technologies*, pages 257-262
DOI: 10.5220/0002281402570262

- Concat is a language one can interactively work with on the console; we regard *interactivity* as essential for an experimental approach to LDSD and LOP
- Concat is *homoiconic*, i.e. code can be treated as data and data as code
- Concat has a very *simple syntax*. Programs are created by concatenating words and so-called quotations, and there are just three tokens with special meaning: whitespace, [and]
- Similarily, Concat has very *simple semantics*. We distinguish the level of words and quotations from the level of functions processing stacks
- Both levels of Concat maintain a relationship called a *homomorphism*; that means that there is a structure preserving mapping from the syntactic level of words and quotations to the semantic level of functions and stacks

There are immediate implications that follow from these characteristics: (1) Concat has a sound mathematical foundation, which enables formal treatment and reasoning over programs. (2) There are no variable bindings in Concat, that means there are no structural ties beyond the homomorphism mentioned. And that has two other important consequences especially for code engineering: (3) Concat supports macros out of the box without further ado. (4) One can cut out any fragment of code at whitespaces. Presumed that you leave the code within squared brackets intact, any such fragment still represents a valid program. This is something, which is impossible in, say, Java, C#, Lisp or Haskell. Concat enables code reuse and refactoring of code to an extent unknown in other languages.

We think that Concat offers many interesting properties. We formally define Concat in Sec. 3 after we have briefly touched upon related work in Sec. 2. We hold the view that concatenative languages deserve much more attention than is the case. They are inspiring, usable and practical despite and because of their simplicity, see Sec. 4 – a position surely debatable. We draw some conclusions in Sec. 5.

2 RELATED WORK

Much of the foundational work on concatenative languages was done by Manfred von Thun in conjunction with the development of the Joy language.[1] Today, several implementations of concatenative languages exist. Cat is a purely functional language that unlike Joy and like Concat supports static type checking.[2] Factor is a programming language designed for use in practice. It has a concatenative core and supports object-oriented programming.[3]

Concatenative languages are closely related to stack-based languages.[4] The former are characterized by the homomorphic relationship between words/ quotations and functions, the latter by the use of a stack as the central concept in the execution model. A language may be both stack-based and concatenative, but this must not necessarily be the case. Forth (Rather et al., 1996) and PostScript (Adobe Systems Inc., 1999) are popular "high-level" stack-based languages that are not concatenative. Several assembly and intermediate languages also use a stack-based model of execution.

In a concatenative language, even those words that may intuitively be perceived as data, for example numbers and strings, denote functions. Thus, concatenative languages are not only functional in the sense that functions have no side effects, but also in the sense that "everything is a function". This form of purity and the non-existence of variables relates them closely to function-level programming as defined in (Backus, 1978) and the point-free style of functional programming (Gibbons, 1999).

3 FORMAL FOUNDATIONS

In this section we will define the concatenative language Concat. Due to space limitations we restrict our presentation to a dynamically typed version of Concat. Actually, Concat is statically typed enabling the programmer to define arbitrary types as encodings.

A specialty of concatenative languages is that there is the level of words and quotations (Sec. 3.1 and 3.2) and the level of functions and stacks (Sec. 3.3 and 3.4). Both levels have their own concepts and their own semantics. However, the levels are constructed in such a way that there is a close relationship between the two (Sec. 3.5).

3.1 Words and Quotations

On the syntactic level, Concat is defined by only some few concepts: vocabularies of words, quotations, stack pools, programs, concatenation and substitution.

[1] http://www.latrobe.edu.au/philosophy/phimvt

[2] http://www.cat-language.com

[3] http://factorcode.org

[4] http://concatenative.org

Definition 3.1 (Vocabulary of Words). A *vocabulary* is a set of elements $V = \{w_1, w_2, \dots\}$; its elements are called *words*.

A quotation is recursively defined as:

Definition 3.2 (Quotation). Let $[\,]$ be the *empty quotation*. Given a vocabulary V of words, a *quotation* q using V is a finite sequence of elements written as $q = [s_1\ s_2\ s_3\ \dots]$ with each element being either a word of V or a quotation using V.

Definition 3.3 (Stack Pool). Given a vocabulary V, the *stack pool* S_V is the set of all possible quotations using V.

Definition 3.4 (Program). A program in a concatenative language is an element p of the program space P_V, $p \in P_V \subseteq S_V$.

Any program is a quotation but not every quotation is a (valid) program.

Definition 3.5 (Concatenation). The binary operation of *concatenation* $\oplus : S_V \times S_V \to S_V$ corresponds to list concatenation.

The concatenation of two programs results in a new program. The following properties hold with $p, p', p'' \in S_V$:

$$p \oplus [\,] \Leftrightarrow [\,] \oplus p \Leftrightarrow p$$

$$(p \oplus p') \oplus p'' \Leftrightarrow p \oplus (p' \oplus p'') \Leftrightarrow p \oplus p' \oplus p''$$

The first property declares the empty quotation as the neutral element of concatenation, the second property is the law of associativity. Programs and their concatenation constitute a monoid.

For the sake of a simpler notation, we replace the concatenation operator $\oplus$ by a whitespace character, and do not use the outer squared brackets for programs.

Definition 3.6 (Substitution Rule). Given a vocabulary V, a *substitution rule* r is a unique mapping from one program to another program: $r : S_V^n \to S_V^m$ with $m, n \in \mathbb{N}\backslash\{0\}$.

Now we have everything together to define a generic substitution system that rewrites concatenations. The execution semantics are fairly simple.

Definition 3.7 (Substitution Evaluation). Given a sequence of substitution rules and a program, *substitution evaluation* is defined as follows: walk through the sequence of substitution rules, rewrite the program if there is a match (probing from right to left!) and repeat this process until no more substitution rules apply.

Before we advance to the level of functions, we would like to provide some simple examples of substitution rules. The attentive reader might notice that the rules look very much like operators written in postfix position. This is for a good reason, which will become clear when we talk about the connection to the function level. The rightmost position on the left-hand side of a substitution rule almost always is a word. Substitutions essentially dispatch from right to left.

3.2 Examples of Substitution Rules

Substitution rules have a left-hand side (LHS) and a right-hand side (RHS). Inside substitution rules, capital letters prefixed by a `$`, `#` or `@` denote variables used for matching words and quotations on the LHS and for value replacement on the RHS. If prefixed by `$`, the variable matches a single word only. If prefixed by `#`, a single word or quotation is matched. If prefixed by `@`, any number of words or quotations is matched.

The following rule defines a swap operation. Remember that the concatenation operator $\oplus$ is replaced by whitespace for improved readability and that the outer squared brackets are implicit:

```
#X #Y swap ==> #Y #X
```

On the LHS `#X` and `#Y` match the two words or quotations preceding `swap` in a given concatenation. On the RHS, the recognized words or quotations are inserted in their corresponding places. Take for example the concatenation "`2 [ 3 4 ] swap`", which is resolved by applying the above rule to "`[ 3 4 ] 2`".

The following substitution rules might help get an idea how simple but powerful the substitution system is.

```
[ @REST #TOP ] call ==> @REST #TOP
[ @X ] [ @Y ] append ==> [ @X @Y ]
true [ @TRUE ] [ @FALSE ] if ==> [ @TRUE ] call
false [ @TRUE ] [ @FALSE ] if ==> [ @FALSE ] call
#X dup ==> #X #X
```

The first rule, `call`, calls a quotation by dequoting it i.e. by releasing the content of the quotation. Syntactically, this is achieved by removing the squared brackets. The second rule, `append`, takes two quotations (including empty quotations) and appends their content in a new quotation. The third and fourth rule define the behavior of `if`. If there is a `true` followed by two quotations, the quotation for truth is called; if `false` matches, the failure quotation is called. Apparently, quotations can be used to defer execution. The last rule simply duplicates a word or quotation.

Due to space limitations we cannot show nor prove that some few substitution rules suffice to have a Turing complete rewriting system. Substitution

rules play the role macros have in other languages such as Lisp/Scheme.

3.3 Functions and Stacks

The concepts on the semantic level of functions and stacks parallel the concepts on the syntactic level. On the one hand there are words, quotations and concatenations, on the other hand there are functions, quotation functions and function compositions. The empty quotation is mapped on the identity function. We will discuss this structural similarity in Sec. 3.5.

Definition 3.8 (Pool of Stack Functions) . Given a stack pool S_V, a pool of *stack functions* $\mathcal{F}(S_V, S_V)$ is the set of all stack functions $f : S_V \to S_V$.

Definition 3.9 (Quotation Function) . Given a quotation $q \in S_V$, the corresponding *quotation function* $f_q \in \mathcal{F}(S_V, S_V)$ is defined to be

$$f_q(s) \to s \oplus q \oplus \text{append} \quad \forall s \in S_V$$

A function that throws its representation as a word onto a stack is called *constructor function* or *constructor* for short.

Definition 3.10 (Function Composition) . Given a stack pool S_V, the *composition* of two functions $f, g \in \mathcal{F}(S_V, S_V)$ with $f : A \to B$, $g : B \to C$ and $A, B, C \subseteq S_V$ is defined by the *composite function* $g \circ f : A \to C$. The following properties hold with $f, g, h \in \mathcal{F}(S_V, S_V)$:

$$f \circ Id \Leftrightarrow Id \circ f \Leftrightarrow f$$

$$(h \circ g) \circ f \Leftrightarrow h \circ (g \circ f) \Leftrightarrow h \circ g \circ f$$

That means that Id is the neutral element of function composition and that function composition is associative. Function composition constitutes a monoid as well.

From the above definition of function composition we can deduce the definition of the identity function:

Definition 3.11 (Identity Function) . For any vocabulary V, the *identity function* $Id \in \mathcal{F}(S_V, S_V)$ is defined as $Id(s) \to s$ for all $s \in S_V$.

We can now compute results with a given composition of stack functions and quotation functions.

Definition 3.12 (Function Evaluation) . Given a stack pool S_V, the *evaluation* of a function $f \in \mathcal{F}(S_V, S_V)$ with $f : A \to B$ and $A, B \subseteq S_V$ is defined to be the application of f on some $s \in A$: $f(s)$.

3.4 Examples of Function Definitions

In the context of functions, we refer to quotations as *stacks*. A function in Concat expects a stack and returns a stack. It will take some values from the input stack, do some computation and possibly leave some results on the input stack to be returned.

Function definitions look like substitution rules. The rightmost position on the LHS is a word denoting the name of a function. Everything else that follows to the left are pattern matchers picking up words and quotations from the stack. The position next to the function name stands for the top of the stack, then comes the position underneath etc.

```
$X $Y + ==> #<(+ $X $Y)>#
```

In the example, `$Y` expects a word on top of the stack and picks it up; `$X` picks up the word underneath. If there are more words or quotations on the stack, they are left untouched. The RHS of a function definition says that the top two values on the input stack are replaced by a single new word or quotation, which is the result of some computation enclosed in `#<` and `>#`.

Within these delimiters a computation can be specified in any suitable language; we use Scheme in this example. Before the computation is executed, the items picked up by `$X` and `$Y` are filled into the template at the corresponding places.

A function definition can also be read as having a so-called *stack effect* (and so can substitution rules): The function `+` takes two words from the stack and pushes a single word onto the stack. Looking at stack effects helps in selecting functions that fit for function composition. A function or composite function cannot consume more items from a stack than there are.

If a suitable language for specifying computations is used, function composition can be directly implemented by combining and rewriting the templates. That is one reason why we have chosen Scheme as the specification language for functions. To provide an example, take the definition to compute the inverse of a number:

```
$X inverse ==> #<(/ 1 $X)>#
```

The concatenation "`+ inverse`" can – on the functional level – be automatically derived as a composite function:

```
$X1 $X2 + inverse ==> #<(/ 1 (+ $X1 $X2))>#
```

Here, `$X1` and `$X2` are automatically generated by Concat. If template rewriting is too complicated to achieve in another target language, the behavioral effect of function composition can be simulated by passing a stack step by step from one function to another.

Any of the above substitution rules (Sec. 3.2) can also be defined as function definitions. One example, although rather trivial, demonstrates this for `dup`. The

two values pushed onto the stack are “computed” on the function level.

```
#X dup ==> #< #X >#  #< #X >#
```

3.5 Connecting the Levels

The previous section already indicates that the level of words and quotations (the syntactic level of Concat) and the level of functions and stacks (the semantic level) are connected. As a matter of fact, we establish a mapping from programs to functions and from concatenation to function composition. Mathematically speaking, this is called a *homomorphism*.

There is a subtle detail. The homomorphism implies that the search strategy looking for substitution matches must scan a concatenation from right to left. On each word or quotation we look through the list of substitution rules top down for a match. After a successful substitution has occurred, the search might continue to the left or start over again at the right. The first way (continuing) has the same effect, function composition has. The second way (starting over) is equivalent to passing a stack from function to function i.e. without really making use of function composition. Either way, the lookup for matching substitutions has to restart top down again.

When writing programs in Concat, we have the choice to either define substitutions that work on a purely syntactical level by rewriting concatenations of words and quotations. Or we define functions whose operational behavior is outside the reach of Concat – it is done in another computational world Concat has only an interface with but no more. For Concat, functions and composite functions are black boxes. Interestingly, we can seamlessly combine substitution and function evaluation.

The two approaches to interpret a given concatenation lead to two different readings. Take the following example, a simple addition:

```
3 0 +
```

Notationally, all there is are words and quotations. Assumed that there is the substitution rule “`$X 0 + ==> $X`”, the result on the word/quotation level is mechanically retrieved as `3`. On the level of functions, all there is are functions and stacks. So `3` is a constructor function that takes a stack and pushes a unique representation of itself – the word(!) `3` – onto the stack and returns the changed stack. So does `0`. Taken together with the function `+`, function composition results in a function accepting some stack and leaving `3` on top. On this level, we experience a stack being passed from function to function. This is also called *trace mode*.

A slightly more complicated example is the following concatenation:

```
6 5 dup [ 3 > ] call [ + ] [ * ] if
```

The word `dup` duplicates `5` and `call` unquotes `[ 3 > ]`, leading to `6 5 5 3 > [ + ] [ * ] if`. Assumed that a function definition for `>` (greater than) is given, `5 3 >` results in `true`. Now `if` rewrites the concatenation to `6 5 [ + ] call`. The result of `6 5 +` is `11`.

4 THINKING CONCATENATIVE

The following subsections aim to inspire the reader of the richness that lurks behind the concatenative paradigm. We barely scratch the surface on a subject worth further investigation.

4.1 Pattern Recognition Agents

The input to Concat can be viewed as a static but possibly very long sequence of words and quotations. Substitution rules and function definitions could be viewed as agents working on the input. Each agent has some sensors that allow the agent to recognize a set of specific subsequences of words/quotations somewhere in a program. If a pattern is recognized, the agent takes the input sensed, transforms it into a new sequence of words and quotations and replaces the input by the transformation.

Essentially, there is a pool of agents ready to process any subsequence they find a match for. This model has some similarities with biochemical processes. Let us take protein biosynthesis in a cell of a living organism as an example. After a copy of the DNA has been created (transcription), complex organic molecules called ribosomes scan the code of the DNA copy in a way comparable to pattern matching. The ribosomes read a series of codons as an instruction of how to make a protein out of an sequence of amino acids (translation). These processes could be understood as numerous computing agents working together.

It is an interesting observation that Concat can be used in a way that is close to how nature works in creating complex living systems. This might inspire a lot of interesting and interdisciplinary research questions. Also cognitive processes rely very much on pattern recognition.

4.2 Stream Processing

Another, dynamic view is to regard the input to Concat being continuously filled with new words and quo-

tations at the outmost right and added to the top of the stack, respectively. A continuous stream of words and quotations flows in. With a certain lookahead Concat applies substitution rules and function definitions as usual. Any context information needed for processing the stream must be either left on top of the stack. Or we introduce a "meta-stack", with the stream-stack being on top, so that context information can be left somewhere else on the meta-stack.

We have built a system called channel/filter/rule (CFR) for advanced protocol analysis in computer networks (Reichert et al., 2008). The incoming stream of data stems from a recording or a live trace of a monitoring device intercepting the communication of two or more interacting parties. Stateless selection (filters) and stateful processing (rules) help in abstracting and extracting information that represent the information flow on the next protocol layer (channel). We suspect that such a stream processing system can be easily realized with Concat. We have not done a prototype, yet. This is research in progress.

4.3 Refinement & Process Descriptions

Refinement is a very important notion in computer science, especially in the formal and theoretical branch. As a matter of fact, refinement is a well-understood concept. The formalization of refinement dates back to the 1970s. Refinement is a means to reduce underspecification. A specification S_2 is said to refine the behavior of a specification S_1 if for each input the output of S_2 is also an output of S_1. Semantically, this notion is captured by logical implication. The denotation $[\![S_2]\!]$ implies $[\![S_1]\!]$.

Programming languages typically do not support refinement. However, it is trivial to provide refinement in Concat, because it is built in: the notion of refinement is captured by unidirectional substitution.

Some researchers bring the notion of refinement and software development processes explicitly together; one prominent example is FOCUS (Broy and Stølen, 2001). In a yet unpublished paper we show that it is straight forward to formally describe development processes in Concat using refinement. We believe Concat to be well-suited for process modeling.

4.4 Flexibility & Expressiveness

Another area for which we cannot take credit for is a demonstration of the extreme flexibility of concatenative languages – this is best shown by pointing to Factor, a modern concatenative language implementation. Factor is dynamically typed and functionally impure (for practical reasons) though functional programming is a natural style in Factor. Its dominating programming model is a stack being passed from one function to the next.

Factor is powered by a kernel written in C++ of about 13.000 lines of code. Everything else is written in Factor itself. Factor's syntax is extensible, it has macros, continuations and a powerful collection library.

Factor comes with an object system with inheritance, generic functions, predicate dispatch and mixins – all this is implemented in Factor. Lexical variables and closures are implemented as a loadable library – in Factor. An optimizing compiler outputs efficient machine code – the compiler is written in Factor. Bootstrapping the system helps all libraries in Factor benefit from such optimizations. Right now, Factor supports a number of OS/CPU combinations among which are Windows, MacOS and Linux.

Programs in Factor are extremely short and compact. Refactoring programs in Factor is easy as is in any concatenative language: any fragment of words can be factored out – hence the name "Factor". These features have helped the developers continuously improve the code base and its libraries. Factor outperforms other scripting languages like Ruby, Groovy or Python not only in runtime but also in the number of features supported by the language.

5 CONCLUSIONS

Remarkably, the definition of Concat fits on a single page of paper (Sec. 3.1/3.3). Yet, the concatenative paradigm shows a lot of interesting features and inspiring approaches. We barely scratched the surface of a subject worth further investigation and research. We think that concatenative programming is a much overlooked paradigm that deserves wider recognition.

ACKNOWLEDGEMENTS

Part of this work was supported by the Thomas Gessmann-Stiftung.

REFERENCES

Adobe Systems Inc. (1999). *PostScript language reference (3rd ed.)*. Addison-Wesley.

Backus, J. (1978). Can programming be liberated from the von Neumann style?: A functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641.

Broy, M. and Stølen, K. (2001). *Specification and Development of Interactive Systems:* FOCUS *on Streams, Interfaces, and Refinement*. Springer.

Gibbons, J. (1999). A pointless derivation of radix sort. *J. Funct. Program.*, 9(3):339–346.

Rather, E. D., Colburn, D. R., and Moore, C. H. (1996). The evolution of Forth. *History of programming languages*, II:625–670.

Reichert, T., Klaus, E., Schoch, W., Meroth, A., and Herzberg, D. (2008). A language for advanced protocol analysis in automotive networks. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 593–602. ACM.