Seminar Report

Kayak

Christoph Klotz

19 February 2013

Supervisor: Dr. Stéphane Gimenez

Abstract

This document is about Kayak an esoteric reversible programming language. It shows on which operations it is build, the problems that might occur and some code snippets for explanation.

Contents

1 Introduction

Kayak is a programming language in which every operation and therefore every program is invertible. Any Kayak procedure can be run either forwards or backwards, or even both in the same program. The syntax of Kayak itself is providing the capability of reversing procedures.
After the explanation what all the fuzz is about, there will be a short overview of the history of reversible computing. In Section 4 the operating principles of kayak is shown. Additionally there are illustrated some problems you might stumble upon if you are programming in this language.

2 Motivation

Although reversible programming may seem unnecessary or unimportant, some people are convinced that this type of programming may be the future of computing. The laws of physics are presumed to be reversible, which means that no information can be destroyed, only shuffled. Computing on the other hand is usually thought of an irreversible process because it can involve Many-to-One procedures.[3] The information the computer lost in these procedures are converted to heat and gets emitted to the environment.

> "The von Neumann architecture is a physically unrealistic model which requires hardware garbage collection (in the form of a heat sink) to support it. Microprocessors which are reversible at the level of their fundamental logic gates can potentially emit radically less heat than irreversible processors, and someday that may make them more economical than irreversible processors."[1]

3 The History of Reversible Programming

3.1 Landauer's principle

In 1961 Rolf Landauer argued that any logically irreversible manipulation of information will gain heat which a computer can't use to reproduce the information again. His principle is widely accepted as a physical law but also was challenged for the last twenty years.

3.2 Reversible Turing Machine

Charles H. Bennett described a reversible Turing machine which had an additional tape. This tape keeps the history of the computations the machine made and if the output get computed backwards, the machine got back in its initial state without performing any irreversible operation.

3.3 Reversible Gates

The most common logical gates have numerous inputs but only one output, here we are back at the Many-to-One problem mentioned in Section 2. To make the

gate invertible we have to consider gates with an equal amount of inputs and outputs. Since not all 2-in/2-out gates can be inverted, Edward Fredkin and Tommaso Toffoli described 3-in/3-out gates which could be inverted and are universal logic elements. But if not every output signal is going to be processed or saved, these gates would not be better than any irreversible gate. So they used Bennetts method from Section 3.2 to save the additional, "garbage" signals, and it was possible to compute the gate backwards and get to the initial state.

4 The Language

Kayak is an esoteric programming language which complement that it is quite hard to read and write code. By the implementation of this language it is only possible to invert and juggle bits around and you are expected to compute serious stuff with this afterwards.

4.1 Identifiers

Except for the nine operators of this language, (i.e. `<`, `>`, `(`, `)`, `[`, `]`, `{`, `}` and `|`) and white-spaces, every symbol or a combination of these are identifiers. White-spaces are only used to delimit identifiers and make the code more readable. So there are no reserved words either and for example "`!§$%&/()=?`" is a valid identifier.

4.2 Data Types

Kayak only has one data type which is the infinite stacks of bits. These stack only can hold bits (i.e. ones an zeros) and are initialized as a stack of infinite zeros. So because they are infinite they can be popped without underflow and in every program/procedure there can be an arbitrary number of these infinite stacks. If you wish, you can count the scoped temporary one-bit-register as a data type too, but it is only one bit. This register is empty or contains a boolean value and is scoped to the current procedure.

4.3 Comments

Comments begin with a "`<`", end with a "`>`" and can be nested. You may be careful with expressions like "` if a >= b then....`" because here the comment would end halfway.

4.4 Variables

If you call a variable which has not been initialized yet, it gets initialized as a stack consisting of infinitely many zeros. A variable is only visible in the procedure it is initialized and also there is no such thing as a global variable. Unless they are passed out with the end parameters (see Section 4.6.2), they will not exist any further. At this point any untransmitted variables must have their initial state, which would be a stack with infinitely many zeros, to prevent the interpreter from complaining. This is the interpreter's way of preventing

information loss and encourage the programmer to implement his code (more) recursively.

4.5 Operating with Variables

If the variable already exists and the temporary bit-register is empty, the top element of the stack is popped and is put into the register (Figure 1a). But if the register is occupied the bit in the register is pushed on the called variable (Figure 1b). For example if we assume an empty register, `var1 var2` would transfer the top bit from `var1` to `var2`.
The `"|"` operator complements the value in the one-bit-register. And if we assume an empty register again, `var1|var2` would pop the top bit of `var1` to the register, complements this one bit and pushes it on the stack `var2`. If `"|"` is called while the temporary one-bit-register is empty the interpreter throws an error.

4.6 Procedures

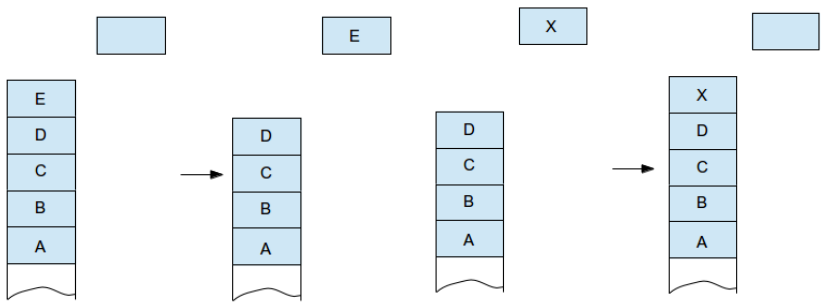
Every Kayak program consists of one or more procedures. Procedure definitions look like in Figure 2.

4.6.1 Name(s)

Every procedure name has two parts. One half is at the beginning of the definition (`name1` in Figure 2) and the other half is at the end (`name2` in Figure 2). To call a procedure you have to provide both parts of the name, so it is also possible that two or more procedures have the same name at the beginning if they have different names at the end. Both parts must not be empty, except for the main procedure (see Section 4.8).

4.6.2 Arguments

A procedure takes any amount of arguments but should have at least one to be useful. These arguments are named twice in the definition and are separated



(a) A variable is called with an empty register

(b) A variable is called with an occupied register

Figure 1: Operations on variables

```
name1(arg1|arg2) {body} (arg3|arg4)name2
```

Figure 2: The looks of a procedure definition

by the normally otherwise used operator "|", once at the beginning, like `arg1` and `arg2`, and once at the end of the definition, like `arg3` and `arg4` in Figure 2. When a procedure is called, the argument names at the beginning of the procedure are bound to the arguments of the procedure call, then the procedure computes something with these variables and when the procedure exits, it updates the content of the variables named at that end of the procedure definition. For example, `swap(a|b) {} (b|a)paws` is a procedure which swaps its two arguments, while `noth(a|b) {} (a|b)hton` is a do-nothing procedure of two arguments.

4.6.3 Calling

Calling a procedure looks like `name1(arg1|arg2)name2`. It performs a call to the procedure `name1()name2` where the arguments `arg1` and `arg2` are names of variables in the calling procedure. These variables will be modified by the called procedure except you have a do-nothing procedure like in Section 4.6.2. Recursion is allowed and is the only looping mechanism implemented.

4.6.4 Calling in Reverse

If you want to call a procedure backwards, you have to write its procedure name in reverse which would be `2eman(arg2|arg1)1eman` in the case of the procedure in Section 4.6.3. This is for example very useful for arithmetic functions, because the reverse of an addition procedure is a subtraction procedure. You can give procedures palindromic names (like in Section 4.6.2), but it is recommended to do so only if the procedure is self-inverse, because there is no way you could execute them backwards and so executing the whole program backwards would fail.

4.7 Conditional Execution

To break out of recursion you need something that tells the interpreter to do something else than calling the recursive procedure again. This is the job of the last two operators "[" and "]" which are used like in Figure 3. The code enclosed by these two parentheses is only executed if there is a one in the temporary on-bit register. If not it is skipped and the procedure moves on. The register which lead the procedure into the conditional procedure is not visible in this procedure any more. It is an error if you use "[" if the temporary register is empty or "]" when the register is occupied. In the first case the interpreter can not determine if the bit is one or zero because its empty and in the second case it would be the same if the procedure was called backwards. For example `a[b|b]a` pops the first bit of a into the the first register, if in the register is a one, b will be popped to a new (second) temporary register, get complemented,

```
[body]
```

Figure 3: The looks of conditional code

will be pushed back on the stack b, the procedure will be exited and the bit in the first temporary register will be pushed back on the stack a. You can also complement the first register after the first "[" and "]" combination to make a if-then-else like procedure (e.g. `a[b|b]|[c d]a`).

4.8 The Main Procedure

The main procedure is the top-level procedure executed by the interpreter. It can distinguish this procedure from the others by its unique name, it is the empty name. So it begins and ends with an argument section. The main procedure can be declared with only one or two arguments. The top-level procedure with one argument contains the standard-input at the beginning and the standard-output at the end. If there are two arguments, one of the these is a bit bucket, which contains random bits at the beginning and a modified version of this bit bucket at the end. This is the way of information disposal of this language. This "garbage" information was created by many-to-one transformations, and since kayak does not save this information, we must get rid of it after executing a program. The standard input and output are always innermost and the bit bucket is always outermost.

4.9 Input and Output Encoding

The standard input and output are encoded as 9 bit per byte on a stack. The top bit validates the following byte encoding an ASCII character with the least significant bit right under the validation bit. This is repeated for every byte and the first byte of standard input as well as standard output are the one closest to the top of the stack. Below the standard input and output there is again a stack with infinitely many zeros, so you can check the End-Of-Input by checking every ninth bit, the validation bit.

4.10 The Interpreter

Rudiak-Gould describes his interpreter as an “hastily-written utility”[1], that is because he wrote it only a few hours before the deadline of his essay, but except for the error reporting it works very well. The interpreter takes only one argument, (e.g. 99bottles.kayak). If there is a file with this name, in the same directory as the interpreter is executed in, the program will be run in forward direction. Else the interpreter is looking for a file which is matches with the arguments character wise reversal (e.g. kayak.selttob99) and if it is found, the program will be run in backward direction. Like in the procedures mentioned in Section4.6.4 the programs can have palindromic names, but it is recommended to do so only if the program is self-inverse. Running a program in reverse is a simple task for the interpreter because the only additional step it

```
a(o){z o z|o z|o z o z o z o z o z|o z|o}(o).
```

Figure 4: Procedure which adds the character "a" to the stack which was bound to o

has to do is to read the program backwards or reverse the whole string of the code character-wise.

5 Difficulties

There were several difficulties learning and programming in kayak. Although the author of this language wanted to give a good introduction of the programming paradigms, operations and coding approaches, I learned more by reading his samples, making my own little programs and and the attempts to combine them.

5.1 Debugging

Like I mentioned in Section 4.10 the interpreter was created in a rash way, and the error reporting system was implemented very poorly. You will not get any line numbers or other hints where your code is faulty. If you are lucky the interpreter gives you a stack with some procedure calls but that is about it. So the interpreter tells what is the error (e.g. “nonempty stack on function exit” or “can’t use | operator with no value in the register”) but if I don’t know where I made that mistake it can happen that I have to search the whole program for it. It is possible to test every procedure one by one, but this also can consume a lot of time.

5.2 No Character/String Support

I mentioned in Section 4.2 that in kayak only exist one major data type, the infinite bit stack. So there is no character support. To get the programming language to generate usable output I had to implement every single character I used for the song. Figure 4 is an example of such an implementation. The calling procedure has to call the procedure with the output stack as the argument to bind it to `o`. Afterwards the procedure puts the bits of the characters ASCII code onto this output stack beginning with the most significant bit. To validate this character an additional 1 is added to the top of the stack and the program returns to the calling procedure with a new output stack. Since `z` is called in this procedure the first time and is not an argument of the calling procedure, it gets initialized as a stack with infinitely many zeros, so it will alway pop a zero and you can push it on `o` complemented or not. To make the code more readable you can combine some procedures in another one. So you get the possibility to add words or phrases to the output. Due to the fact that the bytes on the top are read first, you have to put the characters in reverse order on the output stack as you want them to be put out.

```
div10(r|a|dump){
    r[r[r[r[r[r[r[r[r[i|i div10(r|a|dump).]r]r]r]r]r]r]r]r]r
    i[sub10(r). z|a]dump
}(r|a|dump).
```

Figure 5: Divide by 10 and Modulo 10 procedure (not reversible)

```
multi10(in){
    in[multi10(in). z|in z|in z|in z|in z|in z|in z|in z|in]in
}(in).
```

Figure 6: Multiply a number by 10

5.3 No Integer Support

Because this language lacks of integer support it was necessary to implement procedures with which the interpreter is able to count down from an arbitrary number and also to output the right numbers in each verse. It would have been a possibility to implement some integer support with some simple arithmetic operations, but it was much easier to put as much ones on a stack as an integer value. On every iteration one one gets put away, and the next loop computes the recursive procedure with one one less. By looping I also was able to successively determine the individual digits of such a stack of ones. I created a procedure (Figure 5) which divided the stack, representing an integer with ones, by ten and put out the result and the remainder. The remainder instantly can be put onto the output stack (procedure in Figure 7) because the output is generated backwards like mentioned in Section 5.2. At first I implemented it as a long-winded procedure which checked one for one and put out the whole number at once. Saying there were nested about 100 conditional executions and now it takes only 9 of them. As a little feature I implemented a parser which looks at the input of the user and starts the song at an arbitrary number. To do so i had to implement a loop which takes the first character, parse it to an "integer" (represented by a stack of ones) and multiplies it by 10 (Figure 6) if there is another character on the input stack which would be put onto the multiplied stack and maybe multiplied again, and so on. I only had to do this because my divide by 10 procedure is not reversible.

5.4 Coding Reversibly

It is not always that easy to program reversible code, but it is possible to code non reversible procedures and get away with it. If you are very lucky, the code you generated runs in both direction, if you are not so lucky the code runs in one direction and if you have bad luck you have to use the bit-bucket to dump unwanted information, to prevent "nonempty stack" errors. In Figure 6 i don't need the bit-bucket and can run it forwards (but not backwards), and in Figure 5 i need such a dump stack, to get rid of a one i wasn't able to dispose in another

```
append(in|a){
    a[a[a[a[a[a[a[a[
    9(in).]|[
    8(in).]|a]|[
    7(in).]|a]|[
    6(in).]|a]|[
    5(in).]|a]|[
    4(in).]|a]|[
    3(in).]|a]|[
    2(in).]|a]|[
    1(in).]|a]|[
    0(in).]|a
}(in|a).
```

Figure 7: Appends a digit to the output stack according to the ones in "a"

way.

6 Turing Completeness

To prove the Turing completeness of a programming language, you only have to show that you can emulate a Turing machine or another Turing complete language or calculus with it. The inventor of kayak already did this by making a converter from brainfuck to kayak in perl. He implemented for every brainfuck operator a small reversible kayak procedure and let the converter put them together according to the brainfuck program to get an equivalent kayak program.

7 Conclusion

This paper give a short overview, what reversible computing is good for, what was already achieved and that there is a way of reversible programming. It mentions whole operations-pool, what you can achieve with it and in which problems you might run creating a program in kayak.

References

[1] http://esoteric.voxelperfect.net/files/kayak/doc/kayak.html

[2] http://esolangs.org/wiki/Kayak

[3] http://mathworld.wolfram.com/Many-to-One.html

[4] http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TM-151.pdf

[5] http://arxiv.org/pdf/physics/0210005v2.pdf