

AN ASSEMBLER IN A NUTSHELL

Dan Popa

Univ. of Bacău , Spiru Haret nr. 8, Bacău, România

popavdan@yahoo.com

Abstract: This document is focused on the category theory and the new revised version of the Haskell 98 language, illustrating how can both be used in order to quickly prototyping language processors. As a proof, the reader is invited to see how a functor is becoming an assembler of a simple language. This simultaneously illustrates the power of both tools, The Category Theory and the functional language Haskell itself. The resulted assembler is universal and adaptable. We can easily modify the morphism which produce the machine code without being necessary to modify anything else. The functor will transform the new morphism in a new assembler in an instant. As an example, an other assembler, inspired by the well known book “Compilers and Compiler Generators” was built in less than an hour. A conclusion concerning such tools and their necessity is drawn .

1. Introduction

The facts: Pushed by the new rules came from Europe, roumanian universities are attempting to drop or even to consider as optional some sets of courses. For examples, the University of Bacău is ready to consider Graph Theory and Compiler Constructions as optional courses. Such Theories are forced to leave their places and become replaced by double semesters courses as Data Bases or even Programming Languages courses (where Word and Excel are studied by engineers). Other Universities (like Iasi) had similar customs, sometimes ejecting the Category Theory Course or misplacing it before the Compiler Construction Course. This custom made the students unable to see how powerful mathematical theories are able to solve the problems of the computer scientists. Such a problem, how to quickly build an adaptable assembler is treating, as an example, in the following paragraphs.

2. Basics

Some basic definitions are included.

A fine definition of category may be find in [1]:

A category C consists of two collections, $\mathbf{Ob}(C)$, whose elements are the objects of C , and $\mathbf{Ar}(C)$, the arrows (or morphisms or maps) of C . To each arrow is assigned a pair of objects, called the source (or domain) and the target (or codomain) of the arrow. The notation $f: A \rightarrow B$ means that f as an arrow with source A and target B . If $f: A \rightarrow B$ and $g: B \rightarrow C$ are two arrows, there is an arrow $g \circ f: A \rightarrow C$ called the composite of g and f . The composite is not defined otherwise. We often write gf instead of $g \circ f$ when there is no danger of confusion. For each object A there is an arrow id_A (often written 1_A or just 1 , depending on the context), called the identity of A , whose source and target are both A . These data are subject to the following axioms:

1. For $f: A \rightarrow B$,
 $f \circ \text{id}_A = \text{id}_B \circ f = f$;
2. For $f: A \rightarrow B$, $g: B \rightarrow C$, $h: C \rightarrow D$,
 $h \circ (g \circ f) = (h \circ g) \circ f$

A category consists of two “collections”, the one of sets and the one of arrows.

In the same book [2] (paragraph 1.2. pp.11) the definition of the functor was given as you see below:

Like every other kind of mathematical structured object, categories come equipped with a notion of morphism. It is natural to define a morphism of categories to be a map which takes objects to objects, arrows to arrows, and preserves source, target, identities and composition.

If C and D are categories, a functor $F: C \rightarrow D$ is a map for which

1. If $f: A \rightarrow B$ is an arrow of C , then
 $Ff: FA \rightarrow FB$ is an arrow of D ;
2. $F(\text{id}_A) = \text{id}_{FA}$; and
3. If $g: B \rightarrow C$, then $F(g \circ f) = Fg \circ Ff$.

3. An example

A well known example, which can be found in many textbooks is quoted by Andrea Schalk in [3]:

Example 3.1 Let $M: \mathbf{Set} \rightarrow \mathbf{Set}$ be the functor that maps a set A to all the words that can be formed over the alphabet A , and whose action on morphisms is described as follows.

For $f: A \rightarrow B$ in \mathbf{Set} let $Mf: MA \rightarrow MB$ be given by

$$Mf(a_1 \cdot \dots \cdot a_n) = f(a_1) \cdot \dots \cdot f(a_n)$$

This functor will be used as a sort of adaptable assembler generator in the followings examples. The program will be written in Haskell 98, the recently (2003) revised version of Haskell.

4. History of Haskell 98

Due to the fact that the history of Haskell is best described using the words of the authors we were decided to include the following

text which is quoted from the published version of the Haskell 98 Report and was reproduced in [4]:

<< In September of 1987 a meeting was held at the conference on Functional Programming Languages and Computer Architecture (FPCA '87) in Portland, Oregon, to discuss an unfortunate situation in the functional programming community: there had come into being more than a dozen nonstrict, purely functional programming languages, all similar in expressive power and semantic underpinnings. There was a strong consensus at this meeting that more widespread use of this class of functional languages was

being hampered by the lack of a common language. It was decided that a committee should be formed to design such a language, providing faster communication of new ideas, a stable foundation for real applications development, and a vehicle through which others would be encouraged to use functional languages. This document describes the result of that committee's efforts: a purely functional programming language called Haskell, named after the logician Haskell B. Curry whose work provides the logical basis for much of ours.

The committee's primary goal was to design a language that satisfied these constraints:

1. It should be suitable for teaching, research, and applications, including building large systems.
2. It should be completely described via the publication of a formal syntax and semantics.
3. It should be freely available. Anyone should be permitted to implement the language and distribute it to whomever they please.
4. It should be based on ideas that enjoy a wide consensus.
5. It should reduce unnecessary diversity in functional programming languages.

The committee intended that Haskell would serve as a basis for future research in language design, and hoped that extensions or variants of the language would appear, incorporating experimental features.

Haskell has indeed evolved continuously since its original publication. By the middle of 1997, there had been four iterations of the language design (the latest at that point being Haskell 1.4). At the 1997 Haskell Workshop in Amsterdam, it was decided that a stable variant of Haskell was needed; this stable language (is the subject of the Report), and is called Haskell

98. Haskell 98 was conceived as a relatively minor tidy-up of Haskell 1.4, making some simplifications, and removing some pitfalls for the unwary.

It is intended to be a "stable" language in sense the *implementors are committed to supporting Haskell 98 exactly as specified, for the foreseeable future.*

The original Haskell Report covered only the language, together with a standard library called the Prelude. By the time Haskell 98 was stabilized, it had become clear that many programs need access to a larger set of library functions (notably concerning input/output and simple interaction with the operating system). If these program were to be portable, a set of libraries would

have to be standardized too. A separate effort was therefore begun by a distinct (but overlapping) committee to fix the Haskell 98 Libraries. >>

5. How to write an assembler in 8 lines of code

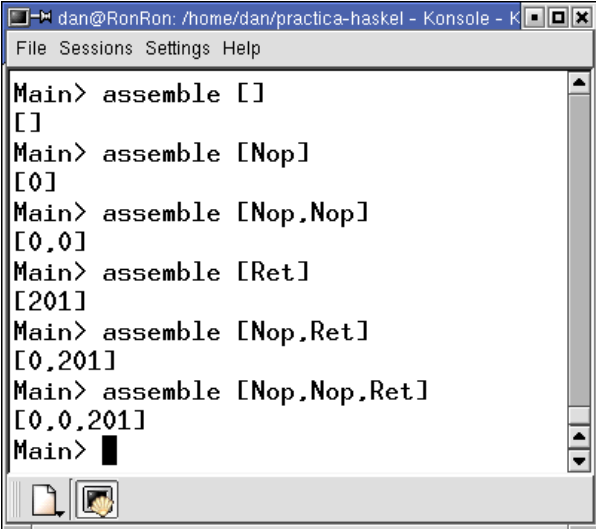
As a first example, a small assembler for a tiny language composed by only two keywords NOP and RET can be immediately written .

```
data Instr = Nop | Ret
-- Arrow
f  :: Instr -> [Int]
f Nop  = [ 0 ]
f Ret  = [201 ]
f _    = []
-- Functor
m  :: (Instr -> [Int]) -> [Instr] -> [Int]
m f []  = []
m f (a:l) = f a l ++ m f l
-- Assambler
assemble  :: [Instr] -> [Int]
assemble x = m f x
```

Remark: Any element of the **Instr** datatype may, theoretically produce a list of machine codes. In Haskell, there is a tradition of considering strings as lists.

6. Testing the assembler

The Mandrake 8.2 Linux and the well known interpreter Hugs 98 was used to test the previous program. The results may be seen here:



```
dan@RonRon: /home/dan/practica-haskell - Konsole - K
File Sessions Settings Help
Main> assemble []
[]
Main> assemble [Nop]
[0]
Main> assemble [Nop,Nop]
[0,0]
Main> assemble [Ret]
[201]
Main> assemble [Nop,Ret]
[0,201]
Main> assemble [Nop,Nop,Ret]
[0,0,201]
Main> █
```

8. Adapting the assembler

The next goal was to adapt the previous assembler to a new language having a more complex instruction set. The language is described in the Chap. 4 of [5].

9. The new program

```

-- Dan Popa 22.iunie.2005
-- Instructions -- Instructiunile de asamblat
data Instr =NOP | CLA | CLC | CLX | CMC | INC | DEC |
           INX | DEX | TAX | INI | INH | INB | INA |
           OTI | OTC | OTH | OTB | OTA | PSH | POP |
           SHL | SHR | RET | HLT |
           LDA Int | LDX Int | LDI Int | LSP Int | LSI Int |
           STA Int | STX Int | ADD Int | ADX Int | ADI Int |
           ADC Int | ACX Int | ACI Int | SUB Int | SBX Int |
           SBI Int | SBC Int | SCX Int | SCI Int | CMP Int |
           CPX Int | CPI Int | ANA Int | ANX Int | ANI Int |
           ORA Int | ORX Int | ORI Int | BRN Int | BZE Int |
           BNZ Int | BPZ Int | BNG Int | BCC Int | BCS Int |
           JSR Int
-- Arrow f -- Morfismul de asamblare a fiecarei instructiuni
-- The semantics -- el ne da semantica de generare a codului
f :: Instr -> [Int]
f NOP = [ 00]
f CLA = [ 01]
f CLC = [ 02]
f CLX = [ 03]
f CMC = [ 04]
f INC = [ 05]
f DEC = [ 06]
f INX = [ 07]
f DEX = [ 08]
f TAX = [ 09]
f INI = [ 10]
f INH = [ 11]
f INB = [ 12]
f INA = [ 13]
f OTI = [ 14]
f OTC = [ 15]
f OTH = [ 16]
f OTB = [ 17]
f OTA = [ 18]
f PSH = [ 19]
f POP = [ 20]
f SHL = [ 21]
f SHR = [ 22]
f RET = [ 23]
f HLT = [ 24]
-- Double byte instr.
f (LDA b) = [ 25, b]
f (LDX b) = [ 26, b]
f (LDI b) = [ 27, b]
f (LSP b) = [ 28, b]
f (LSI b) = [ 29, b]
f (STA b) = [ 30, b]
f (STX b) = [ 31, b]
f (ADD b) = [ 32, b]
f (ADX b) = [ 33, b]
f (ADI b) = [ 34, b]
f (ADC b) = [ 35, b]
f (ACX b) = [ 36, b]
f (ACI b) = [ 37, b]
f (SUB b) = [ 38, b]
f (SBX b) = [ 39, b]
f (SBI b) = [ 40, b]
f (SBC b) = [ 41, b]
f (SCX b) = [ 42, b]
f (SCI b) = [ 43, b]
f (CMP b) = [ 44, b]
f (CPX b) = [ 45, b]
f (CPI b) = [ 46, b]
f (ANA b) = [ 47, b]
f (ANX b) = [ 48, b]

```

```

f (ANI b) = [ 49, b]
f (ORA b) = [ 50, b]
f (ORX b) = [ 51, b]
f (ORI b) = [ 52, b]
f (BRN b) = [ 53, b]
f (BZE b) = [ 54, b]
f (BNZ b) = [ 55, b]
f (BPZ b) = [ 56, b]
f (BNG b) = [ 57, b]
f (BCC b) = [ 58, b]
f (BCS b) = [ 59, b]
f (JSR b) = [ 60, b]
f _ = []
-- Functorul M -- functor -- l inseamna a2...an
m :: (Instr -> [Int]) -> [Instr] -> [Int]
m f [] = []
m f (a1:l) = f a1 ++ m f l

```

```

-- Asamblorul final rezultat din m si f
-- The final assembler
assemble :: [Instr] -> [Int]
assemble x = m f x

```

As you can see, all we had to do was to write the new mnemonics and the new lists of machine codes. That's all. The implementation functor (and consequently, of the assembler itself) remains unchanged.

10. Testing the new assembler

In order to test the new assembler the source from example 4.3 (from the same book) was used, because of the presence of the assembled code in the previous example of the same chapter.:

```

INI
SHR
BCC 13
STA 19
LDA 20
INC
STA 20
LDA 19
BNZ 1
LDA 20
OTI
HLT

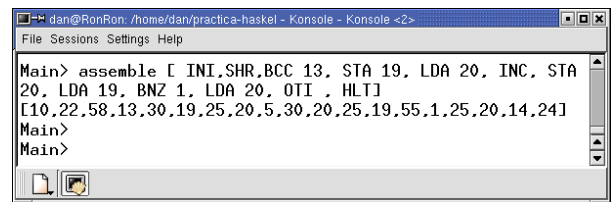
```

The source above should be translated in the following sequence of decimal numbers:

```

10 22 58 13 30 19 25 20 5 30 20 25 19 55 1 25
20 14 24

```



And the translation succeeds.

6. References

[1-2] Barr Michael, Wells Charles, *Toposes, Triples and Theories*, McGill University, 2002, pg.1-11

[3] Schalk Andrea, *Some notes on monads*
Department of Computer Science, University of
Manchester, March, 12, 2002 (www resource)

[4] Hal Daume, *Yet Another Haskell Tutorial* Copyright
(c) Hal Daume III, 2002-2004.

<http://www.isi.edu/~hdaume/htut/>

[5] Terry, P.D, *Compilers and Compiler Generators* (c)
P.D. Terry, 2000