Reducing Structural Complexity of Software by Data Streams*

JÁN KOLLÁR [1]
ĽUBOMÍR WASSERMANN [1]
VALENTINO VRANIĆ [2]
MICHAL VAGAČ [3]

[1]Technical University of Košice, Department of Computers and Informatics
Letná 9, 042 00 Košice, Slovakia
`(Jan.Kollar,Lubomir.Wassermann)@tuke.sk`

[2]Slovak University of Technology, Faculty of Informatics and Information Technologies
Institute of Informatics and Software Engineering
Ilkovičova 3, 842 16 Bratislava 4, Slovakia
`vranic@fiit.stuba.sk`

[3]Matej Bel University, Faculty of Sciences
Tajovského 40, 974 01 Banská Bystrica, Slovakia
`vagac@fpv.umb.sk`

Abstract. Language architecture is developed from context-free grammar enriched by communication channels. Four types of formal communication channels and one type of informal communication channel are identified for functional languages, to provide a systematic background for human-machine communication. Formal channel positions are determined by a grammar, and informal channels by a programmer. Structural complexity is radically decreased since data streams are approximately as concise as computer machine code, but they are semantically equivalent to high level functional programs. Using simple example of a functional language, we present the principle of functional language architecture and its driving by a data stream. In particular, we show how a program is initially recorded, how it is automatically generated, and how it is adapted to verbose, quiet and collaborative modes. As a result, we propose data stream-oriented architecture, in which structural complexity of current programs is rapidly decreased, since the execution of language architecture machine code means the execution of high-level programs.

Keywords: Data stream driven computing, language architecture, executable modeling, domain specific languages, integrated model and language semantics.

(Received June 21, 2009 / Accepted August 17, 2009)

1 Introduction

Automatic software construction by machines is software engineering vision [6, 5]. Generative [3], aspect oriented [11], and context oriented programming [10] are successful software engineering practices for minimizing accidental complexity of software construction. In our opinion however, sometimes too complicated solutions are provided, because increasing growth of semantic complexity of software is accompanied with the growth of structural complexity.

Crucial problem of executable modeling is how to

*This work was supported by VEGA project No. 1/4073/07 "Aspect-oriented Evolution of Complex Software Systems" define the semantics of structural elements expected to be evolved in the future. It is well known for natural languages, that meaningful sentences can be formulated just understanding terms and expressions, from which they are composed. That is why semiformal models post-mortem formalisation to get executables is enormous effort due to high degree of semantic ambiguities. For example, looking deeply in the substance of UML class diagrams and design patterns, we may notice that both are just abstract syntactic trees, with freely semantically interpretable non-terminals.

Modeling, prototyping, design and architectonic patterns are exploited in current software engineering practice, but this time no commonly accepted theory for automatic software production exists. Despite that UML action semantics was published already in 2001 [15], executable modeling is applied just in specific applications [17]. Even worse, no common methodology for model transformation is available [4].

As we may notice, language is more than model and metalanguage is more than metamodel [12]. Software language engineering approaches based on grammars [9, 13] are useful in top-down direction to produce programs from grammars, as well as in bottom-up direction for deriving grammars from programs [1]. In this paper, in constrast to program transformation, applied to metamodel [7], we propose integrated metaprogram and program execution. Instead of two dimensional compiler construction [16] we ignore original source form, not constructing a compiler at all. In initial stages of our design of executable language model, we have visualized grammar tracing, similar to program tracing, introduced in [2]. But in our solution, instead of program inspection, grammar in the form of metaprogram (i.e. language executable model, or language architecture) is executed.

2 Reducing Structural Complexity

The main idea behind our solution, presented in this paper, is as follows:

Let us have a grammar G, and a data stream S. Then S is consumed by language model/architecture derived from grammar G and this consumption executes program P. The execution in G should be semantically equivalent to P, i.e. $P = G\ S$, where $[\![\, P \,]\!] = [\![\, S \,]\!] + [\![\, G \,]\!]$.

Current architecture logic is hardwired as a finite state automaton, which is defined by regular grammar. Program P is executed by the application of automaton to the machine representation S of program P. But since of equivalence of grammar G and automaton, we may abstract automaton to grammar. Then the application $(G\ S)$ of a grammar G to machine code S is meaningful execution of P. Omitting the role of architecture, source program P (written for example in some high level language) and its target machine code S has the same semantics, and then we may write $[\![\, P \,]\!] = [\![\, S \,]\!]$. But this is just the simplification, because the meaning of source program is given by composition of target code structural semantics $[\![\, \text{S} \,]\!]$ and architecture semantics $[\![\, G \,]\!]$. The we have $[\![\, P \,]\!] = [\![\, S \,]\!] + [\![\, G \,]\!]$, which is an accurate relation between the semantics of source program P and its target code $[\![\, S \,]\!]$ executed in architecture represented by grammar $[\![\, G \,]\!]$.

Structural complexity of a program written in a high level language is far more lower than if using a machine language for the same problem.

The problem is that current formal programming languages supported by translators have limits of abstraction given by semantically weak machine languages of current architectures.

That is why our data stream S, which represents program construction, should be structurally at a very low level of current machine code, but semantically equivalent to current high level program P.

Then we will be able to build new kind of architectures – language architectures, which will by applied to miniaturized forms of current high level programs (and software systems) in the form of new target code. We would like to integrate current concepts of modeling and programming, replacing current modeling practices by the transformation of formal languages of new generation.

In this paper, we present no software tool for the development of programs, systems and models in new language architectures. We introduce just principles and simple implementation, using Haskell [14] IO monad, to be as concise as possible. Practically, we are using Hugs 98 interpreter [8], because it is fully sufficient for our purposes. It is supposed that a reader is familiar with functional programming concepts including monads.

3 Language Architecture

Language based architecture is language executable model, defined by grammar with communication channels, in which data streams are consumed and produced, either by a user or by an architecture, and they may be stored in the architecture memory.

Let us define a simple functional language grammar, with communication channels according to (1).

$$\begin{array}{lcl} Language & \rightarrow & \text{print } Exp \\ Exp & \rightarrow & Mul\ \underline{\mathrm{C}_1?}\ \{\ \underline{\mathrm{S}_1?}(\ +\mid -\ )\ Mul\ \} \\ Mul & \rightarrow & Term\ \underline{\mathrm{O}_1?}\ [\ \underline{\mathrm{S}_2?}(\ *\mid /\ )\ Mul\ ] \\ Term & \rightarrow & \underline{S_3?}(\underline{\mathrm{val}?} \mid Exp) \end{array} \tag{1}$$

where $[\ \varphi\ ] = \varepsilon \mid \varphi$, for syntactic expression φ and empty symbol ε, and $\{\ \varphi\ \}$ is transitive closure, i.e.

$$\{\ \varphi\ \} = \varepsilon \mid \varphi \mid \varphi\,\varphi \mid \varphi\,\varphi\,\varphi \mid \ \ldots$$

In (1), communication channels are underlined, and they are associated with metalanguage constructs – transitive closure, optional occurrence, and set of syntactic expressions, as well as with selected terminal symbol val.

Note, channels $\mathrm{C_k}$ and $\mathrm{O_k}$ are represented by one bit, $\mathrm{S_k}$ can be represented in most cases by one byte (since the number or set elements rarely exceeds 256), but each terminal channel representation depends on its value type.

Position of each communication channel above is given by grammar. Communication channels derivable from grammar are formal.

The fifth type of communication channel is informal, it is the subject of annotation by a programmer, and it does not affect program execution. Informal channels are not marked in grammar (1) in this paper, although we are using them in the implementation (as prompts to a user).

The semantics of language architecture is defined by function `evaluate`, see Fig 1.

```
data Symbols = Language | Exp | Mul | Term |
               PRINT | ADD | SUB | MUL | DIV | EXP | VAL Int
               deriving Show

data T a = T a [T a] deriving Show

evaluate :: T Symbols -> IO ()
evaluate (T PRINT [st] ) = putStr (show (evalExp st))

evalExp :: T Symbols -> Int
evalExp (T ADD [st1, st2] ) = evalExp st1 + evalExp st2
evalExp (T SUB [st1, st2] ) = evalExp st1 - evalExp st2
evalExp (T MUL [st1, st2] ) = evalExp st1 * evalExp st2
evalExp (T DIV [st1, st2] ) = evalExp st1 ‘div‘ evalExp st2
evalExp (T (VAL x) [] )     = x
```

Figure 1: Language architecture semantics

Communication channels consume data stream which determines the execution in the language architecture. Informally, the communication channels work as follows:

1. If $\mathrm{C_k}?$ consumes 1, then the constructs in transitive closure are activated, if 0, then they are not activated.
2. If $\mathrm{O_k}?$ consumes 1, then optional constructs are activated, if 0, then they are not activated.
3. If $\mathrm{S_k}?$ consumes number n, then n-th construct selected from set of constructs is activated.
4. $\mathrm{val}?$ consumes the value, which defines the semantics of terminal symbol val.

Suppose source expression `print 4*(3+2-1)-5` is translated to the syntactic tree

```
T PRINT [T SUB [T MUL [T (VAL 4) [],
T SUB [T ADD [T (VAL 3) [],T (VAL 2)
[]],T (VAL 1) []]],T (VAL 5) []]]
```

Then the evaluation by function `evaluate` prints value 11 on the screen.

We remind, our goal is to get the same result by supplying the architecture with data stream, instead of supplying compiler/interpreter by source code.

4 Data Streams as a Language Architecture Instructions

Constructing initial data stream manually without supporting tool is enormous effort, because when we are typing stream data, we must look at grammar. Nevertheless, if hard task is done and data stream is recorded to a memory, architecture is able to reproduce computation.

4.1 Stream Recording

To record input stream, we set all communication channels to "read from user and write to memory" mode. This is done by function `getStreamItem` in Fig. 2 and continued in Fig. 3.

Invoking computation by `(construct Language [])`, and typing the stream of numbers

```
0 4 1 0 1 0 3 0 1 0 0 2 0 1 1 0 1 0 0
0 1 1 0 5 0 0.
```

we get in IO monad pair of two items – (1) constructed syntactic tree, and (2) input stream represented as `DataStream`, which is recorded, i.e

```
(T PRINT [T SUB [T MUL [T (VAL 4) [],
T SUB [T ADD [T (VAL 3) [], T (VAL 2)
[]],T (VAL 1) []]],T (VAL 5) []]],
[S 0,V 4,O 1,S 0,S 1,S 0,V 3,O 0,C 1,
S 0,S 0,V 2,O 0,C 1,S 1,S 0,V 1,O 0,
C 0,O 0,C 1,S 1,S 0,V 5,O 0,C 0])
```

```
aToI = foldl op 0 where x ‘op‘ y = x * 10 + ord y - ord ’0’

data DataStream = S Int | O Int | C Int | V Int

getStreamItem :: (Int -> DataStream) -> [DataStream]
                                      -> IO (Int,[DataStream])
getStreamItem itemType stream =
  getLine >>= \ns -> return (aToI ns) >>=
              \n -> return (n,stream++[itemType n])

construct :: Symbols -> [DataStream] -> IO (T Symbols,[DataStream])

construct Language  stream =
  construct Exp stream >>=
  \(exp,stream) -> return ((T PRINT [exp]),stream)

construct Exp  stream = construct Mul stream >>=
                        \(mul,stream) -> closure mul stream
  where
    closure mul stream = putStr "#C1# [0/1]? " >>=
                         \() -> getStreamItem C stream >>=
                         \(n,stream) -> caseSel n stream
      where
        caseSel n stream
          | n == 1 = select stream >>=
                     \(op,stream) -> construct Mul stream >>=
                     \(mul’,stream) -> closure (T op [mul,mul’]) stream
          | n == 0 = return (mul,stream)

        select stream = putStr "#S1# [0:+,1:-]: " >>=
                        \() -> getStreamItem S stream >>=
                        \(n,stream) -> return (([ADD,SUB] !! n),stream)
```

Figure 2: Data stream recording

```
construct Mul stream = construct Term stream >>=
                       \(term,stream) -> optional term stream
  where
    optional term stream = putStr "#O1# [0/1]? " >>=
                           \() -> getStreamItem O stream >>=
                           \(n,stream) -> caseSel n stream
      where
        caseSel n stream
          | n == 1 = select stream >>=
                     \(op,stream) -> construct Mul stream >>=
                     \(mul’,stream) -> return ((T op [term,mul’]),stream)
          | n == 0 = return (term,stream)

        select stream = putStr "#S2# [0:*,1:/]: " >>=
                        \() -> getStreamItem S stream >>=
                        \(n,stream) -> return (([MUL,DIV] !! n),stream)

construct Term stream = putStr "#S3# [0:val:?,1:Exp]: " >>=
                        \() -> getStreamItem S stream >>=
                        \(n,stream) -> caseSel n stream
  where
      caseSel n stream
        | n == 0 = terminal stream >>=
                   \(val,stream) -> return ((T (VAL val) []),stream)
        | n == 1 = construct Exp stream
      terminal stream = putStr ("val: ") >>=
                        \() -> getStreamItem V stream
```

Figure 3: Data stream recording – continued

4.2 Program Reproduction

To accept recorded stream, all communication channels are switched to "read from memory" mode, see Fig 4 and Fig 5, and program is reproduced by expression

```
construct Language (dataStream
datastream) >>= \(lang,ds) ->
evaluate lang
```

Then, instead of constructing stream in memory, data stream `datastream` is accepted by function `streamItem`. For pure reproduction, function `getStreamItem` is not applied at all.

For the purpose of simplicity, instead of using some external file for recorded stream, we just have transfered it via clipboard, to define constant function `datastream`. Function `dataStream` transforms stream items to string representation of numbers.

Reproduced output on the screen in verbose mode is shown in Fig. 7. This looks like recording, except that this time user types nothing.

Informal channel (user interface) does not affect the computation, hence switching computation from verbose mode (Fig. 7) to quiet mode means just removing each occurrence of monadic chain
`putStr (......) >>= \() ->`, as can be seen in Fig. 6.

Then no lines with prompts will appear on the screen, see Fig. 8.

4.3 Program Alternation

By stream alternation it is possible to change recorded program, and also human-language architecture cooperation, allocating some activities to a user, and others to the language architecture.

For example, all data stream values `(V x)` from data stream `datastream` can be filtered out, obtaining substream, and val communication channel can be switched to "read from user" mode.

```
getStreamItem :: [String] -> IO (Int,[String])
getStreamItem (d:ds) = getLine >>= \ns -> return ((aToI ns),(d:ds))

streamItem :: [String] -> IO (Int,[String])
streamItem (d:ds) = putStr (d++"\n") >>= \() -> return ((aToI d),ds)

construct :: Symbols -> [String] -> IO (T Symbols,[String])

construct Language ds  = construct Exp ds >>=
                         \(exp,ds) -> return ((T PRINT [exp]),ds)

construct Exp ds = construct Mul ds >>=
                   \(mul,ds) -> closure mul ds
  where
    closure mul ds = putStr "#C1# [0/1]? " >>=
                     \() -> streamItem ds >>=
                     \(n,ds) -> caseSel n ds
      where
        caseSel n ds
          | n == 1 = select ds >>=
                     \(op,ds) -> construct Mul ds >>=
                     \(mul',ds) -> closure (T op [mul,mul']) ds
          | n == 0 = return (mul,ds)

        select ds = putStr "#S1# [0:+,1:-]: " >>=
                    \() -> streamItem ds >>=
                    \(n,ds) -> return (([ADD,SUB] !! n),ds)
```

Figure 4: Automated program reproduction

Then, instead of recorded data stream below

```
0 4 1 0 1 0 3 0 1 0 0 2 0 1 1 0 1 0 0
0 1 1 0 5 0 0.
```

the next stream of data will be consumed from memory, except all numbers, designated by lowercase letters that will be supplied by a user.

```
0 a 1 0 1 0 b 0 1 0 0 c 0 1 1 0 d 0 0
0 1 1 0 e 0 0.
```

By other words, we provide user with the domain specific language for printing values of all expressions `print a*(b+c-d)-e`.

The screen output is shown in Fig. 9.

5 Acknowledgment

This paper is the result of the project implementation Centre of Information and Communication Technologies for Knowledge Systems (project number: 26220120020) supported by the Research & Development Operational Programme funded by the ERDF and was supported by bilateral project *Software Evolution by Language Adaptation* Grant No. BI-SK/08-09-002 and Grant No. SK-SI-0001-08 between Slovakia and Slovenia.

6 Conclusion

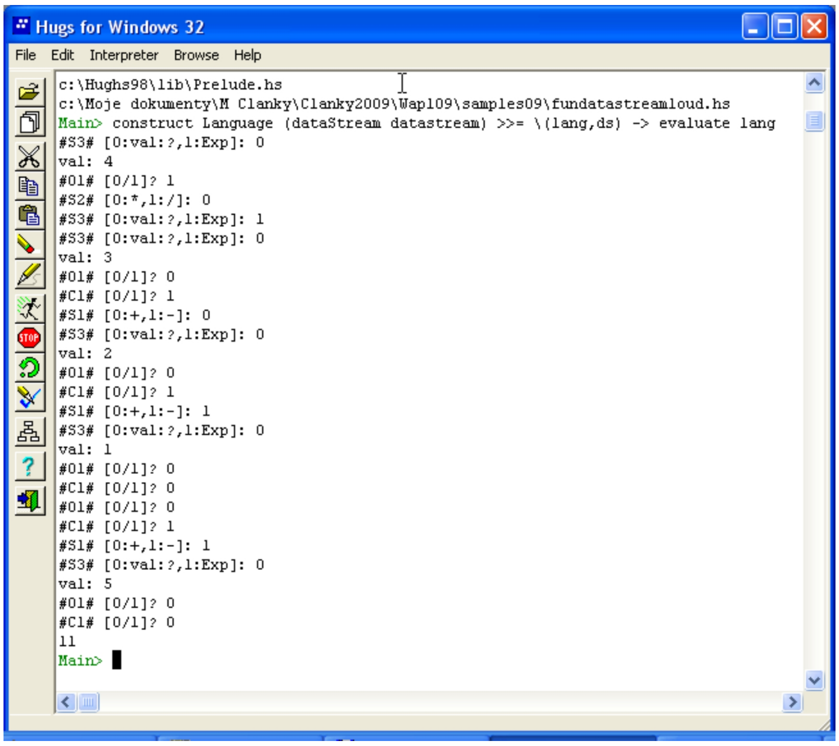
The main contribution of this paper is as follows. Executable language model can be implemented as language architecture driven by data streams. Data streams are language architecture programs that represent source language program as a history of program construction. This history is formalized, with a very concise representation.

```
construct Mul ds = construct Term ds >>=
                   \(term,ds) -> optional term ds
  where
    optional term ds = putStr "#O1# [0/1]? " >>=
                       \() -> streamItem ds >>=
                       \(n,ds) -> caseSel n ds
      where
        caseSel n ds
          | n == 1 = select ds >>=
                     \(op,ds) -> construct Mul ds >>=
                     \(mul’,ds) -> return ((T op [term,mul’]),ds)
          | n == 0 = return (term,ds)

        select ds = putStr "#S2# [0:*,1:/]: " >>=
                    \() -> streamItem ds >>=
                    \(n,ds) -> return (([MUL,DIV] !! n),ds)

construct Term ds = putStr "#S3# [0:val:?,1:Exp]: " >>=
                    \() -> streamItem ds >>=
                    \(n,ds) -> caseSel n ds
  where
      caseSel n ds
        | n == 0 = terminal ds >>=
                   \(val,ds) -> return ((T (VAL val) []),ds)
        | n == 1 = construct Exp ds

      terminal ds = putStr ("val: ") >>=
                    \() -> streamItem ds

datastream =  [S 0,V 4,O 1,S 0,S 1,S 0,V 3,O 0,C 1,S 0,S 0,V 2,O 0,
               C 1,S 1,S 0,V 1,O 0,C 0,O 0,C 1,S 1,S 0,V 5,O 0,C 0])
```

Figure 5: Automated program reproduction – continued

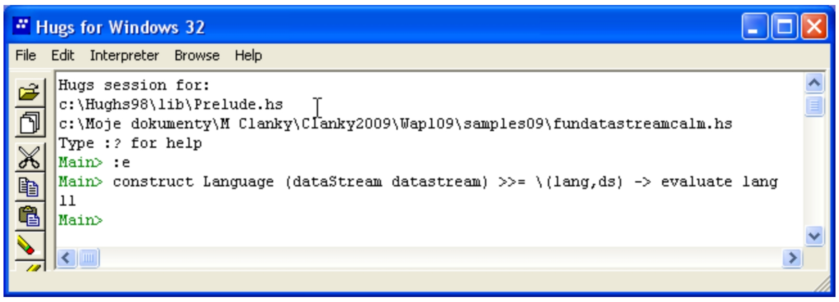
```
streamItem (d:ds) = return ((aToI d),ds)

construct :: Symbols -> [String] -> IO (T Symbols,[String])

construct Language ds  = construct Exp ds >>=
                         \(exp,ds) -> return ((T PRINT [exp]),ds)

construct Exp ds = construct Mul ds >>=
                   \(mul,ds) -> closure mul ds
  where
    closure mul ds = streamItem ds >>=
                     \(n,ds) -> request n ds
      where
        request n ds
          | n == 1    = select ds >>=
                        \(op,ds) -> construct Mul ds >>=
                        \(mul’,ds) -> closure (T op [mul,mul’]) ds
          | otherwise = return (mul,ds)

        select ds = streamItem ds >>=
                    \(n,ds) -> return (([ADD,SUB] !! n),ds)

construct Mul ds = construct Term ds >>=
                   \(term,ds) -> optional term ds
  where
    optional term ds = streamItem ds >>=
                       \(n,ds) -> request n ds
      where
        request n ds
          | n == 1    = select ds >>= \(op,ds) -> construct Mul ds >>=
                        \(mul’,ds) -> return ((T op [term,mul’]),ds)
          | otherwise = return (term,ds)

        select ds = streamItem ds >>=
                    \(n,ds) -> return (([MUL,DIV] !! n),ds)

construct Term ds = streamItem ds >>=
                    \(n,ds) -> request n ds
  where
      request n ds
        | n == 0 = terminal ds >>=
                   \(val,ds) -> return ((T (VAL val) []),ds)
        | n == 1 = construct Exp ds

      terminal ds = streamItem ds
```

Figure 6: Stream driven quiet computation



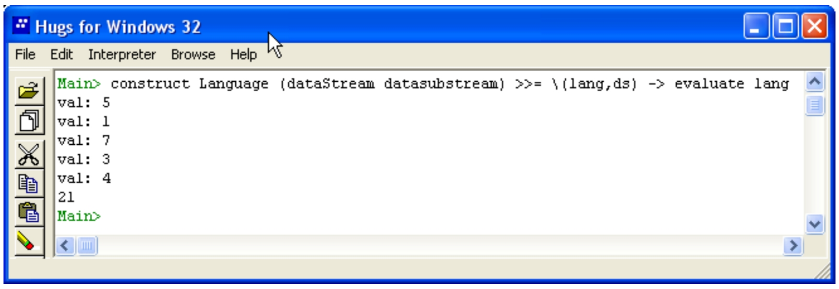
```
c:\Hughs98\lib\Prelude.hs
c:\Moje dokumenty\M Clanky\Clanky2009\Wap109\samples09\fundatastreamloud.hs
Main> construct Language (dataStream datastream) >>= \(lang,ds) -> evaluate lang
#S3# [0:val:?,1:Exp]: 0
val: 4
#01# [0/1]? 1
#S2# [0:*,1:/]: 0
#S3# [0:val:?,1:Exp]: 1
#S3# [0:val:?,1:Exp]: 0
val: 3
#01# [0/1]? 0
#C1# [0/1]? 1
#S1# [0:+,1:-]: 0
#S3# [0:val:?,1:Exp]: 0
val: 2
#01# [0/1]? 0
#C1# [0/1]? 1
#S1# [0:+,1:-]: 1
#S3# [0:val:?,1:Exp]: 0
val: 1
#01# [0/1]? 0
#C1# [0/1]? 0
#01# [0/1]? 0
#C1# [0/1]? 1
#S1# [0:+,1:-]: 1
#S3# [0:val:?,1:Exp]: 0
val: 5
#01# [0/1]? 0
#C1# [0/1]? 0
11
Main>
```

Figure 7: Stream driven verbose computation



```
Hugs session for:
c:\Hughs98\lib\Prelude.hs
c:\Moje dokumenty\M Clanky\Clanky2009\Wap109\samples09\fundatastreamcalm.hs
Type :? for help
Main> :e
Main> construct Language (dataStream datastream) >>= \(lang,ds) -> evaluate lang
11
Main>
```

Figure 8: Stream driven quiet computation



```
Main> construct Language (dataStream datasubstream) >>= \(lang,ds) -> evaluate lang
val: 5
val: 1
val: 7
val: 3
val: 4
21
Main>
```

Figure 9: Alternation to a domain specific computation

For example, considering 10 S, 6 O, 5 C and 5 val channel communications for our print expression sample, data stream takes $(80 + 6 + 5 + 5 \times 64) = 411$ bits, i.e. less than 52 bytes, which is comparable with the number of computer machine instructions. However, stream is not machine code for computer architecture, but semantically strong code for extensible high-level language architecture, as we have illustrated for functional language case in this paper.

To make more reliable conclusions about the succesfull applications of streams in practice, it is necessary to develop imperative and object oriented executable language models, as well as appropriate method for automated (or at least assisted) recording of source programs in the form of streams. Second, it is necessary to select language of streams, which would be potentially useful as a target communicating language between human body and language oriented computers.

References

[1] Črepinšek, M., Mernik, M. Inferring Context-Free Grammars for Domain-Specific Languages, Conf. on Language Descriptions, Tools and Applications, LDTA 2005, April 3, Edinburgh, Scotland, UK, pp. 64–81, 2005

[2] da Cruz, D., Berón, M., Henriques, P. R, Pereira, M. J. V. Strategies for Program Inspection and Vizualizations, Proc. CSE'2008 International Scientific Conference on Computer Science and Engineering, Sep.24–26, High Tatras Stará Lesná, Slovakia, pp. 107-117, 2008

[3] Czarnecki, K., Eisenecker, U. E. Generative Programming: Methods, Tools, and Applications. Addison Wesley, 832 pp., 2005

[4] Czarnecki, K., Helsen, S. Feature-based survey of model transformation approaches, IBM Systems Journal, Vol.45, No.3, pp. 621–645, 2006

[5] Greenfield, J., Short, K., Cook, S., Kent, S. Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley, 500 pp, 2004

[6] van Gurp, J., Bosch, J., Svahnberg, M. On the notion of variability in software product lines. In Proceedings 2nd Working IEEE / IFIP Conference on Software Architecture (WICSA), pp. 45–54, 2001

[7] Javed, F., Mernik, M., Gray, M., Zhang, J., Bryant, B. R. Using a Program Transformation Engine to Infer Types in a Metamodel Recovery System, Acta Electrotechnica et Informatica, Vol.8, No.1, pp. 3–30, 2008

[8] Jones, M.P. et al. The Hugs98 User Manual. http://www.haskell.org/hugs/

[9] Lämmel, R. Grammar Adaptation, In Proc. Formal Methods Europe (FME'01), volume 2021 of LNCS, Springer-Verlag, pp. 550–570, 2001

[10] von Löwis, M., Denker, M., Nierstrasz, O. Context-oriented programming: beyond layers, Proceedings of the International Conference on Dynamic languages, ACM International Conference Proceeding Series; Vol. 286, Lugano, Switzerland, pp. 143–156, 2007

[11] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, V. G. An Overview of AspectJ. ECOOP'01, LNCS, vol. 2072, pp. 327–355, 2001

[12] Kleppe, A. A Language Description is More than a Metamodel. In: the 4th International Workshop on (Software) Language Engineering, 2007

[13] Klint, P., Lämmel, R., Verhoef, C. Toward an Engineering Discipline for Grammarware, ACM Transactions on Software Engineering and Methodology, Vol.14, No. 3, July 2005, pp. 331–380, 2005

[14] Peyton-Jones, S. (ed.): Haskell 98 Language and Libraries (The Revised Report), 2002 http://haskell.org/definition/haskell98-report.pdf

[15] Sunyé, G., Ho, W., Le Guennec, A. and Jézéquel, J. M. Using UML Action Semantics for executable modeling and beyond, Proceedings of the 13th Conference on Advanced Information Systems Engineering, Springer, pp. 433–447, 2001

[16] Wu, X., Roychoudhury, S., Bryant, B. R., Gray, J. G., Mernik, M. A Two-Dimensional Separation of Concerns for Compiler Construction. Proceedings of the 2005 ACM symposium on Applied computing, pp. 1365–1369, 2005

[17] Zhao, J., Liu, S., Wang, X., Chen, L., Wei, C. Research and design of an executable modeling language based on MOF, IEEE 9th International Conference on Computer-Aided Industrial Design & Conceptual Design, Kumming, China, 22-25 Nov. 2008, pp. 399–404, 2008