# Experiments with CamLab Program

S.O. Salnikova
Department of Cybernetics
Moscow Engineering Physics Institute
Moscow, Russia
e-mail: withmusic@rambler.ru

**Abstract**[1]

We'll examine the program CamLab, that simulate the work of Categorical abstract machine (CAM), and calculate various expressions, as a result we'll formulate rules how to work with the program, it's merits and demerits.

## 1. Introduction

The program CamLab is the illustration of the Categorical abstract machine work and conversion of $\lambda$-terms into the categorical code.

Categorical abstract machine – it's the version of computation theory, based on categorical combinatory logic. CAM uses special combinators, that in point of fact represent the instructions in a programming system.

The primary task of CAM is to calculate the value of expression. Expressions, written as $\lambda$-terms, first are converted to the form of instructions, understandable to CAM, then are executed consistently, instruction after instruction.

The Program is consists of two parts. First part is conversion $\lambda$-terms into the categorical code. Program displays all calculus, including de Brauijn's encoding. The second part illustrates the Categorical abstract machine work, represented by the table, including Term, Code and Stack columns.

Thus, while working with this program, one can understand how does compilation of the compiling code occur and than how can expression values be received.

But the program does not compile all expressions. The main is find out, what form of expressions is understandable for the program, and formulate transforming rules of the initial expression to the wanted form.

In the first part we will give the theory to understand, what the program is based on, thus creating the necessary prerequisites for the further conclusions.

Next we will give an example of the "good" expression, illustrating the work of the program.

In the third part we will view four expressions, which could not be compiled at first. We will try to establish the reasons and to give the prove, resting on the theory.

As the reasons are different for the all expressions, finally we will formulate the set of rules, using which one can effectively work with the program CamLab.

## 2. Theory

Program CamLab works both with lambda-terms and with expressions, already transformed to the categorical code. How does the compiling to the categorical code occur?

The theoretical footing is combinatory logic, cartesian closed category and the de Brauijn's numbers [1].The cartesian closed category can be interpreted like that:

1. in the category are used composition and identity;
2. in the closed category ordered pairs, the first and the second projections are added;
3. in the cartesian closed category functional spaces are added, so currying, applying and exponentiation operations are legal.

For the expression calculus the conception of environment is important. Because the calculus value of the expression represents as a reflection, in which the first element is identifier I and the second is the environment $\rho$, and the result is the value of the identifier I:

$$\|\cdot\| \cdot : \text{identifier} \times \text{environment} \rightarrow \mathit{value}$$

Let the environment be in form of:

$$[[\ldots[[(\ ), v_n], v_{n-1}]\ldots], v_0]$$

The computation theory can be represented as:

**Proceedings of the 9th International Workshop on Computer Science and Information Technologies CSIT'2007, Ufa, Russia, 2007**

(ass) $(x \circ y)z = x(yz)$

$(fst)$ $\mathrm{Fst}(x, y) = x$

$(snd)$ $\mathrm{Snd}(x, y) = y$

$(dpair)$ $\langle x, y \rangle z = (xz, yz)$

$(ac)$ $\varepsilon(\Lambda(x)y, z) = x(y, z)$

$(quote)$ $('x)y = x$

The de Brauijn's encoding is needed to avoid the collisions of binding variables while replacing the formal parameters by the actual ones. Note that important knowledge of a variable is the depth of it's binding, i.e. the number of symbols $\lambda$ between the variable and it's binding $\lambda$ (excepting the last operator). For example,

$$\lambda y.(xy.x)y = \lambda.(\lambda\lambda.\underline{1})\underline{0} \qquad (1)$$

Thus using the de Brauijn's encoding and the computation theory rules we can transform expression into the categorical code.

After compiling, calculation of the expression value occurs, using CAM instructions [2, 3].

The machine structure is consists of three parts:

T –term, in which area the calculation environment is formed, and at the the final result of calculation is written;

C -code, in which area the categorical form is written;

S –stack, in which area intermediate value is written (auxiliary memory).

The work of abstract machine is based on eight instructions, seven of which are static combinators, i.e. they are available before the calculation:

$\langle \quad , \quad \rangle \quad \mathrm{Fst} \quad \mathrm{Snd} \quad \Lambda \quad '$

The eighth instruction is the dynamic combinator $\varepsilon$, which forms instruction during the process of value calculation. Here is the table of CAM instructions in order to understand how is the each instruction executed:

**Table 1. Cycle of CAM working**

| *Initial configuration* | | | *Resulting configuration* | | |
|---|---|---|---|---|---|
| T | C | S | T' | C' | S' |
| $(s,t)$ | $car.C$ | $S$ | $s$ | $C$ | $S$ |
| $(s,t)$ | $cdr.C$ | $S$ | $t$ | $C$ | $S$ |
| $s$ | $(quoteC).C$ | $S$ | $C$ | $C$ | $S$ |
| $s$ | $(curC).C_1$ | $S$ | $(C : s)$ | $C_1$ | $S$ |
| $s$ | $push.C$ | $S$ | $s$ | $C$ | $s.S$ |
| $t$ | $swap.C$ | $s.S$ | $s$ | $C$ | $t.S$ |
| $t$ | $cons.S$ | $s.S$ | $(s,t)$ | $C$ | $S$ |
| $(C : s,t)$ | $app.C_1$ | $S$ | $(s,t)$ | $C@C_1$ | $S$ |

For the convenience the following mnemonic notation is used:

**Table 2. Mnemonic Notation for CAM Instructions**

| *Fst* | *Snd* | < | , | > | *E* | \ | ' |
|---|---|---|---|---|---|---|---|
| *Car* | *Cdr* | *Pus h* | *Swap* | *Cons* | *App* | *Cur* | *quo te* |

Now we know what work of the program is based on. Let's try to understand what expressions it works with.

## 3. Experiment

Here we will calculate a value of the expression, which cites as the example in the program.

The example is:

$$(\lambda x.((\lambda z.z[x.2])+))3 \qquad (2)$$

It is not difficult to understand that this expression is calculating the value of adding *2* to *3*. So we should receive *5* as a result.

Compiling into the categorical code gives:

*De Brauijn's encoding:*

$(\lambda x.((\lambda z.z[x,2])+))3$

$(\lambda.((\lambda z.z[\underline{1},2])+))3$

$(\lambda.((\lambda.\underline{0}[\underline{1},2])+))3$

$\lambda$ -term compiling:

$(\lambda.((\lambda.Snd[\underline{1},2])+))3$

$(\lambda.((\lambda.Snd[FstSnd,2])+))3$

$(\lambda.((\lambda.Snd[FstSnd, quote(2)])+))3$

$(\lambda.((\lambda.Snd < FstSnd, quote(2) >)+))3$

$(\lambda. < cur(< Snd, < FstSnd, quote(2) >> app),$

$cur(cdradd) > app)3$

$cur(< cur(< Snd, < FstSnd, quote(2) >> app),$

$cur(cdradd) > app)quote(3)$

*Compiled expression:*

$< cur(< cur(< Snd, < FstSnd, quote(2) >> app),$

$cur(cdradd) > app), quote(3) > app$

In the *Tab.3* we can see value calculation, using CAM instructions.

**Table 3. CAM's Work**

| N | T | C | S |
|---|---|---|---|
| 1 | () | < cur ( < Snd , < Fst Snd , quote (2) > > app ) , cur ( cdr add ) > app ) , quote (3) > app | [] |
| 2 | () | push cur ( < cur ( < Snd , < Fst Snd , quote (2) > > app ) , cur ( cdr add ) > app ) , quote (3) > app | [] |

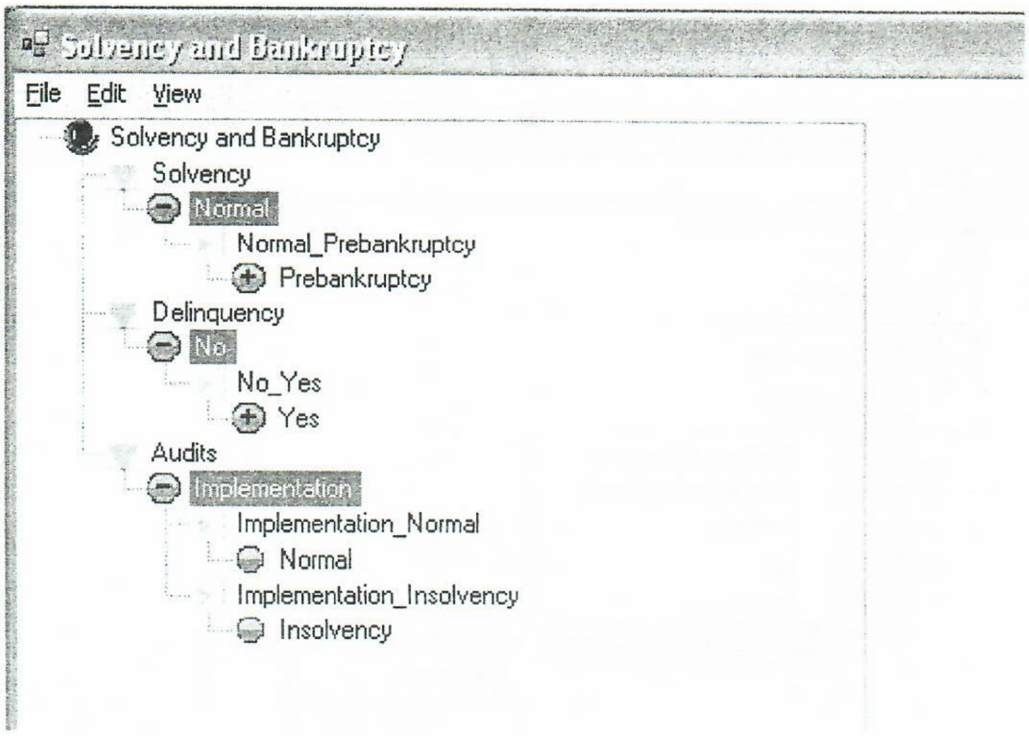


**Figure 2. Navigation Tool in Current States Mode**

- Mode «Current States». This mode builds the navigation tree on the basis of Current State Memory content (Fig. 2). That's why only current states objects can be navigated directly. All the rest model objects are not available in this mode. Mode «Current State» is a default mode. Force change to this mode can be performed by module *CurrentMode*;
- Mode «Full Model». Unlike the previous one this mode provides for user the whole hierarchy of model objects. Current states are just specially displayed in context of whole model. Mode «Full Model» can be performed by loading module named *FullTreeMode*.
- Mode «Prehistory». Current State Memory contains not only information about all current situations of dynamic XML document, but also data about previous iterations. So, in Current State Memory each situation can be characterized by its source (the jump or pass activated this state), previous state, etc. This data is available for user in *Prehistory* mode. In this mode the appropriate module creates hierarchical structure which shows the hierarchy of Current State Memory objects. Move to *Prehistory* mode is performed by module named *CreatePrehistory*.

Any user made iteration can be canceled. This task is performed by module named *Rollback*. Rollback function removes from the Current State Memory the situation which has become current after performing the last iteration. In result previous situation (according to the Current State Memory content) becomes current.

## Edit Module

As dynamic XML document is correct XML document, it's editing can be made with using standard XML-editors, such as Notepad, Microsoft XML Notepad, etc. But all this tools do not provide verification mechanism and do not into consideration inbuilt dynamic model.

That's why it was necessary to develop a tool oriented not just for XML document, but especially for XML document with inbuilt dynamic model.

In paper [1] it was suggested to use hierarchical situational models for inbuilt model development. Four main types of elements were defined in structure of dynamic XML document model. They are *model*, *submodel*, *state* and *jump*. Each of these elements has its own complicated structure.

All that elements have their own specific features of editing. These features were developed in appropriate Edit Module of Processing Tool for dynamic XML documents. Performing this module moves user interface to model editing mode. In this mode the list of available edit actions for the selected object is generating.

For example, for object type *Submodel* the following edit operations are available (Fig. 3):

- *Add New Submodel*. As inbuilt dynamic model of dynamic XML document is asynchronous the order of submodels has a meaning. Performing described operation allows adding new submodel to the position previous to the selected one.

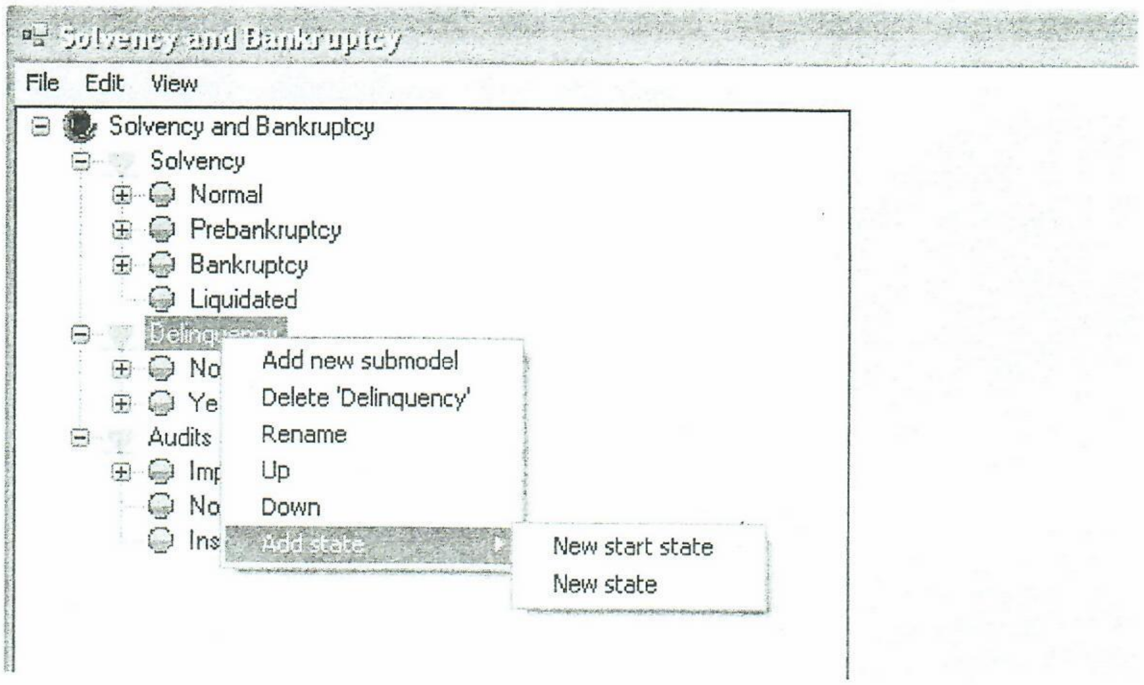


**Figure 3. Edit Mode. Edit Functions for Submodel**

- *Up (Down)*. These two operations allow user to change the current position of selected submodel node within either the whole model or the parent state.
- *Add New State.* This function allows adding new state into the selected submodel. As the model is asynchronous the order of submodel's states doesn't matter. The state becomes current as a result of performing appropriate jump (pass). The exclusion is Start State which becomes current automatically after submodel activation. That's why operation of submodel state creation is available in two variants:
  - New Start State;
  - New State.
- *Rename.* Performing this operation changes the name of selected submodel that is attribute *name* value of appropriate dynamic XML document tag.
- *Delete.* This operation allows user to delete selected submodel from the inbuilt dynamic model. Besides all child states of selected submodel will be deleted too.

All changes made in model in Edit Mode are reflected in the source file with dynamic XML document. Among other edit operation there is an operation of creating dynamic XML document. Besides there was developed a mechanism which allows save any changes on any edit step into file different from the source.

Any model changing operation is followed by the performing functions of *Verification Module*. For that reason there was developed XSD-schema [4] which defines the structure of dynamic XML document according to the conception described in paper [1]. *Verification Module* checks the structure of dynamic XML document for conformity with that schema.

## Text Edit Module

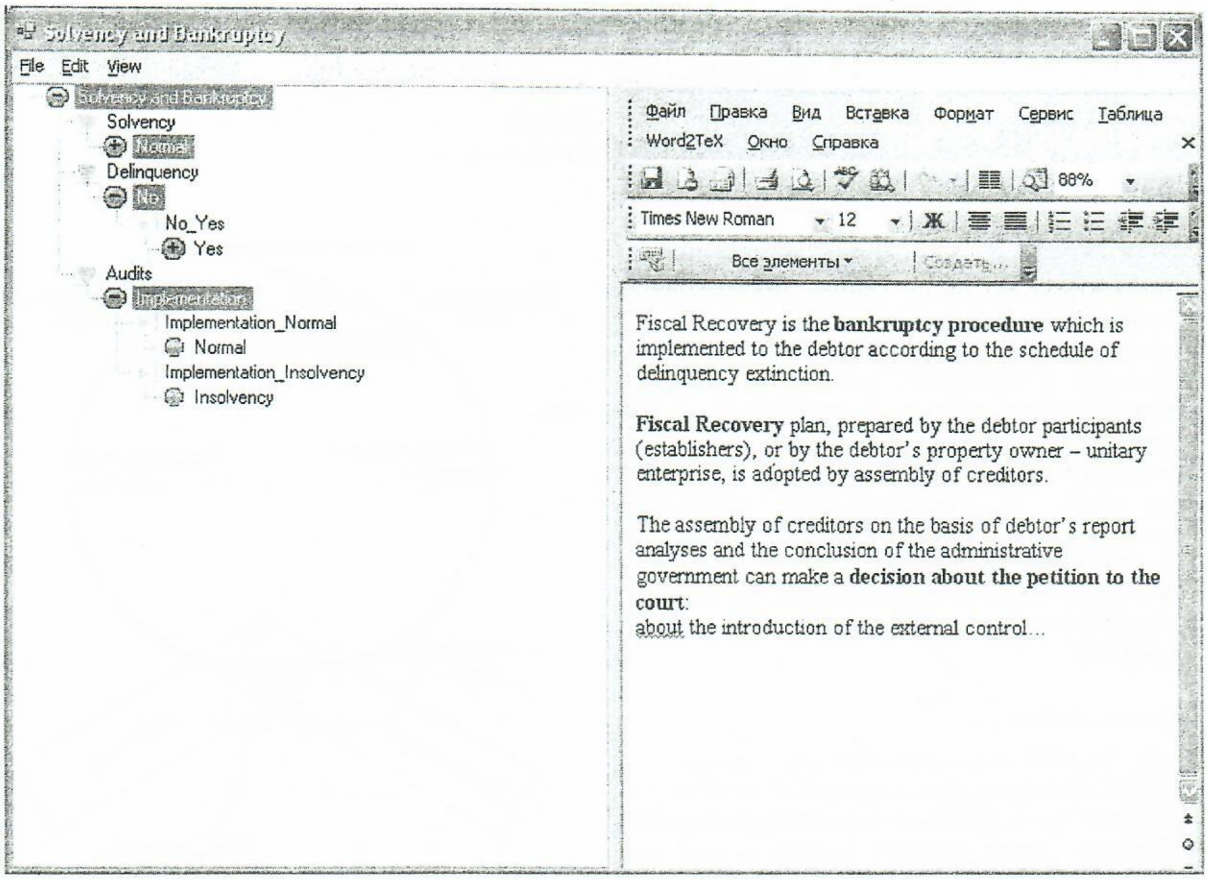
According to the conception of dynamic XML documents each state of inbuilt model has some associated object, text for example. In the developed Processing Tool these objects are stored in dynamic XML document in WordML format. Choosing a state loads *Text Edit Module* with appropriate WordML-fragment associated with selected situation as input parameter.

*Text Edit Module* provides the ability to add new and to change existing WordML-fragments. This module engines textual processor Microsoft Word 2003 which supports functions for processing documents in WordML format. The special feature of that module performing is that the textual processor starts not in separate window, but inside the main window (or form) of Processing Tool interface (Fig. 4).

All changes made in fragment including text formatting operations, objects inserting, etc, are rewrite into the source file with dynamic XML document in WordML format.

## 5. Conclusion

- Popularity of XML technologies makes it actual to apply this technology to conception of dynamic documents.
- Dynamic XML document is document in XML format with inbuilt dynamic model.



Fiscal Recovery is the **bankruptcy procedure** which is implemented to the debtor according to the schedule of delinquency extinction.

**Fiscal Recovery** plan, prepared by the debtor participants (establishers), or by the debtor's property owner – unitary enterprise, is adopted by assembly of creditors.

The assembly of creditors on the basis of debtor's report analyses and the conclusion of the administrative government can make a **decision about the petition to the court**:
about the introduction of the external control...

**Figure 4. Displaying Associated WordML Fragments of the Selected Situation**

- To provide the user work with dynamic XML documents there was developed Processing Tool. It performs such functions as navigation in inbuilt model, edit, creation of dynamic XML document, processing text and other objects associated with model situations. Each stage of processing dynamic XML document is followed by the operations of verification that is check of agreement of tested file to the conception of dynamic XML documents.

## References

1. Mironov, V.V. Dynamic XML Documents: Conception of Development and Application in Juridical Area // V. V. Mironov, G. R. Shakirova // Proceedings of the Workshop on Computer Science and Information Technologies (CSIT'2006), Karlsruhe, Germany, September 28–29, 2006. Vol. 1, P. 68–72
2. Mironov, V.V. Dynamic Electronic Documents / V. V. Mironov, T. A. Garifullin // USATU Bulletin. 2004. No 1 (9). P. 185–189 (In Russian).
3. Mironov, V.V. Interpretation of XML Documents with Inbuilt Dynamic Model / V. V. Mironov, G. R. Shakirova // USATU Bulletin. 2007. No 2 (20). P. 88–97 (In Russian).
4. XML Specification. http://www.w3.org/TR/REC-xml. [electronic source]