ON APPLYING STOCHASTIC PROBLEMS IN HIGHER-ORDER THEORIES

Valerie NOVITZKÁ, Viliam SLODIČÁK
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice, Letná 9, 042 00 Košice
e-mail: valerie.novitzka@tuke.sk, viliam.slodicak@tuke.sk

SUMMARY

In the article presented we deal with one method of stochastic programming – probabilistic programming. It is a programming in which the probabilities of the values of the variables are of interest. Our approach is that solving problem we can construct in logical reasoning over some mathematical theories. In this approach we use category theory for construction of type theory and of the logical system. Then we formulate the logical theory to enclose the solution of the problem. These steps we show at well-known examples: the random number generator and the Blackjack game.

Keywords: *probabilistic programming, type theory, logical theory, linear logic*

1. INTRODUCTION

We usually model many decision problems by mathematical programs which seek to maximize or minimize some objective which is a function of the decisions. Decisions are represented by variables. Objective and constraints are functions of the variables, and problem data. Stochastic programs are mathematical programs where some of the data incorporated into objective or constraints are uncertain. One of the methods of stochastic programming - probabilistic programming - refers to programming in which the probabilities of the values of variables are of interest [5]. The term "predicative programming" describes the programming according to the first-order semantics. The purpose of this paper is to show how the stochastic programming can be applied in the mathematical theory of programming.

2. BASIC NOTIONS

For every problem we need to define variables. Usually we deal with inputs and outputs – that's why we define:

- list of input variables: $\sigma = x_1, x_2, \mathrm{K}$
- list of output variables: $\sigma' = x_1', x_2', \mathrm{K}$

Next we introduce types for these variables and define typed context.

In this part we briefly introduce basic notions and recent results which are necessary for applying them in next research.

2.1. Probabilistic and predicative programming

Probabilistic programming is a programming in which the probabilities of the values of the variables are of interest. For example, if we know the probability distribution from which the inputs are drawn, we may calculate the probability distributions of outputs. *Predicative programming* is a way of writing programs so that each programming step is proven as it is made [5]. First step is to decide what quantities are of interest, and to introduce a variable for each such quantity. A *specification* is defined as a boolean expression whose variables represent the quantities of interest. In a specification, some variables may represent inputs, and some may represent outputs. A specification is implemented on a computer when, for any values of the input variables, the computer generates values of the output variables to satisfy the specification.

2.2. The Linear logic

In mathematical logic, linear logic is a type of substructural logic that denies the structural rules of weakening and contraction; it allows only restricted versions of that rules. Girard's linear logic [4] (introduced in 1987) becames a natural mean for research and applications in computer science. It has offered great promise, as a formalism particularly well-suited to serve at the interface between logic and computer science. It is able to describe systems that are changed during they are used. Using Curry-Howard correspondence, propositions of linear logic are interpreted as types [3]. This paradigm has been a cornerstone of new approach concerning connections between intuitionistic logic, functional programming and category theory. The interpretation in linear logic is of hypotheses as resources: every hypothesis must be consumed exactly once in a proof. The most important feature of linear logic is that formulae are consider as actions. This differs from usual logics where the governing judgement is of truth, which may be freely used as many times as necessary. While classical and intuitionistic logics treat with the sentences that are always true or false, in linear logic formulae describe *actions* and the truth values depend on an internal state of a dynamic system. For instance, linear implication $\varphi \multimap \psi$ is causal, i.e. the action described by φ is a cause of the action described by ψ; the formula φ does not hold after

linear implication. Linear logic uses two conjunctions: multiplicative $\varphi \otimes \psi$ expressing that both actions will be performed ('$\otimes$' is read as "times") and additive $\varphi \& \psi$ expressing that only one of two actions will be performed and we shall decide which one (external indeterminism). Intuitionistic linear logic uses additive disjunction '$\oplus$' which expresses that only one of two actions will be performed but we cannot decide which one; this is a statement of internal indeterminism. Dual of multiplicative conjunction is multiplicative disjunction that uses the symbol '$\wp$' and is read as "par".

We consider here intuitionistic linear logic because we would like to use it to describe program execution. Precisely, reduction of linear terms corresponding to proofs in intuitionistic linear logic can be regarded as computation of programs [10].

2.3. The Type theory

As the first step in problem solving we have to introduce types of data structures and operations on them [8]. We enclose types and operations for problem solution in the well-known notion of algebraic specification - a many-typed signature. A *many-typed signature* $\Sigma = (T, \mathcal{F})$ is a couple consisting of finite set T of (the names of) *basic types* necessary for a given problem denoted by (possibly indexed) symbols $\tau_1, \tau_2, \mathrm{K}, \tau_i$ and of finite collection $\mathcal{F}$ of *function symbols*. Every function symbol $f \in \mathcal{F}$ is of the form $f : \tau_1, \mathrm{K}, \tau_n \to \tau_{n+1}$ for some natural number n, i.e. it takes inputs of types $\tau_1, \mathrm{K}, \tau_n$ and yields an output of a type τ_{n+1}.

To form terms we assume a countably infinite set of variables $Var = (v_1, v_2, \mathrm{K})$ that range over basic types. Every variable in a term has to be typed, i.e. it has assigned a unique basic type from a signature Σ, written $v : \tau$ as a variable declaration. A finite sequence $\Gamma = (v_1 : \tau_1, \mathrm{K}\ v_n : \tau_n)$ of variable declarations is called a *type context*. A sequent of term calculus has a form

$$\Gamma \vdash M : \tau$$

and it is read as a term M of type τ with variables in Γ. From basic types we construct more complex Church's types using type constructors '$\times$', '$+$' and '$\to$' [7]. If $\tau, \theta \in T$ then $\tau \times \theta$ is a product type, $\tau + \theta$ is a coproduct (sum) type, $\tau \to \theta$ is an arrow (function) type.

In constructing the classifying category of type contexts containing variable declarations of Church's types we have the advantage that we can use Church's types instead type contexts as category objects [8, 12]. The product type ensures that any term $M : \tau$ of a Church's type τ

$$v_1 : \theta_1, \mathrm{K}, v_n : \theta_n \vdash M : \tau$$

is in one-to-one correspondence with a term

$$v : \theta_1 \times \mathrm{K} \times \theta_n \vdash N : \tau$$

which's context consists of a single variable v of product type $\theta_1 \times \mathrm{K} \times \theta_n$. Now we are able to construct type theory as linear classifying category $LinCl(\Sigma)$, which is a symmetric monoidal closed category containing linear types as objects and linear terms as morphisms.

2.4. The Category theory

Category theory [1] is a part of mathematics. It was introduced in 1945 and its importance for theoretical computer science growth in last decade. Categorical methods are already well-established for the semantical foundation of type theory, data type specification frameworks and graph transformation [2, 15]. Categories are structures which enable to work with objects of arbitrary complexity. Fundamentals of category theory are relations between objects. These relations are expressed by morphisms. A fibration (Fig. 1) is a special functor which allows indexing and substitution. The properties of fibration can be found in [6, 9]. The classifying category $LinCl(\Sigma)$ is a base category for fibration. Every subcategory is a fibre over each type. A fibre contains linear logic over that type and type indexes the fibre. The objects of fibre are formulas and morphisms are sequents. Term t in classifying category induces the substitute functor t^*, which is a part of proof tree. Then all the total category expresses the linear logic over the given signature together with defined set of axioms.

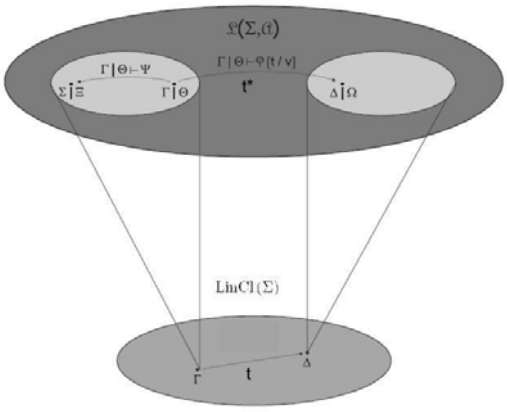

Fig. 1 Linear logic over type theory

In the Fig. 1

- $LL(\Sigma, A)$ is fibration containing linear logic over type theory with axioms A, its subcategories over contexts are fibres, i.e. logics over contexts

- $LinCl(\Sigma)$ is the type theory for solved problem
- t is the term in classifying category
- t^* is the substitute functor, which is a part of proof tree
- l is the fibration

2.5. The indeterminism

In this article we deal with indeterminism. According to some authors [5], nondeterminism comes in several varieties: angelic, demonic, oblivious and prescient. To illustrate the differences, consider

$$x := rand(2); y := 0 \textbf{ or } y := 1$$

and we want the result $x' = y'$. If **or** is angelic nondeterminism, it chooses between its operands $y := 0$ and $y := 1$ in such a way that the desired result $x' = y'$ is always achieved.

If **or** is demonic indeterminism, it chooses between its operands in such a way that the desired result is never achieved. Both angelic and demonic indeterminism require knowledge of the value of variable x when choosing between assignments to variable y.

Oblivious nondeterminism is restricted to making a choice without looking at the current (or past) state. It achieves $x' = y'$ half the time. Now consider

$$x := 0 \textbf{ or } x := 1; y := rand(2)$$

and we want $x' = y'$. If **or** is angelically prescient, x will be chosen to match the future value of y, always achieving $x' = y'$. If **or** is demonically prescient, x will be chosen to avoid the future value of y, never achieving $x' = y'$. If **or** is not prescient, then $x' = y'$ is achieved half the time.

In predicative programming, indeterminism is disjunction. Angelic, demonic, oblivious, and prescient are not kinds of indeterminism, but ways of refining indeterminism. In the example

$$x := rand(2); y := 0 \vee y := 1$$

with desired result $x' = y'$ we can refine the indeterminism angelically as $y := x$ or demonically as $y := 1 - x$ or obliviously as either $y := 0$ or $y := 1$.

In the example

$$x := 0 \vee x := 1; y := rand(2)$$

with desired result $x' = y'$ we first have to replace $rand(2)$ by boolean variable r having probability 1/2. Then we can refine the indeterminism with angelic prescience as $x := r$ or with demonic prescience as $x := 1 - r$ or without prescience as either $x := 0$ or $x := 1$.

2.6. Constructing the theory

The solution of the problem we can enclose into the logical theory [11]. A logical theory is a list of basic type symbols $\tau_1, \ldots, \tau_n$, terms called basic constant symbols $c_1, \ldots, c_m$ in the parameters $\tau_1, \ldots, \tau_n$, and the sentences $\alpha_1, \ldots, \alpha_k$ in the parameters $\tau_1, \ldots, \tau_n$, $c_1, \ldots, c_m$. We consider

$$T' = (\tau_1, \ldots, \tau_n, c_1, \ldots, c_m, \alpha_1, \ldots, \alpha_k)$$

as a theory, where $(\tau_1, \ldots, \tau_n, c_1, \ldots, c_m)$ is the basic language of T' and $\alpha_1, \ldots, \alpha_k$ are the axioms of T'. This theory is in correspondence with the theory $T = (\tau, c, \alpha)$ where

- the list of types we replace by one product type $\tau = \tau_1 \times \ldots \times \tau_n$
- the list of constants we replace by one constant $c = (c_1, \ldots, c_m)$
- the list of axioms $\alpha_1, \ldots, \alpha_k$ we can replace by the axiom $\alpha_1 \wedge \ldots \wedge \alpha_k$ but only in classical logic

For constructing models of the theories we use special categories called *toposes* (or *topoi*) [11, 13]. The objects of topos correspond to types, morphisms correspond to constant symbols and for axioms there is the subobject classifier [14].

3. EXAMPLES OF STOCHASTIC PROBLEMS

Our idea is that to solve large scientific problems by mathematical machines we always start with the formulation of their theoretical foundations [10]. We need to formalize these theoretical foundations as logical reasoning in some mathematical theories because the programs should really prove the correctness of their results. Program consists of data structures and algorithms. Data structures always have some types. These types can frequently be very complex structures as algebraic structures, vector spaces, etc. In such cases set theory does not suffice our needs to describe and represent them. Mathematics provides a useful discipline - category theory that enables us to work with the structures of arbitrary complexity and describe their properties and relations between them. Using of category theory in computer science has extremely growth in the last decade [2]. In the next parts we present some aspects about probabilistic predicative programming and follow the interpretation of this style in categorical terms [12].

To illustrate the combination of indeterminism and probability, we look at well-known problems of random number generator and Blackjack game. In these situations we have to deal with values that are uncertain from view of external observer. In [12] we showed how to construct the solution of very famous Monty Hall problem in categorical terms.

3.1. Random Number Generator

Many programming languages provide a random number generator; sometimes called a "pseudo-random" number generator. The usual notation is functional and the usual result is a value whose distribution is uniform over nonempty finite range. If $n : nat$, we use the notation $rand(n)$ for a generator that produces natural numbers uniformly distributed over the range $0,..n$ - from (including) 0 to (excluding) n. So $rand(n)$ has value r with probability $(r : 0,..n)/n$.

Functional notation for a random number generator is inconsistent. Since $x = x$ is a law, we should be able to simplify $rand(n) = rand(n)$ to T, but we cannot because the two occurrences of $rand(n)$ might generate different numbers. Since $x + x = 2 \times x$ is a law, we should be able to simplify $rand(n) + rand(n) = 2 \times rand(n)$, but we cannot. To restore consistency, we replace each use of $rand(n)$ with a fresh integer variable r whose value has probability $(r : 0,..n)/n$ before we do anything else. Or we can replace each use of $rand(n)$ with a fresh variable $r : 0,..n$ whose value has probability $1/n$. But this is a mathematical variable, not a state variable; in other words, there is no r'. For example, in one state variable x:

$$
\begin{aligned}
& x := rand(2); x := x + rand(3) \\
& \equiv \sum r : 0,..2. \sum s : 0,..3. (x := r; x := x + s) \\
& \equiv \sum r : 0,..2. \sum s : 0,..3. (x' = r + s) \\
& \equiv (x' = 0)/6 + (x' = 1)/3 + (x' = 2)/3 + (x' = 3)/6
\end{aligned}
$$

which says that x' has values 0 and 3 one-sixth of the time and values 1 and 2 one-third of the time.

In the process of construction the categorical solution we have to define:

- Types: $\theta = 0,..2$, $\tau = 0,..3$ (subtypes *nat*)
- Inputs: $\Gamma = (r : \theta, s : \tau)$
- Outputs: $\Gamma' = (x' : \tau)$
- Terms: $t : \Gamma \to \Gamma'$, $x' = t(r,s) : r, s\ \alpha\ r + s$

We formulate the logical formulas describing the problem: $\psi_1 : r = rand(2)$ and $\psi_2 : s = rand(3)$:

$$\psi_1 \text{—} o \psi_2$$

But the result of the problem we can describe by another formulas: $\varphi_i : x' = i$. So the whole result has the form

$$\varphi_1 \oplus \varphi_2 \oplus \varphi_3 \oplus \varphi_4$$

$$(x' = 0) \oplus (x' = 1) \oplus (x' = 2) \oplus (x' = 3)$$

All that process was formulated in [12]. Finally the logical theory for the problem is

$$T = (\theta \times \tau; t; \varphi_1 \oplus \varphi_2 \oplus \varphi_3 \oplus \varphi_4)$$

3.2. The Blackjack Game

This example is a simplified version of the card game known as *Blackjack*. Player is dealt a card from a deck; its value is in the range 1 through 13 inclusive. Player may stop with just one card, or has a second card if he wants. Player's object is to get a total as near as possible to 14 , but not over 14 . The strategy is to take a second card if the first is under 7. Assuming each card value has equal probability (actually, the second card drawn has a diminished probability of having the same value as the first card drawn, but let's ignore that complication), we represent a card as $1 + rand(13)$. In one variable x, the game is

$$
\begin{aligned}
& x := 1 + rand(13); \\
& \textbf{if } (x < 7) \textbf{ then } x := x + 1 + rand(13) \textbf{ else } ok
\end{aligned}
$$

First we introduce variables $c, d : 0,..13$ for two uses of $rand$: each with probability 1/13. The program becomes

$$
\begin{aligned}
& x := 1 + c; \\
& \textbf{if } (x < 7) \textbf{ then } x := x + d + 1 \textbf{ else } ok
\end{aligned}
$$

or by substitution

$$\textbf{if } (1 + c < 7) \textbf{ then } x' = c + d + 2 \textbf{ else } x' = c + 1$$

Then x' has the distribution

$$\sum c, d : 0,..13. \begin{pmatrix} \textbf{if } (1 + c < 7) \textbf{ then } x' = c + d + 2 \\ \textbf{else } x' = c + 1 \end{pmatrix} \cdot (1/13)^2$$

by several omitted steps

$$
\begin{aligned}
K = & ((2 \le x' < 7) \cdot (x' - 1) + (7 \le x' < 14) \cdot 19 \\
& + (14 \le x' < 20) \cdot (20 - x'))/169
\end{aligned}
$$

Similarly as in the previous example, we define:

Types: $\tau = 0,..13$

Inputs: $\Gamma = (c : \tau, d : \tau)$

Outputs: $\Gamma' = (x' : \tau)$

Terms: $t_1 : \Gamma \to \Gamma'$; $x' = t_1(c,c) : c, d\ \alpha\ c + d + 2$,
$t_2 : \Gamma \to \Gamma'$; $x' = t_2(c) : c\ \alpha\ c + 1$

Logical formulas describing the problem are then:

$$\varphi_0 : (c = 1 + rand(13)) \otimes (d = 1 + rand(13))$$

$$\varphi_1 : (x < 7)$$

$\psi_1 : x' = c + d + 2$

$\psi_2 : x' = c + 1$

and the whole problem can be expressed as

$$\varphi_0 \multimap o((\varphi_1 \multimap o\psi_1) \& (\varphi_1^{\perp} \multimap o\psi_2))$$

The logical theory for the problem is

$$T = (\tau; (t_1, t_2); \varphi_0 \multimap o((\varphi_1 \multimap o\psi_1) \& (\varphi_1^{\perp} \multimap o\psi_2)))$$

In this example we used strategy „under 7“. There are many variations of that game, i.e. „under 8“ strategy, etc. We can find distribution and then construct the solution in similar ways.

4. CONCLUSION

In this article we presented some aspects about constructing the solution of solving stochastic problem in the mathematical theory of programming. We formulated necessary steps: type theory for given problem, logical theory of the problem and showed them in examples of random number generator and the Blackjack game. We are able to model the logical theory in topos, so our next goal is to find the relation between the topos of theories and the category of linear logic for the given problem.

This work was supported by VEGA Grant No.1/2181/05: Mathematical Theory of Programming and Its Application in the Methods of Stochastic Programming.

REFERENCES

[1] Barr M., Wells C.: Category Theory for Computing Science, Prentice Hall International, 1990, ISBN 0-13-120486-6, pp. 1-432

[2] Ehrig H.: Applied and computational category theory, in Bulletin of the EATCS no. 89, 2006, European Association for Theoretical Computer Science, pp.134-135

[3] Girard J., Taylor P., Lafont Y.: Proofs and Types, Cambridge University Press, 1990, pp. 1–175

[4] Girard J.-Y.: Linear logic: Its syntax and semantics, Cambridge University Press, 2003, pp. 1–42

[5] Hehner E.: Probabilistic Predicative Programming, Department of Computer Science, University of Toronto, Toronto, Canada, 2004, http://citeseer.ist.psu.edu/626001.html

[6] Jacobs B.: Categorical Logic and Type Theory, 1999, Elsevier

[7] Novitzká V.: Church's types in logical reasoning on programming, in Acta Electrotechnica et Informatica, Košice, 2006, pp. 27–31

[8] Novitzká V., Mihályi D., Slodičák V.: How to combine Church's and linear types, in ECI'2006, Košice-Herľany, elfa s.r.o., 2006, pp. 128-133

[9] Novitzká V., Mihályi D., Slodičák V.: Categorical models of logical systems in the mathematical theory of programming, in MaCS'06 6th Joint Conference on Mathematics and Computer Science, Book of Abstracts, University of Pécs, Hungary, 2006, pp. 13-14

[10] Novitzká, V., Mihályi, D., Slodičák, V. Foundations of Correct Programming of Mathematical Machines, elfa s.r.o., 2007, pp. 1–5

[11] Novitzká V., Slodičák V., Verbová A.: On Modeling Higher-Order Logic, in Ivan Plander (ed.): Proceeding from International Scientific Conference Informatics 2007, Bratislava, June 2007, pp. 156-162

[12] Slodičák V., Novitzká V., Verbová A.: On applying stochastic programming in mathematical theory of programming, International Multiconference on Computer Science and Information Technology: 1st Workshop on Advances in Programming Languages (WAPL'07), Wisla, Poland, October 15-17, Polish Information Processing Society, 2007, 2, pp. 1147-1150, ISSN 1896-7094

[13] Barr M., Wells C. Toposes, Triples and Theories. Springer-Verlag, 2002, pp. 1–326.

[14] Awodey S. Logic In Topoi: Functorial semantics for higher-order logic. PhD thesis, The University of Chicago, Chicago, IL, Marec 1997.

[15] Taylor P. Practical Foundations of Mathematics. Cambridge University Press, 1999, pp. 1–572. ISBN 0-521-63107-6.

[16] Vokorokos L., Kleinová A., Látka O.: Network Security on the Intrusion Detection System Level, Proceedings of IEEE 10th International Conference on Intelligent Engineering Systems, London, Jun 26th – 28th, 2006, pp. 270-275, ISBN 1-4244-9708-8.

BIOGRAPHIES

Valerie Novitzká defended her PhD Thesis: On semantics of specification languages at Hungarian Academy of Sciences in 1989. She works at Department of Computers and Informatics from 1998, firstly as Assistent Professor, from 2004 as Associated Professor. Her research areas covers category theory, categorical logic, type theory, classical and linear logic and theoretical foundations of program development.

Viliam Slodičák was born in 1981. He graduated at Technical university of Košice, Slovakia. He is working on his PhD. degree at the Department of Computers and Informatics FEEI, Technical university of Košice, Slovakia. His scientific research area are topos theory, categorical logic and linear logic.