Linear logic automata

Max I. Kanovich

Russian Humanities State University, Moscow, Russia and Tohoku University, 1-1-2-Chome, Katahira, Aoba-ku, Sendai, Japan

Received 31 January 1995; communicated by A. Nerode

Abstract

A Linear Logic automaton is a hybrid of a finite automaton and a non-deterministic Petri net. LL automata commands are represented by propositional Horn Linear Logic formulas. Computations performed by LL automata directly correspond to cut-free derivations in Linear Logic.

A programming language of LL automata is developed in which typical sequential, non-deterministic and parallel programming constructs are expressed in the natural way.

All non-deterministic computations, e.g. computations performed by programs built up of guarded commands in the Dijkstra's approach to non-deterministic programming, are directly simulated within the framework of Linear Logic automata, and thereby within the Horn framework of Linear Logic.

1. Introduction and summary

Linear Logic was introduced by Girard [9] as a resource-sensitive refinement of classical logic. Linear Logic turned out to be more expressive than traditional classical or intuitionistic logic, even if we consider the modalized versions of those logics. In particular, Lincoln et al. [25] proved the undecidability of full propositional Linear Logic. Their undecidability proof [25] consists of a reduction from the halting problem for and-branching two counter machines without zero-test (specified in the same [25]) to a decision problem in propositional Linear Logic. Later, in [17, 18] the halting problem for standard many-counter Minsky machines proper (introduced originally in [31, 24]) was proved to be simulated directly in propositional Linear Logic, even if we used nothing but Horn-like propositional formulas. In the latter case, we invoked our

This research was supported in part by the International Science Foundation under Grant No. NFQ000, and by the Project on Parallel Symbolic Computing conducted by Takayasu Ito (Tohoku University).

* Corresponding author. E-mail: maxk@ito.ecei.tohoku.ac.jp.

0168-0072/96/$15.00 © 1996 Elsevier Science B.V. All rights reserved

SSDI 0168-0072(95)00035-6
intuition about derivability in Linear Logic based on the adequate complexity interpretation developed for Horn fragments of Linear Logic in [17, 18].

These considerations give rise to the task of looking for a general computational model behind derivations in Linear Logic in which all or most useful programming constructs can be directly expressed.

In this paper we aim to introduce such a general automata model behind Horn Linear Logic, and establish the exact level of the expressive power of Linear Logic Automata introduced.

As to an intuitive motivation, the main idea is the following:

(a) Atomic propositions, as well as "products" of them, are understood as "states" or "configurations" of a system.

(b) Purely Horn implications \((X \rightarrow Y)\) are perceived as instructions to change the "state" \(X\) for the "state" \(Y\).

(c) The case where the reaction of the system is assumed to be non-deterministic is described with the help of instructions of the form \((X \rightarrow (Y_1 \oplus Y_2 \oplus Y_m))\). The firing of such an instruction results in the change of the "state" \(X\) either for the "state" \(Y_1\), or for the "state" \(Y_2\), or \(\ldots\), or for the "state" \(Y_m\). But we do not know in advance which of \(m\) alternatives will be chosen at a given occasion.

In order to distinguish the information and resources aspects in the dynamic behaviour of a given system, we consider configurations of the system as pairs \((q_i, \kappa)\) where

(i) the first component \(q_i\) is conceived of as an information state, or a state of the automaton,

(ii) while the second component \(\kappa\) is conceived of as a distribution of resources, or a state of the memory.

Accordingly, the execution step performed by an LL Automata command will consist of two parts:

(i) The current state of the automaton \(q_i\) will be changed for the corresponding state of the automaton \(q_j\) in the finite automata manner.\(^1\)

(ii) While the current distribution-of-resources \(\kappa\) will be rearranged in the Petri nets style.

Petri Nets are well-known as one of the most fundamental formal models for the representations and analysis of the resource sensitive behaviour of interacting parallel processes. The notion of non-deterministic Petri nets was proposed in [20].

1.1. Automata over resources

According to what has been said, a Linear Logic Automaton is introduced as a hybrid of a finite automaton and a non-deterministic Petri net. Namely, a Linear Logic

\(^1\)This \(q_i\) will be changed for one of the corresponding states of the automaton \(q_{ik}\) in the case of the non-deterministic reaction of the system.
Automaton $A$ is defined as a tuple $A = (Q, Q_0, F, T)$ where $Q$ is a finite set of states of the automaton, $Q_0$ is the initial state, $F$ is a finite set of final states, and $T$ is a finite set of commands.

LL Automata commands are represented by Horn-like Linear Logic formulas of two forms:

$$(q_i \otimes (p_1^{c_1} \otimes p_2^{c_2} \otimes \cdots \otimes p_n^{c_n})) \rightarrow (q_i \otimes (p_1^{d_1} \otimes p_2^{d_2} \otimes \cdots \otimes p_n^{d_n}))$$

$$(q_i \rightarrow (q_{j_1} \oplus q_{j_2} \oplus \cdots \oplus q_{j_m})),$$

where

(a) "$q$-part" describes the action over the states of the automata,

(b) and "$p$-part" indicates that $c_1, c_2, \ldots, c_n$ tokens should be removed from and

then $d_1, d_2, \ldots, d_n$ tokens should be added to the corresponding places.\(^2\)

E.g., the execution step performed by an LL Automaton command of the purely Horn form $((q_i \otimes (p_1^1)) \rightarrow (q_i \otimes (p_2^3 \otimes p_3^5)))$ consists of the following two actions:

(i) $q_i$ is changed for $q_j$,

(ii) 4 tokens are removed from the first place, and then 3 tokens and 5 tokens are

added to the second and third places, respectively.

While the performance of an LL Automaton command of the branching form $(q_i \rightarrow (q_{j_1} \oplus q_{j_2}))$ yields a non-deterministic result:

(i) $q_i$ is changed either for $q_{j_1}$, or for $q_{j_2}$.

(See details in Definition 3.4.)

Whereas any computation $C$ performed by a given Linear Logic Automaton $A$ should envisage all possible reactions of the system, such a $C$ forms generally a finite rooted tree built up of configurations, with each of its branches leading from the initial configuration (in the root of $C$) to a final configuration.

It should be noted that our definitions of Linear Logic Automata and their computations gained nothing from Linear Logic but the syntax of LL Automata commands together with the naive interpretation of it. Of course, for such notational purposes we could use the syntax of any propositional logical system.

The proper reason behind the choice of the Linear Logic syntax and calling such automata over resources Linear Logic Automata is that there exists\(^3\) a direct correspondence between

(a) computations $C$ performed by a given LL Automaton $A$:

$$A = (Q, Q_0, \{q_0, q_1, q_2, \ldots, q_k\}, F),$$

that are leading from a given state of the memory $\kappa$ into states of the memory belonging to some set $\{\xi_0, \xi_1, \xi_2, \ldots, \xi_k\}$,

\(^2\) The memory is assumed to consist of registers (places) that can contain non-negative integers (non-negative numbers of tokens). In the Petri nets interpretation, $k$ tokens accommodated in some place indicate that exactly $k$ resources are available at this place.

\(^3\) In contrast to any other propositional logical system.
(b) and cut-free Linear Logic derivations of the following Horn-like sequent\(^4,\,5\) (See Theorem 3.1.)

\[
q_{A'}, \kappa, !\mathcal{F} \vdash ((q_0 \otimes \tilde{\zeta}_0) \oplus (q_1 \otimes \tilde{\zeta}_1) \oplus (q_2 \otimes \tilde{\zeta}_2) \oplus \cdots \oplus (q_k \otimes \tilde{\zeta}_k)).
\]

As a rule, Automata are being constructed to perform one of the following two tasks:

(i) They compute functions from the states of the memory to the states of the memory, or

(ii) They recognize predicates over the states of the memory.

This means that outputs\(^6\) produced by an automaton for a given input \(\kappa\) are assumed to be unique.

As for Linear Logic Automata, the problem is that for one and the same input \(\kappa\) some LL Automaton \(\mathcal{A}\) can develop non-isomorphic computations that yield different outputs, so that it is impossible to define the proper output \(\mathcal{A}[\kappa]\). In other words, we can meet, for instance, with the case where the corresponding sequent has at least two "disjoint" derivations in Linear Logic represented by different proof nets [9, 12] (see Example 4.1).

In order to provide the desired uniqueness and thereby the correct definitions of functions computable by LL automata and predicates recognizable by LL automata, we introduce the notion of properly terminated computations. A properly terminated computation \(\mathcal{C}\) is specified as a computation such that, for a certain final state of the automaton \(q_0\), the complete garbage collection is guaranteed along all branches of \(\mathcal{C}\) leading to this state \(q_0\).

Now the above uniqueness problem is solved as follows.

For the purposes of computation of function and predicate recognition we invoke the Linear Logic automata that can provide us with the strongest version of the desired uniqueness, namely, all properly terminated computations, performed by such an LL automaton \(\mathcal{A}\) for a given input \(\kappa\), are to be isomorphic with respect to their outputs and, moreover, with respect to their whole structure (see Definitions 4.3 and 6.1).

On the basis of LL Automata computability we are able to develop a standard Automata Theory in the natural way. In particular,

(a) The class of all partial multivalued functions computable by LL automata is proved to be closed with respect to basic sequential operations, like sequential composition (Theorem 5.1), if-then-else selection, etc.

(b) Moreover, the class of all LL computable functions is proved to be closed also with respect to basic parallel operations, like non-deterministic selection (Theorem 9.2) and parallel repetition (Theorem 9.3).

\(^4\) Where \(\kappa, \zeta_0, \zeta_1, \ldots, \zeta_k\) are represented by certain tensor products of propositional literals \(\bar{\kappa}, \bar{\zeta}_0, \bar{\zeta}_1, \ldots, \bar{\zeta}_k\), respectively.

\(^5\) Where \(\mathcal{F}\) stands for the multiset resulting from putting the modal storage operator \(!\) before each formula in \(\mathcal{F}\).

\(^6\) Possibly multivalued.
(c) The class of all partial predicates recognizable by LL Automata is proved to be closed with respect to all Boolean operations, even if we use both “parallel” and “sequential” versions of the Boolean operations (Theorem 7.1).

1.2. A basic non-deterministic programming language

In order to estimate the expressive power of LL Automata from the programming point of view, we introduce a programming language, \( \mathcal{M} \mathcal{V} \mathcal{F} \), aimed at programming Multi-Valued Functions from the states of the memory to the states of the memory.

Being a version of the language of non-deterministic programs, \( \mathcal{M} \mathcal{V} \mathcal{F} \) incorporates typical sequential and non-deterministic programming constructs, like alternative and repetitive constructs introduced by Dijkstra [6, 7].

The basic unit of the syntax of \( \mathcal{M} \mathcal{V} \mathcal{F} \) is the guarded statement which is defined to be one of the following expressions:

(i) an assignment statement,
(ii) a sequential construct,
(iii) a “sequential” if-then-else construct,
(iv) a “non-deterministic” alternative construct,
(v) a “sequential-parallel” while construct,
(vi) a “parallel” repetitive construct.

As building blocks for alternative and repetitive constructs, we use guarded commands of the form

\[
\square B \rightarrow S
\]

where \( S \) is a guarded statement, and the guard \( B \) is a Boolean expression.

The formal semantics of \( \mathcal{M} \mathcal{V} \mathcal{F} \) is given through a compositional interpretation of the guarded statements \( S \) as terms in the input–output manner. Namely, following the schemata of a guarded statement \( S \), we compose a multivalued mapping from initial states of the memory \( \kappa \) to final states of the memory \( \zeta \):

\[
S: \kappa \rightarrow S[\kappa]
\]

where \( S[\kappa] \) is the set of all final states \( \zeta \), to which \( S \) can “lead” from \( \kappa \).

**Example 1.1.** The argument that the final state of the memory in which a program terminates is not necessarily deterministic can be illustrated with the following guarded statement \( S_0 \):

\[
S_0 = \begin{cases} 
\text{if} & (x_1 \leq x_2) \rightarrow x_3 := 2 \\
(x_2 \leq x_1) \rightarrow x_3 := 1 \\
\text{fi}
\end{cases}
\]

“For given \( x_1 \) and \( x_2 \), the program \( S_0 \) determines a position \( x_3 \) of the maximum \( \max\{x_1, x_2\} \) in the ordered pair \( (x_1, x_2) \).”
It is readily seen that, for instance, in the case where \( x_1 = x_2 = 8 \), our \( S_0 \) yields non-deterministic output: \( x_3 \) becomes either 1, or 2:

\[
S_0([8, 8, 0, 0, \ldots, 0, \ldots]) = \{(8, 8, 2, 0, \ldots, 0, \ldots), (8, 8, 1, 0, \ldots, 0, \ldots)\}.
\]

1.3. Summary of the paper

It is well-known that programming languages like \( \mathcal{MV} \mathcal{F} \) are extremely powerful. In particular, any execution control, that is conceivable from the programming point of view, is constructible within the framework of \( \mathcal{MV} \mathcal{F} \).

The main expressiveness result of this paper is that each of guarded statements \( S \) is directly simulated by a Linear Logic Automaton \( \mathcal{A}_S \) which is assembled following the schemata of the given \( S \) (Theorem 9.1).

The important virtue of our expressiveness result is that the most useful kinds of sequential, non-deterministic and parallel execution controls are easily represented within the paradigm of Linear Logic Automata.

This argument can be demonstrated by the natural direct simulation of

(a) the non-deterministic selection (Fig. 9 directly corresponds to Fig. 7),
(b) parallel repetition (Fig. 10 is isomorphic to Fig. 8),
(c) sequential composition (shown in Fig. 6), etc.

It is remarkable that more or less difficult cases of our simulation that we meet with are only the following two cases:

(i) the case of the "primitive" assignment: \( x_m := 0 \),
(ii) and the case of the "primitive" predicate: \( (x_i \leq x_j) \).

Coming back to Linear Logic proper, we can give a complete characterization of non-deterministic programs in terms of Linear Logic derivability.

According to Corollary 9.1, for each of guarded statements \( S \), we can construct a multiset \( \mathcal{F}_S \), consisting of Horn-like formulas, such that, whatever state of the memory \( \kappa \) we take, \( S[\kappa] \) coincides exactly with the minimal set of states of the memory \( \{\zeta_1, \zeta_2, \ldots, \zeta_k\} \) such that a sequent of the form

\[
q', \bar{\kappa}, !\mathcal{F}_S \vdash (q_0 \oplus (q_1 \otimes (\bar{\zeta}_1 \otimes \bar{\zeta}_2 \otimes \cdots \otimes \bar{\zeta}_k)))
\]

is derivable in Linear Logic. In addition to that, any cut-free derivation of the latter "minimal" sequent, read from its axiomatic vertices to its root, forms a terminated computation of \( \mathcal{F} \) that starts from the given state of the memory \( \kappa \).

In closing, we make two remarks concerning semantics of programming languages. The guarded commands approach to non-deterministic programming was introduced by Dijkstra [6, 7].

The Dijkstra’s semantics of non-deterministic programs was defined by means of the predicate transformer \( \text{wp} \):

We use the notation \( \text{wp}(S, R) \), where \( S \) denotes a statement and \( R \) some condition on the state of the system, to denote the weakest precondition for the initial state \( \kappa \) of the system such that activation of \( S \) is guaranteed to lead to a properly terminating
activity leaving the system in a final state $\zeta$ satisfying the postcondition $R$ (even

The formal semantics of $\mathcal{M}$-$\mathcal{F}$ introduced in this paper, and thereby our computa-
tional semantics of Horn Linear Logic derivability, can be conceived of as a natural
semantics behind the Dijkstra’s semantics.

As for the Dijkstra’s predicate transformer $wp$ proper, now it can be introduced by the following:

$$wp(S, R)(\kappa) \equiv \left(\text{\textit{S}[\kappa] is defined} \land \forall \zeta ((\zeta \in S[\kappa]) \rightarrow R(\zeta))\right)$$

The justification for our proposal is that all properties of $wp$ formally declared
in [6, 7] can be proved within the framework of our approach.

The LL Automata semantics can be also conceived of as a natural semantics behind the
Hoare’s approach to program semantics introduced in [15]. In particular, the Hoare’s statement:

If $P$ (the precondition) holds before executing $S$, then $R$ (the postcondition) holds
when $S$ terminates.

can be formally introduced in one of the following two ways:

(A) $\forall \kappa (P(\kappa) \rightarrow ((\text{\textit{S}[\kappa] is defined}) \rightarrow \forall \zeta ((\zeta \in S[\kappa]) \rightarrow R(\zeta))))$ (this item represents the partial correctness case).

(B) $\forall \kappa (P(\kappa) \rightarrow ((\text{\textit{S}[\kappa] is defined}) \land \forall \zeta ((\zeta \in S[\kappa]) \rightarrow R(\zeta))))$ (that represents the total correctness case).

Such a proposal can be also justified by validation of Hoare’s proof system [15], even
if we extend Hoare’s rules to non-deterministic programs.

2. Horn-like fragments of linear logic

Let us recall here basic definitions from Linear Logic.

We will use only Horn-like formulas that are built up of positive elementary propositions, or literals, by the following connectives: $\otimes, \neg, \oplus, !$.

**Definition 2.1.** The tensor product of a positive number of positive literals is called a simple product. A single literal $p$ is also called a simple product.

Simple products will be denoted by $X, Y, Y_1, Y_2, \ldots, Y_m, W, Z, \text{etc.}$

**Definition 2.2.** Horn-like formulas are introduced by the following:

(a) A Horn implication is a formula of the form $(X \rightarrow Y)$.

(b) A $\oplus$-Horn implication is a formula of the form $(X \rightarrow (Y_1 \oplus Y_2 \oplus \cdots \oplus Y_m))$. 
Table 1

The inference rules of intuitionistic linear logic

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>I</strong></td>
<td>$A \vdash A$</td>
</tr>
<tr>
<td><strong>L∞</strong></td>
<td>$\Sigma_1 \vdash A, \Sigma_2 \vdash C$</td>
</tr>
<tr>
<td>$\Sigma_1, (A \rightarrow B), \Sigma_2 \vdash C$</td>
<td></td>
</tr>
<tr>
<td><strong>B</strong></td>
<td>$\Sigma_1 \vdash B, \Sigma_2 \vdash C$</td>
</tr>
<tr>
<td><strong>R∞</strong></td>
<td>$\Sigma_1 \vdash A, \Sigma_2 \vdash B$</td>
</tr>
<tr>
<td>$\Sigma \vdash (A \rightarrow B)$</td>
<td></td>
</tr>
</tbody>
</table>

Definition 2.3. Let a multiset $\Gamma$ consist of Horn implications and $\oplus$-Horn implications. Then a sequent of the form

$$W, !\Gamma \vdash (Z_1 \oplus Z_2 \oplus \cdots \oplus Z_k)$$

will be called a $(!, \oplus)$-Horn sequent.

The intuitionistic nature of the Horn fragments of Linear Logic is revealed by the following:

Lemma 2.1. Any $(!, \oplus)$-Horn sequent is derivable in Linear Logic if and only if it is also derivable in Intuitionistic Linear Logic (its rules are listed in Table 1).

Proof. The straightforward Boolean evaluation shows that, in the absence of negations and negative literals and constants, any sequent of the form $\Sigma \vdash$ is not derivable in Linear Logic.

Therefore, each of cut-free derivations of a given $(!, \oplus)$-Horn sequent is, at the same time, a derivation in intuitionistic linear logic. □

3. Computations performed by LL automata

A Linear Logic Automaton is a hybrid of a finite automaton and a non-deterministic Petri net. (The concept of non-deterministic Petri nets was introduced in [20].)

A formal definition of a Linear Logic Automaton is as follows:

Definition 3.1. A Linear Logic Automaton $\mathcal{A}$ is a tuple

$$\mathcal{A} = (\mathcal{S}, q_{\mathcal{A}}, \mathcal{L}, \tau)$$

\[7\text{Where } !\Gamma \text{ stands for the multiset resulting from putting the modal storage operator }! \text{ before each formula in } \Gamma.\]
where $\mathcal{Z}$ is a finite set of states of the automaton, $q_0$ is the initial state, $\mathcal{F}$ is a finite set of final states, and $\mathcal{T}$ is a finite set of commands.  

Such a hybrid of finite automata and non-deterministic Petri nets should allow us to distinguish both information and resources aspects in the dynamic behaviour of systems under consideration.

To do that, we define configurations of the systems as pairs $(q, \kappa)$ where

(i) the first component $q$ is conceived of as an information-state, or a state of the automaton,

(ii) while the second component $\kappa$ is conceived of as a distribution of resources, or a state of the memory.

The memory is assumed to consist of registers $\text{reg}_1, \text{reg}_2, \ldots, \text{reg}_n, \ldots$ that can contain non-negative integers.

From the Petri nets point of view, each of registers $\text{reg}_m$ is supposed to be a place that can accommodate an unlimited number of tokens of the $m$th sort. The number $k_m$ of tokens accommodated in the place $\text{reg}_m$ indicates that exactly $k_m$ resources are available at this place.

**Definition 3.2.** A configuration is a pair of the form $(q, \kappa)$ where $q$ is the current state of the automaton, and $\kappa$ is the current state of the memory, i.e. $\kappa$ is the linearly ordered sequence of non-negative integers contained in the corresponding registers:

$$\kappa = (k_1, k_2, \ldots, k_n, \ldots).$$

Such a $\kappa$ is also conceived of as the current distribution of resources (the current marking in terms of Petri nets).

We will use the following two sorts of literals:

(a) $q_0, q_1, q_2, \ldots, q_i, \ldots$ (to represent states of the automaton,) and

(b) $p_1, p_2, \ldots, p_n, \ldots$ (to represent states of the memory).

**Definition 3.3.** Each register $\text{reg}_m$ is associated with literal $p_m$.

Any state of memory $\kappa = (k_1, k_2, \ldots, k_n, \ldots)$, will be represented by the following simple tensor product

$$\mathcal{\kappa} = (p_1^{k_1} \otimes p_2^{k_2} \otimes \cdots \otimes p_n^{k_n} \otimes \cdots).$$

In particular, the trivial state of the memory $\emptyset = (0, 0, \ldots, 0, \ldots)$ is represented by $\overline{\emptyset} \equiv 1$.

---

8 We will call LL Automata commands instructions, or transitions, as well.

9 Here, and henceforth, $G^0 = 1$, $G^k = (G \otimes G \otimes \cdots \otimes G)$.

10 We will assume a finite number of non-zero registers at the current moment.
In its turn, any configuration of the form \((q_i, \kappa)\) will be represented by a simple tensor product of the form \((q_i \otimes \kappa)\). Henceforth, we consider both forms of representation for configurations as identical: \((q_i, \kappa) = (q_i \otimes \kappa)\).

**Definition 3.4.** We will use the following two kinds of LL automaton instructions:

1. **Purely Horn instructions** \(\tau\) of the form
   \[
   \tau = ((q_i \otimes (p_1^{a_1} \otimes p_2^{a_2} \otimes \cdots \otimes p_n^{a_n})) \to (q_j \otimes (p_1^{b_1} \otimes p_2^{b_2} \otimes \cdots \otimes p_n^{b_n})))
   \]
   where the corresponding non-negative integer vectors \(\vec{c}\) and \(\vec{d}\):
   \[
   \vec{c} = (c_1, c_2, \ldots, c_n), \quad \vec{d} = (d_1, d_2, \ldots, d_n),
   \]
   represent preconditions and postconditions of \(\tau\), respectively.

   Applying such an ordinary instruction \(\tau\) to a given configuration of the form
   \[
   (q_i, \vec{a}) = (q_i \otimes (p_1^{a_1} \otimes p_2^{a_2} \otimes \cdots \otimes p_n^{a_n})),
   \]
   LL automaton \(A\)
   
   (1a) changes its state \(q_i\) for \(q_j\),
   
   (1b) removes \(c_m\) tokens from each place \(reg_m\),
   
   (1c) and then adds \(d_m\) tokens to each place \(reg_m\).

2. **Non-deterministic, or branching, instructions** \(\tau\) of the form
   \[
   \tau = (q_i \to (q_{j_1} \otimes q_{j_2} \otimes \cdots \otimes q_{j_m})).
   \]

   Applying such a branching instruction \(\tau\) to a given configuration of the form
   \[
   (q_i, \vec{a}) = (q_i \otimes (p_1^{a_1} \otimes p_2^{a_2} \otimes \cdots \otimes p_n^{a_n})),
   \]
   LL automaton \(A\)
   
   (2a) changes its state \(q_i\) either for \(q_{j_1}\), or for \(q_{j_2}\), or \ldots, or for \(q_{j_m}\).

The difference between ordinary and non-deterministic instructions can be perceived as follows: When we fire a certain instruction, we make thereby our own choice from a given set \(\mathcal{F}\).

1. If we choose an ordinary instruction to be fired, we know the result of this firing to be deterministic.

2. On the contrary, having chosen a non-deterministic instruction to be fired, we meet with the non-deterministic situation that is out of our control. In particular, we do not know in advance which action from the set of the alternative ones will be chosen at a given occasion.

The following oracle analogy can be invoked. Let us imagine that there exists the LL Automata Oracle who is being consulted by Linear Logic automata. And when we have chosen a non-deterministic instruction to be fired, actually we have put a question to the LL Automata Oracle.

"Which of \(m\) alternatives is allowed to be performed in a given case?"

Only after having got his answer, we will know what we may do.
Definition 3.5. A computation $C$ performed by a given Linear Logic Automaton $A = (Q, q_0, Q_0, T)$, is a finite rooted tree built up of configurations such that

(a) The root of $C$ is of the form $(q_0, \kappa) = (q_0 \otimes (p_{1}^{a_1} \otimes p_{2}^{a_2} \otimes \cdots \otimes p_{n}^{a_n}))$.

(b) Each edge of $C$ is labelled by an instruction from $T$:

- If a vertex $(q_i, a) = (q_i \otimes (\cdots \otimes p_{j}^{a_j} \otimes \cdots))$, has exactly one son $(q_j, b)$:

$$ (q_j, b) = (q_j \otimes (\cdots \otimes p_{j}^{b_j} \otimes \cdots)) $$

then its outgoing edge from $(q_i, a)$ to $(q_j, b)$ should be labelled by an ordinary instruction $\tau$ of the form

$$ ((q_i \otimes (p_{1}^{c_1} \otimes p_{2}^{c_2} \otimes \cdots \otimes p_{n}^{c_n})) \rightarrow (q_j \otimes (p_{1}^{d_1} \otimes p_{2}^{d_2} \otimes \cdots \otimes p_{n}^{d_n}))) $$

such that for all $k$:

$$ \begin{align*}
  c_k &\leq a_k \quad \text{(the applicability conditions)} \\
  b_k &= (a_k - c_k) + d_k.
\end{align*} $$

- Let $(q_i, a)$ be a branching vertex with exactly $m$ sons. Then all these sons should be of the form $(q_j, a_i), (q_j, a_2), \ldots, (q_j, a_m)$ (with just the same $a$) and each of $m$ edges outgoing from $(q_i, a)$ should be labelled by one and the same branching instruction $\tau : \tau = (q_i \rightarrow (q_{j_1} \oplus q_{j_2} \oplus \cdots \oplus q_{j_m}))$.

(c) Finally, any vertex $(q, \zeta)$ of $C$ is terminal if and only if $q$ is one of the final states of $A$.

Any computation $C$ is conceived of as a multivalued mapping from initial configurations $(q_0, \kappa)$ into final configurations $(q, \zeta)$. 

Definition 3.6. For a given computation $C$, with $(q_0, \kappa) \rightarrow_C (q, \zeta)$ we denote that $C$ leads from $(q_0, \kappa)$ into $(q, \zeta)$; that means

(a) the root of $C$ is of the form $(q_0, \kappa)$,

(b) and there exists a terminal vertex of the form $(q, \zeta)$ in $C$.

Our computational interpretation is proved to be sound and complete with respect to Linear Logic:

Theorem 3.1 (Completeness). For a given Linear Logic automaton $A$:

$$ A = (Q, q_0, \{q_0, q_1, q_2, \ldots, q_k\}, T), $$

whatever states of the memory $\kappa, \zeta_0, \zeta_1, \zeta_2, \ldots, \zeta_k$ we take, a Horn-like sequent of the form

$$ q_0, \kappa, \vdash (q_0 \otimes \zeta_0) \oplus (q_1 \otimes \zeta_1) \oplus (q_2 \otimes \zeta_2) \oplus \cdots \oplus (q_k \otimes \zeta_k) $$

\[11\] Computations might be defined as acyclic directed graphs of a certain kind. For simplicity, we use an unfolded version of computations.

\[12\] Here final states $q_0, q_1, q_2, \ldots, q_k$ are not necessarily distinct.
is derivable in Linear Logic if and only if there exists a computation $\mathcal{C}$ performed by $\mathcal{A}$ such that
(a) The root of $\mathcal{C}$ is of the form $(q_\mathcal{A}, \kappa)$.
(b) For any $q$ and $\xi$, if $(q_\mathcal{A}, \kappa) \rightarrow (q, \xi)$, then for some $i : (q, \xi) = (q_i, \zeta_i)$.
Moreover, there is an exact correspondence between cut-free derivations of the above Horn-like sequent and terminated computations of $\mathcal{A}$ leading from $\kappa$ into $\{\zeta_0, \zeta_1, \zeta_2, \ldots, \zeta_k\}$.

Proof. We can generalize our Completeness Theorem from [17, 18] where a slightly different version of resource-sensitive Horn computations is considered. $\square$

4. Functions computable by L.L. automata

As a rule, Automata are being constructed to perform one of following two tasks:
(i) They compute functions from the states of the memory to the states of the memory, or
(ii) They recognize predicates over the states of the memory.
It should be pointed out that our Completeness Theorem 3.1 does not guarantee that the set of all $\zeta$ that occur at terminal vertices of a given computation $\mathcal{C}$ performed by a Linear Logic Automaton $\mathcal{A}$ coincides exactly with the corresponding set $\{\zeta_0, \zeta_1, \zeta_2, \ldots, \zeta_k\}$. Moreover, we cannot give such a guarantee because in Linear Logic derivability of a sequent of the form $\Sigma \vdash A$ implies derivability of any sequent of the form $\Sigma \vdash (A \oplus B)$.

Example 4.1. The above uniqueness problem can be sharpened with the help of the LL Automaton $\mathcal{A} = (\{q_\mathcal{A}, q_1, q_2\}, q_\mathcal{A}, \{q_1, q_2\}, \mathcal{T})$, where $\mathcal{T}$ consists of two Horn instructions $\tau_1$ and $\tau_2$:

$$\tau_1 = q_\mathcal{A} \neg \circ (q_1 \otimes p_1),$$
$$\tau_2 = q_\mathcal{A} \neg \circ (q_2 \otimes p_2).$$

It is readily seen that
(a) The following sequent is derivable in Linear Logic:
$$q_\mathcal{A}, 1, !\mathcal{T} \vdash ((q_1 \otimes p_1) \oplus (q_2 \otimes p_2)).$$
(b) While there exist exactly two distinct computations $\mathcal{C}_1$ and $\mathcal{C}_2$ with the root\(^{13}\)
$$\mathcal{C}_1$$
one of them, say $\mathcal{C}_1$, leads from $(q_\mathcal{A} \otimes 1)$ to $(q_1 \otimes p_1)$ and does not lead to $(q_2 \otimes p_2)$,
(b2) whereas the other $\mathcal{C}_2$ leads from $(q_\mathcal{A} \otimes 1)$ to $(q_2 \otimes p_2)$ and does not lead to $(q_1 \otimes p_1)$.

\(^{13}\) Recall that $\theta = (0, 0, \ldots, 0, \ldots)$ and $\tilde{\theta} \equiv 1$. 
The effect is that there is no way of determining the proper output $\mathcal{A}[\theta]$ in our Example 4.1.

Nevertheless, we can get the desired uniqueness and thereby the correct definition of LL computable functions by invoking the idea of the complete garbage collection along specified branches of LL automata computations.

Henceforth, we specify a final state $q_0$ for these purposes.

**Definition 4.1.** For a given LL Automaton $\mathcal{A} = (2, q_0, L_0, \mathcal{T})$ where $L_0$ contains $q_0$, a computation $C$ performed by $\mathcal{A}$ is said to be a *properly terminated* computation if for every its terminal vertex of the form $(q_0, \zeta)$ the state of memory $\zeta$ is *trivial*: $\zeta = \varnothing = (0, 0, \ldots, 0, \ldots)$.

We introduce the strongest version of the concept of

LL Automaton $\mathcal{A}$ computes a multi-valued function $f$

providing that all computations performed by $\mathcal{A}$ on a given input $\kappa$ are to be of one and the same structure.

**Definition 4.2.** Let $\mathcal{A}$ be a Linear Logic Automaton of the form\(^{14}\) $\mathcal{A} = (2, q_0, \{q_0, q_1\}, \mathcal{T})$. Computations $C_1$ and $C_2$ performed by $\mathcal{A}$ are called *isomorphic* if there is a one-to-one correspondence $\mathcal{I}$ between the trees $C_1$ and $C_2$ such that

(i) $\mathcal{I}$(the root of $C_1$) = the root of $C_2$. Moreover, the initial configurations of $C_1$ and $C_2$ are identical: the root of $C_1$ = the root of $C_2$.

(ii) The number of edges between any two vertices of $C_1$ is equal to the number of edges between the corresponding vertices in $C_2$.

(iii) Any vertex $(q, A)$ of $C_1$, one of whose terminal descendants is of the form $(q_1, \zeta)$, coincides exactly with the corresponding vertex $\mathcal{I}((q, A))$ in $C_2$.

(iv) Any terminal vertex $(q, \zeta)$ of $C_1$ coincides exactly with the corresponding terminal vertex $\mathcal{I}((q, \zeta))$ in $C_2$.

Introducing the concept of functions computable by LL Automata, we should take into account the following peculiarities of Linear Logic Automata:

(a) They are able to compute multivalued functions.

(b) Any LL Automaton $\mathcal{A}$ can "tackle" only a *finite* number of registers. The effect is that any multivalued function $f$ computed by an LL Automaton $\mathcal{A}$ can be conceived of as a multivalued mapping from certain $n$-dimensional integer vector space $V$ into $V$.

**Definition 4.3.** Let $n$ be the number of the first registers that can be tackled by a partial multivalued function $f$ mapping the states of the memory into the states of the memory.

\(^{14}\)To simplify definitions, all final states differing from $q_0$ are glued together into $q_1$. 
We will say that a Linear Logic Automaton \( \mathcal{A} = (\mathcal{A}, q_{\mathcal{A}}, \{q_0, q_1\}, \mathcal{T}) \), computes the partial multivalued function \( f \) if, whatever state of the memory \( \kappa \) of the form \( \kappa = (k_1, k_2, \ldots, k_n, 0, 0, \ldots) \) we take, (a) All properly terminated computations \( \mathcal{C} \) performed by \( \mathcal{A} \) such that their roots are of the form \( (q_{\mathcal{A}}, \kappa) \), are isomorphic in the sense of Definition 4.2. (b) \( \mathcal{A}[\kappa] \) is declared to be defined if and only if one can construct a properly terminated computation \( \mathcal{C} \) performed by \( \mathcal{A} \) such that the root of \( \mathcal{C} \) is of the form \( (q_{\mathcal{A}}, \kappa) \). (c) If \( \mathcal{A}[\kappa] \) is defined then there is a non-empty set of states of the memory \( \{\zeta_1, \zeta_2, \ldots, \zeta_k\} \) such that for every properly terminated computation \( \mathcal{C} \) with the root \( (q_{\mathcal{A}}, \kappa) \), the following holds: The set \( \mathcal{C}[\kappa] \), consisting of all \( \zeta \) such that \( (q_{\mathcal{A}}, \kappa) \rightarrow (q_1, \zeta) \), coincides exactly with the set \( \{\zeta_1, \zeta_2, \ldots, \zeta_k\} \): \[ \mathcal{C}[\kappa] = \{\zeta \mid (q_{\mathcal{A}}, \kappa) \rightarrow (q_1, \zeta)\} = \{\zeta_1, \zeta_2, \ldots, \zeta_k\}. \] Furthermore, having had such a unique \( \{\zeta_1, \zeta_2, \ldots, \zeta_k\} \), we set: \( \mathcal{A}[\kappa] = \mathcal{C}[\kappa] = \{\zeta_1, \zeta_2, \ldots, \zeta_k\} \). (d) Finally, \( \mathcal{A}[\kappa] = f(\kappa) \).

Remark. It is worthwhile observing that, according to our Definition 4.3, there is no meaningless properly terminated computation \( \mathcal{C} \) performed by \( \mathcal{A} \) such that all its terminal vertices are of the form \((q_0, \theta)\) only.

Example 4.2. We illustrate the above definitions with the following LL Automaton \( \mathcal{A}_0 \) we will be using in our main theorems: \[ \mathcal{A}_0 = (\{q_{\mathcal{A}_0}, q_0, q_1, g\}, q_{\mathcal{A}_0}, \{q_0, q_1\}, \mathcal{T}_0) \] where \( \mathcal{T}_0 \) consists of the following instructions: \[ \mathcal{T}_0 = \begin{cases} ((q_{\mathcal{A}_0} \otimes p_m) \rightarrow q_{\mathcal{A}_0}), \\ (q_{\mathcal{A}_0} \rightarrow (q_1 \otimes g)), \\ ((g \otimes p_k) \rightarrow g), \quad (k = 1, 2, \ldots, m - 1, m + 1, \ldots, n), \\ (g \rightarrow q_0). \end{cases} \] Here a new literal \( g \) will serve as a garbage collector to wipe registers clean by killing all literals except for \( p_m \).

Lemma 4.1. Let \( \kappa \) be a state of the memory of the form \[ \kappa = (k_1, k_2, \ldots, k_m, \ldots, k_n, 0, 0, \ldots). \] Then \( \mathcal{A}[\kappa] = \{\zeta\} \) where the state of the memory \( \zeta: \zeta = (z_1, z_2, \ldots, z_m, \ldots, z_n, 0, 0, \ldots) \), satisfies the following conditions: \[ z_m = 0, \quad z_i = k_i \quad (i \neq m). \]
Proof. Starting with the initial configuration \((q_{\mathcal{A}_0}, \kappa) = (q_{\mathcal{A}_0} \otimes p_m^k \otimes X)\) where

\[
X = (p_1^k \otimes \cdots \otimes p_{m-1}^k \otimes p_{m+1}^k \otimes \cdots \otimes p_n^k),
\]

we develop the desired computation in the following way (see Fig. 1 assuming that \(k\) is equal to \(k_m\)):

(i) First, one and the same Horn instruction \(\tau_0 = ((q_{\mathcal{A}_0} \otimes p_m) \rightarrow q_{\mathcal{A}_0})\), is applied successively \(k_m\) times, so that the number of the occurrences of literal \(p_m\) is being exhausted from \(k_m\) to 0. In other words, we develop a branch, it being called main, by creating a chain of \(k_m\) edges and labelling each of these edges by one and the same Horn instruction \(\tau_0\). The effect is that we get into the configuration \((q_{\mathcal{A}_0} \otimes X)\).

(ii) Then, we apply our branching instruction. Namely, we create two outgoing edges and label these new edges by \((q_{\mathcal{A}_0} \rightarrow (q_1 \otimes g))\). As a result, we produce two son configurations: \((q_1 \otimes X)\) (that will be the desired final configuration on the main branch) and \((g \otimes X)\) (that will be the root of a g-branch).

(iii) Now we will process the latter son developing its g-branch. With the help of killing commands \(((g \otimes p_k) \rightarrow g)\), we contract \(X\) to the trivial 1, getting into the following configuration on the g-branch: \((g \otimes 1)\).

(iv) Finally, applying the instruction \((g \rightarrow q_0)\), we terminate properly the g-branch of our computation at the final configuration \((q_0 \otimes 1)\).

In the opposite direction, we will prove that an arbitrary properly terminated computation \(\mathcal{C}\) performed by \(\mathcal{A}_0\) cannot be but of the form described above.

For a given computation \(\mathcal{C}\), we develop its main branch as follows (Fig. 1).
Let $k$ be the length of the longest non-branching segment that starts from the initial configuration $(q_{\mathcal{A}_0}, \kappa) = (q_{\mathcal{A}_0} \otimes p_m^k \otimes X)$. It means that at the $k$th step we get into the end of the segment, which is a non-terminal configuration of the form $(q_{\mathcal{A}_0} \otimes p_m^{k-k} \otimes X)$. Furthermore, this configuration should be the father of two sons: $(q_1 \otimes p_m^{k-k} \otimes X)$ (the final configuration on the main branch) and $(q \otimes p_m^{k-k} \otimes X)$, respectively.

Let us examine the descendants of the latter son. Taking into account the applicability conditions, we can conclude that all its non-terminal descendants must be non-branching vertices of the form $(q \otimes p_m^{k-k} \otimes X')$, and, therefore, its terminal descendant must be of the form $(q_0 \otimes p_m^{k-k} \otimes X')$. Recalling that $V$ is a properly terminated computation yields that $X' \in 1$, and $k_0 = 0$. Hence, the final configuration on the main branch that has just been identified is to be of the desired form: $(q_1 \otimes X)$. 

**Remark.** From the programming point of view, the effect of Lemma 4.1 is that our LL Automaton $\mathcal{A}_0$ simulates directly an assignment statement of the form $x_m := 0$ where the variable $x_m$ represents the current value of the $m$th register.

**Example 4.3.** Actually Lemma 4.1 resolved the most difficult case of assignment statements. Indeed, an assignment statement of the form $x_m := x_m - c$ (where $c$ is a positive integer) can be simulated in one step by the LL Automaton $\mathcal{A}_0 = (\{q_{\mathcal{A}_0}, q_1\}, q_{\mathcal{A}_0}, \{q_1\}, \mathcal{T} \mathcal{C})$ with $\mathcal{T} \mathcal{C}$ being the singleton that consists of one Horn instruction $(q_{\mathcal{A}_0} \otimes p_m^c \otimes q_1)$. 

**Lemma 4.2.** Let $\kappa$ be a state of the memory: $\kappa = (k_1, k_2, \ldots, k_m, \ldots, k_n, \ldots)$. Then "$\mathcal{A}_c[\kappa]$ is defined" if and only if $k_\kappa \geq c$, and, if $k_\kappa \geq c$, $\mathcal{A}_c[\kappa] = \{\zeta\}$ where the state of the memory $\zeta = (z_1, z_2, \ldots, z_m, \ldots, z_n, \ldots)$, satisfies the following conditions:

$$z_m = k_m - c, \quad z_i = k_i \quad (i \neq m).$$

**Proof.** It follows immediately from Definition 3.5 (see Fig. 2). 

**Example 4.4.** In its turn, an assignment statement of the form $x_m := x_m + c$ can be simulated in one step by the LL Automaton $\mathcal{A}_c = (\{q_{\mathcal{A}_c}, q_1\}, q_{\mathcal{A}_c}, \{q_1\}, \mathcal{T} \mathcal{C})$, with $\mathcal{T} \mathcal{C}$ being the singleton that consists of one Horn instruction $(q_{\mathcal{A}_c} \otimes p_m^c \otimes q_1).$

5. **Sequential composition of LL automata**

The class of LL computable functions is proved to be closed under basic sequential operations, like sequential composition, if-then-else selection, etc.
In particular, sequential composition is expressed within the framework of LL Automata in the following way:

**Theorem 5.1.** For any LL Automata $A_1$ and $A_2$ computing multivalued functions that can tackle only first $n$ registers, we construct an LL Automaton $A$ computing a multivalued function such that, whatever state of the memory $\kappa$ of the form $\kappa = (k_1, k_2, \ldots, k_n, 0, 0, \ldots)$ we take,

(i) $A[\kappa]$ is defined if and only if $A_1[\kappa]$ is defined and for every state of the memory $\rho$ taken from $A_1[\kappa]$, $A_2[\rho]$ is also defined.

(ii) if $A[\kappa]$ is defined then $A[\kappa] = \bigcup_{\rho \in A_1[\kappa]} A_2[\rho]$.

**Proof.** Suppose we have two LL Automata $A_1$ and $A_2$:

$$A_1 = (Z_1, q_{A_1}, \{q_0, q_1\}, \mathcal{F}_1), \quad A_2 = (Z_2, q_{A_2}, \{q_0, q_2\}, \mathcal{F}_2),$$

such that (with the help of renaming states) $Z_1 \cap Z_2 = \{q_0\}$. The desired composition $A$ is introduced as follows:

$$A = (Z, q_A, \{q_0, q_2\}, \mathcal{F})$$

where

(a) $Z = Z_1 \cup Z_2$,

(b) $q_A = q_{A_1}$,

(c) $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2 \cup \mathcal{F}_1$, with $\mathcal{F}_1$ being the singleton that consists of one interface instruction $(q_1 \rightarrow q_{A_1})$.

**Lemma 5.1.i.** If $A_1[\kappa]$ is defined and, for every $\rho$ from $A_1[\kappa]$, $A_2[\rho]$ is also defined, then there is a properly terminated computation $C_1$ performed by $A$ such that

$$C[\kappa] = \bigcup_{\rho \in A_1[\kappa]} A_2[\rho].$$

**Proof.** Such a computation $C$ is shown in Fig. 3: First, by means of instructions from $\mathcal{F}_1$ a computation $C_1$ yields all $\rho$ from $A_1[\kappa] = \{\rho_1, \rho_2, \ldots, \rho_k\}$. Then we apply the interface instruction $(q_1 \rightarrow q_{A_2})$ to each of $A_1$-final vertices of the form $(q_1, \rho_i)$, getting thereby into the corresponding configurations $(q_{A_2}, \rho_i)$ (that are initial configurations for $A_2$).

Finally, with the help of $A_2$-computations $C_{2,i}$ developed from the corresponding inputs $(q_{A_2}, \rho_i)$, we produce the desired

$$\{\tilde{z}_{1,1}, \ldots, \tilde{z}_{1,n_1}, \ldots, \tilde{z}_{k,1}, \ldots, \tilde{z}_{k,n_k}\} = \bigcup_{\rho \in A_1[\kappa]} A_2[\rho]. \quad \Box$$

**Lemma 5.1.ii.** Let $C$ be a properly terminated computation performed by $A$ such that the root of $C$ is of the form $(q_A, \kappa)$. Then $C$ is to be of the form shown in
Fig. 3, and furthermore $A_1[\kappa]$ must be defined, and, for every $\rho$ from $A_1[\kappa]$, $A_2[\rho]$ must be defined as well.

**Proof.** Whereas $\mathcal{C}_1 \cap \mathcal{C}_2 = \{q_0\}$, on any path leading from the root of $\mathcal{C}$ to a terminal vertex of the form $(q_2, \zeta)$ there is exactly one edge of the form $((q_1, \rho), (q_2, \zeta))$. Let $\mathcal{C}_1$ be the result of truncation of the tree $\mathcal{C}$ by cutting off all edges of the form $((q, \overline{a}), (q', \overline{b}))$ where either $q \in \mathcal{C}_2$, or $q = q_1$.

Actually this $\mathcal{C}_1$ is a properly terminated computation performed by $A_1$. According to Definition 4.3, the effect is that $A_1[\kappa]$ is to be defined, $\mathcal{C}_1$ is to be unique in the sense of Definition 4.2, and for some positive $k$:

$$A_1[\kappa] = \mathcal{C}_1[\kappa] = \{\rho_1, \rho_2, \ldots, \rho_k\}.$$  

Similarly, the subtrees $\mathcal{C}_2, \ldots, \mathcal{C}_2, k$ with the corresponding roots $(q_{d_2}, \rho_1), \ldots, (q_{d_2}, \rho_k)$ are proved to be unique properly terminated computations performed by $A_2$, provided that all $A_2[\rho_1], \ldots, A_2[\rho_k]$ are defined as well. \[\square\]

6. Predicates recognizable by LL automata

For the purposes of predicate recognition we introduce the concept of Boolean LL automata.

Similar to the case of multivalued functions, we consider those partial predicates over the states of the memory that can “tackle” only a finite number of registers.

The truth values **true** and **false** will be “designated” by specific final states of the automaton $q_{\top}$ and $q_{\bot}$, respectively.
Definition 6.1. Let \( B(x_1, x_2, \ldots, x_n) \) be an \( n \)-ary partial predicate over non-negative integers. We will say that a Linear Logic Automaton \( \mathcal{A} = (\mathcal{I}, q_{df}, \{q_o, q_T, q_\perp\}, \mathcal{F}) \), recognizes the partial predicate \( B \) if, whatever state of the memory \( \kappa \) of the form \( \kappa = (k_1, k_2, \ldots, k_n, 0, 0, \ldots) \) we take,

(a) All properly terminated computations \( \mathcal{C} \) performed by \( \mathcal{A} \) such that their roots are of the form \( (q_{df}, \kappa) \), are isomorphic in the sense of Definition 4.2.

(b) \( \mathcal{A}[\kappa] \) is declared to be defined if there exists a properly terminated computation \( \mathcal{C} \) performed by \( \mathcal{A} \) such that the root of \( \mathcal{C} \) is of the form \( (q_{df}, \kappa) \).

(c) If \( \mathcal{A}[\kappa] \) is defined then one can pick a \( q_1 \) from the pair \( \{q_T, q_\perp\} \) so that for every properly terminated computation \( \mathcal{C} \) with the root \( (q_{df}, \kappa) \) the following holds:

(c1) In \( \mathcal{C} \) there is one and exactly one terminal vertex of the form \( (q_1, \kappa) \).

(c2) All terminal vertices that differ from \( (q_1, \kappa) \) are of the trivial form \( (q_0, \kappa) \).

Furthermore, having had such a unique \( q_1 \), we set:

\[
\mathcal{A}[\kappa] = \mathcal{C}[\kappa] = \begin{cases} 
\text{true}, & \text{if } q_1 = q_T, \\
\text{false}, & \text{if } q_1 = q_\perp.
\end{cases}
\]

(d) Finally, \( \mathcal{A}[\kappa] = B(k_1, k_2, \ldots, k_n) \).

Example 6.1. We illustrate the concept of Boolean LL automata with one of the key "primitive" predicates \( B_<(x_i, x_j) = "x_i < x_j" \). \( B_<(x_i, x_j) \) is proved to be simulated by the Boolean LL Automaton

\[
\mathcal{A}_< = (\{q_{df}, q_0, q_1, q_d\}, q_{df}, \{q_0, q_T, q_\perp\}, \mathcal{F}_<),
\]

where \( \mathcal{F}_< \) is the union of sets \( \mathcal{F}_1, \mathcal{F}_2, \) and \( \mathcal{F}_g \) consisting of the following instructions:

\[
\mathcal{F}_1 = \begin{align*}
& (q_{df} \rightarrow q_1), \\
& ((q_1 \otimes p_i \otimes p_j) \rightarrow (q_1 \otimes t)), \\
& (q_1 \rightarrow (q_3 \oplus g_1)), \\
& ((g_1 \otimes p_k) \rightarrow g_1) \quad (k = 1, 2, \ldots, i-1, i+1, \ldots, n), \\
& ((g_1 \otimes t) \rightarrow g_1), \\
& (q_1 \rightarrow q_0), \\
& ((q_3 \otimes t) \rightarrow (q_3 \otimes p_i \otimes p_j)), \\
& (q_3 \rightarrow (q_T \oplus g)).
\end{align*}
\]

Here an additional literal \( t \) will be used to store the value of \( x_i \) and \( x_j \), a new literal \( g_1 \) will serve as a garbage collector to wipe registers clean by killing all literals except
for $p_j$.

$$\mathcal{F}_2 = \begin{cases} 
(q_{\mathcal{A}} \rightarrow q_2), \\
((q_2 \otimes p_i \otimes p_j) \rightarrow (q_2 \otimes t)), \\
(q_2 \rightarrow (q_4 \oplus g_2)), \\
((g_2 \otimes p_k) \rightarrow g_2) \quad (k = 1, 2, \ldots, j - 1, j + 1, \ldots, n), \\
((g_2 \otimes t) \rightarrow g_2), \\
((g_2 \otimes p_i) \rightarrow q_0), \\
((q_4 \otimes t) \rightarrow (q_4 \otimes p_i \otimes p_j)), \\
(q_4 \rightarrow (q_1 \oplus g)).
\end{cases}$$

Here a new literal $g_2$ will be exploited to clean registers up by killing all literals except for $p_j$.

$$\mathcal{F}_y = \begin{cases} 
((g \otimes p_k) \rightarrow g) \quad (k = 1, 2, \ldots, i, \ldots, j, \ldots, n), \\
(g \rightarrow q_0).
\end{cases}$$

Here a literal $g$ serves as a garbage collector to wipe registers clean by killing all literals except for $t$.

**Lemma 6.1.** Let $\kappa$ be a state of the memory of the form

$$\kappa = (k_1, k_2, \ldots, k_i, \ldots, k_j, \ldots, k_n, 0, 0, \ldots).$$

Then $\mathcal{A} \preceq [\kappa] = B_\preceq (k_i, k_j)$.

**Proof.** According to Figs. 4 and 5, we develop a properly terminated computation $\mathcal{C}$ so that $\mathcal{C}[\kappa] = B_\preceq (k_i, k_j)$.

Case 1: Suppose that $B_\preceq (k_i, k_j)$ is true. Then, starting with the initial configuration

$$(q_{\mathcal{A}_0}, \kappa) = (q_{\mathcal{A}_0} \otimes p_i^{k_i} \otimes p_j^{k_j} \otimes X)$$

where

$$X = (p_1^{k_1} \otimes \cdots \otimes p_{i-1}^{k_{i-1}} \otimes p_{i+1}^{k_{i+1}} \otimes \cdots \otimes p_{j-1}^{k_{j-1}} \otimes p_{j+1}^{k_{j+1}} \otimes \cdots \otimes p_n^{k_n}),$$

we develop the desired computation $\mathcal{C}$ in the following way (See Fig. 4 assuming that both $k$ and $d$ are equal to $k_i$):

(i) First, having chosen the instruction $(q_{\mathcal{A}_0} \rightarrow q_1)$ to be applied, we move along the main branch, getting exactly into the configuration $(q_1, \kappa) = (q_1 \otimes p_i^{k_i} \otimes p_j^{k_j} \otimes X)$.

(ii) Then, one and the same instruction $\tau_0 = ((q_1 \otimes p_i \otimes p_j) \rightarrow (q_4 \otimes t))$, is applied successively $k_i$ times, so that the number of the occurrences of literals $p_i$ and $p_j$ is decreasing by $k_i$, and the number of the occurrences of literal $t$ is increasing.
monotonically from 0 to $k_i$. In other words, on the main branch we create a chain of $k_i$ edges and label each of these edges by one and the same instruction $\tau_0$, getting into the configuration $(q_1 \otimes p_i^{k_i-1} \otimes t^k \otimes X)$.

(iii) Here we apply our branching instruction $(q_1 \cdots (q_3 \otimes g_1))$ to produce the following son configurations: $(q_3 \otimes p_j^{k_j-1} \otimes t^k \otimes X)$ (on the main branch) and $(g_1 \otimes p_j^{k_j-1} \otimes t^k \otimes X)$ (that will be the root of a $g_1$-branch).

(iv) With the help of killing commands from $\mathcal{F}_1$:

\[
\begin{cases}
((g_1 \otimes p_k) \cdots (g_1)) & (k = 1, 2, \ldots, i-1, i+1, \ldots, n), \\
((g_1 \otimes t) \cdots (g_1)) \\
(g_1 \cdots q_0),
\end{cases}
\]

we develop the $g_1$-branch, contracting our $g_1$-configuration to $(g_1 \otimes 1)$ and eventually to the trivial final configuration $(q_0 \otimes 1)$.

(v) Coming back to the main branch, we apply repeatedly ($k_i$ times) the restoring instruction $((q_3 \otimes t) \cdots (q_3 \otimes p_i \otimes p_j))$, getting into the configuration $(q_3 \otimes p_i^{k_i} \otimes$
and thereby restoring the original number of the occurrences of literals $p_i$ and $p_j$.

(vi) In its turn, we apply the branching instruction: $(q_3 \ominus (q_T \otimes g))$ to produce the following son configurations: $(q_T, \kappa) = (q_T \otimes p_i^{k_i} \otimes p_j^{k_j} \otimes X)$ (that is the desired final configuration on the main branch) and $(g \otimes p_i^{k_i} \otimes p_j^{k_j} \otimes X)$ (the latter will be the root of a $g$-branch).

(vii) Similar to the previous items, with the help of killing commands from $T_g$, we terminates properly the $g$-branch of our computation at the final configuration $(q_0 \otimes 1)$.

Case 2: The case where $B_\leq(k_i, k_j)$ is false is handled in the same way. The only difference is that here we develop the desired computation $\mathcal{C}$ following the pattern shown in Fig. 5.

In the opposite direction, we should prove that an arbitrary properly terminated computation $\mathcal{C}$ performed by $\mathcal{A}_\leq$ cannot be but of the form described above. Let $\mathcal{C}$ be a properly terminated computation with the root: $(q_{\mathcal{A}_\leq}, \kappa) = (q_{\mathcal{A}_\leq} \otimes p_i^{k_i} \otimes p_j^{k_j} \otimes X)$. The
instruction used at the first edge of $\mathcal{C}$ cannot but be either $(q,\delta_{\mathcal{C}} \to q_1)$, or $(q,\delta_{\mathcal{C}} \to q_2)$.
This means that we meet with two cases to be considered. For a more complex case, suppose that the latter instruction $(q,\delta_{\mathcal{C}} \to q_2)$ was applied to the initial configuration, putting us into the following configuration (See Fig. 5): $(q_2, \kappa) = (q_2 \otimes p_i^k \otimes p_j^k \otimes X)$.

Let $k$ be the length of the longest non-branching segment that starts from this configuration. It means that at the $k$th step along the main branch that we are searching for we get into the end of the segment, which is a non-terminal configuration of the form $(q_2 \otimes p_i^{k-k} \otimes p_j^{k-k} \otimes t^k \otimes X)$. In addition to that, this configuration cannot be but the father of two sons: $(q_4 \otimes p_i^{k-k} \otimes p_j^{k-k} \otimes t^k \otimes X)$ (that is declared to be on the main branch) and $(g_2 \otimes p_i^{k-k} \otimes p_j^{k-k} \otimes t^k \otimes X)$, respectively. Let us examine the descendants of the latter son.

Taking into account the applicability conditions, we can conclude that all its non-terminal descendants must be non-branching vertices of the form $(g_2 \otimes p_i^{k-k} \otimes p_j^{k-k} \otimes t^{k-k} \otimes X')$, and thereby its terminal descendant is to be of the form $(q_0 \otimes p_i^{k-k} \otimes p_j^{k-k} \otimes t^{k-k} \otimes X')$. The proper termination of the $g_2$-branch that has just been found provides us with:

$$k_i - k \geq 1, \quad k_j - k = 0.$$

Furthermore, the effect is that $B_{\leq}(k_i, k_j)$ is false.

Let $d$ be the length of the longest non-branching segment that starts from the above configuration that has only just got on the main branch: 15

$$(q_4 \otimes p_i^{k_i-k_i} \otimes p_j^{k_j-k_j} \otimes t^{k_i} \otimes X).$$

It follows that we can develop our main branch up to the end of the segment, the configuration $(q_4 \otimes p_i^{k_i-k_i+d} \otimes p_j^d \otimes t^{k_i-d} \otimes X)$, which is to be the father of two sons: $(q_4 \otimes p_i^{k_i-k_i+d} \otimes p_j^d \otimes t^{k_i-d} \otimes X)$ (that is declared to be the final configuration on the main branch) and $(g \otimes p_i^{k_i-k_i+d} \otimes p_j^d \otimes t^{k_i-d} \otimes X)$ that will be the root of a $g$-branch ending at a vertex of the form $(q_0 \otimes p_i^{k_i} \otimes p_j^{k_j} \otimes t^{k_i-d} \otimes X')$. In its turn, the proper termination of the $g$-branch yields that

$$X' = 1, \quad k' = 0, \quad k'' = 0, \quad k_j - d = 0.$$

Hence, actually the main branch we have been developing has ended at the desired configuration $(q_4 \otimes p_i^k \otimes p_j^k \otimes X) = (q_4 \otimes \kappa)$. This means that $\mathcal{C}[\kappa] = B_{\leq}(k_i, k_j)$.

The remaining case where the instruction used at the first edge of $\mathcal{C}$ is of the form $(q,\delta_{\mathcal{C}} \to q_1)$ is handled similarly in full accordance with Fig. 4.

Finally, we can conclude that, according to whether $B_{\leq}(k_i, k_j)$ is true or not, either arbitrary properly terminated computations $\mathcal{C}$ with the root $(q,\delta_{\mathcal{C}} \to \kappa)$ are to be of the form shown in Figure 4, or those computations are to be of the form shown in Fig. 5. \hfill \square

15 Here $k$ became $k_i$. 

7. Boolean algebra of LL recognizable predicates

The class of LL recognizable predicates is proved to be closed under Boolean operations, even if we use both their "parallel" and "sequential" versions. By analogy with McCarthy's "sequential" conjunction [30],

(a) The "parallel" disjunction \((B_1 \lor B_2)\) is declared to be defined precisely when both \(B_1\) and \(B_2\) are defined,

(b) Whereas the "sequential" disjunction \((B_1 \text{ or } B_2)\) is declared to be defined if
   (b1) either \(B_1\) is \text{false} and \(B_2\) is defined,
   (b2) or the first \(B_1\) is \text{true} (saying nothing about \(B_2\)).

Theorem 7.1. The class of all partial predicates recognizable by Linear Logic automata is closed with respect to all Boolean operations.

Proof. The case of negation is readily handled by interchanging the roles of final states \(q_T\) and \(q_\perp\) in the corresponding LL automata.

Let us consider the case of the "parallel" disjunction. Suppose that \(n\)-ary partial predicates \(B_1(x_1, x_2, \ldots, x_n)\) and \(B_2(x_1, x_2, \ldots, x_n)\) are recognized by LL Automata \(\mathcal{A}_1\) and \(\mathcal{A}_2\), respectively:

\[\mathcal{A}_1 = (\mathcal{B}_1, q_{\mathcal{B}_1}, \{q_0, q_{\mathcal{T},1}, q_{\perp,1}\}, \mathcal{T}_1),\]
\[\mathcal{A}_2 = (\mathcal{B}_2, q_{\mathcal{B}_2}, \{q_0, q_{\mathcal{T},2}, q_{\perp,2}\}, \mathcal{T}_2),\]
where \(\mathcal{B}_1 \cap \mathcal{B}_2 = \{q_0\}\). Renaming states of \(\mathcal{A}_2\), we introduce a new "copy" \(\mathcal{A}_2':\)

\[\mathcal{A}_2' = (\mathcal{B}_2', q_{\mathcal{B}_2'}, \{q_0, q_{\mathcal{T},2'}, q_{\perp,2'}\}, \mathcal{T}_2'),\]

which is used to recognize \(B_2\) with \(\mathcal{A}_2'\). The desired "parallel disjunct" \(\mathcal{A}\) is defined as follows:

\[\mathcal{A} = (\mathcal{B}, q_{\mathcal{B}}, \{q_0, q_{\mathcal{T}}, q_{\perp}\}, \mathcal{T})\]

where

(a) \(\mathcal{B} = \mathcal{B}_1 \cup \mathcal{B}_2 \cup \mathcal{B}_2'\),

(b) \(q_{\mathcal{B}} = q_{\mathcal{B}}\),

(c) \(\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2 \cup \mathcal{T}_2' \cup \mathcal{T}_1\), with \(\mathcal{T}_1\) consisting of the following interface instructions:

\[\mathcal{T}_1 = \begin{cases} 
(q_{\mathcal{T},1} \dashv q_{\mathcal{B}_1}), \\
(q_{\perp,1} \dashv q_{\mathcal{B}_1'}), \\
(q_{\mathcal{T},2} \dashv q_{\mathcal{T}}), \\
(q_{\perp,2} \dashv q_{\mathcal{T}}), \\
(q_{\mathcal{T},2'} \dashv q_{\mathcal{T}}), \\
(q_{\perp,2'} \dashv q_{\perp}). \end{cases}\]

For a given state of the memory \(\kappa\) of the form \(\kappa = (k_1, k_2, \ldots, k_n, 0, 0, \ldots)\), there are three cases to be considered.
Case 1: Suppose that

\[ B_1(k_1, k_2, \ldots, k_n) \text{ is true,} \]
\[ B_2(k_1, k_2, \ldots, k_n) \text{ is defined.} \]

Then an arbitrary properly terminated computation \( \mathcal{C} \) performed by \( \mathcal{A} \) such that the root of \( \mathcal{C} \) is of the form \( (q_{\mathcal{A}, \kappa}) \), is to be of the form shown in Fig. 6(a) where

\[ q_2 = \begin{cases} 
q_{T,2}, & \text{if } B_2(k_1, k_2, \ldots, k_n) \text{ is true}, \\
q_{\bot,2}, & \text{if } B_2(k_1, k_2, \ldots, k_n) \text{ is false}. 
\end{cases} \]

In addition, \( \mathcal{C}[\kappa] = \text{true} \).

Case 2: Suppose that

\[ B_1(k_1, k_2, \ldots, k_n) \text{ is false,} \]
\[ B_2(k_1, k_2, \ldots, k_n) \text{ is defined.} \]

Then an arbitrary properly terminated computation \( \mathcal{C} \) performed by \( \mathcal{A} \) such that the root of \( \mathcal{C} \) is of the form \( (q_{\mathcal{A}, \kappa}) \), is to be of the form shown in Fig. 6(b) where

\[ q_2' = \begin{cases} 
q_{T,2}', & \text{if } B_2(k_1, k_2, \ldots, k_n) \text{ is true}, \\
q_{\bot,2}', & \text{if } B_2(k_1, k_2, \ldots, k_n) \text{ is false}. 
\end{cases} \]

\[ q' = \begin{cases} 
q_T, & \text{if } B_2(k_1, k_2, \ldots, k_n) \text{ is true}, \\
q_{\bot}, & \text{if } B_2(k_1, k_2, \ldots, k_n) \text{ is false}. 
\end{cases} \]

Furthermore, \( \mathcal{C}[\kappa] = B_2(k_1, k_2, \ldots, k_n) \).

Case 3: In the case where

\[ B_1(k_1, k_2, \ldots, k_n) \text{ is undefined, or } B_2(k_1, k_2, \ldots, k_n) \text{ is undefined,} \]

Fig. 6. The "parallel" disjunction.
there is no properly terminated computation $C$ performed by $A$ such that the root of $C$ is of the form $(q_d, \kappa)$, and, hence, $A[\kappa]$ is undefined as well.

Now, bringing together all the cases considered, we can conclude that

$$A[\kappa] = (B_1(k_1, k_2, \ldots, k_n) \lor B_2(k_1, k_2, \ldots, k_n)).$$

By means of the corresponding changes in $T_I$, the remaining Boolean operations are handled in the same way. ☐

8. Language $\mathcal{MVF}$. Syntax and semantics

In this section we introduce a programming language, $\mathcal{MVF}$, aimed at programming Multi-Valued Functions from the states of the memory to the states of the memory.

Being a version of the language of non-deterministic programs, $\mathcal{MVF}$ incorporates typical sequential and non-deterministic programming constructs, like alternative and repetitive constructs introduced by Dijkstra [6, 7].

The formal semantics of $\mathcal{MVF}$ is given through a compositional interpretation of the statements $S$ of the language in the input-output manner.

Namely, following the schemata of a guarded statement $S$, we compose a multivalued mapping from initial states of the memory $\kappa$ to final states of the memory $\zeta$: $S: \kappa \rightarrow S[\kappa]$ where $S[\kappa]$ is the set of all final states $\zeta$, to which $S$ can "lead" from $\kappa$.

**Definition 8.1.** The only data type considered in $\mathcal{MVF}$ is the type of non-negative integers. An infinite sequence of registers $\text{reg}_1, \text{reg}_2, \ldots, \text{reg}_m, \ldots$ is assumed to be given. Each register can contain non-negative integers. The current value of an $m$th register is represented by the variable $x_m$. The current state of the memory $\kappa$ is represented by the linearly ordered sequence of non-negative integers contained in the corresponding registers: $\kappa = (k_1, k_2, \ldots, k_m, \ldots)$. The basic unit of the syntax of $\mathcal{MVF}$ is the notion of guarded statements that are built up in the inductive way.

**Definition 8.2.** A guarded statement $S$ is defined to be one of the following expressions:

(i) an elementary assignment statement,
(ii) a sequential construct,
(iii) an if-then-else construct,
(iv) an alternative construct,
(v) a while construct,
(vi) a repetitive construct.

**Definition 8.3.** To contract the set of elementary statements without loss of generality, we use only elementary assignment statements of one of the three "primitive" forms:

(a) $x_m := c$, 

(b) \( x_m := x_m + c \),
(c) \( x_m := x_m - c \),
where \( c \) is a non-negative integer constant.

The formal semantics of those elementary statements is introduced straightforwardly. A less trivial case is the following:

**Definition 8.4.** Let \( S \) be an assignment statement of the form \( x_m := n - c \). For a given state of the memory \( \kappa \):

\[
\kappa = (k_1, k_2, \ldots, k_{m-1}, k_m, k_{m+1}, \ldots),
\]

if \( k_m \geq c \) then \( S[\kappa] \) is defined to be \( \{ \zeta \} \):

\[
S[\kappa] = \{ \zeta \}, \quad \text{where} \quad \zeta = (k_1, k_2, \ldots, k_{m-1}, k_m - c, k_{m+1}, \ldots).
\]

Otherwise, \( S[\kappa] \) is declared to be *undefined*.

**Definition 8.5.** A sequential construct \( S \) is defined in the standard way:

begin
\[
S_1;
S_2;
\ldots
S_n;
end
\]

The semicolons in sequential constructs have the usual naive meaning: "The guarded statements \( S_i \) should be executed successively in the natural order."

**Definition 8.6.** Let \( S \) be a sequential construct of the form

\[
S = \left\{ \begin{array}{ll}
\text{begin} \\
S_1; \\
S_2;
\text{end}
\end{array} \right.
\]

If \( S_1[\kappa] \) is defined and for every state of the memory \( \rho \) from \( S_1[\kappa] \) the set \( S_2[\rho] \) is also defined, then

\[
S[\kappa] = \bigcup_{\rho \in S_1[\kappa]} S_2[\rho].
\]

Otherwise, \( S[\kappa] \) is declared to be *undefined*.

Our compositional semantics of sequential constructs \( S \) from Definition 8.5 in the case where \( n > 2 \) is introduced with a straightforward induction, such an \( S \) being
treated as equivalent to a sequential construct of the form

\[
\begin{align*}
\text{begin} \\
\text{begin} \\
S_1; \\
S_2; \\
\ldots \\
S_{n-1}; \\
\text{end}; \\
S_n; \\
\text{end}
\end{align*}
\]

**Definition 8.7.** An if-then-else construct \( S \) is of the standard form as well:

\[
\begin{align*}
\text{if} & \quad B \quad \text{then} \quad S_1 \\
\text{else} & \quad S_2 \\
\text{endif}
\end{align*}
\]

**Definition 8.8.** The compositional semantics of the above if-then-else construct \( S \) is introduced in the natural way:

\[
S[\kappa] = \begin{cases} 
S_1[\kappa], & \text{if } B(\kappa) \text{ is true,} \\
S_2[\kappa], & \text{if } B(\kappa) \text{ is false,} \\
\text{undefined}, & \text{if } B(\kappa) \text{ is undefined.}
\end{cases}
\]

Guarded commands are introduced as building blocks for alternative and repetitive constructs.

**Definition 8.9.** A guarded command is an expression of the form

\[
\square B \rightarrow S
\]

where \( S \) is a guarded statement, and the guard \( B \) is a Boolean expression, i.e. a Boolean combination of "primitive" predicates of the forms

\[
(x_i = x_j), \quad (x_i < x_j), \quad \text{and} \quad (x_i \leq x_j).
\]

**Definition 8.10.** An alternative construct \( S \) has the syntax:

\[
\begin{align*}
\text{if} \\
\square B_1 \rightarrow S_1 \\
\square B_2 \rightarrow S_2 \\
\ldots \\
\square B_n \rightarrow S_n \\
\text{fi}
\end{align*}
\]
According to [6, 7], the naive semantics of such an alternative construct $S$ is as follows (see Fig. 7):

When $S$ is executed for a given state of the memory,
(a) The guards $B_i$ are evaluated.
(b) If each of the guards is false, the program will abort.
(c) Otherwise, an arbitrary guarded statement $S_i$ with the true guard $B_i$ is allowed to be selected for execution.

This is formalized in the following way.

**Definition 8.11.** Let $S$ be an alternative construct of the above form in Definition 8.10. For a given state of the memory $\kappa$, if at least one of guards $B_i$ is true and for every true $B_i$ the set $S_i[\kappa]$ is defined, then

$$S[\kappa] = \bigcup_{B_i \text{ is true}} S_i[\kappa].$$

Otherwise, $S[\kappa]$ is declared to be undefined.

**Proposition 8.1.** It is readily seen that the if-then-else construct $S$ taken from Definition 8.7 is semantically equivalent to the following alternative construct:

```
if
  if $B$ then $S_1$
  else $\neg B$ then $S_2$
fi
```

**Definition 8.12.** A while construct $S$ has the form:

```
do while $B$
  $S'$
endloop
```

Whereas $S'$ within the above while construct $S$ is allowed to be non-deterministic, such an $S$ is provided with the following "parallel" semantics:
Definition 8.13. Let $S$ be a while construct of the form in Definition 8.12. For a given state of the memory $\kappa$, $S[\kappa]$ is declared to be defined if and only if there exists a finite rooted tree built up of states of the memory such that (see Fig. 8)

(a) The root coincides exactly with $\kappa$.
(b) For each of terminal vertices $\zeta$, $B$ is false.
(c) For each of non-terminal vertices $\rho$, $B$ is true, and the set of all sons of this vertex $\rho$ coincides exactly with $S'[\rho]$.

Furthermore, having got such a tree, the set of all its terminal vertices $\zeta$ is declared to be $S[\kappa]$.

Definition 8.14. A repetitive construct $S$ is an expression of the form:

\[
\text{do}
\begin{align*}
\quad 
\& B_1 \to S_1 \\
\& B_2 \to S_2 \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \qa
exists a finite rooted tree built up of states of the memory such that
(a) The root coincides exactly with \( \kappa \).
(b) For each of terminal vertices \( \zeta \), all guards \( B_i \) are false.
(c) For each of non-terminal vertices \( \rho \), at least one of guards \( B_i \) is true and the set of all sons of this vertex \( \rho \) coincides exactly with \( S'[^\rho] \) where \( S' \) denotes the following "one-step" instance of \( S \):

\[
S' = \begin{cases} 
\text{if} & B_1 \rightarrow S_1 \\
\text{if} & B_2 \rightarrow S_2 \\
& \ldots \\
\text{if} & B_n \rightarrow S_n 
\end{cases}
\]

Furthermore, having got such a tree, the set of all its terminal vertices \( \zeta \) is declared to be \( S[\kappa] \).

**Proposition 8.2.** It is readily seen that the repetitive construct \( S \) taken from Definition 8.14 is semantically equivalent to the following while construct:

\[
\text{do while } (B_1 \lor B_2 \lor \ldots \lor B_n) \\
\text{if} \\
\quad B_1 \rightarrow S_1 \\
\quad B_2 \rightarrow S_2 \\
\quad \ldots \\
\quad B_n \rightarrow S_n \\
\text{fi}
\]

endloop

with while condition \( B = (B_1 \lor B_2 \lor \ldots \lor B_n) \), being the parallel disjunction of all guards \( B_i \).

9. MVś statements \( \Rightarrow \) LL automata

9.1. Synthesizing LL automata for MVś statements

In this section we prove our main expressiveness result.

**Theorem 9.1.** Let \( S \) be a guarded statement, all of its variables belong to some finite set \( \{x_1, x_2, \ldots, x_n\} \). Then we can construct a Linear Logic automaton \( \mathcal{A}_S \) such that, whatever state of the memory \( \kappa \) of the form

\( \kappa = (k_1, k_2, \ldots, k_n, 0, 0, \ldots) \)

we take, the following holds: \( \mathcal{A}_S[\kappa] = S[\kappa] \).
Proof. We assemble the desired LL Automaton $\mathcal{A}_S$ by induction on the schemata of $S$. The case of an elementary assignment statement of the form $x_m := c$ is handled in the following way.

We represent this statement by the sequential construct:

\[
\begin{align*}
\text{begin} & \\
& x_m := 0; \\
& x_m := x_m + c; \\
\text{end}
\end{align*}
\]

and then apply Theorem 5.1 to LL automata $\mathcal{A}_0$ and $\mathcal{A}_c$ taken from Examples 4.2 and 4.4, respectively. Elementary assignment statements of the form $x_m := x_m \pm c$ are simulated by LL Automata $\mathcal{A}_c$ and $\mathcal{A}_{-c}$ taken from Examples 4.4 and 4.3, respectively.

The case of $S$ being a sequential construct is handled with the help of Theorem 5.1.

The case of $S$ being an alternative construct is considered in the following theorem.

Theorem 9.2. Let $S$ be an alternative construct of the form taken from Definition 8.10 where

(i) For some $m$, each of variables occurred in $S$ belongs to \{\(x_1, x_2, \ldots, x_m\}\).

(ii) Each of guards $B_i$ is recognized by an LL automaton $\mathcal{B}_i = (2_{\mathcal{A}_i}, q_{\mathcal{A}_i}, \{q_0, q_{\top, \mathcal{A}_i}, q_{\bot, \mathcal{A}_i}\}, \mathcal{T}_{\mathcal{A}_i})$.

(iii) The “parallel” disjunction $B = (B_1 \lor B_2 \lor \cdots \lor B_n)$, is recognized by an LL Automaton $\mathcal{B} = (2_{\mathcal{A}}, q_{\mathcal{A}}, \{q_0, q_{\top, \mathcal{A}}, q_{\bot, \mathcal{A}}\}, \mathcal{T}_{\mathcal{A}})$.

(iv) Each of guarded statements $S_i$ is computed by an LL Automaton $\mathcal{A}_i = (2_{\mathcal{A}_i}, q_{\mathcal{A}_i}, \{q_0, q_{\mathcal{A}_i}\}, \mathcal{T}_{\mathcal{A}_i})$.

(v) For every different $i$ and $j$,

\[
2_{\mathcal{A}_i} \cap 2_{\mathcal{A}_j} = 2_{\mathcal{A}_i} \cap 2_{\mathcal{A}_i} = 2_{\mathcal{A}_i} \cap 2_{\mathcal{A}_i} = 2_{\mathcal{A}_i} \cap 2_{\mathcal{A}_i} = \{q_0\}.
\]

Then $S$ is computed by the following LL automaton $\mathcal{A} = (2_{\mathcal{A}}, q_{\mathcal{A}}, \{q_0, q'\}, \mathcal{T})$, where

(a) $2 = \{q', g\} \cup 2_{\mathcal{A}} \cup \bigcup_{i=1}^n 2_{\mathcal{A}_i} \cup \bigcup_{i=1}^n 2_{\mathcal{A}_i}$,

(b) $q_{\mathcal{A}} = q_{\mathcal{A}}$,

(c) $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_{\mathcal{A}} \cup \bigcup_{i=1}^n \mathcal{T}_{\mathcal{A}_i} \cup \bigcup_{i=1}^n \mathcal{T}_{\mathcal{A}_i}$, with $\mathcal{T}_1$ consisting of the following interface instructions:

\[
\mathcal{T}_1 = \begin{cases}
(q_{\top, \mathcal{A}} \leftarrow (q_{\mathcal{A}_1} \oplus q_{\mathcal{A}_2} \oplus \cdots \oplus q_{\mathcal{A}_n})), \\
(q_{\bot, \mathcal{A}} \leftarrow q_{\mathcal{A}}) & (i = 1, 2, \ldots, n), \\
(q_{\bot, \mathcal{A}} \leftarrow g) & (i = 1, 2, \ldots, n), \\
((g \otimes p_k) \leftarrow g) & (k = 1, 2, \ldots, m), \\
(g \leftarrow q_0), \\
(q_i \leftarrow q') & (i = 1, 2, \ldots, n).
\end{cases}
\]
Proof. There are two lemmas to be proved.

Lemma 9.2.i. For a given state of the memory $\kappa$, if at least one of guards $B_i$ is true and for every true $B_i$ the set $S_i[\kappa]$ is defined, then there is a properly terminated computation $\mathcal{C}$ performed by $\mathcal{A}$ such that

$$\mathcal{C}[\kappa] = \bigcup_{\mathcal{A}_i[\kappa] = \text{true}} \mathcal{A}_i[\kappa].$$

Proof. Such a computation $\mathcal{C}$ is shown in Fig. 9:

(a) First, by means of instructions from $\mathcal{T}_A$ we move from the initial configuration $(q_{A_1}, \kappa)$ into the configuration $(q_{B_i}, \kappa)$.

(b) Then we apply our branching instruction $(q_{A_1} \rightarrow (q_{B_i} \oplus q_{B_2} \oplus \cdots \oplus q_{B_n}))$ to produce $n$ configurations $(q_{B_i}, \kappa), (q_{B_2}, \kappa), \ldots, (q_{B_n}, \kappa)$, where each $(q_{B_i}, \kappa)$ will be the root of its own “ith subtree”.

(c) The next step is to evaluate each of guards $B_i$ by means of instructions from the corresponding $\mathcal{T}_B$.
(d) For each $\mathcal{B}_i$-final vertex of the form $(q_T, \mathcal{A}_i, \kappa)$, we develop our $i$th subtree in the following way. By applying the interface instruction $(q_T, \mathcal{A}_i, q_{\mathcal{A}_i})$ we get into an initial configuration for $\mathcal{A}_i$ of the form: $(q_{\mathcal{A}_i}, \kappa)$. Then with the help of instructions from $\mathcal{F}_{\mathcal{A}_i}$ we compute all $\zeta$ from $\mathcal{A}_i[\kappa] = \{\zeta_{i,1}, \ldots, \zeta_{i,k}\}$, and by the interface instruction $(q_i \rightarrow q')$ we complete eventually our $i$th subtree.

(e) For each $\mathcal{B}_j$-final vertex of the form $(q_\perp, \mathcal{B}_j, \kappa)$, we develop our $j$th subtree in a different way. By applying the interface instruction $(q_\perp, q_\perp, -o g)$ we get into a configuration of the form: $(g, \kappa)$. Then with the help of killing commands ($(g \otimes p_{\mathcal{A}_i}) -o g$) we wipe our memory clean, getting into $(g, \theta)$, and by the interface instruction $(g \rightarrow q_0)$ we complete our $j$th subtree at the trivial final configuration $(q_0, \theta)$.

It is clear that

$$\mathcal{C}[\kappa] = \bigcup_{\mathcal{A}_i[\kappa] = \text{true}} \{\zeta_{i,1}, \ldots, \zeta_{i,k}\} = \bigcup_{\mathcal{A}_i[\kappa] = \text{true}} \mathcal{A}_i[\kappa]. \quad \square$$

**Lemma 9.2.ii.** Let $\mathcal{C}$ be a properly terminated computation performed by $\mathcal{A}$ such that the root of $\mathcal{C}$ is of the form $(q_{\mathcal{A}}, \kappa)$. Then $\mathcal{C}$ is to be of the form described above in Fig. 9, and furthermore at least one of guards $B_i$ is to be true, and for every true $B_i$ the set $S_i[\kappa]$ is to be defined.

**Proof.** According to applicability conditions, on any path leading from the root of $\mathcal{C}$ to a terminal vertex of the form $(q', \zeta)$ there is exactly one vertex of the form $(q_T, \mathcal{A}, \kappa)$. Let $\mathcal{C}_0$ be the result of truncation of the tree $\mathcal{C}$ by cutting off all proper descendants of these unique vertices.

Actually this $\mathcal{C}_0$ is a properly terminated computation performed by $\mathcal{B}$. Definition 6.1 has the effect on that $\mathcal{C}_0$ is to be unique in the sense of Definition 4.2, $\mathcal{B}[\kappa]$ is to be true, and in the whole tree $\mathcal{C}$ there is exactly one vertex of the form $(q_{\mathcal{A}}, \kappa)$, which, in its turn, is to be the father of $n$ sons of the form: $(q_{\mathcal{A}_i}, \kappa), (q_{\mathcal{A}_2}, \kappa), \ldots, (q_{\mathcal{A}_n}, \kappa)$. For each $i$, let us consider the "$i$th subtree" with the root $(q_{\mathcal{A}_i}, \kappa)$ (see Fig. 9).

There are two cases to be considered.

**Case 1:** Suppose that some terminal vertices of the $i$th subtree are of the form $(q', \zeta)$.

Similar to the previous item, we can show that the $i$th subtree is to be composed of the unique computations performed by $\mathcal{B}_i$ and $\mathcal{A}_i$ on the input $\kappa$, and $\mathcal{B}_i[\kappa]$ is to be true, and $\mathcal{A}_i[\kappa]$ is to be defined.

**Case 2:** Suppose that the $i$th subtree is meaningless: all its terminal vertices are of the form $(q_0, \theta)$.

According to Definition 6.1, there is no meaningless computation performed by $\mathcal{B}_i$ proper. Therefore, taking into account applicability conditions, in the $i$th subtree we can find exactly one vertex of the form $(q_{\perp, \mathcal{A}_i}, \kappa)$. Definition 6.1 has also the effect on that $\mathcal{B}_i[\kappa]$ is to be false, and our $i$th subtree is to be composed of the unique computation performed by $\mathcal{B}_i$ on the input $\kappa$ and a "$g$-branch" that starts from $(q_{\perp, \mathcal{A}_i}, \kappa)$. \ensuremath{\square}
Theorem 9.3. Let $S$ be a while construct of the form taken from Definition 8.12 where

(i) The while condition $B$ is recognized by an LL automaton $\mathcal{B} = (\mathcal{Q}_\mathcal{B}, \mathcal{q}_0, \{q_0, q_{\top, \mathcal{B}}, q_{\perp, \mathcal{B}}\}, \mathcal{T}_\mathcal{B})$.

(ii) The guarded statements $S'$ is computed by an LL automaton $\mathcal{A}' = (\mathcal{Q}_{\mathcal{A}'}, \mathcal{q}_{\mathcal{A}'}, \{q_0, q'_1\}, \mathcal{T}_{\mathcal{A}'})$.

(iii) $\mathcal{Q}_\mathcal{B} \cap \mathcal{Q}_{\mathcal{A}'} = \{q_0\}$

Then $S$ is computed by the following LL automaton $\mathcal{A} = (\mathcal{Q}, \mathcal{q}_\mathcal{A}, \{q_0, q_1\}, \mathcal{T})$, where

(a) $\mathcal{Q} = \{q_1\} \cup \mathcal{Q}_\mathcal{B} \cup \mathcal{Q}_{\mathcal{A}'}$,

(b) $q_\mathcal{A} = q_{\perp, \mathcal{B}}$,

(c) $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_\mathcal{B} \cup \mathcal{T}_{\mathcal{A}'}$, with $\mathcal{T}_1$ consisting of the following interface instructions:

$$\mathcal{T}_1 = \begin{cases} (q_{\top, \mathcal{B}} \rightarrow q_{\mathcal{A}'}) , \\ (q_{\perp, \mathcal{B}} \rightarrow q_1) , \\ (q' \rightarrow q_{\mathcal{A}}) . \end{cases}$$

Proof. There are two lemmas to be proved.

Lemma 9.3.i. Suppose we have a finite rooted tree $T$ built up of states of the memory such that (see Fig. 8)

(a) The root of $T$ coincides with a given state of the memory $\kappa$.

(b) For each of its leaves $\zeta$, $B(\zeta)$ is false.

(c) For each of its non-terminal vertices $\rho$, $B(\rho)$ is true, and the set of all sons of this vertex $\rho$ coincides exactly with $S'[\rho]$.

Then there is a properly terminated computation $\mathcal{C}$ performed by $\mathcal{A}$ such that $\mathcal{C}[\kappa]$ coincides exactly with the set of all leaves of the tree $T$.

Proof. Running from the root of $T$ to $T$'s leaves, we develop the desired computation $\mathcal{C}$ as follows (see Fig. 10):

(a) First, by means of instructions from $\mathcal{T}_\mathcal{B}$ we move from the initial configuration $(q_{\mathcal{A}'}, \kappa)$ into a configuration of the form $(q_{1,1}, \kappa)$, where

$$q_{1,1} = \begin{cases} q_{\top, \mathcal{B}} , & \text{if } B(\kappa) \text{ is true} , \\ q_{\perp, \mathcal{B}}, & \text{if } B(\kappa) \text{ is false} . \end{cases}$$

(b) Suppose that $B(\kappa)$ is true, and, hence, all the sons of the $T$'s root form the set $\mathcal{S}'[\kappa] = \{\rho_1, \ldots, \rho_k\}$. Then, by applying the interface instruction $(q_{\top, \mathcal{B}} \rightarrow q_{\mathcal{A}'})$, we get into an initial configuration for $\mathcal{A}'$ of the form $(q_{\mathcal{A}'}, \kappa)$. With the help of instructions from $\mathcal{T}_{\mathcal{A}'}$ we compute all $\rho$ from $\mathcal{S}'[\kappa]$, getting into configurations of the form $(q'_j, \rho_1), \ldots, (q'_j, \rho_k)$. And, by applying the interface instruction $(q' \rightarrow q_{\mathcal{A}})$ to each of $\mathcal{A}'$-final vertices just been computed, we get into the configurations $(q_{\mathcal{A}}, \rho_1), \ldots, (q_{\mathcal{A}}, \rho_k)$ that directly correspond to the root's sons $\rho_1, \ldots, \rho_k$. 
Suppose that one of the current vertices in the tree $T$, say $\rho_1$, is a leaf, and hence, $B(\rho_1)$ is false. Then by means of instructions from $T_B$ we move from the initial configuration $(q_{A},\rho_1)$ into a configuration of the form $(q_{A}',\rho_1)$, and eventually into a final configuration of the form $(q_{A},\rho_1)$, with the help of the interface instruction $(q_{A},\rho_1 \rightarrow q_{A})$.

For each of non-terminal vertices in the tree $T$, say $\rho_k$, this entire process of developing $e$ is repeated (see Fig. 10).

As a result, all leaves of the tree $T$ form the set $e[\kappa]$. $\square$
Lemma 9.3.ii. Let $\mathcal{C}$ be a properly terminated computation performed by $\mathcal{A}$ such that the root of $\mathcal{C}$ is of the form $(q, \kappa)$. Then $\mathcal{C}$ is to be of the form described above in Fig. 10, and furthermore we can transform $\mathcal{C}$ into a finite rooted tree $T$ built up of states of the memory such that

(a) The root of $T$ coincides exactly with $\kappa$.
(b) For each of $T$'s leaves $\zeta$, $\mathcal{A}[\zeta] = \text{false}$.
(c) For each of non-terminal vertices $\rho$ of $T$, $\mathcal{A}[\rho] = \text{true}$, $\mathcal{A}'[\rho]$ is defined, and $\mathcal{A}'[\rho]$ coincides exactly with the set of all sons of this vertex $\rho$ in $T$.

Proof. Starting from the root of $\mathcal{C}$, we will construct the desired tree $T$ by induction (see Fig. 8):

(a) The root of $T$ is defined to be $\kappa$. The root of $T$ is associated with the initial configuration in $\mathcal{C}$: $(q, \kappa)$.

(b) Suppose that $\mathcal{A}[\kappa] = \text{true}$. Similar to Lemma 9.2.ii, we can truncate our $\mathcal{C}$ so that the resulting $\mathcal{C}_0$ will be the unique properly terminated computation performed by $\mathcal{A}$ on the input $\kappa$. The main final vertex of $\mathcal{C}_0$ is to be of the form $(q, \mathcal{T}, \mathcal{A}, \kappa)$ (see Fig. 10), and furthermore within $\mathcal{C}$ this vertex is to be the father of the single son $(q, \mathcal{A}', \kappa)$. Reproducing the arguments of Lemma 9.2.ii once more, we truncate the subtree with the root $(q, \mathcal{A}', \kappa)$ so that the resulting tree $\mathcal{C}'$ will be the unique properly terminated computation performed by $\mathcal{A}'$ on the input $\kappa$. The effect is that $\mathcal{A}'[\kappa]$ is to be defined, for some positive $k$: $\mathcal{A}'[\kappa] = \{\rho_1, \ldots, \rho_k\}$, and $\mathcal{C}'$-final vertices are of the form $(q, \rho_1), \ldots, (q, \rho_k)$, and each of them is to be the father of the single son, so that the following anti-chain of their sons are being identified in $\mathcal{C}$: $(q, \rho_1), \ldots, (q, \rho_k)$. In this case we extend the fragment of $T$ (that has already been constructed) as follows (see Fig. 8): We create $k$ outgoing edges of the form $(\kappa, \rho_1), \ldots, (\kappa, \rho_k)$, and associate these new vertices in $T$: $\rho_1, \ldots, \rho_k$ with the above $\mathcal{A}$-initial configurations in $\mathcal{C}$: $(q, \rho_1), \ldots, (q, \rho_k)$, respectively.

(c) Suppose that for one of these new vertices in the tree $T$, say $\rho_1$, we have: $\mathcal{B}[\rho_1] = \text{false}$. Similar to the previous item, we can prove that the subtree with the root $(q, \rho_1)$ in our $\mathcal{C}$ is to be composed of the unique computation performed by $\mathcal{B}$ on the input $\rho_1$ and the single edge $((q, \mathcal{A}, \rho_1), (q, \rho_1))$. Furthermore, such a $\rho_1$ will be eventually a leaf in the tree $T$.

(d) Suppose that for one of these new vertices in the tree $T$, say $\rho_k$, we have $\mathcal{B}[\rho_k] = \text{true}$. Then we will examine the subtree with the root $(q, \mathcal{A}, \rho_k)$ in our $\mathcal{C}$. It is readily seen that such a subtree can be conceived of as a properly terminated computation performed by $\mathcal{A}$ on the input $\rho_k$. Hence, the entire process of extending $T$ described above can be repeated (see Figs. 10 and 8).

Thus, our inductive process results in the desired tree $T$ of the form in Fig. 8, and in the unique form of the given $\mathcal{C}$ described in Fig. 10. \(\square\)

Now, bringing together all the cases considered and Propositions 8.1 and 8.2, we can complete Theorem 9.1. \(\square\)
9.2. $\mathcal{MV}$ programs $\Rightarrow$ Linear Logic

In conclusion we give a complete characterization of guarded statements $S$ in terms of Linear Logic proper:

**Corollary 9.1.** Let $S$ be a guarded statement, all of its variables belong to some set \{x_1, x_2, \ldots, x_n\}. Then, for some literals $q'$, $q_0$, and $q_1$, we can construct a multiset $\mathcal{F}_S$, consisting of Horn implications and $\oplus$-Horn implications, such that, whatever state of the memory $\kappa$ of the form $\kappa = (k_1, k_2, \ldots, k_n, 0, 0, \ldots)$ we take, the following holds:

(A) $S[\kappa]$ is defined if and only if there is a Linear Logic derivable $(!\oplus)$-Horn sequent of the form

$$q', \kappa, !\mathcal{F}_S \vdash (q_0 \oplus (q_1 \otimes (Z_1 \otimes Z_2 \oplus \cdots \oplus Z_k))).$$

(b) If $S[\kappa]$ is defined then $S[\kappa]$ coincides exactly with the minimal set of states of the memory \{x_1, x_2, \ldots, x_k\} such that a sequent of the form

$$q', \kappa, !\mathcal{F}_S \vdash (q_0 \oplus (q_1 \otimes (\zeta_1 \otimes \zeta_2 \oplus \cdots \oplus \zeta_k)))$$

is derivable in Linear Logic.

Moreover, there is a direct correspondence between cut-free derivations of the latter "minimal" sequent and terminated computations of $\mathcal{F}$ that start from the given state of the memory $\kappa$.

**Proof.** According to Theorem 9.1, $S$ can be computed by an LL Automaton $\mathcal{S}_S$ of the form $\mathcal{S}_S = (\mathcal{S}_S, q', \{q_0, q_1\}, \mathcal{F}_S)$.  

(A) With this $\mathcal{F}_S$ and our Completeness\footnote{Where $\zeta_0$ becomes $\emptyset$, and $\emptyset \equiv \mathbf{1}$, and $(q_0 \otimes \zeta_0) \equiv q_0$.} the first item of Corollary 9.1 is evident.

(B) According to Theorem 3.1, a sequent of the form

$$q', \kappa, !\mathcal{F}_S \vdash (q_0 \oplus (q_1 \otimes (\zeta_1 \otimes \zeta_2 \oplus \cdots \oplus \zeta_k)))$$

is derivable in Linear Logic if and only if there exists a properly terminated computation $C$, performed by $\mathcal{S}_S$ on the input $\kappa$, such that $S[\kappa] = \mathcal{S}_S[\kappa] = C[\kappa] \subseteq \{\zeta_1, \zeta_2, \ldots, \zeta_k\}$. The effect is that $S[\kappa]$ coincides exactly with the intersection of all "derivable" sets \{\zeta_1, \zeta_2, \ldots, \zeta_k\}. $\square$

Coming back to the case of computations yielding deterministic results, we can characterize all partial recursive functions in terms of Horn-like Linear Logic sequents as follows:

**Corollary 9.2.** For any $n$-ary partial recursive function $f$, we can construct a multiset $\mathcal{F}_f$, consisting of Horn implications and $\oplus$-Horn implications, such that, whatever non-negative integers $k_1, k_2, \ldots, k_n$, and $z$ we take, the following holds: $f(k_1, k_2, \ldots, k_n)$
= z if and only if a Horn-like sequent of the form

\[ q', p_1^{k_1}, p_2^{k_2}, \ldots, p_n^{k_n}, \exists \mathcal{F}_f \vdash (q_0 \oplus (q_1 \otimes p_1)) \]

is derivable in Linear Logic.

Moreover, any cut-free derivation of the latter sequent, read from its axiomatic vertices to its root, forms a computation leading from the inputs \( k_1, k_2, \ldots, k_n \) to the output \( f(k_1, k_2, \ldots, k_n) \).

**Proof.** Every \( n \)-ary partial recursive function \( f \) is computed by a guarded statement \( S_f \), and, according to Theorem 9.1, by an L.L. automaton \( \mathcal{A}_f \) of the form \( \mathcal{A}_f = (2_f, q', \{q_0, q_1\}, T_f) \). It remains to use Theorem 3.1. 0

9.3. Properties of \( \mathcal{MVF} \) programs

In closing, there are two remarks concerning semantics of programming languages. The guarded commands approach to non-deterministic programming was introduced by Dijkstra [6, 7]. The basic point of the Dijkstra's semantics of non-deterministic programs was the concept of the predicate transformer \( wp \):

**Definition 9.1.** According to [6], "we use the notation \( wp(S, R) \), where \( S \) denotes a statement and \( R \) some condition on the state of the system, to denote the weakest precondition for the initial state \( \kappa \) of the system such that activation of \( S \) is guaranteed to lead to a properly terminating activity leaving the system in a final state \( \zeta \) satisfying the postcondition \( R \) (even in the case of possibly non-deterministic behavior)."

The formal semantics of \( \mathcal{MVF} \) introduced in this paper, and thereby our computational semantics of Horn Linear Logic derivability, can be conceived of as a natural semantics behind the Dijkstra's semantics.

As for the Dijkstra's predicate transformer \( wp \) proper, the intuitive Definition 9.1 can be reformulated as follows:

**Definition 9.2.** Let \( R \) be any condition on states of the system. For a given guarded statement \( S \), we define the weakest precondition \( wp(S, R) \) by the following:

\[ wp(S, R)(\kappa) = "(S[\kappa] \text{ is defined}) \text{ and } \forall \zeta((\zeta \in S[\kappa]) \rightarrow R(\zeta))". \]

We can furthermore justify our approach by the following theorem:

**Theorem 9.4.** All properties of the Dijkstra's predicate transformer \( wp \) declared in [6, 7] can be proved within the framework of Definition 9.2.

**Proof.** Based on Theorem 9.1 and Corollary 9.1, we can convert all formal definitions of [6, 7] into valid theorems. 0
The foregoing LL Automata semantics can be also conceived of as a natural semantics behind the Hoare’s approach to program semantics introduced in [15]. In particular, the Hoare’s statement:

If $P$ (the precondition) holds before executing $S$, then $R$ (the postcondition) holds when $S$ terminates.

can be formally introduced in one of the following two ways:

(A) \( \forall \kappa (P(\kappa) \rightarrow ((S[\kappa] \text{ is defined}) \rightarrow \forall \zeta ((\zeta \in S[\kappa]) \rightarrow R(\zeta)))) \) (this item represents the partial correctness case).

(B) \( \forall \kappa (P(\kappa) \rightarrow ((S[\kappa] \text{ is defined}) \text{ and } \forall \zeta ((\zeta \in S[\kappa]) \rightarrow R(\zeta)))) \) (this represents the total correctness case).

Such a proposal can be also justified by validation of Hoare’s proof system [15], even if we extend Hoare’s rules to non-deterministic programs.

10. Concluding remarks

It was a common point that only restricted classes of computations can be expressed within the propositional framework of logical systems. Contrary to this common point, strong connections between propositional fragments of Linear Logic and very rich complexity classes have been established (see [25, 17, 34, 20], etc.).

The emphasis in this paper is on whether propositional Linear Logic is capable of handling the well-known standard and non-standard constructions of traditional Programming in the natural way. (See also [35] where the related problem of comprehensive computational understanding of Linear Logic is discussed.)

Based on our Linear Logic Automata model, which has been introduced with a particular stress laid on the paradigm

\[ \text{Computations} \leftrightarrow \text{Derivations}, \]

we have demonstrated that the proof machinery of Linear Logic can provide us with the natural and direct simulation of all programming constructs taken from traditional sequential programming and non-deterministic programming: \(^{17}\)

\[ \text{Non-Deterministic Programs} \]

Theorem 9.1

\[ \text{Linear Logic Automata} \]

Theorem 3.1

\[ \text{Horn Linear Logic} \]

\(^{17}\) Even if we use nothing but Horn-like propositional formulas.
Acknowledgements

I am very much obliged to P. Lincoln, J. Mitchell, V. Pratt, A. Scedrov, and N. Shankar for their inspiring introduction to the complexity problems related to Linear Logic.

I owe special thanks to J.-Y. Girard, Y. Lafont and L. Regnier for providing supportive and stimulating environment to discuss all these new concepts during my stay in Laboratoire de Mathématiques Discrètes.

I am greatly indebted to A. Nerode for valuable discussions on models of automata and automata computations. It was his proposal to look for a general automata model behind Linear Logic in which all or most useful computational constructs can be expressed.

I also appreciate the helpful comments I have received from S. Artemov’s Seminar on Logical Methods in Computer Science at the Moscow State University.

I am also grateful to Takayasu Ito for insightful comments and advice related to the guarded commands approach.

References