Guarded programs and Proofs in Linear Logic (extended abstract)

D. Galmiche & R. Couturier
CRIN - CNRS
Campus Scientifique - B.P. 239
54506 Vandœuvre-lès-Nancy Cedex
France.
e-mail : galmiche@loria.fr

1 Introduction

There are several approaches to model concurrent and object-oriented computations using logic. Linear Logic (LL), that is a resource sensitive refinement of classical logic [6], provides the means to handle many operational aspects of programming languages from a proof theoretical perspective [1, 4]. Moreover a logical characterization of typical features of concurrent, imperative and object-oriented languages could yield a clear semantical counterpart of their operational meaning [2]. In [4] we have shown that Full Intuitionistic Linear Logic (FILL) seems adapted as logical foundation for a framework to involve object, concurrent and logic programming dealing with non-determinisms and concurrency. It considered the both paradigms *proofs as computations (formulas as states)* and *proof search as computation* and the relationships between extended Petri nets and a logical framework (calculus based on FILL with formulas as processes) where we can modelize some important concurrent and object-oriented programming concepts. One of our goals is to model systems that can dynamically modify their state as in [2] and to analyze them from a proof theoretical perspective involving concurrent and non-deterministic computations. In this context, we want to study how to simulate the behavior of imperative paradigms [2]. A first step for this study will be to consider the work of M. Kanovich on simulating guarded programs in Linear Logic [9] that is based on non-deterministic Petri nets with states and to clarify the relationship with our previous work. Let us recall that the guarded commands approach to non-deterministic and parallel programming was introduced by Dijkstra [3]. In this paper we propose new logical systems to deal with guarded programs and their executions and we analyze a direct correspondence between a fragment of linear logic and guarded programs (without intermediate use of extended Petri nets as in [9]). Moreover we explain how to manage structures, non-determinisms and concurrency in this context. For that we propose compiling rules, for the different instructions, to simulate guards and guarded programs into LL and their executions into adequate LL proofs [5].
These results and their applications, with the encoding of structures into LL, have to be compared and enriched with other works dealing with imperative paradigms, concurrent objects and proofs in based-on LL frameworks [2, 4].

2 Guarded programs

The guarded commands approach to non-deterministic and parallel programming was introduced by Dijkstra [3].

2.1 Guarded commands

These commands are introduced as building blocks for sequential, alternative and repetitive structures and the program semantics is defined by means of predicate transformer $wp(S, R)$ [3]. Each instruction block in a structure is executable if and only if the preceding guard is satisfied. When several guards are satisfied a block of instructions is chosen in a non-deterministic way among the activable blocks. If no guard is satisfied we leave the structure. In a repetitive structure, we execute an instructions block until there exists a guard that is satisfied and then evaluate again all the guards. As such a program involves some non-deterministic choices in the structures the program execution is not unique.

2.2 Simulation of guarded commands

M. Kanovich has proposed a system to simulate guarded programs in Linear Logic (LL) [9]. This proposal consists in introducing a natural semantics behind the initial one. If one consider each non-deterministic statement S as a multi-valued mapping from initial program state κ to final state ζ then the predicate transformer wp can be defined by $wp(S, R)(\kappa) \equiv (S[\kappa] \neq \emptyset \; and \; \forall\zeta(\zeta \in S[\kappa] \rightarrow R(\zeta)))$. All formal definitions in [3] can be converted to theorems. The rules of the logical system are the one of Intuitionistic Linear Logic (ILL). After defining the language of guarded commands he proves that each type of instructions can be simulated with non-deterministic Petri nets [8] extended with states. Moreover he proves that the reachability problem for such nets is equivalent to the derivability problem for specific sequents in Linear Logic. These sequents, called Horn-like sequents, have the following form $\widetilde{q_1}, \widetilde{\kappa}, !\widetilde{T_S} \vdash (\widetilde{q_0} \oplus (\widetilde{q_2} \otimes (\widetilde{\zeta_1} \oplus \widetilde{\zeta_2} \oplus \ldots \oplus \widetilde{\zeta_n})))$ where q_1 is the initial state, q_0 is the final failure state, q_2 the final success state, T_S is the theory encoding the system specified by the extended net (it includes the logical rules representing the instructions of the guarded program S), κ is the set of variables of the system, ζ_i represents the different sets of variables corresponding to the different possible executions of the system. Let us remark that the ˜sign means a translation into LL of the expression.

Let us precise the meaning of such a sequent. Starting with the initial state $\widetilde{q_1}$ with a set of variables $\widetilde{\kappa}$ and the logical rules of the theory $\widetilde{T_S}$ (representing the instructions of the guarded program), one obtains the final failure state $\widetilde{q_0}$ or the final success $\widetilde{q_2}$ state with one of the sets of variables $\widetilde{\zeta_i}$. A guarded program is correct if there exists a proof of such a sequent in the Horn Linear logic [8].

2.3 Some remarks

Let us mention now some problems to simulate guards, with the example of the guard $x \leq y$. To simulate it we need to introduce 8 rules in the theory [9] with the following principle: we simultaneously decrease by 1 the values of x and y while none of the both variables is null (moreover we increment at the same time a temporary variable). Then, if y is equal to zero we change the state and we can increment the variables x and y and decrease by 1 the temporary variable until it will be zero. At any moment in this process we can change the state and reach in a non-deterministic way the final failure state. Thus the proof of a guard $x \leq y$ can be long and we have to apply the rules in a certain way to obtain the final success state.

The major points in the previous proposal [9] are the explicit uses of non-deterministic Petri nets with states and of the correspondence between reachability in these nets and provability in a LL fragment, to define the simulation of guarded program in LL.
Moreover some points have to be clarified in this seminal approach: how to directly and systematically translate (or compile) a guarded program into LL (using or not extended nets) ? how to manage a repetitive or alternative structure ? are the guarded programs we are able to simulate interesting ? how to manage some non-determinisms and concurrency ? what is the relationship with other based-on LL proposals that model systems which can dynamically modify their state as per proofs as computations perspective ?

3 Compiling guarded programs into LL

In oder to compile guarded programs into LL, we propose different types of rules concerning four cases: assignments, guards, structures and instructions blocks.

3.1 Assignments

In an assignment the value of a variable is the value of an expression that is also composed by variables or constants connected by arithmetic operators.
To compile the assignment $v := expression$ we consider a LL rule of the form:
$s_i \otimes v^a \otimes {v_1}^{a_1} \otimes \ldots \otimes {v_n}^{a_n} \multimap s_{i+1} \otimes v^{exp} \otimes {v_1}^{a_1} \otimes \ldots \otimes {v_n}^{a_n}$
where *exp* obtained by replacing all occurrences of variables by their values in *expression*.

3.2 Guards

In a repetitive and alternative structure all guards have to be tested before to execute a block of instructions. To know if a guard i can be activated, we introduce a variable g_i. A guard being valid or not we associate two LL rules to each guard that we compile
$s_i \otimes {v_1}^{a_1} \otimes \ldots \otimes {v_n}^{a_n} \multimap^{gar} s_{i+1}{v_1}^{a_1} \otimes \ldots \otimes {v_n}^{a_n} \otimes g_i,\ s_i \multimap s_{i+1}$
where *gar* is obtained by replacing all variable occurrences in the guard by their values. The condition *gar* allows the application of the rule when the condition is true.

3.3 Structures

The structures are more difficult to compile and there are different ways to do it. To simulate a structure, we have to evaluate all guards. Each evaluation of guard i implies the creation of a variable g_i. If there exists a block (and thus a g_i) that can be activated we choose a block in a non-deterministic way, else we leave the structure. After the execution of a block, if the structure is alternative then we execute the next instruction, else if the structure is repetitive we evaluate again all the guards until none is satisfied. In the both cases we suppress all the g_i variables after the execution of a block,.
The structure to compile begins with a state i and we consider structures with m guards, indexed from 0 to $m-1$. As each guard needs two LL rules we have $2m$ rules:
$s_i \otimes V_0 \multimap^{gar_0} s_{i+1} \otimes V_0 \otimes g_i$
$s_i \multimap s_{i+1}$
$\vdots$
$s_{i+j} \otimes V_j \multimap^{gar_j} s_{i+j+1} \otimes V_j \otimes g_{i+j}$
$s_{i+j} \multimap s_{i+j+1}$

$\vdots$
$s_{i+m-1} \otimes V_{m-1} \multimap^{gar_{m-1}} s_{i+m} \otimes V_{m-1} \otimes g_{i+m-1}$
$s_{i+m-1} \multimap s_{i+m}$
where V_j is the set of variables of guard j, $j \in [0, m-1]$.
Let us remark that the variables that indicate the activability of a block are indexed form i to $i+m-1$ and thus we can execute several structures in parallel, each structure having a different i.

To know if there exists a guard that is true, we have to test the number of g_i with the two following rules: $s_{i+m} \otimes {g_{i+j}}^{nb} \multimap^{nb>0} s_{i+m+1} \otimes {g_{i+j}}^{nb}$ and $s_{i+m} \multimap s_{end}$.
The first one can be applied when the number of variables $g_{i+j}, j \in [0, m-1]$ is greater than zero. Else we can apply the second rule that allows to execute the next instruction. Let us recall that the term ${g_{i+j}}^{nb}$ allows to instantiate all occurrences of variables g_{i+j}. These variables are different but by using the term ${g_{i+j}}^{nb}$, we want to count the variables g_{i+j}.
If we consider the non-deterministic choice of blocks to execute, it is realized by the following rule: $s_{i+m+1} \multimap (s_{i+m+2} \oplus \ldots \oplus s_{i+m+2+j} \oplus \ldots \oplus s_{i+m+2+m-1})$, s_{i+m+2} is the first instruction of the first block and the block 0, $s_{i+m+2+m-1}$ is the first instruction of the last block $m-1$.
Before to execute a block of instructions j, we have to know if it can be activated, i.e., if g_{i+j} is present. In this case the block j can be executed else we have to choose another block by passing in the state s_{i+m+1}. It is realized by the $2m$ rules:
$s_{i+m+2} \otimes g_i \multimap s_{i+2m+2}$
$s_{i+m+2} \multimap s_{i+m+1}$
$s_{i+m+3} \otimes g_{i+1} \multimap s_{f_0+1}$
$s_{i+m+3} \multimap s_{i+m+1}$
$\vdots$
$s_{i+m+2+j} \otimes g_{i+j} \multimap s_{f_{j-1}+1}$
$s_{i+m+2+j} \multimap s_{i+m+1}$
$\vdots$
$s_{i+m+2+m-1} \otimes g_{i+m-1} \multimap s_{f_{m-2}+1}$
$s_{i+m+2+m-1} \multimap s_{i+m+1}$
where s_{f_j} is the state of the last instruction of the block j, $j \in [0, m-1]$. The last instruction of a block j has the form $s_{f_j} \ldots \multimap s_{f_{m-1}+1} \ldots, j \in [0, m-1]$, $s_{f_{m-1}+1}$ being the state that suppresses the variables g_{i+j}. Therefore this state has to be considered after each execution of a block.
When a block is executed, we have to suppress the variables g_{i+j} with the rule: $s_{f_{m-1}+1} \otimes {g_{i+j}}^{nb} \multimap s_{f_{m-1}+2}$ if the structure is alternative ($s_{f_{m-1}+2}$ is the state of the next instruction) or the rule $s_{f_{m-1}+1} \otimes {g_{i+j}}^{nb} \multimap s_i$ if the structure is repetitive (s_i is the state allowing to start the guards evaluation). Moreover, the next instruction (after the structure) starts with the state $s_{f_{m-1}+2}$.

3.4 Blocks of instructions

The blocks of instructions are composed with assignment statements. The sequentiality between the instructions is realized by the increasing of the state index between the different rules.

Example 3.1 *The compilation of* $v_3 = v_1; v_1 = v_2; v_2 = v_3$ *is realized by the rules:*
$s_i \otimes {v_1}^{a_1} \otimes {v_3}^{a_3} \multimap s_{i+1} \otimes {v_1}^{a_1} \otimes {v_3}^{a_1}$
$s_{i+1} \otimes {v_1}^{a_1} \otimes {v_2}^{a_2} \multimap s_{i+2} \otimes {v_1}^{a_2} \otimes {v_2}^{a_2}$
$s_{i+2} \otimes {v_2}^{a_2} \otimes {v_3}^{a_1} \multimap s_{i+3} \otimes {v_2}^{a_1} \otimes {v_3}^{a_1}$

3.5 Dynamic compiling

This method is different from the previous one in the way to choose the block to execute. The previous method use the rule $s_{i+m+1} \multimap (s_{i+m+2} \oplus \ldots \oplus s_{i+m+2+j} \oplus \ldots \oplus s_{i+m+2+m-1})$ that allows to execute all blocks (having the g_{i+j} variable).
The principle of this method consists in dynamically introducing a new rule of the form $s_{i+m+1} \multimap (s_{i+m+2} \oplus \ldots \oplus s_{i+m+2+j} \oplus \ldots \oplus s_{i+m+2+m-1})$ where the states $s_{i+m+2+j}$ are in the rule iff the g_{i+j} variables are present. This rule suppress the g_{i+j} variables at the same time it introduces the states $s_{i+m+2+j}$ in the dynamic rule.
Then we have not to suppress the g_{i+j} variables after the execution of a block. The two previous rules $s_{f_{m-1}+1} \otimes {g_{i+j}}^{nb} \multimap s_{f_{m-1}+2}$ and $s_{f_{m-1}+1} \otimes {g_{i+j}}^{nb} \multimap s_i$ are transformed into $s_{f_{m-1}+1} \multimap s_{f_{m-1}+2}$ (alternative case) or $s_{f_{m-1}+1} \multimap s_i$ (repetitive case).
We can prove that the result of the structure simulation with the first method is identical to the result obtained with the second one.

3.6 Important remarks

Here, we use meta-rules. For instance, to simulate a guard we fix a condition for the application of the operator $\multimap$ that uses values of system variables. If it is true we can apply the rule else we apply another rule that works on the same state.
Instead to introduce a dynamic rule in a structure we can consider that the rule exists after the compilation. But in this case all states j representing the first instruction of a block j will be suppressed of a non-deterministic rule, dedicated to a block choice, if the variable g_{i+j} is not present, thus when the guard associated to the j block is not true.
Before to compile a program that we want to execute in parallel, we need to construct a dependency graph associated to the program that indicates the parts that can be executed with several processes. Then, we need a rule that duplicates the number of processes and each of it has its block of instructions to execute. Finally, we need a rule to synchronize all these processes.

4 Proof systems for simulation

We define the notions of generalized implications and generalized sequents [7] adapted to our problem.

Definition 4.1 Generalized implications
(i) A deterministic implication *is a formula of the form :* $(S \otimes V) \multimap (S' \otimes V')$.
(ii) A non deterministic implication *is a formula of the form:* $S \multimap (S_1 \oplus \ldots \oplus S_n)$.
(iii) A parallel implication *is a formula of the form:* $S \multimap (S_1 \otimes \ldots \otimes S_n)$.

Let us precise some notations. S represents a state or a set of states, V represents a set of variables of the form $v_1 \otimes \ldots \otimes v_n$ (where v_i is a variable), R represents the current transition rule (that is a generalized implication), T_s represents the theory encoding the system (multi-set of generalized implications) and F_S represents a set of the final states of the form $((S_1 \otimes V_1) \oplus \ldots \oplus (S_n \otimes V_n))$. We recall that we use the connective $\oplus$ (resp. $\otimes$) on states to represent non determinism (resp. parallelism).
Compared to [9] no variable will have~sign, the states are numbered in increasing order and the states indexes respect the sequentiality of the guarded program. Moreover, the variables and their values are represented by ${v_1}^{a_1} \otimes \ldots \otimes {v_n}^{a_n}$ with a_i value of v_i.

Definition 4.2 Generalized sequents
For any T_s and R consisting of generalized implications a sequent of the form $!T_s, S, V, [R] \vdash F_S$, where S represents the state of the system and V the variables, is called a generalized sequent.

The notation $[R]$ means that the current rule R is not always present in the sequent. In fact, the application of a rule with R present corresponds to a system's evolution. An important point to notice is that a sequent can have different evolutions depending on the rules that are applied. For for the ones corresponding to *(i)* the state of the system and the variables can change but for the ones corresponding to *(ii)* and *(iii)* only the state of the system can change.

4.1 A system for non deterministic programs

Definition 4.3 *We call* SFNDP *(system for non deterministic programs) the formal system defined by the following rules:*

$$\frac{!T_s, S, V, R \vdash F_S}{!T_s \otimes !R, S, V \vdash F_S}\ L\otimes!$$

$$\frac{!T_s, S_2, V_2 \otimes V_2{}' \vdash F_S}{!T_s, S_1, (V_1 \otimes V_2{}'), (S_1 \otimes V_1) \multimap (S_2 \otimes V_2) \vdash F_S}\ L\multimap_{ordi}$$

$$\frac{!T_s, S_1, V \vdash F_S \quad \ldots \quad !T_s, S_i, V \vdash F_S \quad \ldots \quad !T_s, S_n, V \vdash F_S}{!T_s, S, V, S \multimap (S_1 \oplus \ldots \oplus S_i \oplus \ldots \oplus S_n) \vdash F_S}\ L\multimap\oplus$$

$$\frac{}{!T_s, S_i, V_i \vdash ((S_1 \otimes V_1) \oplus \ldots \oplus (S_i \otimes V_i) \oplus \ldots \oplus (S_n \otimes V_n))}\ R\oplus_i$$

with the structural rules (L!, W!, C!) of HLL [7] and the cut rule.

Theorem 4.1 *A generalized sequent of the form $!T_s, S, V, [R] \vdash F_S$ is derivable in LL iff it is derivable in the $SFNDP$ system.*

Proof 4.1 *Proof by induction on derivations.*

4.2 A system for non deterministic parallel programs

Definition 4.4 *We call* SFNDPP *(system for non deterministic parallel programs) the system defined by the rules of $SFNDP$ with the three following rules:*

$$\frac{!T_s, S_1 \otimes \ldots \otimes S_n, V \vdash F_S}{!T_s, S, V, S \multimap (S_1 \otimes \ldots \otimes S_n) \vdash F_S}\ L\multimap\otimes$$

$$\frac{!T_s, S_1 \otimes \ldots \otimes S_n, V_1 \otimes \ldots \otimes V_n \vdash S_1' \otimes V_1' \otimes \ldots \otimes S_n' \otimes V_n' \quad !T_s, S_1' \otimes \ldots \otimes S_n', V_1' \otimes \ldots \otimes V_n' \otimes V \vdash F_S}{!T_s, S_1 \otimes \ldots \otimes S_n, V_1 \otimes \ldots \otimes V_n \otimes V \vdash F_S}\ Cut$$

$$\frac{!T_s, S_1, V_1 \vdash S_1' \otimes V_1' \quad \ldots \quad !T_s, S_i, V_i \vdash S_i' \otimes V_i' \quad \ldots \quad !T_s, S_n, V_n \vdash S_n' \otimes V_n'}{!T_s, S_1 \otimes \ldots \otimes S_i \otimes \ldots \otimes S_n, V_1 \otimes \ldots \otimes V_i \otimes \ldots \otimes V_n \vdash S_1' \otimes V_1' \otimes \ldots \otimes S_i' \otimes V_i' \otimes \ldots \otimes S_n' \otimes V_n'}\ R\otimes$$

Let us describe the principles of a proof of a parallel program. Here we consider sequential programs having some parts that will be executed in parallel. We can consider three different consecutive steps: a) the duplication of the principal process, b) the evolution of each process until its synchronization state, c) the synchronization when all processes are in their synchronization state. The first one is obtained with the $L \multimap \otimes$ rule that duplicates the current process into n processes. Then we have to prove that the synchronization state of each process allows to the system to reach a final state. Such a proof is obtained with the *cut* rule. We have to prove that each process changes until to reach a state allowing a synchronization with the other processes. The rule $R\otimes$ allows to make this proof.

4.3 Some remarks

Let us notice, having in mind the compiling rules presented in the previous section, that we have constraints for the applications of the rules. If we want to test the following guard : x<y with x=5 and y=8 it is translated into the rule: $s_1 \otimes {v_x}^a \otimes {v_y}^b \multimap^{a<b} s_{i+1} \otimes {v_x}^a \otimes {v_y}^b$. that is only applicable if the instantiation of variables a and b (values of x and y) is such that a<b is true. Then if, with the values of x and y, the variables a and b are instantiated by 5 and 2, the rule is not applicable. Then we have to respect this constraint to correctly simulate a program with guarded commands.

Moreover, if we want to test the guard number $i : x = y$, then we have the two following rules: $s_i \otimes {v_x}^a \otimes {v_y}^b \multimap^{a=b} s_{i+1} \otimes {v_x}^a \otimes {v_y}^b \otimes g_i$, $s_i \multimap s_{i+1}$. To correctly simulate the guard, we have to apply at first the first rule. If it is applicable (if a value of x equal b value of y) then the variable g_i indicates that the guard i is activable. Else, we apply the second rule (always applicable) that allows to change the state of the system.

Here we have proposed some rules to simulate a non-deterministic parallel program. Compared to [7] we have more compact rules that means a reduced number of rules but more compact rules (including more actions). Moreover, the application of the rules can only be done following the execution order of the program. In [7] we can prove the end of a program before having proved the beginning using the rule $L \multimap$ and the *cut* rule. On our case, it is not possible because the $L \multimap$ ($L \multimap_{ordi}$, $L \multimap \oplus$ and $L \multimap \otimes$) rules are applied on the current state and not on a state that is not yet reached. Concerning the parallelism, we have to use a *cut* to simulate a part of parallel program. This cut application needs that we know the synchronization state and the values of variables of each process. To go over this difficulty we have to do the proof of each process until it reachs its synchronization state. Therefore we can continue the proof construction after this parallel part.

5 Simulation of guarded programs in LL

We have proposed a detailed method to compile guarded programs into LL rules with a complete management of the structures. We have a global evaluation of guards in a deterministic way with two rules for simple or composed guards and thus do not fix limitations for the assignment statements. Moreover, we compile directly the instructions of a guarded program into rules of LL without intermediate use of the extended non-deterministic Petri nets [9]. Taking into account the proof systems we have proposed we can prove the following results

Theorem 5.1 *The execution of a guarded program S can be simulated into the SFNDP system with a proof of a generalized sequent $!T_s, S, V, [R] \vdash F_S$.*

Moreover, there is an exact correspondence between cut-free derivations of the sequent and terminated computations of S.

Proof 5.1 *by induction on the type of instructions of a guarded program, using the previous compiling rules.*

Theorem 5.2 *The execution of a parallel guarded program S can be simulated into the SFNDPP system with a proof of a generalized sequent* $!T_s, S, V, [R] \vdash F_S$.

Proof 5.2 *by induction on the type of instructions of a guarded program, using the previous compiling rules.*

6 Concluding remarks

It is clear that there are connections between this approach to simulate guarded programs into LL, works in LL frameworks to model systems that can dynamically modify their state and works on linear process calculus to represent concepts of logic, concurrent and object-oriented programming. We will study relationships between structures and computations in imperative languages and objects, classes and methods in such LL frameworks.

References

[1] J.M. Andreoli and R. Pareschi. Linear objects: logical processes with built-in inheritance. *New Generation Computing*, 9:445–473, 1991.

[2] G. Delzanno and M. Martelli. Objects in Forum. In *International Logic Programming Symposium, ILPS'95*, Portland, Oregon, December 1995.

[3] E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivations of programs. In *Current Trends in Programming Methodology, vol 1, Software Specification and Design*, pages 233–242, 1977.

[4] D. Galmiche and E. Boudinet. Proofs, concurrent objects and computations in a FILL framework. In *Workshop on Object-based Parallel and Distributed Computation, OBPDC'95*, Tokyo, Japan, 1995. to appear LNCS, Springer Verlag.

[5] D. Galmiche and G. Perrier. On proof normalization in linear logic. *Theoretical Computer Science*, 135(1):67–110, 1994.

[6] J.Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[7] M. Kanovich. Linear logic as a logic of computations. *Annals of Pure and Applied Logic*, 67:183–212, 1994.

[8] M. Kanovich. Petri nets, Horn programs, Linear logic and Vector games. In *Int. Symposium on Theoretical Aspects of Computer Software, TACS'94, LNCS 789*, pages 642–666, Sendai, Japan, April 1994.

[9] M. Kanovich. Simulating guarded programs in linear logic. In *Int. Workshop on Theory and Practice of Parallel Programming, LNCS 907*, pages 45–69, Sendai, Japan, November 1994.

A An example

Let us consider a program that sorts three elements in an increasing order. It has the form

x:=9; y:=5; z:=3;
do
x>y→t:=x;x:=y;y:=t;
y>z→t:=y;y:=z;z:=t;
od

The rules obtained after the compiling step are:
$R1 : s_1 \otimes x^a \multimap s_2 \otimes x^9$
$R2 : s_2 \otimes y^b \multimap s_3 \otimes y^5$
$R3 : s_3 \otimes z^c \multimap s_4 \otimes z^3$
$R4 : s_4 \otimes x^a \otimes y^b \multimap^{a>b} s_5 \otimes x^a \otimes y^b \otimes g_4$
$R5 : s_4 \multimap s_5$
$R6 : s_5 \otimes y^b \otimes z^c \multimap^{b>c} s_6 \otimes y^b \otimes z^c \otimes g_5$
$R7 : s_5 \multimap s_6$
$R8 : s_6 \otimes {g_{4+j}}^{nb} \multimap^{nb>0} s_7$
$R9 : s_6 \multimap s_{15}$
$R10 : s_7 \multimap (s_8 \oplus s_9)$
$R11 : s_8 \otimes x^a \otimes t^d \multimap s_{10} \otimes x^a \otimes t^a$
$R12 : s_{10} \otimes x^a \otimes y^b \multimap s_{11} \otimes x^b \otimes y^b$
$R13 : s_{11} \otimes y^b \otimes t^a \multimap s_{14} \otimes y^a \otimes t^a$
$R14 : s_9 \otimes y^b \otimes t^d \multimap s_{12} \otimes y^b \otimes t^b$
$R15 : s_{12} \otimes y^b \otimes z^c \multimap s_{13} \otimes y^c \otimes z^c$
$R16 : s_{13} \otimes z^c \otimes t^b \multimap s_{14} \otimes z^b \otimes t^b$
$R17 : s_{14} \multimap s_4$
s_{15} : final state

Let us remark that $j \in [0,1]$ for the rule R8. Rules R4 to R7 are the rules that evaluate the guards. Rules R8 and R9 test if we have to leave a loop or not. Rule R10 makes the non-deterministic choice on the block to execute.

The following proof corresponds to the execution of the guarded program

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\pi_1 \qquad \pi_2}{!T_s, s_7, x^9 \otimes y^5 \otimes z^3, s_7 \multimap (s_8 \oplus s_9)(R10) \vdash F_s}\; L \multimap \oplus}{!T_s, s_6, x^9 \otimes y^5 \otimes z^3 \otimes g_4 \otimes g_5, R8 \vdash F_s}\; L \multimap_{ordi}}{!T_s, s_5, x^9 \otimes y^5 \otimes z^3 \otimes g_4, R6 \vdash F_s}\; L \multimap_{ordi}}{!T_s, s_4, x^9 \otimes y^5 \otimes z^3, R4 \vdash F_s}\; L \multimap_{ordi}}{!T_s, s_3, x^9 \otimes y^5, R3 \vdash F_s}\; L \multimap_{ordi}}{!T_s, s_2, x^9, R2 \vdash F_s}\; L \multimap_{ordi}}{!T_s, s_1, , R1 \vdash F_s}$$

with $F_s = (s_{15} \otimes x^3 \otimes y^5 \otimes z^9 \otimes t^5) \oplus (s_{15}, x^3 \otimes y^5 \otimes z^9 \otimes t^9)$. At this level, we have to look π_1 and π_2, knowing that the two variables g_4 et g_5 have disappear with the use of rule R10. π_1 is a proof that the program is correct by exchanging at first x and y (first guard and first block). π_2 is a proof that the program is correct by exchanging at first y and z (second guard and second block).