# Synthesizing Imperative Object-Oriented Programs using a Logic of Actions[1]

Patrick BELLOT

École Nationale Supérieure des Télécommunications
Computer Science department, Software Engineering group
46 rue Barrault, 75634 Paris Cedex 13, France

Email: Patrick.Bellot@inf.enst.fr, Tél: 33-1 45 81 78 81, Fax: 33-1 45 81 31 19.

**Abstract**

In the article, we present a logical system which allows to specify and to synthesize imperative object-oriented programs. It is a logic of actions. Formulae can describe actions, situations and mathematical assertions. The specifications are written in this logic which is a linear sequent calculus. Then a proof of the assertion is done. The proof uses logical and extra-logical axioms and inference rules. Proofs in this logic may have very simple linear structures. Then a program is extracted from the proof using a realization technique. This program is an object-oriented program which relies heavily on threads for parallelism and synchronization. We give two examples. The first is the well-known World of Blocks. The second example is called the Candy Machine and is an example of communicating process.

**Key words:** imperative program synthesis, logic of actions, linearity, realizability, object oriented programming, logical specifications.

## 1 Introduction

**Computing vs. programming.** Our system, the $\Omega$ logic, is designed to synthesize imperative programs. We make the distinction between *computing-by-proof* and *programming-by-proof*. Computing by proof is concerned with computations, the object of functional programming languages such as LISP or SML. Programming is mainly concerned with actions such as sorting a disk file. In this case, classical program synthesizing systems such as [7, 14, 16, 19, 21, 22, 23] are not adapted because they rely on more or less classical logics and produce $\lambda$-expressions. Even if imperative execution can be modeled with functions, it is not realistic.

**Logics and actions.** Classical and even intuitionistic logics are also not adapted for processing actions unless we use the rather heavy scheme of pre-conditions and post-conditions which do not describe the action but its consequences. But some actions may have no quantifiable consequences such as displaying an information on a screen. Alternatively, we can add a temporal parameter in formulae but temporal logics are far more complex and still talk about situations. Moreover, as J-Y. Girard showed it [11, 12, 13], the structural rules of classical and intuitionistic logics do not take into account the notion of resources which is essential when we deal with real world problems. One could decide to use non-commutative or partially non-commutative logics but the syntax of these logics is only for specialists.

**The formulae.** $\Omega$ logic is a linear logic because it takes into account that formulae are resources which cannot be freely duplicated or erased. Our atomic formulae can be either *situation* formulae which describe a situation such as *"The block A is on the block B"* or *action* formulae describing actions such as *"To put the block A on the block B"*. Finally

[1]Extended Abstracts of LOPSTR'98, Eighth International Workshop on Logic-based Program Synthesis and Transformation, 15-19 June 1998, Manchester, UK. Technical Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-98-6-1. http://www.cs.man.ac.uk/cstechrep/titles98.html

an atomic formulae can be a *mathematical* formula, or mathematical assertion, describing an eternal truth. Such formulae, which can be discarded and duplicated, are handled with modalities as in Linear Logic.

**The connectives.** The connectives of $\Omega$ logic which allow to build complex formulae from atomic ones have two meanings. The first meaning is logical and close to the meaning of the same connective in linear logic. The second meaning is in term of actions. For instance, the *times* connective, written $\otimes$, can be thought as the first linear conjunction but if $A$ and $B$ are actions then $A \otimes B$ is the action of doing $A$ and $B$ concurrently.

**Situations as actions.** This two meanings are the same as far as we think a situation formula as the action of *"doing things in such a way that the situation formula holds"*.

**The realizations.** Realizability [15] is a technique allowing to associate some object to a proved formulae. The object, called the *realization*, is built with the proof of the formula. Our realizations are objects in the sense of object-oriented programming. They all have a method $prove()$ which realizes the formula. If the formula describes an action, the method $prove()$ does the action. If the formula is a situation, the situation is verified or else created if needed. For instance, the realization of the formula *"The block A is on the block B"* can be a program which verifies that the block $A$ is on the block $B$. It can also be a program which puts the block $A$ on the block $B$. In any case, the situation must hold after the execution of the program. Moreover, the method $prove()$ must have a result. For instance, if the action is to open a disk file, the result may be a descriptor of the file. If no significant result is needed, the *nil* value, written [ ] is returned.

**The specifications.** A specification is a formula written in the $\Omega$ logic. For instance, the formula

$$\forall x, y, z \cdot top(x) \otimes on(x, y) \Rightarrow top(x) \otimes on(x, floor) \otimes top(y) \qquad (1)$$

is a formula saying that we can move any top block $x$ from its support $y$ onto the floor. To get a program doing this move, we have to prove the formula and to extract its realization from the proof. We will see how to use the realization to move a particular block.

# 2 The $\Omega$ logic

**Formulae and realizations.** We could give first the axioms and inference rules of $\Omega$ logic and *then* the method for building the realization of a theorem. But the construction of realizations can be described in the logical axioms and inference rules. Therefore our logic handles realized formulae $R : A$, read it $R$ *realizes* $A$, where $R$ is a realization and $A$ is a formula. The realization is supposed to be an object with a method $prove()$. Formulae are built from atomic formulae using the connectives we will describe.

**The theorems.** $\Omega$ logic is a sequent calculus. We prove sequents written:

$$R_1 : A_1, R_2 : A_2, \ldots, R_n : A_n \vdash S_1 : B_1, S_2 : B_2, \ldots, S_m : B_m$$

meaning that *if* $R_i$ realizes $A_i$ for $i \in [1, n]$ *then* $S_j$ realizes $B_j$ for $j \in [1, m]$. On both sides of the assertion sign $\vdash$, the realized formulae are implicitly linked by a *conjunction*. It means that all formulae and all realizations are available at the same time. The $R_i, i \in [1, n]$ are all different realizations *variables*. The $S_j, j \in [1, m]$ are realizations built using the realization variables $R_i, i \in [1, n]$ and individual variables occurring in the formulae $B_1, \ldots, B_m$. We also assume the realizations $R_i, i \in [1, n]$ to be *independent*, i.e. they can be executed in any order, sequentially or concurrently, without changing their semantics. That is a rather strong assumption which appears to be realistic.

**The axiomatization.** The axiomatization has been made difficult because of the necessity of being linear. Every hypothesis which is not a mathematical assertion must be consumed one and exactly one time. Let us take the formula (1) above. The hypotheses $top(x)$ and $on(x, y)$ must be used exactly one time during the proof and the conclusion must describe *everything* that has been done. Other difficulties come from the fact that our realizations are imperative programs. Such programs cannot be handled in any way. We will give a significant part of the axioms and inference rules.

**Notations.** $\Gamma$, $\Delta$, $\Theta$ are formulae sequences. $\mathcal{X}, \mathcal{Y}, \mathcal{Z}$ are realization variables sequences. $\mathcal{R}$, $\mathcal{S}$, $\mathcal{T}$ are realizations sequences. If $\mathcal{R} = R_1, \ldots, R_n$ has the same length as $\Gamma = G_1, \ldots, G_n$, then $\mathcal{R} : \Gamma$ denotes the sequence of realized formulae $R_1 : G_1, \ldots, R_n : G_n$. If $\mathcal{R}$ and $\mathcal{X}$ have the same length as $[\mathcal{R}/\mathcal{X}]\mathcal{S}$ denotes the simultaneous substitution of the $R_i$ to the $X_i$ in the $S_j$.

**The structural rules.** These rules are unusual because of the unusual fully conjunctive structure of our sequents. The rules are the following named *identity*, *transitivity* and *monotony*[2]:

$$\mathcal{X} : \Gamma \vdash \mathcal{X}' : \Gamma' \tag{2}$$

$$\frac{\mathcal{X} : \Gamma \vdash \mathcal{R} : \Delta \quad \mathcal{Y} : \Delta \vdash \mathcal{S} : \Theta}{\mathcal{X} : \Gamma \vdash [\mathcal{R}/\mathcal{Y}]\mathcal{S} : \Theta} \tag{3}$$

$$\frac{\mathcal{X} : \Gamma \vdash \mathcal{R} : \Delta}{\mathcal{X} : \Gamma, \mathcal{Y} : \Theta \vdash \mathcal{R} : \Delta, \mathcal{Y} : \Theta} \tag{4}$$

**The proofs structure.** Besides classical tree structure of proof, the transitivity axiom (3) allows a linear structure. If we assume proofs of sequents $\Gamma_i \vdash \Gamma_{i+1}$ for $i \in [0, n-1]$ then we can build a proof of $\Gamma_0 \vdash \Gamma_n$. We admit the presentation: $\Gamma_0 \vdash \Gamma_1 \vdash \cdots \vdash \Gamma_{n-1} \vdash \Gamma_n$. This is a linear structure of proofs which is very useful when the sequents $\Gamma_i \vdash \Gamma_{i+1}$ are axioms for $i \in [0, n-1]$. Moreover, let us assume an axiom $\Gamma \vdash \Delta$, using the monotony axiom (3), we can deduce $\Theta_1, \Gamma, \Theta_2 \vdash \Theta_1, \Delta, \Theta_2$ for any $\Theta_1, \Theta_2$. We admit $\Theta_1, \Gamma, \Theta_2 \vdash \Delta, \Theta_2$ as a generalized instance of $\Gamma \vdash \Delta$.

Using linear proof and generalized instances of axioms, proofs are much more simpler than classical tree-like proofs. It seems [3, 4, 25] that human and automatic search for proofs is also less complex. We will see an example of such a simple proof in section 5.

**The $\otimes$ conjunction.** $A \otimes B$, read it $A\ times\ B$, means executing concurrently $A$ and $B$. This connective will be especially explained because it shows a lot of the problems encountered. If $A$ and $B$ are situation formulae, $\otimes$ is close to the linear conjunction with same name. The realization of a formula $A \otimes B$ is an object which $prove()$ method concurrently does $A$ and $B$ and which returns a pair $[a, b]$, $a$ and $b$ being respectively significant of $A$ and $B$ executions. The first axiom allows to introduce $\otimes$:

$$R : A, S : B \vdash \pi(R, S) : A \otimes B \tag{5}$$

where $\pi(R, S).prove()$ concurrently computes $r = R.prove()$ and $s = S.prove()$. It returns $[r, s]$. $R$ and $S$ are assumed to be *independent*, thus it is possible to execute them concurrently. It is not allowed to split a formula $A \otimes B$. Firstly because we cannot be sure that its realization has the form $\pi(R, S)$ with $R : A$ and $S : B$ and secondly, because even in this case we cannot be sure that $R$ and $S$ are independent. One just have to think of $A$ and $B$ as being communicating processes sharing a buffered channel, $A$ may write on the channel and $B$ may read on the channel. $A$ and $B$ may be run concurrently. $A$ may be run alone but $B$ cannot since it would wait forever for an incoming data. Therefore, unlike usual systems we need rules for expliciting the natural properties of the connectives. For instance:

$$R : A \otimes B \vdash \pi_c(R) : B \otimes A \tag{6}$$

where $\pi_c(R)$ is an object which $prove()$ method computes $[r, s] = R.prove()$ and returns $[s, r]$. The following is a generalization of commutativity:

$$R : A_1 \otimes \cdots \otimes A_n \vdash \pi_p(\{i_1, i_2, \ldots, i_n\}, R) : A_{i_1} \otimes A_{i_2} \otimes \cdots \otimes A_{i_n} \tag{7}$$

where $i_1, \ldots, i_n$ is a permutation of $1, \ldots, n$. $\pi_p(\{i_1, i_2, \ldots, i_n\}, R)$ is an object which $prove()$ method computes $[[..[a_1, a_2], ..], a_n] = R.prove()$ and returns $[[..[a_{i_1}, a_{i_2}], ..], a_{i_n}]$.

$$R : (A \otimes B) \otimes C \vdash \alpha_r(R) : A \otimes (B \otimes C) \tag{8}$$

[2] $\mathcal{X}' : \Gamma'$ is a permutation of $\mathcal{X} : \Gamma$

where $\alpha_r(R)$ is an object which $prove()$ method computes $[[u,v],w] = R.prove()$ and returns $[u,[v,w]]$.

$$R : A \otimes (B \otimes C) \vdash \alpha_l(R) : (A \otimes B) \otimes C \tag{9}$$

where $\alpha_l(R)$ is an object which $prove()$ method computes $[u,[v,w]] = R.prove()$ and returns $[[u,v],w]$.

$$\frac{\mathcal{X} : \Gamma, X : A \vdash \mathcal{R} : \Delta, T : C}{\mathcal{X} : \Gamma, R : A \otimes B \vdash \mathcal{R} : \Delta, \mu_l(R, \lambda X \cdot T) : C \otimes B} \tag{10}$$

$\mu_l(R, \lambda X \cdot T)$ is an object which $prove()$ method computes $c = [R/X]T.prove()$. $[R/X]T$ is $T$ where $R$ has been substituted to $X$. During the computation of $c$, when $[a,b] = R.prove()$ is computed, only $a$ is returned. $b$ is saved to build the final result $[c,b]$. Here is the right-component form of this axiom:

$$\frac{\mathcal{X} : \Gamma, X : B \vdash \mathcal{R} : \Delta, T : D}{\mathcal{X} : \Gamma, R : A \otimes B \vdash \mathcal{R} : \Delta, \mu_r(R, \lambda X \cdot T) : A \otimes D} \tag{11}$$

$\mu_r(R, \lambda X \cdot T)$ is an object which $prove()$ method computes $d = [R/X]T.prove()$. During the computation of $d$, when $[a,b] = R.prove()$ is computed, only $b$ is returned. $a$ is saved for returning the final result which is $[a,d]$.

**The $\&$ conjunction.** $A \;\&\; B$, read it $A$ *and* $B$, means that we can choose between executing $A$ or $B$. Its realization is an object $R$ such that $R.prove().left()$ realizes $A$ and $R.prove().right()$ realizes$B$. If $A$ and $B$ are situations, $A \;\&\; B$ is close to the linear conjunction with the same name. Its axiomatization is trivial.

**The $\oplus$ disjunction.** $A \oplus B$, read it $A$ *plus* $B$, means executing either $A$ or $B$ without knowing *a priori* which one. Its realization is an object $R$ which method $prove()$ executes either $A$ or $B$ and returns a left-characterized result if $A$ has been executed and a right-characterized result if $B$ has been executed. The left or right characterization allows to know afterwards which formula has been executed. If $A$ and $B$ are situations, $A \oplus B$ is close to the linear conjunction with the same name. Its axiomatization is more restrictive than usual ones because of the imperative nature of the realizations. It is *not yet* considered as finalized.

**The implication $\Rightarrow$.** $A \Rightarrow B$, read it $A$ *implies* $B$, that we are able to build a realization of $B$ provided we have one for $A$. Its realization is an object $R$ such that $R.prove().imply(S)$ is a realization of $B$ when $S$ is a realization of $A$. When $A$ and $B$ are situations, $A \Rightarrow B$ is close to the linear implication. The rule for its introduction is:

$$\frac{\mathcal{X} : \Gamma, X : A \vdash \mathcal{R} : \Delta, T : B}{\mathcal{X} : \Gamma \vdash \mathcal{R} : \Delta, \iota(\lambda X \cdot T) : A \Rightarrow B} \text{ if } X \notin \mathcal{X} \tag{12}$$

where $\iota(\lambda X \cdot T)$ is an object which $prove()$ method returns the object itself and which $imply(R)$ method returns $[R/X]T$.

**N.B.** This is an example of the difficulties encountered in the axiomatization: it is mandatory that the program $\iota(\lambda X \cdot T).prove().imply(S).prove()$ to be executed in a precise manner. First, a thread (a light process) is run for computing $\iota(\lambda X \cdot T).prove()$. This thread computes $T$ but stops at the time of computing $X$ which, anyway, is just a realization variable. Then, when $imply(S)$ is computed, $S$ is substituted to $X$ in $T$ and the thread restarts at the next $prove()$ call. Under these conditions only, we can write the very useful axiom:

$$R : (A \Rightarrow B) \otimes A \vdash \iota_t(R) : B \tag{13}$$

where $\iota_t(R)$ computes $[u,v] = R.prove()$ and returns $u.imply(\zeta(v))$. $\zeta(v)$ is an object such that $\zeta(v).prove() = v$. For eliminating an implication, we use:

$$R : A \Rightarrow B, S : A \vdash R.prove().imply(S) : B \tag{14}$$

**The causality $\rightarrow$.** $A \rightarrow B$, read it $A$ *causes* $B$ means that when a particular $A$ is executed, $B$ will be executed too. Thus $A \rightarrow B$ appears as a deferred $B$ which execution is started by a particular $A$. To build a $A \rightarrow B$, we need at least one $A$ and a $B$. Let us mention that we can associate several different $A$'s to a single $A \rightarrow B$ formula, the first to be executed will make $B$ to be executed. Its axiomatization relies heavily on threads synchronization and is *not yet* considered as finalized.
**Independent sequentiality $\gg$.** $A \gg B$, read $A$ *then* $B$, means executing $A$ and then executing $B$. Both $A$ and $B$ are available after the execution. Its axiomatization is simple.
**Dependant sequentiality $>$.** $A > B$, read it $A$ *prefix* $B$, means executing $A$ first in order to execute $B$. Only $B$ is available after the execution. Its axiomatization is simple too.
**Universal quantification $\forall$.** $\forall x \cdot A$ means that we can do $[t/x]A$ for any chosen value $t$ but only for one value. We introduce it with:

$$\frac{\mathcal{X} : \Gamma \vdash \mathcal{R} : \Delta, T : A}{\mathcal{X} : \Gamma \vdash \mathcal{R} : \Delta, \kappa\,(\lambda x \cdot T) : \forall x \cdot A} \quad x \notin FV(\Gamma \cup \mathcal{R} : \Delta) \tag{15}$$

where $\kappa\,(\lambda x \cdot T)$ is an object $R$ such that $R.prove().forall(t)$ returns $(\lambda x \cdot T)(t)$. For elimination:

$$R : \forall x \cdot A \vdash R.prove().forall(t) : [t/x]A \tag{16}$$

**Existential quantification $\exists$.** $\exists x \cdot A$ means that we can do $[t/x]A$ for a particular value of $t$. Its axiomatization which relies on Heyting's intuitionistic semantic of proofs is *not complete* for syntactical reasons.
**The $!$ modality.** $!A$, read it *many* $A$, means that $A$ can be freely erased or duplicated. It is the case of Plato-Cantor eternal truth which proofs are resource-free. The realization of $!A$ is an object which $prove()$ method returns the object itself and it $extract()$ method returns a realization of $A$. The rules we will need are the following:

$$R : !A \vdash R : !A, R : !A \tag{17}$$

$$R : !A \vdash R.prove().extract() : A \tag{18}$$

$$R : !A \vdash \tag{19}$$

We also have to add specific rules such as the following which is relative to $\otimes$ and could be generalized:

$$R : !A \otimes B \vdash \pi_2\,(R) : B \tag{20}$$

where $\pi_2\,(R)\,.prove()$ method computes $[a, b] = R.prove()$ and returns $b$.
**N.B.** Of course, $\Omega$ logic handles the dual $?$ modality but it is rather useless in the current context.

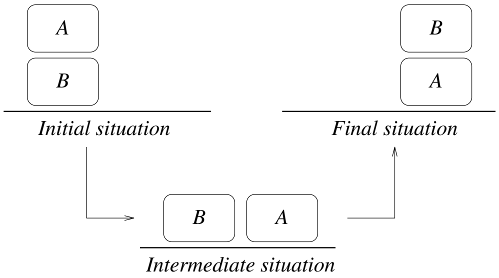
# 3 The realizations

The realizations have been implemented as objects in the JAVA language. The interest of JAVA is that it has immediate tools for creating and managing threads. It also has easily usable tools for building interfaces. And last but not least, its garbage collector is essential because of the intrinsic complexity of the execution of the realizations. It is roughly impossible to know globally what is going on even if each piece of code is very simple, thus it is also impossible to know when data are no longer needed.

Our implementation relies on a general type *Data* which is derived in a lot of subtypes: *NilData*, *PairData*, *IntegerData*, *StringData*, *Pi*, *Forall*, etc. One advantage of an object-oriented implementation is that it can be easily extended with other types of data, realizations or formulae.

# 4 Axiomatization of the world of blocks

The world of blocks [17, 18] is a floor on which there is blocks. Given two configurations of blocks called *initial* and *final*, the problem is to find the block movements needed to get the final situation from the initial one. An example is given on the figure below.




To model this problem, we introduce terms to represent the blocks and $floor$ to represent the floor. We also introduce two predicates. The first is $top(x)$, where $x$ is either $floor$ or a block, meaning that the top of $x$ is free. The second is $on(x, y)$, where $x$ is a block and $y$ is either $floor$ or a block, meaning that $x$ is on $y$. To simplify the proofs, we assume that the floor is large enough to support all the blocks. This is translated into the following axiom:

$$\vdash [\,] : top(floor) \tag{21}$$

where $[\,]$ is a data such that $[\,].prove() = [\,]$. The information $top(floor)$ is free, we can erase it:

$$R : A \otimes top(floor) \vdash \pi_1(R) : A \tag{22}$$

where $\pi_1(R).prove()$ computes $[u, v] = R.prove()$ and returns $u$.
**N.B.** Another difficulty of axiomatization: we remark that in the previous axiom for eliminating $top(floor)$ we are obliged to consider $top(floor)$ linked in a $\otimes$ conjunction. This is because we cannot simply drop the realization of $top(floor)$. Because it may do something and because of linearity, it has to be executed even if the formula, i.e. the information, disappears.

To move a block $x$ on $z$, $z$ being the floor or another block, we use the axiom:

$$R : top(x) \otimes on(x, y) \otimes top(z) \vdash Move(R, x, z) : top(x) \otimes on(x, z) \otimes top(y) \tag{23}$$

where $Move(R, x, z).prove()$ executes $R.prove()$ to get the pre-condition of the move. Then it effectively moves the block $x$ from $y$ on $z$. It returns $[[[\,], [\,]], [\,]]$ because the conclusion is made of three formulae linked by $\otimes$ and no significant result is needed.
**N.B.** Yet another difficulty of axiomatization: we remark the necessity to mention $y$, the initial support of $x$, even if $y$ is not mandatory for the movement. This comes from the linear management of informations: all modifications have to be mentioned and the fact that $top(y)$ after the move is one of them.

# 5 A first and easy theorem

Let us prove that any top block can be moved on the floor. The proof is presented as a simple linear proof:

| | | |
|---|---|---|
| | $X : top(x) \otimes on(x, y)$ | |
| $\vdash$ | $X : top(x) \otimes on(x, y), [\,] : top(floor)$ | axiom 21 ($top(floor)$ intro) |
| $\vdash$ | $\pi(X, [\,]) : top(x) \otimes on(x, y) \otimes top(floor)$ | axiom 5 ($\otimes$ intro) |
| $\vdash$ | $Move(\pi(X, [\,]), x, floor) : top(x) \otimes on(x, floor) \otimes top(y)$ | axiom 23 (bloc move) |

Using axiom 12 ($\Rightarrow$ intro) and axiom 15 ($\forall$ intro), we have:

$$\vdash F : \forall x \cdot \forall y \cdot top(x) \otimes on(x,y) \Rightarrow top(x) \otimes on(x, floor) \otimes top(y)$$

with: $F \equiv_{def} \kappa\,(\lambda x \cdot \kappa\,(\lambda y \cdot \iota\,(\lambda X \cdot Move(\pi(X,[\,]), x, floor))))$. We will see in section 8 how to use this realization.

# 6 An iteration axiom for stacks

If we want to work on stacks of blocks, we need an axiom allowing to iterate an action. Let us name a stack $x - y$ where $x$ is the top of stack and $y$ is the bottom block. Let us assume we want to prove a property $u(x,y)$ of stacks $x - y$. We would have to prove it for one-block stacks, i.e. something of the form:

$$\mathcal{X} : \Gamma \vdash A(x) : u(x,x) \tag{24}$$

**Syntax.** We choose to write $A(x)$ instead of $A$ in order to write $A(t)$ instead of $(\lambda x \cdot A)\,(t)$ or $[t/x]A$.

Then we prove an iteration step: assuming tha property is true for a stack $x - y$, we prove it for the stack $z - y$ where $on(z,x)$. This is a result of the form:

$$\mathcal{Y} : \Delta, R : u(x,y), T : on(z,x) \vdash B(R,T,z,x,y) : u(z,y) \tag{25}$$

Let us forget about $\Gamma$ and $\Delta$ which only dull the formalism. The conclusion is:

$$\mathcal{X} : \Gamma, \mathcal{Y} : !\Delta \vdash C(x,y) : u(x,y) \tag{26}$$

where $C(x,y).prove()$ begins by testing whether $x = y$. If true, it returns $A(x)$. Otherwise, it returns the object $B(C(under(x), y), [\,], x, under(x), y)$ where $under(x)$ is a program call which returns the block or the $floor$ under $x$.

# 7 Formalization of stacks

We introduce the predicate $stack(x,y)$ meaning that $x - y$ is a stack. First axiom tells that a block is a one-element stack:

$$\vdash [\,] : stack(x,x) \tag{27}$$

Second axiom states that a stack can be made of a block on a smaller stack:

$$R : stack(z,y) \otimes on(x,z) \vdash R : stack(x,y) \tag{28}$$

The information $on(x,z)$ is implicit in $stack(x,y)$ since $z$ is part of the stack. Conversely:

$$R : stack(x,y) \otimes on(x,z) \vdash R : stack(z,y) \otimes on(x,z) \tag{29}$$

Here, the information $on(x,z)$ must be preserved.

# 8 A more interesting theorem

If we want to move a stack $a - b$, we need $a - b$ to be effectively a stack, i.e. $stack(a,b)$. We need the top of $a - b$ to be free, i.e. $top(a)$. For linear management of informations, we consider $c$ such that $on(b,c)$. Finally, $d$ that will receive the stack must also be free, i.e. $top(d)$. Initial situation is: $stack(a,b) \otimes top(a) \otimes on(b,c) \otimes top(d)\cdot$ Final situation is: $stack(a,b) \otimes top(a) \otimes on(b,d) \otimes top(c)\cdot$ And the theorem to prove:

$$\vdash \forall a \cdot \forall b \cdot \forall c \cdot \forall d \cdot \left(\begin{array}{c} stack(a,b) \otimes top(a) \\ \otimes \\ on(b,c) \otimes top(d) \end{array}\right) \Rightarrow \left(\begin{array}{c} stack(a,b) \otimes top(a) \\ \otimes \\ on(b,d) \otimes top(c) \end{array}\right) \tag{30}$$

We prove it using iteration axiom on formula:

$$u(x,y) \equiv_{def} \forall c \cdot \forall d \cdot \left(\begin{array}{c} stack(a,b) \otimes top(a) \\ \otimes \\ on(b,c) \otimes top(d) \end{array}\right) \Rightarrow \left(\begin{array}{c} stack(a,b) \otimes top(a) \\ \otimes \\ on(b,d) \otimes top(c) \end{array}\right)$$

Then two applications of rule 15 ($\forall$ intro) allow to conclude. The proof is very long since we have to manage all details including commutativity et associativity. But it is not so difficult. We get the following realization, $U_5$, from the proof. It is inherently unreadable but it works perfectly.

$$U_5 \equiv_{def} \kappa\,(\lambda x \cdot \kappa\,(\lambda y \cdot C(x,y)))$$

$$C(x,y).prove() \equiv_{def} x = y \longrightarrow A(x)\ ;\ B(C(under(x),y),[\,],x,y)$$

$$B(R,T,z,y) \equiv_{def} \kappa\,(\lambda c \cdot \kappa\,(\lambda d \cdot \iota\,(\lambda X \cdot U_4(R,X,T,z,c,d,x))))$$

$$U_4(R,X,T,z,c,d,x) \equiv_{def} \pi_s \left(\{1,3,4,5\}, \pi_1 \left(\pi_p \left(\begin{array}{l} \{4,2,1,6,3,5\} \\ U_3(R,X,T,z,c,d,x) \end{array}\right)\right)\right)$$

$$U_3(R,X,T,z,c,d,x) \equiv_{def} \mu_l \left(3, \pi_p \left(\begin{array}{l} \{5,6,2,1,3,4\} \\ U_2(R,X,T,z,c,d) \end{array}\right), \lambda S \cdot Move(S,z,x)\right)$$

$$U_2(R,X,T,z,c,d) \equiv_{def} R.prove().forall(c).prove().forall(d).prove().imply(U_1(X,T,z))$$

$$U_1(X,T,z) \equiv_{def} \pi_p \left(\{6,3,4,5,1,2\}, \mu_l \left(\begin{array}{l} 3 \\ \pi_p\,(\{2,5,3,4,1\}, \pi\,(X,T)) \\ \lambda R \cdot Move(\pi\,(R,[\,])\,, z, floor) \end{array}\right)\right)$$

$$\text{A(x)} \equiv_{def} \kappa \left(\lambda c \cdot \kappa \left(\lambda d \cdot \iota \left(\lambda X \cdot \pi_p \left(\begin{array}{l} \{4,1,2,3\} \\ \mu_l \left(\pi_p \left(\begin{array}{l} \{2,3,4,1\} \\ X \end{array}\right), \lambda R \cdot Move(R,x,d)\right) \end{array}\right)\right)\right)\right)$$

**Simplifications.** Brute realizations can oftenly be transformed in simpler realizations doing the same work. A trivial example is the simplification of $\pi_c\,(\pi\,(A,B))$ in $\pi\,(B,A)$. Numerous purely syntactic simplification rules are given in [1].

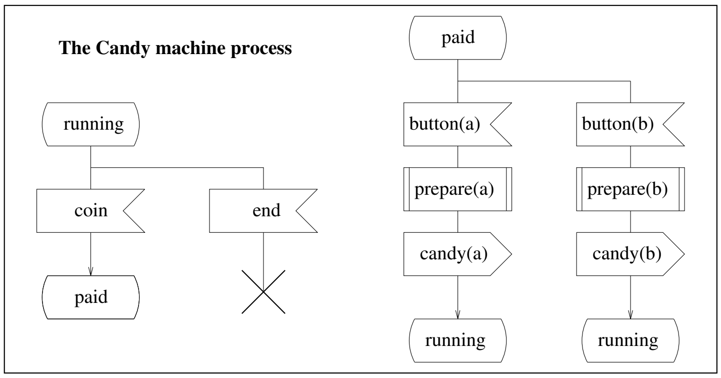
**Using the realization.** To use this realization, we need two blocks $a$ and $b$ such that $a - b$ is a stack. We need $c$, floor or block, on which $b$ is. We need $d$, floor or block, such that $top(d)$. And we need a realization $R$ of $stack(a,b) \otimes top(a) \otimes on(b,c) \otimes top(d)$. And we have to execute:

$$U_5.prove().forall(a).prove().forall(b).prove().forall(c).prove().imply(R).prove()$$

This program runs well and does what it is intended to. However, it is quite impossible to follow dynamically its execution. Note that the "linear" information $top(a) \otimes top(d)$ ensures that $a \neq d$.

# 9 The Candy machine

The Candy machine is a machine where a user can put a coin and then push either on button $a$ or on button $b$. In the first case, he receives a candy $a$. In the second case, he receives a candy $b$. Such a communicating process is specified as a protocol written in *SDL*, *Specification Description Language* [24], which graphical syntax is shown in the figure below. The machine can be in two states: *running* and *paid*. Initially in the *running* state, it is waiting for an input message *coin* or a message *end*. If it receives *coin*, it goes in state *paid*. If it receives messages *end*, it stops. When in state *paid*, it waits for an input message *button(x)* where *x* is either *a* or *b*, then it sends the output message *candy(x)* and returns to state *running*. Such SDL diagrams can be obviously and mechanically translated in $\Omega$ logic formulae.




For instance, the first one is translated by introducing predicates $running$ and $paid$ meaning that the machine is in the mentioned state, by introducing constants and functions to represent the messages and by introducing the predicates $in(x)$ and $out(x)$ meaning that the machine respectively receives or sends the message $x$. Then assuming we have realizations for the atomic actions, we can prove the translated specification and get the appropriate program for the machine. This example uses the causality connective ($\rightarrow$) because the input message causes an action.

# 10 Conclusions

**About axiomatization.** Linearity of axioms and rules is very difficult to obtain for extra-logical axioms. It is not so easy to detect that an axiom duplicates or forgets an information.
**About proofs.** As in any formal system, proofs are very delicate to write down. All normally implicit details must be explicited. Programs obviously depends on the proof. A bad proof gives a bad program either in memory complexity or in computational complexity. In the iteration axiom, we considered a $(n + 1)$-blocks stacks as being a top block laid on a $n$-blocks stack. If, instead, we consider it to be a $n$-blocks stack upon one bottom block, we get a very different program: to move a 10-blocks stacks, it needs about 8000 threads and 60000 objects. Yes, you read well! But this is also true for other similar systems [20].
**About the theory.** It seems very difficult to prove the soundness of this approach either by a semantic method or by a syntactic method. At the present time, it is a serious drawback.
**About execution.** Each piece of code is small, simple and have a clear semantics. However, the global execution of a synthesized program is intractable. As far as correctness with regard to specification is the only concern, this is obviously sufficient. But if we want to deal with efficiency, it is a real problem.
**About linearity.** We choose a linear form of logic for almost obvious reason: assume we use a situation $A$ for producing a situation $B$, if you can duplicate formulae then you may

reach a proof-state where both $A$ and $B$ are true even if this is impossible. For instance, $A$ could be $top(x)$ and $B$ could be $on(y, x)$ after a move of $y$ on $x$. Thus, even if linearity is difficult to handle in practical cases, it appears as a mandatory feature of an action-based system.

# 11 Relations with other works

**Functions and actions.** Our system is clearly a proofs-as-programs one: we build the proof and we extract the program from it. There exists a lot of such well-known systems for functional computations. However, it appears, oftenly but not always, that the specifications of functional programs are executable as PROLOG programs provided they are written with the operational view of PROLOG in head. In our approach, the action is the base rock and we don't think it can easily be synthesized as logic programs. Moreover, we could ask the question whether there is a real place for the application of such systems to functions, except for scientific purpose.
**Transformations and proofs-as-programs.** As in most *transformational synthesis* or *proofs-as-programs* systems, the main stumbling rock is the proof and the way it can be obtained. The proof determines the tractability of the program. In transformational systems like LOPS [8], proofs are transformations of specifications into executable logic programs. In our system, programs are extracted from proofs of the specifications. We don't think there is a great difference between the two approaches: they seem to be the same work under different presentations. The main difference lies in what has to be proved in order to get an executable program. We could claim that our approach is better because it deals only with logic formalism and not with logic-like programs, i.e. some form of Horn clauses. However, this would be unfair because the way the theorem is proved is closely linked to the program as we already mentioned it.
**Programming or proving ?** In [10], Tessler theorem is quoted "Automatic programming is whatever has not been compiled yet !". This appears as definitively true in the case of proofs-as-programs systems: proving is just another form of programming. For instance, choosing an induction rule or another means choosing one kind of iteration or another. We think that this is true in all other proofs-based systems. One may easily prefer to program ! However, proofs can be semi-automatically obtained and can be automatically checked, programs cannot. And programs are automatically extracted from proofs. The automatisation of the process lies in an automatic theorem prover and this is impossible in the general case.

# 12 Perspectives

**Proof assistant.** We have developed a graphical formulae editor with a readable LATEX output. We are currently developing a proof assistant which will also extract realizations from proofs. This application is written in JAVA. Thus it will be portable on almost all platforms.
**Proving correctness of specifications.** Let us assume that we have the logical translation $F_{spec}$ of SDL diagram such as in section 9. We can also have a logical specification $U_{spec}$ of the other actor, for instance the user in the case of the Candy machine. By transposing the user action , with the $\perp$ operation [1], and by studying $F_{spec} \otimes U_{spec}^{\perp}$, we can hope to prove the correctness of the specifications. Transposition is syntactically defined [1] but even if it works for the linear menu, it is not clear that it will work for the Candy Machine for instance.
**Other example.** Hanoï Towers is an example which has to be developed in order to finalize $\oplus$ and $\exists$ axiomatizations. A sorting-in-place algorithm also needs to be developed.

# 13 Axioms and rules

In this section, we give a subset of axioms and inference rules. As said in section 2, the axiomatization is not complete.

**Notations.** $\Gamma'$ is a permutation $\Gamma$. $!\Gamma$ is the sequence made of the formulae of $\Gamma$ modalized with $!$.

$$\Gamma \vdash \Gamma' \qquad \frac{\Gamma \vdash \Delta \quad \Delta \vdash \Theta}{\Gamma \vdash \Theta} \qquad \frac{\Gamma \vdash \Delta}{\Gamma, \Theta \vdash \Delta, \Theta}$$

$$A, B \vdash A \otimes B \qquad A \otimes (B \otimes C) \vdash (A \otimes B) \otimes C \qquad \mathbf{1} \otimes B \vdash B \qquad \frac{\Gamma, A \vdash C, \Delta}{\Gamma, A \otimes B \vdash C \otimes B, \Delta}$$

$$A \otimes B \vdash B \otimes A \qquad (A \otimes B) \otimes C \vdash A \otimes (B \otimes C) \qquad A \otimes \mathbf{1} \vdash A \qquad \frac{\Gamma, B \vdash D, \Delta}{\Gamma, A \otimes B \vdash A \otimes D, \Delta}$$

$$A \mathbin{\&} B \vdash A \qquad A \mathbin{\&} B \vdash B \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \mathbin{\&} B}$$

$$A \vdash A \oplus B \qquad B \vdash A \oplus B \qquad \frac{\Gamma, A \vdash A > C \quad \Gamma, B \vdash B > C}{\Gamma, A \oplus B \vdash (A \oplus B) > C}$$

$$A, B \vdash A \gg B \qquad A \gg (B \gg C) \vdash (A \gg B) \gg C \qquad \mathbf{1} \gg B \vdash B$$

$$(A \gg B) \gg C \vdash A \gg (B \gg C) \qquad A \gg \mathbf{1} \vdash A \qquad \frac{\Gamma, B \vdash D, \Delta}{\Gamma, A \gg B \vdash A \gg D, \Delta}$$

$$A > B \vdash B \qquad A \vdash A > A \qquad (A > B) \gg C \vdash A > (B \gg C) \qquad A > (B \gg C) \vdash (A > B) \gg C$$

$$A, A \Rightarrow B \vdash B \qquad A \otimes (A \Rightarrow B) \vdash B \qquad (A \Rightarrow B) \gg A \vdash B \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}$$

$$\frac{\Gamma \vdash A, \Delta}{\Gamma \vdash \forall x \cdot A, \Delta} \quad x \notin FV(\Gamma, \Delta) \qquad \forall x \cdot A \vdash [t/x]A$$

$$\frac{\Gamma \vdash A}{!\Gamma \vdash !A} \qquad !A \vdash !A, !A \qquad !A \vdash A \qquad !A \vdash \mathbf{1}$$

# References


[1] BELLOT, P. Logics, languages and specifications, the science of information processing. In *accessible par* `http://www.enst.fr/~bellot` (ENST Paris (France), 1997 (to be continued)).

[2] BELLOT, P., COTTIN, J.-P., DEMAILLE, A., LENEUTRE, J., AND ZARPAS, E. A logic of actions for imperative program synthesis. In *Logic Colloquium'96, LC'96* (San Sebastian (Spain), july 1996), J. Larrazabal, Ed.

[3] BELLOT, P., COTTIN, J.-P., DEMAILLE, A., LENEUTRE, J., AND ZARPAS, E. Towards bug free software engineering for telecommunication. In *proceeding of The Eighth International Conference (Joint Europe-USA Conference) on Artificial Intelligence Applications, Expersys'96* (Paris (France), october 1996).

[4] BELLOT, P., COTTIN, J.-P., LENEUTRE, J., AND ZARPAS, E. Automated deduction for protocols design. In *Third Workshop on Automated Reasoning: Bridging the Gap between Theory and Practice* (Brighton (England), april 1996), I. Gent, Ed.

[5] BELLOT, P., LENEUTRE, J., ROBINET, B., AND ZARPAS, E. Prolegomena of a logic of causality and dynamicity. *Studia Logica, an International journal for Symbolic Logic*, 61 (1998), 393–421.

[6] BELLOT, P., LENEUTRE, J., AND ZARPAS, E. The B-logic, from the Big Bang to the Big Crunch. In *Logic Colloquium'94, LC'94* (Clermont-Ferrand (France), july 1994), D. Richard, Ed.

[7] BERGER, U. Program extraction from normalization proofs. In *TCLA'93, Typed Lambda Calculi and Applications* (Berlin, 1993), vol. 664 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 91–106.

[8] BIBEL, W., AND HORNING, K. Lops : a system based on a strategical approach to program synthesis. In *in [9]* (1984).

[9] BIERMANN, A., GUIHO, G., AND KODRATOFF, Y. *An Overview of Automatic Program Construction Techniques*. Macmillan, Boston/Dordrecht/London, 1984.

[10] FLENER, P. *Logic Program Synthesis from Incomplete Information*. Kluwer Academic Publishers, Boston/Dordrecht/London, 1995.

[11] GIRARD, J.-Y. Linear logic. *Theoretical Computer Science 50*, 1 (1987), 1–102.

[12] GIRARD, J.-Y. Introduction à la logique linéaire. In *Logique et Informatique : une introduction*, B. Courcelle, Ed. INRIA, 1989.

[13] GIRARD, J.-Y. Towards a geometry of interaction. *Contemporary Mathematics 92* (1989), 69–108.

[14] HOWE, D. Reasoning about functional programs in nuprl. In *Functional Programming, Concurrency, Simulation, and Automated Reasoning* (1991), vol. 693 of *Lecture Notes in Computer Science*, Springer-verlag, pp. 145–164.

[15] KLEENE, S. Realizability. In *Colloquium on Constructivity in Mathematics, Amsterdam, 1957* (1959), North-Holland, Amsterdam.

[16] KRIVINE, J.-L. *Lambda-calcul types et modèles*. Masson, Paris (France), 1990.

[17] MASSERON, M., TOLLU, C., AND VAUZEILLES, J. Generating plans in linear logic, part I: Actions as proofs. *Theoretical Computer Science 113*, 2 (1993), 349–370.

[18] MASSERON, M., TOLLU, C., AND VAUZEILLES, J. Generating plans in linear logic, part II: A geometry of conjunctive actions. *Theoretical Computer Science 113*, 2 (1993), 371–376.

[19] NEPEJVODA, N. A bridge between constructive logic and computer programming. *Theoretical Computer Science 90* (1991), 253–270.

[20] PARIGOT, M. Programming with proofs: a second order type theory. In *ESOP'88, Second European Symposium on programming* (Nancy, 1988), vol. 300 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 145–159.

[21] PARIGOT, M. Classical proofs as programs. In *KGC'93, Third K. Godel Colloquium, Computational logic and Proof theory, Brno* (1993).

[22] PAULIN-MOHRING, C. Réalisabilité et extraction de programmes. In *Logique et informatique : une introduction*, B. Courcelle, Ed. INRIA, 1989, pp. 163–180.

[23] TATSUTA, M. Program synthesis using realizability. *Theoretical Computer Science 90* (1991), 309–353.

[24] TURNER, K. *Using Fromal Description Techniques - An Introduction to Estelle, Lotos and SDL.* Wiley Series in Communication and Distributed Systems, 1993.

[25] ZARPAS, E. Démonstration automatique dans une logique des actions (thesis). Tech. rep., ENST, Paris (France), 1997.