

Disjunctive Tautologies as Synchronisation Schemes

Vincent Danos and Jean-Louis Krivine

Université Paris 7
Équipe Preuves, Programmes, Systèmes
2 place Jussieu, 75251 Paris Cedex 05, France

Abstract. In the ambient logic of classical second order propositional calculus, we solve the specification problem for a family of excluded middle like tautologies. These are shown to be realized by sequential simulations of specific communication schemes for which they provide a safe typing mechanism.

1 Introduction

Since the inception of the proof/program correspondence with the Curry-Howard isomorphism, one of the goals of proof-theory has been the interpretation of logical rules as programming instructions. Only recently has this correspondence been extended to *classical logic*, which is now explained as a typing system for a λ -calculus augmented with a ‘call-with-current-continuation’ primitive, or some similar form of control such as an ‘exception handler’. This extremely interesting explanation, first promoted by Griffin [3] and Felleisen, now admits many variants, as well as neat proof-theoretic renderings by Parigot [6] and Girard [2], for instance. All these variants are given in terms of *sequential languages*, though.

In this paper, we significantly depart from this tradition by interpreting a family of classical formulas, namely pure *disjunctions of literals*, as specifying *synchronisation protocols*. We contend, more generally, that this paradigmatic shift towards a concurrent reading of classical logic, gives rise to illuminating behavioral explanations.

The programming language we’ll be using to make our point is a concurrent extension of a variant of Felleisen’s λC -calculus, so that the world of programs that formulas will be referring to will indeed be a world of concurrent processes. The ambient logic, *i.e.*, the means of expressing behavioural specifications, will be classical second order propositional

logic, equipped with only the following logical operators: \rightarrow and \forall . More comprehensive frames, such as second order predicate calculus or even Zermelo-Fraenkel set theory, are amenable to the same treatment, as shown in [4], but this simple logic suffices for our present purposes. Finally, the main and only tool we shall use is a revised version of the second author's *realizability* method, which itself is an adaptation of the Tait-Girard reducibility method.

The gist of the interpretation is best understood with a simple example. Let's consider the formula $G_1 = \forall R \forall S [(R \rightarrow S) \vee (S \rightarrow R)]$. For one thing, we may recode implication as a disjunction and get a classically equivalent form of G_1 , which is a pure disjunction of literals, namely $G_0 = \forall R \forall S [(\neg R \vee S) \vee (\neg S \vee R)]$. This last form is obviously true, so that G_1 itself is a tautology.

Conversely, the standard second order recoding of disjunction, yields a purely implicative formula, which is an intuitionistic equivalent of G_1 , namely $G = \forall R \forall S \forall X [((R \rightarrow S) \rightarrow X) \rightarrow ((S \rightarrow R) \rightarrow X) \rightarrow X]$. This last formula can as well be reformulated as a rule:

$$G \frac{\Gamma, R \rightarrow S \vdash X \quad \Gamma, S \rightarrow R \vdash X}{\Gamma \vdash X}$$

The operational explanation developed in this paper for such rules is as follows. Suppose we have two processes $P_1[k_1]$, $P_2[k_2]$, both of type X , with free variables, or channels, k_1 , k_2 , of respective types $R \rightarrow S$ and $S \rightarrow R$. By the rule above we can build a compound process of type X , say $[P_1 \mid P_2]$, within which k_1 , k_2 are bound. Close examination with the realizability tool shows that the computational behavior prescribed by G is the following: $[P_1 \mid P_2]$ starts running both P_1 and P_2 concurrently; if both get locked in requesting values for their free variables, k_1 and k_2 , their states can then be described as $(k_1)v_R\pi_S$ and $(k_2)v_S\pi_R$, where v_R , v_S are some terms and π_R , π_S are some sequences, or stacks of terms; $[P_1 \mid P_2]$ then resumes the computation by running $(v_S)\pi_S$ and $(v_R)\pi_R$ concurrently.

That G is a tautology amounts to saying, according to our interpretation, that this cross communication scheme is well-typed, in that if, as expected, v_R , v_S are of respective types R and S , and if π_R , π_S are providing suitable environments in which to evaluate any term of respective types R and S , then both $(v_S)\pi_S$ and $(v_R)\pi_R$ will interact correctly.

This paper gives theoretical support to such concurrent computational explanations. A detailed construction of the framework, language, types

and the realizability tool is given first. We then exercise that tool to extract concurrent behavioral specifications from a few tautologies, such as G , which is the simplest interesting example. Eventually, we home in on a quite general result explaining the family of *purely disjunctive* tautologies as synchronisation protocols.

2 Terms, Types and Models

In this preliminary section we first define our language, and the logic which types it, and then set up the suitable notion of realizability interpretation.

2.1 The Programming Language

We first define the set Λ of *terms*, denoted t , and the set Π of *stacks*, denoted π , with the following grammar:

$$\begin{aligned} t &= x, (t)t, t \mid t, \lambda x.t, \kappa x.t, *_t, *_\pi \\ \pi &= \epsilon, t \cdot \pi \end{aligned}$$

We then define *executables* as finite multisets on $\Lambda \times \Pi$, still denoting \mid the multiset constructor. By definition this constructor is commutative and associative. Those executables are equipped with an *evaluation* relation, written \succ , and defined as the smallest preorder on the set of executables which is compatible with \mid and such that:

$$\begin{aligned} (t)u, \pi &\succ t, u \cdot \pi && \text{(PUSH)} \\ t \mid u, \pi &\succ t, \pi \mid u, \pi && \text{(DIST)} \\ (\lambda x.t), u \cdot \pi &\succ t[*_u/x], \pi && \text{(L-STORE)} \\ *_u, \pi &\succ u, \pi && \text{(L-LOAD)} \\ (\kappa x.t), \pi &\succ t[*_\pi/x], \pi && \text{(K-STORE)} \\ *_\pi, t \cdot \pi' &\succ t, \pi && \text{(K-LOAD)} \end{aligned}$$

Note the analogy between the two binders, λx and κx . The first takes a snapshot, denoted $*_u$, of u , the current top element of the stack, stores it in x and pops the stack, while the second is taking a snapshot, denoted $*_\pi$, of the *whole* stack π , stores it in x as well, and leaves the stack intact. When $*_u$ comes in head position, it simply loads its value u , while $*_\pi$ throws the top element of the stack to its value.

The usual *cc* construction can be recovered as $\lambda h.\kappa k.(h)k$, one interest of our variant formulation being that the analogy just noted is made more obvious.

As an example, set $\delta = \kappa x.x$, then for any π , we get that loop:

$$(\delta)\delta, \pi \succ \delta, \delta \cdot \pi \succ *_{\delta.\pi}, \delta \cdot \pi \succ \delta, \delta \cdot \pi.$$

2.2 The Typing System

Formulas or types, denoted A, B, \dots , are here second order propositional formulas. Typing judgements of the form $x_1 : A_1, \dots, x_n : A_n \vdash t : B$, where t is a term and A_1, \dots, A_n, B are formulas, are generated by the following rules:

$$\begin{array}{c} \text{ax} \frac{}{\Gamma, x : A \vdash x : A} \text{var} \\ \text{abs} \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \rightarrow B} \\ \text{app} \frac{\Gamma \vdash t : A \rightarrow B \quad \Gamma \vdash u : A}{\Gamma \vdash (t)u : B} \\ \text{vi} \frac{\Gamma \vdash t : A}{\Gamma \vdash t : \forall X A} \\ \text{ve} \frac{\Gamma \vdash t : \forall X A}{\Gamma \vdash t : A[B/X]} \\ \text{peirce} \frac{\Gamma, x : A \rightarrow B \vdash t : A}{\Gamma \vdash \kappa x.t : A} \text{cc} \\ \text{mix} \frac{\Gamma \vdash t : A \quad \Gamma \vdash u : A}{\Gamma \vdash t | u : A} \text{par} \end{array}$$

The quantification introduction rule, $\forall i$, is subject to the constraint that X is not free in the context Γ . The first five rules give a standard presentation, known as *natural deduction*, for second order propositional intuitionistic logic. Alongside with the sixth rule, known as *Peirce's law*, we get one possible natural deduction presentation of second order propositional classical logic. The last rule, or the *mix* rule, is just there to add expressive power on the terms side.

2.3 Truth Values and Models

Let \perp be a given set of executables, which, we assume throughout the paper, is closed by \succ^{-1} and $|$. That is to say: 1) if $e \in \perp$ and $e' \succ e$, then $e' \in \perp$, and 2) if $e \in \perp$ and $e' \in \perp$, then $e | e' \in \perp$.

For any set of stacks \mathcal{Z} , set $\mathcal{Z} \rightarrow \perp$ to be the largest set of terms \mathcal{X} such that $\mathcal{X} \times \mathcal{Z} \subset \perp$. Any such set of terms, which can be written as $\mathcal{Z} \rightarrow \perp$ for some set of stacks \mathcal{Z} , will be said to be a *truth value*. Two particular

truth values are of special interest, the largest one $A = \emptyset \rightarrow \perp$, and the smallest one, denoted $\perp = \Pi \rightarrow \perp$. For any $t, \pi \in \perp$, $(*_\pi)t \in \perp$, so \perp is empty iff \perp is.

Given a choice of \perp , we can extend any map $|\cdot|_0^- : Var \rightarrow 2^\Pi$, from propositional variables to 2^Π , to a map $|\cdot|^- : Form(2^\Pi) \rightarrow 2^\Pi$, from formulas with parameters in 2^Π to 2^Π , as follows:

$$\begin{aligned} |\mathcal{Z}|^- &= \mathcal{Z} \\ |X|^- &= |X|_0^- \\ |A \rightarrow B|^- &= (|A|^- \rightarrow \perp) \cdot |B|^- \\ |\forall X A|^- &= \cup_{\mathcal{Z}} |A[\mathcal{Z}/X]|^- \end{aligned}$$

In the last clause, the union is meant to range over all subsets \mathcal{Z} of Π . For instance, we get $|\forall X X|^- = \cup |\mathcal{Z}|^- = \cup \mathcal{Z} = \Pi$, so that $|\forall X X| = \perp$.

We then define the dual map $|\cdot| : Form(2^\Pi) \rightarrow 2^A$ simply by putting:

$$|F| = |F|^- \rightarrow \perp,$$

so $|F|$ is always a truth value. Moreover, when F is closed, its value, $|F|$, only depends on the choice of \perp . Such valuations of classical formulas can be factorized through a ‘not-not’ translation to intuitionistic formulas.

When $\perp = \emptyset$, it is easily seen that $|\cdot|$ can only take two values, namely \emptyset and A , and that, for any closed formula F , $|F| = A$ iff F is valid. In this special case the model collapses down to the usual notion of two-valued model. The generalization of this fact to any choice of \perp is known as the *adequacy* property:

Proposition 1 *Let $x_1 : A_1, \dots, x_n : A_n \vdash t : B$ be derivable, \perp be any set of multisets on $\Lambda \times \Pi$ closed by \succ^{-1} and $|\cdot|$, and $|\cdot|_0^-$ be any map from propositional variables to 2^Π , then, for all $v_1 \in |A_1|, \dots, v_n \in |A_n|$, and for all $\pi \in |B|^-$, $t[v_1/x_1, \dots, v_n/x_n], \pi \in \perp$.*

Proof. The proof is by induction on the typing derivation of t . To ease the reading of the proof, we’ll simply write A and A^- for $|A|$ and $|A|^-$, and $t[v_i/x_i]$ for $t[v_1/x_1, \dots, v_n/x_n]$. We also skip the axiom and quantifier rules, which are trivial, and use no assumption on \perp .

1. Application: $\frac{\Gamma \vdash t : A \rightarrow B \quad \Gamma \vdash u : A}{\Gamma \vdash (t)u : B}$. Pick $v_i \in A_i$ and $\pi \in B^-$.

By induction $t[v_i/x_i] \in A \rightarrow B = A \cdot B^- \rightarrow \perp$ and $u[v_i/x_i] \in A$, so that $t[v_i/x_i], u[v_i/x_i] \cdot \pi \in \perp$. But:

$$(t)u[v_i/x_i], \pi = (t[v_i/x_i])u[v_i/x_i], \pi \succ t[v_i/x_i], u[v_i/x_i] \cdot \pi,$$

$\perp\!\!\!\perp$ being closed by (PUSH), we deduce $(t)u[v_i/x_i], \pi \in \perp\!\!\!\perp$, qed.

2. Abstraction: $\frac{\Gamma, x:A \vdash t:B}{\Gamma \vdash \lambda x.t:A \rightarrow B}$. Pick $v \in A$, $v_i \in A_i$, $\pi \in A^-$ and $\pi' \in B^-$.

We have $*_v, \pi \succ v, \pi$, $\perp\!\!\!\perp$ being closed by (L-LOAD), we deduce $*_v \in A$. So, by induction, $t[v_i/x_i][*_v/x], \pi' \in \perp\!\!\!\perp$. But:

$$(\lambda x.t)[v_i/x_i], v \cdot \pi' = (\lambda x.t[v_i/x_i]), v \cdot \pi' \succ t[v_i/x_i][*_v/x], \pi',$$

$\perp\!\!\!\perp$ being closed by (L-STORE), we deduce $(\lambda x.t)[v_i/x_i], v \cdot \pi' \in \perp\!\!\!\perp$, qed.

3. Peirce: $\frac{\Gamma, x:A \rightarrow B \vdash t:A}{\Gamma \vdash \kappa x.t:A}$. Pick $v \in A$, $v_i \in A_i$, $\pi \in A^-$ and $\pi' \in B^-$.

We have $*_\pi, v \cdot \pi' \succ v, \pi$, $\perp\!\!\!\perp$ being closed by (K-LOAD), we deduce $*_\pi \in A \rightarrow B$. So, by induction, $t[v_i/x_i][*_\pi/x], \pi \in \perp\!\!\!\perp$. But:

$$(\kappa x.t)[v_i/x_i], \pi = (\kappa x.t[v_i/x_i]), \pi \succ t[v_i/x_i][*_\pi/x], \pi,$$

$\perp\!\!\!\perp$ being closed by (K-STORE), we deduce $(\kappa x.t)[v_i/x_i], \pi \in \perp\!\!\!\perp$, qed.

4. Mix: $\frac{\Gamma \vdash t:A \quad \Gamma \vdash t':A}{\Gamma \vdash t|t':A}$.

Pick $v_i \in A_i$ and $\pi \in A^-$. By induction, $t[v_i/x_i], \pi$ and $t'[v_i/x_i], \pi \in \perp\!\!\!\perp$, so, $\perp\!\!\!\perp$ being closed by $|$, we get $t[v_i/x_i], \pi | t'[v_i/x_i], \pi \in \perp\!\!\!\perp$. But:

$$(t|t')[v_i/x_i], \pi = t[v_i/x_i] | t'[v_i/x_i], \pi \succ t[v_i/x_i], \pi | t'[v_i/x_i], \pi,$$

$\perp\!\!\!\perp$ being closed by (DIST), we deduce $(t|t')[v_i/x_i], \pi \in \perp\!\!\!\perp$. □

The proof is perfectly modular, each listed case calling on its own closure conditions on $\perp\!\!\!\perp$ which we have carefully recorded. Indeed, this proof, in some sense, describes a set of computation rules which are compatible with the typing system.

3 The Specification Problem

We can now put to good use this adequacy result. Careful choices of $\perp\!\!\!\perp$ result in proving normalizability results, which has been the traditional application of adequacy. Here, we'll be using it in a somewhat different way, to solve the so-called *specification* problem. That is, given an F , is there any computational behavior which all terms of type F have in common ?

3.1 Booleans

Let's start with a very simple example: $B = \forall X [X \rightarrow (X \rightarrow X)]$.

From now on, as in the adequacy proof, we'll simply write A and A^- in place of $|A|$ and $|A|^-$. The particular choices we'll make for \perp will always be closures by \succ^{-1} and $|$ of a given generating set of executables.

Proposition 2 *Let $\vdash t : B$ be derivable, then for all terms a, b and for all stack π , tab, π evaluates into a multiset on $\{(a, \pi), (b, \pi)\}$.*

Proof. Indeed, let a, b be terms and π be a stack. Take \perp to be the closure of $\{(a, \pi), (b, \pi)\}$ and set $X^- = \{\pi\}$. Then, a and $b \in X$, hence, by adequacy, $t, a \cdot b \cdot \pi \in \perp$, so that by (PUSH), $tab, \pi \in \perp$. \square

For instance $t = \lambda x. \lambda y. x \mid \lambda x. \lambda y. y : B$ does behave in this way, since $tab, \pi \succ a, \pi \mid b, \pi$.

One can refine a specification. Though we didn't develop the formal material pertaining to predicate calculus, the following should be pretty self-explanatory.

Consider a language \mathcal{L} with two individual constants, 0 and 1, set:

$$Bx = \forall X [X0 \rightarrow (X1 \rightarrow Xx)],$$

and suppose $\vdash t : B0$. Take then \perp to be the closure of $\{(a, \pi)\}$ and set $X0^- = \{\pi\}$ and $X1 = A$. Clearly $a \in X0$ and $b \in X1$, so that $t, a \cdot b \cdot \pi \in \perp$, and hence, by (PUSH), $tab, \pi \in \perp$. So we get:

Proposition 3 *Let $\vdash t : B0$ be derivable, then for all terms a, b and for all stack π : tab, π evaluates into a multiset on $\{(a, \pi)\}$.*

Informally, this last proposition says that any $t : B0$ behaves as a certain number of $\lambda x. \lambda y. x$ running concurrently. A property which one might think of as computational consistency for our system.

To recap, adequacy gives a means of decoding the behavior specified by a given formula with respect to a given language. This here is the main thrust. Conversely, one can use it to refute typability. For instance, the proposition above shows in particular that $t = \lambda x. \lambda y. x \mid \lambda x. \lambda y. y$ can't be of type $B0$, nor can be any term with the same behaviour.

3.2 The Excluded Middle

Set $T = \forall X \forall R \forall S [((R \rightarrow S) \rightarrow X) \rightarrow ((R \rightarrow X) \rightarrow X)]$, which is the usual implicative coding of the *excluded middle*, $T_1 = \forall R \forall S [(R \rightarrow S) \vee R]$.

We want to show:

Proposition 4 *Let $\vdash c : T$ be derivable, then for all terms ρ, σ, r and for all stacks π, π_R and π_S , such that for all terms a and b , $\rho a, \pi \succ a, r \cdot \pi_S$ and $\sigma b, \pi \succ b, \pi_R, c\rho\sigma, \pi$ evaluates into a multiset on $\{(r, \pi_R)\}$.*

Proof. Let $\rho, \sigma, r, \pi, \pi_R$ and π_S be as above. Take for \perp the closure of $\{(r, \pi_R)\}$, and set $R^- = \{\pi_R\}$, $S^- = \{\pi_S\}$ and $X^- = \{\pi\}$.

First, we have $r \in R$. Let now $a \in R \rightarrow S$, by definition $a, r \cdot \pi_S \in \perp$, so by \perp being closed: $\rho, a \cdot \pi \in \perp$, hence $\rho \in (R \rightarrow S) \rightarrow X$. Likewise, if $b \in R$, by definition, $b, \pi_R \in \perp$, so by \perp being closed: $\sigma, b \cdot \pi \in \perp$, hence $\sigma \in R \rightarrow X$. By adequacy we get: $c, \rho \cdot \sigma \cdot \pi \in \perp$. \square

This specification can be rendered informally as follows: c launches two executables, or two independent processes, ρ and σ , passing over to each a variable, a and b respectively, and a same context π ; if both processes stop on these variables, c sends ρ 's top stack element to σ , via b , so that σ runs again, while ρ dies.

Let's run an example. We set $((c)\rho)\sigma = \kappa k.(\sigma)\kappa h.(k)(\rho)h$, and by typing $k : X \rightarrow R$ and $h : R \rightarrow S$, we do have $c\rho\sigma : X$ as it should. If ρ and σ behave as in the proposition, we then get the following interaction:

$$\begin{aligned}
c\rho\sigma, \pi &\succ (\sigma)\kappa h. (*_{\pi})(\rho)h, \pi \\
&\succ \kappa h. (*_{\pi})(\rho)h, \pi_R \\
&\succ (*_{\pi})(\rho) *_{\pi_R}, \pi_R \\
&\succ (\rho) *_{\pi_R}, \pi \\
&\succ *_{\pi_R}, r \cdot \pi_S \\
&\succ r, \pi_R,
\end{aligned}$$

with the expected result. And thus we get one possible sequential implementation of the specification. It can be observed that, in this particular example, σ is run first, which means that T can't be understood as specifying an exception handling mechanism where σ would be the handler !

A particular case is when $\sigma = \lambda x.x$, which at the level of types means $X = R$. The behavior is now, if ever $\rho a, \pi \succ a, r \cdot \pi_S$, then $((c)\rho)\lambda x.x, \pi$

evaluates to a multiset on $\{(r, \pi_R)\}$. If we further assume that all terms involved are sequential, *i.e.*, none involve the $|$ operator, then we simply get $((c)\rho)\lambda x.x, \pi \succ r, \pi$, which is the behaviour of cc . In that $T[R/X]$ is trivially equivalent to Peirce's law; this shouldn't be too much a surprise.

3.3 The Symmetric Excluded Middle

We return to G , already presented in the introduction. This formula is not intuitionistically valid, and the logic obtained when adding it to intuitionistic logic is that of formulas which are true in any linear Kripke model. What we set up to prove is:

Proposition 5 *Let $\vdash c : G$ be derivable, then for all terms ρ, σ, r et s and for all stacks π, π_R and π_S , such that for all terms a and b , $\rho a, \pi \succ a, r \cdot \pi_S$ and $\sigma b, \pi \succ b, s \cdot \pi_R, c\rho\sigma, \pi$ evaluates into a multiset on $\{(s, \pi_S), (r, \pi_R)\}$.*

Proof. Let $\rho, \sigma, r, s, \pi, \pi_R$ and π_S be as above. Take for \perp the closure of $\{(s, \pi_S), (r, \pi_R)\}$, and set $R^- = \{\pi_R\}$, $S^- = \{\pi_S\}$ and $X^- = \{\pi\}$.

For one thing $r \in R$ and $s \in S$.

Let now $a \in R \rightarrow S$, by definition $a, r \cdot \pi_S \in \perp$, so by \perp being closed: $\rho, a \cdot \pi \in \perp$, hence $\rho \in (R \rightarrow S) \rightarrow X$. Symmetrically, $\sigma \in (S \rightarrow R) \rightarrow X$. Whence by adequacy we get: $c, \rho \cdot \sigma \cdot \pi \in \perp$. \square

An example is $((c_R)\rho)\sigma = \kappa k^{X \rightarrow S}.(\rho)\lambda x^R.(k)(\sigma)\lambda y^S.x : X$ or c_S the symmetric form exchanging σ and ρ . When ρ and σ behave as in the proposition, $((c_R)\rho)\sigma, \pi \succ r \cdot \pi_R$, whereas $((c_S)\rho)\sigma, \pi \succ s \cdot \pi_S$.

In pure propositional calculus, there is no means to discriminate between these two behaviours. As in the case of the boolean type, we can refine the specification in predicate calculus, by decorating G as:

$$\forall R \forall S [(\forall x R x \rightarrow \forall x S x) \vee \forall x (S x \rightarrow R x)].$$

Then all terms of that type will behave as c_R does.

Note also that, since T trivially implies G , there is a form of compatibility between the two specifications.

We don't rerun the operational explanation given in the introduction. Let us observe, yet, that the proposition is not saying that any c of type G is implementing the cross communication mechanism. In fact that can't be

the case since sequential proofs, such as the one above, just can't express it. But some proofs will, such as $c_S | c_R$. Admittedly, the implementation is not very elegant. The reasonable thing to do, then, seems to extend the language with a new suitable primitive. One subtle point yet is that the naïve rule:

$$[a, r \cdot \pi_S | b, s \cdot \pi_R | \dots] \succ [r, \pi_R | s, \pi_S | \dots],$$

would be wrong in that r, s might respectively contain a and b , so that both executables r, π_R and s, π_S might know a and b and could subsequently deadlock by calling on the same channel.

One other desirable thing would be a general specification result about T - and G -like formulas, perhaps giving some insight on why they should specify a means of synchronisation, and explaining as well what other schemes are typable. This is the object of the next and last subsection.

3.4 Disjunctive Tautologies

Let's consider a finite family of formulas $A_i, i \in I$, each of the form:

$$A_i = B_{i1} \rightarrow (\dots (B_{in_i} \rightarrow C_i) \dots),$$

where B_{ij} s and C_i s are all propositional variables, and set A to be the universal closure of $\bigvee_I A_i$. Such an A will be called a *purely disjunctive* formula.

We define the truth set of A , denoted $tr(A)$, to be the set of triples i, j, k such that $B_{ij} = C_k$. Rewriting all implications in A as disjunctions yields a classically equivalent formula which is the closure of a disjunction of literals, namely $(\bigvee \neg B_{ij}) \vee C_k$, whence A is valid iff its truth set is not empty.

Conversely, by the standard second order encoding of disjunctions as implications, we can obtain an intuitionistic equivalent of A , which we still denote A in the proposition below.

Both T_1 and G_1 , we already ran into, are purely disjunctive; a simpler example is $V = \forall A [A \rightarrow A]$, and a longer one is:

$$W = \forall ABC [(A \rightarrow (B \rightarrow C)) \vee (C \rightarrow A) \vee (C \rightarrow B)].$$

The special thing about them is:

Proposition 6 *Let A be a purely disjunctive formula and let $\vdash c : A$ be derivable, then for all terms ρ_i, b_{ij} and for all stacks π_i , such that for all terms $a, \rho_i a, \pi \succ a, b_{i1} \cdots b_{in_i} \cdot \pi_i, c\rho_1 \dots \rho_n, \pi$ evaluates into a multiset on $\{(b_{ij}, \pi_k) : i, j, k \in tr(A)\}$.*

Proof. Note that if A is false, then, hopefully, $\vdash c : A$ is not derivable, and the statement then vacuously holds.

The proof goes the usual way. Take for \perp the closure of $\{(b_{ij}, \pi_k) : i, j, k \in tr(A)\}$, set $X^- = \{\pi\}$ and $C_i^- = \{\pi_k : C_i = C_k\}$.

First we observe that $b_{ij}, \pi_k \in \perp$ whenever $B_{ij} = C_k$, hence $b_{ij} \in B_{ij}$ if there is some k such that $i, j, k \in tr(A)$. If not, we simply take $B_{ij} = \Lambda$. In all cases, we now have $b_{ij} \in B_{ij}$.

Let $a \in A_i$, then $a, b_{i1} \cdots b_{in_i} \cdot \pi_i \in \perp$, and so does $\rho_i a, \pi$, hence $\rho_i \in A_i \rightarrow X$. Therefore, $c\rho_1 \dots \rho_n, \pi \in \perp$. \square

If we consider any protocol generated by V , then we see it is sequential, c merely feeding in ρ with the identity, should ρ ever stop on a . In fact, V is intuitionistically valid, and any intuitionistic proof of a purely disjunctive formula generates a dummy protocol, by the ... disjunction property. That is, intuitionistic terms, even using $|$, can only fork and will never synchronise back their threads. In fact, $|$ itself behaves like this and can be thought of as a degenerated synchronisation scheme associated to the disjunctive formula $\top \vee \top$, where \top stands for the logical constant ‘true’.

Next, if we turn to W , we see that associated protocols can be non linear and non deterministic either. Specifically, if our three processes are blocked as in $\rho_1 a_1, \pi \succ a_1, v_A \cdot v_B \cdot \pi_C, \rho_2 a_2, \pi \succ a_2, v_C \cdot \pi_A$ and $\rho_3 a_3, \pi \succ a_3, v'_C \cdot \pi_B$, then communication could be implemented by launching concurrently v_C, π_C and v'_C, π_C , or by launching one of them only, etc. There is room in such a specification for creating dynamic patterns of synchronisation.

4 Conclusion

So what? We have shown disjunctive tautologies have as realizers $\lambda\kappa$ -terms which are dying to be read as sequential implementations of more abstract concurrent programs, namely synchronisation schemes. We didn’t, as the referees would point out, develop those concurrent programs in an independent syntax such as the pi- or the join-calculus [7]. That remains

to be done. We didn't either bring in any concrete concurrent example backing the expressiveness of these schemes. As all synchronisations here are by construction deadlock-free, they shouldn't be expected to have immense expressive power anyway.

References

1. V. DANOS, J.-B. JOINET, H. SCHELLINX. A New Deconstructive Logic: Linear Logic (1996). *Journal of Symbolic Logic*.
2. J.-Y. GIRARD. A New Constructive Logic: Classical Logic (1992). *Mathematical Structures in Computer Science*.
3. T. GRIFFIN. A formulae-as-types notion of control (1990). *In Proceedings of POPL'90*.
4. J.-L. KRIVINE. Typed Lambda-Calculus and Classical ZF Set-Theory (2000). *Archive for Mathematical Logic*.
5. L. ONG, C. STEWART. A Curry-Howard Foundation for Functional Computation with Control (1997). *In Proceedings of POPL'97*.
6. M. PARIGOT. Strong Normalization for Second-Order Lambda-Mu Calculus (1993). *In Proceedings of LICS'93*.
7. MOSCOVA PROJECT. The Join-Calculus Language:
<http://pauillac.inria.fr/join/>.