

# Disjunctive Normal Forms specify Multi-Exception Handlers

Emmanuel Beffara\*    Vincent Danos  
Université Paris 7  
Équipe Preuves, Programmes, Systèmes

## Abstract

Within a classical call-by-name  $\lambda$ -calculus, we prove, using Krivine’s realizability, that all terms typable with disjunctive normal forms (disjunctions of conjunctions of literals) share a common computational behaviour: they implement a multi-exception handling mechanism whose exact geometry depends on the tautology. By using dynamic binding, we are able to define equivalent and more efficient primitive control combinators, which are neatly described through a specialized sequent calculus, and are correct in the sense that they realize the intended associated tautology.

## 1. Introduction

This paper runs the proof/program correspondence backwards. Within a specified set of types, namely *disjunctive normal forms* (DNFs), we identify, for each type, a typical common computational behaviour, which we call *multi-exception handling*. After Krivine, we call this solving the *specification problem* for that particular class of formulas. It is understood that this is only a way of saying, since there are many solutions to the same specification problem. Running Curry-Howard backwards has a definite advantage: one might not find anything interesting in terms of instructions, but what one finds has to be correct by construction.

Surely not all classes of formulas are worth asking the question. What does it take to qualify as an interesting class? Well, a hint comes from [3], where some synchronisation schemes were already set in correspondence with a family of so-called *disjunctive* tautologies. None of the formulas of that class was intuitionistically valid (except in trivial cases). This basically is a consequence of the intuitionistic disjunction property. This is also the case for DNFs and they are actually the next simplest class with this anti-intuitionistic feature.

Caveat! Of course any formula is classically equivalent to a DNF, but when it comes to computational interpretation, relying on logical equivalence in classical logic is the last thing one wants to do. Actually, one computational interpretation of this transformation from generic propositional formulas to DNFs must be a compilation into our multi-exception handling combinators. And we intend to explore the matter further.

Coming back to our question, it seems there is a pattern here: anti-intuitionistic classes of formulas specify various forms of control. Whatever inhabits them, precisely because nothing in there is intuitionistically true, should be pure control. This is in essence the heuristics we use.

One could fear that such a vast set of types as DNFs, would be too loose to actually provide an interesting specification problem. Well surprisingly, it is not so. They all do specify a quite distinctive control mechanism, and more sophisticated than with disjunctive tautologies. Proving this is the first contribution of the paper. We go beyond giving mere descriptions of those behaviours, by actually extending our basic language, a call-by-name typed  $\lambda$ -calculus with control, with primitive instructions directly implementing these. Proving these are correct is the second contribution. We use Krivine’s realizability to do this (as documented in [6, 7]).

The new computation rules are, we believe, simple enough to understand, even though they use dynamic binding, and can be rendered in graphic notation as *control charts*. Notation is an issue here, in that to have the new instructions actually used by a programmer, one has to accompany the formal operational semantics with an intuitive notation that will help in building a working representation of whatever control scheme is described. We also flank the graphic notation with a more proof-theoretic description via a specialized sequent calculus, and more importantly with an ordinary programming-language-like syntax to suggest how our new control combinators would fit in the real picture. This last piece of syntax is smoothly extending the CAML [1] notation for exception handling. We hasten to add that all exceptions here are local for one thing, and second, the ambient language being call-by-name, control is of

\*Corresponding author: 2 place Jussieu, 75251 Paris Cedex 05, France, Emmanuel.Beffara@pps.jussieu.fr

quite a different and simpler nature than it is in ordinary programming. Yet it should be possible to rerun our methods in the call-by-value world. This is an important question which we leave for future exploration.

## 2. Preliminaries

Let us first state the definitions for our calculus and the logic that types it. We also state the notion of realizability that we use later on.

### 2.1. The calculus

We define  $\Lambda$ , the set of terms (or  $\lambda\kappa$ -terms), and  $\Pi$ , the set of stacks, as well as the set  $\Lambda \times \Pi$  of executables, by

$$\begin{array}{ll} \text{terms :} & t ::= x \mid tt \mid \lambda x.t \mid \kappa x.t \mid k_\pi \\ \text{stacks :} & \pi ::= \varepsilon \mid t \cdot \pi \\ \text{executables :} & e ::= t * \pi \end{array}$$

The evaluation relation  $\succ$  is defined as the reflexive transitive closure of the following set of rules:

$$\begin{array}{lll} tu * \pi \succ & t * u \cdot \pi & [\text{push}] \\ \lambda x.t * u \cdot \pi \succ & t[u/x] * \pi & [\text{pop}] \\ \kappa x.t * \pi \succ & t[k_\pi/x] * \pi & [\text{save}] \\ k_\pi * t \cdot \pi' \succ & t * \pi & [\text{restore}] \end{array}$$

Note that the binders  $\lambda x$  and  $\kappa x$  are dual in the sense that one substitutes terms while the other substitutes stacks, in the form of the terms  $k_\pi$  which can also be called *continuations*.

### 2.2. The typing system

Types are second order propositional formulas. Given a set  $Var$  of propositional variables, the typing rules are the following:

$$\begin{array}{c} \frac{}{\Gamma, x : A \vdash x : A} [\text{axiom}] \\ \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \rightarrow B} [\rightarrow i] \quad \frac{\Gamma \vdash t : A \rightarrow B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B} [\rightarrow e] \\ \frac{\Gamma \vdash t : A \quad X \notin FV(\Gamma)}{\Gamma \vdash t : \forall X A} [\forall i] \quad \frac{\Gamma \vdash t : \forall X A}{\Gamma \vdash t : A[B/X]} [\forall e] \\ \frac{\Gamma, x : A \rightarrow B \vdash t : A}{\Gamma \vdash \kappa x.t : A} [\text{Peirce}] \end{array}$$

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.

Further on we will use positive and negative literals as types, naturally defining the absurd as  $\perp := \forall X X$  and the negation as  $\neg A := A \rightarrow \perp$ . All the types we consider later on live actually in a fragment of this system that correspond to simple types augmented with the  $\perp$  constant, so types are implicitly considered universally quantified in every variable.

### 2.3. Realizability

Realizability builds models of the propositional logic above by associating truth values in the form of subsets of  $\Pi$  to each closed formula. Let  $\perp \subseteq \Lambda \times \Pi$  be a set of executables closed by anti-reduction, i.e. if  $e \succ e'$  and  $e' \in \perp$  then  $e \in \perp$ . The executables in this set are called the *observables*.

From  $\perp$  we deduce a notion of *orthogonality* between terms and stacks: a set  $X$  of terms is said to be orthogonal to a set  $Z$  of stacks (which we can write  $X \perp Z$ ) if for any term  $t \in X$  and any stack  $\pi \in Z$  we have  $t * \pi \in \perp$ . The orthogonal of a given set of terms or stacks is then the largest set orthogonal to it, i.e.  $X^\perp = \{\pi \mid X \perp \pi\}$  and  $Z^\perp = \{t \mid t \perp Z\}$  (note that we use the same notation in both cases even if the operations are dual).

Let  $e$  be a function from propositional values into the powerset  $\mathcal{P}(\Pi)$ . The truth value  $[A]_e$  associated to a given type  $A$  in the valuation  $e$  is defined inductively as

$$\begin{aligned} [X]_e &= e(X) \\ [A \rightarrow B]_e &= [A]_e^\perp \cdot [B]_e \\ [\forall X A]_e &= \bigcup_{Z \subseteq \Pi} [A]_{e[Z/X]} \end{aligned}$$

where  $e[Z/X]$  is the environment  $e$  where value  $Z$  has been assigned to  $X$ . For closed formulas we can write  $[A]$  instead of  $[A]_e$  since the value is environment-independent (more generally,  $[A]_e$  depends only on the values  $e$  takes on variables free in  $A$ ). We then call *interpretation* of  $A$  the set  $|A|_e = [A]_e^\perp$  of terms orthogonal to  $[A]_e$ , and we say that a term  $t$  *realizes* a type  $A$ , which we write  $t \models A$ , if  $t$  is in the interpretation of  $A$ .

### 2.4. Adequacy

This  $\models$  is a semantic relation between terms and types, while the typing provided a more syntactic relation. For any choice of  $\perp$ , the typing relation is actually a subset of the realization relation:

**Theorem 1 (adequacy).** *Let  $x_1 : A_1, \dots, x_n : A_n \vdash t : B$  be a derivable typing judgement and let  $\perp$  be a set of observables. For any valuation  $e : Var \rightarrow \mathcal{P}(\Pi)$ , any family  $t_1, \dots, t_n$  of terms such that  $t_i \in |A_i|_e$  for each  $i$  and any stack  $\pi \in [B]_e$ , the executable  $t[t_1/x_1, \dots, t_n/x_n] * \pi$  is in  $\perp$ .*

*Proof.* We proceed by induction on the typing derivation. Call  $\Gamma$  the typing environment  $x_1 : A_1, \dots, x_n : A_n$  and consider a given valuation  $e$ , a given family  $t_i \in |A_i|_e$  and a given stack  $\pi \in [B]_e$ . For any term  $t$ , call  $\bar{t}$  the substituted term  $t[t_1/x_1, \dots, t_n/x_n]$ .

**axiom:** We have the judgement  $x : A \vdash x : A$  so if  $t \in |A|_e$  we have  $t * \pi \in \perp$  by definition.

**application:** Let  $t$  and  $u$  be terms such that  $\Gamma \vdash t : A \rightarrow B$  and  $\Gamma \vdash u : A$  are derivable. By induction we have  $\bar{t} \in |A \rightarrow B|_e$  and  $\bar{u} \in |A|_e$ , so  $\bar{u} \cdot \pi$  is in  $|A|_e \cdot [B]_e = [A \rightarrow B]_e$ , therefore  $\bar{t} * \bar{u} \cdot \pi$  is in  $\perp$ , and so is  $\bar{t}\bar{u} * \pi$  since  $\perp$  is closed by anti-reduction.

**abstraction:** Assume  $\Gamma, x : A \vdash t : B$  is derivable. By induction, for any term  $u \in |A|_e$ , we know that  $\bar{t}[u/x] * \pi$  is in  $\perp$ , and so is  $\bar{\lambda}x.t * u \cdot \pi$  by anti-reduction, so we can actually deduce that  $\bar{\lambda}x.t$  is orthogonal to  $|A|_e \cdot [B]_e = [A \rightarrow B]_e$ .

**continuation:** If  $\Gamma, x : A \rightarrow B \vdash t : A$  is derivable, then, by induction, for any term  $u \in |A \rightarrow B|_e$  and any stack  $\pi \in [A]_e$  we have  $\bar{t}[u/s] * \pi \in \perp$ . Besides, for any stack  $v \cdot \pi' \in [A \rightarrow B]_e$  the term  $v$  is in  $|A|_e$ , so  $v * \pi$  is in  $\perp$ , and so is  $k_\pi * v \cdot \pi'$  by anti-reduction, which proves that  $k_\pi$  is in  $|A \rightarrow B|_e$ , so the executable  $\bar{t}[k_\pi/x] * \pi$  is in  $\perp$ , as well as  $kx.t * \pi$  by anti-reduction.

**quantification:** Write  $B = \forall X A$  and suppose that  $\Gamma \vdash t : A$  is derivable. From the definition of  $[\forall X A]$  we deduce the existence of a stack set  $\mathcal{Z}$  for which  $[A]_{e[z/X]}$  contains  $\pi$ . Since the variable  $X$  does not appear in any of the  $A_i$ , the value of each  $|A_i|_e$  does not depend on  $e(X)$ , so for each  $i$  we have  $t_i \in |A_i|_{e[z/X]}$ , and therefore  $\bar{t} * \pi$  is in  $\perp$  by induction hypothesis.

**un-quantification :** Assume  $\Gamma \vdash t : \forall X A$ , this means that for every stack set  $\mathcal{Z}$  and every stack in  $[A]_{e[z/X]}$  we have  $t * \pi \in \perp$ . This is true in particular for  $\mathcal{Z} = [B]_e$  for a given type  $B$ . If we prove that the valuation of  $A[B/X]$  in  $e$  is equal to the valuation of  $A$  in  $e[[B]_e/X]$ , then we get the expected result, and this substitution lemma is proved easily by induction on  $A$ , from the definition of substitution.

Adequacy thus holds for all types.  $\square$

In the case of closed formulas, this theorem reduces into the following adequacy lemma:

**Proposition 2.** *For any term  $t$  and any type  $A$ , if  $\vdash t : A$  is derivable then  $t \models A$  holds for any  $\perp$ .*

### 3. Specification theorems

The class of formulas we are considering here is the so-called *disjunctive normal forms* (or DNFs), that is disjunctions of conjunctions of literals. Given a set  $Var$  of propositional variables, we write  $Lit$  for the set of literals over  $Var$ , i.e.  $Lit = Var \cup \neg Var$ . Two literals  $X$  and  $\neg X$  are said to be *opposite*. A *clause* is a conjunction of literals, or equivalently a finite subset of  $Lit$ , and a DNF is a disjunction of clauses, or a finite subset of the powerset  $\mathcal{P}(Lit)$ . We could use multisets instead of sets, however this would lead to undue notational complication, so we don't. We will write clauses and DNFs indifferently as formulas with  $\vee$  and  $\wedge$  or as sets of sets of literals, whichever is more suited to the context.

We will study the specification problem for this class of formulas. However, simple types are built with  $\rightarrow$  as the only connector, so we must define how DNFs are converted into types. For any type  $\tau$  and any ordered clause  $\{L_1, \dots, L_k\}$  we define:

$$\{L_1, \dots, L_k\} \rightarrow \tau := L_1 \rightarrow \dots \rightarrow L_k \rightarrow \tau$$

The ordering we use will be either clear from the context or indifferent. Likewise, given a formula  $\Gamma = \{c_1, \dots, c_n\}$  and a fresh variable  $Z$ , we interpret  $\Gamma$  as a type by defining

$$\Gamma := \forall Z (c_1 \rightarrow Z) \rightarrow \dots \rightarrow (c_n \rightarrow Z) \rightarrow Z$$

again using an appropriate ordering. Any term of type  $\Gamma$  will thus take  $n$  functions as arguments. Take note that  $\Gamma$  seen as a type is intuitionistically isomorphic to  $\Gamma$  as a DNF, so conversion is computationally neutral.

#### 3.1. A first specification

Our purpose now is to identify a computational behaviour common to all  $\lambda\kappa$ -terms that are typable using a given DNF. For this purpose, we characterize tautologies among DNFs using a notion of *section*:

**Definition 1 (section).** *Let  $\Gamma = c_1 \vee \dots \vee c_n$  be a disjunctive normal form. A section of  $\Gamma$  is an element of the product  $c_1 \times \dots \times c_n$ .*

One can rephrase this by saying that a section is a choice of one literal in each clause. We then write  $\sigma(c)$  for the literal chosen in clause  $c$  and  $L \in \sigma$  if literal  $L$  is chosen in some clause. A section can be interpreted as a potential counter-example, and therefore it comes as no surprise that the formula is true exactly when there is no such counter-example:

**Proposition 3.** *A disjunctive normal form  $\Gamma$  is a tautology if and only if every section of  $\Gamma$  contains two opposite literals.*

*Proof.* Suppose that  $\Gamma$  is a tautology and there is a section  $\sigma$  that does not contain opposite literals. We can then define a boolean valuation  $\nu$  of the propositional variables such that  $\nu(X) = \top$  if  $\neg X \in \sigma$  and  $\nu(X) = \perp$  if  $X \in \sigma$ . This valuation thus makes each clause of  $\Gamma$  false, which is contradictory.

Suppose now that  $\Gamma$  is not a tautology, so there is a valuation  $\nu$  such that  $\nu(\Gamma)$  is false. For each clause  $c$ , since  $\nu_c = \perp$ , there is a literal  $\sigma(c)$  such that  $\nu(\sigma(c)) = \perp$ , and this defines a section in which all literals are false in  $\nu$ , therefore  $\sigma$  cannot contain opposite literals.  $\square$

Sections are clearly connected to provability by the proposition above, but they also happen to play a rôle in the specification. Before embarking on the precise statement, let us explain intuitively what is happening. Assume some  $t$  of type  $\Gamma$  is given a sequence of arguments, say  $f_c$ , with  $c \in \Gamma$ . What  $t$  does is to pass each  $f_c$  a set of exceptions indexed by  $c$ , and if *all* of the  $f_c$ s raise an exception, then  $t$  selects two of them matching opposite types and run their arguments one against the other. That there always must be two opposite exceptions is precisely what the proposition above says.

**Theorem 4.** *Let  $\Gamma$  be a tautology in disjunctive normal form and let  $t$  be a  $\lambda\kappa$ -term of type  $\Gamma$ . Let  $\pi$  be a stack and  $\vec{f}$  a family of terms indexed by the clauses of  $\Gamma$ . Suppose there exists a section  $\sigma$  of  $\Gamma$  and a family of terms  $\nu_c$  and stacks  $\pi_c$  such that for each clause  $c$ ,*

$$\begin{aligned} \sigma(c) \in \neg Var &\Rightarrow \forall \vec{\alpha} \exists \pi' & f_c * \vec{\alpha} \cdot \pi \succ \alpha_{\sigma(c)} * \nu_c \cdot \pi' \\ \sigma(c) \in Var &\Rightarrow \forall \vec{\alpha} & f_c * \vec{\alpha} \cdot \pi \succ \alpha_{\sigma(c)} * \pi_c \end{aligned}$$

where  $\vec{\alpha}$  is indexed on the literals of  $c$ . Then there exists a pair of clauses  $(c, c')$  such that  $\sigma(c) = \neg\sigma(c')$  and

$$t * \vec{f} \cdot \pi \succ \nu_c * \pi_{c'}.$$

*Proof.* First we split  $\Gamma$  into  $\Gamma^+ = \{c \mid \sigma(c) \in Var\}$  and  $\Gamma^- = \{c \mid \sigma(c) \in \neg Var\}$ . Define  $\perp$  as the closure by anti-reduction of

$$\{\nu_c * \pi_{c'} \mid c \in \Gamma^-, c' \in \Gamma^+, \sigma(c) = \neg\sigma(c')\}$$

and instantiate the propositional variables by

$$[Z] = \{\pi\} \quad \text{and} \quad [X] = \{\pi_{c'} \mid c' \in \Gamma^+, \sigma(c') = X\}$$

where  $Z$  is the fresh variable used to translate  $\Gamma$  into a type and  $X$  ranges over variables occurring in  $\Gamma$ .

Let  $c$  be a clause in  $\Gamma^-$ . For any family  $\vec{\alpha}$  of terms such that  $\alpha_{\neg X} \models \neg X$  and any family  $\vec{a}$  of terms such that  $\alpha_X \models X$ , by hypothesis,  $f_c * \vec{\alpha} \cdot \vec{a} \cdot \pi$  reduces into  $\alpha_{\sigma(c)} * \nu_c \cdot \pi'$  for some  $\pi'$ . Here  $\sigma(c)$  is a negative literal  $\neg X$ , and the term  $\nu_c$  realizes  $X$  since for each element  $\pi_{c'}$  in  $[X]$  we have  $\sigma(c) =$

$\neg\sigma(c')$  and so  $\nu_c * \pi_{c'} \in \perp$ , therefore  $\alpha_{\sigma(c)} * \nu_c \cdot \pi'$  is in  $\perp$ , so is  $f_c * \vec{\alpha} \cdot \vec{a} \cdot \pi$  by anti-reduction, and so  $f_c \models c \rightarrow Z$ .

Similarly, let  $c'$  be a clause in  $\Gamma^+$ . For any family  $\vec{\alpha}$  of terms such that  $\alpha_{\neg X} \models \neg X$  and any family  $\vec{a}$  of terms such that  $\alpha_X \models X$ , by hypothesis  $f_{c'} * \vec{\alpha} \cdot \vec{a} \cdot \pi$  reduces into  $\alpha_{\sigma(c')} * \pi_{c'}$ , which is in  $\perp$  since  $\pi_{c'} \in [\sigma(c')]$ , therefore  $f_{c'} \models c' \rightarrow Z$ .

By hypothesis,  $\vdash t : \Gamma$  is derivable, so the adequacy lemma proves that  $t$  realizes  $\Gamma$ . We have just shown that for each clause  $c$ , the term  $f_c$  realizes the type  $c \rightarrow Z$ , and by definition  $\pi$  is in  $[Z]$ , so  $\vec{f} \cdot \pi$  is in  $[\Gamma]$ , so  $t * \vec{f} \cdot \pi$  is in  $\perp$ , which means that it reduces into some  $\nu_c * \pi_{c'}$  with  $\sigma(c) = \neg\sigma(c')$ .  $\square$

### 3.2. Sharper specifications

The specification extracted above is quite shallow. There is an important restriction, namely that the values  $\nu_c$  and the stacks  $\pi_c$  that the processes pass when rising exceptions may not contain any occurrence of the functions' arguments (the  $\alpha_L$  in the quantifications). We can do better by relaxing the conditions on the arguments:

$$\begin{aligned} \sigma(c) \in \neg Var &\Rightarrow \forall \vec{\alpha} \exists \pi' & f_c * \vec{\alpha} \cdot \pi \succ \alpha_{\sigma(c)} * \nu_c[\vec{\alpha}] \cdot \pi' \\ \sigma(c) \in Var &\Rightarrow \forall \vec{\alpha} & f_c * \vec{\alpha} \cdot \pi \succ \alpha_{\sigma(c)} * \pi_c[\vec{\alpha}] \end{aligned}$$

and proving that reduction leads to  $\nu_c[\vec{f}] * \pi_{c'}[\vec{u}]$  for some sets of terms  $\vec{f}$  and  $\vec{u}$ . Proving just this is mainly the same as proving theorem 4. However, proving something about the terms in  $\vec{f}$  and  $\vec{u}$  is harder.

In the particular case of the excluded middle, i.e.  $\Gamma = \neg X \vee X$ , the following development result holds:

**Proposition 5.** *Let  $t$  be a term of type  $\neg X \vee X$ . Let  $\pi$  be a stack and let  $f$  and  $g$  two terms. Suppose there is a stack  $\pi_g$  and a family of contexts  $(\nu_i[\cdot])_{0 \leq i \leq n}$  such that*

$$\begin{aligned} \forall \alpha \exists \pi' & f * \alpha \cdot \pi \succ \alpha * \nu_0[\alpha] \cdot \pi' \\ \forall \alpha & g * \alpha \cdot \pi \succ \alpha * \pi_g \\ \forall i < n \forall \alpha \exists \pi' & \nu_i[\alpha] * \pi_g \succ \alpha * \nu_{i+1}[\alpha] \cdot \pi' \end{aligned}$$

then there exists a term  $u$  such that  $t * f \cdot g \cdot \pi$  reduces into  $\nu_n[u] * \pi_g$ .

This allows values passed to exceptions to contain occurrences of exceptions, without lifting the constraint on the stacks. This result could probably be extended to the general case, but the task seems notationally daunting. Even in this relatively simple case, such a refined specification is quite technical to prove, and we skip the argument. Yet, we wanted to insist that sharper specifications can be extracted.

## 4. Synthesizing combinators

Of course all DNFs are provable in our system, and therefore one can find  $\lambda\kappa$ -terms that will have them as

types. Compared to the specification as described in the theorem above, they appear to implement it in a quite clumsy way. It is tempting to have new combinators in the calculus doing the same job, only better.

In order to synthesize new control primitives, we now define a logic with an associated sequent calculus to prove tautologies with, and deduce a computational structure from the proofs, by using a specialized Curry-Howard correspondence.

#### 4.1. The or-and logic

The logic  $\mathcal{L}_{\vee\wedge}$  is defined as follows, given a set  $Var$  of propositional variables:

$$\begin{aligned} \text{variables : } & X \in Var \\ \text{literals : } & L ::= X \mid \neg X \\ \text{clauses : } & c ::= \top \mid L \wedge \dots \wedge L \\ \text{formulas : } & \Gamma ::= \perp \mid c, \dots, c \\ \text{sequents : } & \Vdash \Gamma \end{aligned}$$

Again, clauses are understood as finite sets and formulas are finite sets of clauses, and  $\top$  and  $\perp$  are their respective empty sets. In particular, ignoring order and duplication, we derive sequents using a single  $n$ -ary rule:

$$\frac{\Vdash \Gamma_1, \Delta \quad \dots \quad \Vdash \Gamma_n, \Delta}{\Vdash (\neg X_1 \wedge \dots \wedge \neg X_n), (X_1 \wedge \Gamma_1), \dots, (X_n \wedge \Gamma_n), \Delta}$$

with  $n \geq 0$ , where the notation  $L \wedge \Gamma$  represents the distribution of the literal  $L$  over the clauses in  $\Gamma$ , i.e.

$$L \wedge (c_1, \dots, c_n) := (L \wedge c_1), \dots, (L \wedge c_n)$$

For  $n = 0$  our unique rule reduces to an axiom  $\Vdash \top, \Delta$ , noticing that  $\top$  is the conjunction of zero literals.

First of all, we have to show that this system is complete, i.e. that it actually proves our tautologies. For this we need the following lemma:

**Lemma 6.** *Every tautology in disjunctive normal form has a totally negative clause.*

*Proof.* Let  $\Gamma$  be a DNF. Suppose that  $\Gamma$  has no purely negative clause, then each clause has at least one positive literal, which defines a totally positive section. From proposition 3, we conclude that  $\Gamma$  is not a tautology.  $\square$

We also need the following notion of quotient:

**Definition 2.** *Let  $\Gamma$  be a disjunctive normal form and  $X$  be a variable. The quotient of  $\Gamma$  by  $X$  is defined as*

$$\Gamma/X = \{c \setminus \{X\} \mid c \in \Gamma, \neg X \notin c\}.$$

For instance,

$$\{\{X, Y\}, \{\neg X, Z\}, \{Z\}\}/X = \{\{Y\}, \{Z\}\}.$$

This operation provides a reduction of  $\Gamma$  under the assumption that variable  $X$  is true. Indeed we have the following consistency result:

**Proposition 7.** *Let  $\Gamma$  be a disjunctive normal form and  $X$  be a variable. If  $\Gamma$  is a tautology, so is  $\Gamma/X$ .*

*Proof.* By construction,  $\Gamma/X$  has its variables in  $Var \setminus \{X\}$ . Let  $v$  be a valuation of this set of variables. Define  $v'$  as the extension of  $v$  to  $Var$  such that  $v'(X) = \top$ . Since  $\Gamma$  is a tautology,  $v'(\Gamma)$  is true, so there is a clause  $c$  in  $\Gamma$  such that  $v'(c) = \top$ . Since  $v'(X) = \top$ ,  $c$  does not contain  $\neg X$ , so  $c \setminus \{X\}$  is a clause of  $\Gamma/X$ , and  $v(c \setminus \{X\}) = \top$ , so  $v$  validates  $\Gamma/X$ .  $\square$

**Proposition 8 (completeness).** *Let  $\Gamma$  be a disjunctive normal form. If  $\Gamma$  is a tautology, then  $\Vdash \Gamma$  can be derived in  $\mathcal{L}_{\vee\wedge}$ .*

*Proof.* We actually prove a slightly stronger result, namely that under the same conditions, for any set of clauses  $\Delta$  the sequent  $\Vdash \Gamma, \Delta$  is derivable in  $\mathcal{L}_{\vee\wedge}$ . We proceed by induction on the number of variables in  $\Gamma$ : if there is no variable,  $\Gamma$  is reduced to one trivial clause  $\top$ , and  $\Vdash \top, \Delta$  is derived using the nullary rule. Otherwise, lemma 6 proves that  $\Gamma$  has a totally negative clause, so we can write

$$\Gamma = (\neg X_1 \wedge \dots \wedge \neg X_k), \Gamma'$$

We will thus derive  $\Vdash \Gamma, \Delta$  by applying the  $k$ -ary rule to the  $k$  formulas we get by supposing that one of these variables is true. For each  $i$ , define

$$\begin{aligned} \Gamma_i &= \{c \mid c \wedge X_i \in \Gamma, \neg X_i \notin c\} \\ \Delta_i &= \{c \mid c \in \Gamma, X_i \notin c, \neg X_i \notin c\} \end{aligned}$$

The union of these two formulas is actually the quotient  $\Gamma/X_i$ , and from proposition 7 we know that the formula  $\Gamma_i, \Delta_i$  is a tautology. Since it contains strictly fewer variables than  $\Gamma$ , the induction hypothesis proves that the sequent  $\Vdash \Gamma_i, \Delta_i, \Gamma', \Delta$  is derivable. Besides, from this definition, obviously  $\Delta_i$  is a subset of  $\Gamma'$ , so the sequent  $\Vdash \Gamma_i, \Gamma', \Delta$  is derivable. Moreover, since for each  $i$  the formula  $X_i \wedge \Gamma_i$  is also included in  $\Gamma'$ , we can write

$$\frac{\Vdash \Gamma_1, \Gamma', \Delta \quad \dots \quad \Vdash \Gamma_n, \Gamma', \Delta}{\Vdash (\neg X_1 \wedge \dots \wedge \neg X_n), \Gamma', \Delta}$$

which concludes the proof.  $\square$

We also have to prove that this is actually a proof system, in the sense that the formulas it derives are (classical) tautologies. This can be proved using the characterization with sections:

**Proposition 9 (soundness).** *If a sequent  $\Vdash \Gamma$  is derivable in  $\mathcal{L}_{\vee\wedge}$  then the disjunctive normal form  $\Gamma$  is a tautology.*

*Proof.* We prove by induction on the derivation that any section of  $\Gamma$  contains opposite literals. First we can remark that if  $\Gamma$  contains the empty clause  $\top$ , it has no section and the result holds trivially. Otherwise, let  $\sigma$  be a section of  $\Gamma$ . The last rule is

$$\frac{\Vdash \Gamma_1, \Delta \quad \dots \quad \Vdash \Gamma_n, \Delta}{\Vdash (\neg X_1 \wedge \dots \wedge \neg X_n), (X_1 \wedge \Gamma_1), \dots, (X_n \wedge \Gamma_n), \Delta}$$

for some  $n \geq 1$ , so  $\sigma$  contains a  $\neg X_i$ . If it also contains  $X_i$ , then the proof is finished. Otherwise, its restriction to  $X_i \wedge \Gamma_i, \Delta$  is actually a section of  $\Gamma_i, \Delta$ , which contains opposite literals by induction hypothesis.  $\square$

It is also interesting to prove the soundness of  $\mathcal{L}_{\vee\wedge}$  by “compiling” proofs into classical sequent calculus. To do this, we need a lemma for the introduction of variables:

**Lemma 10.** *If the sequent  $\vdash \Gamma, \Delta$  is derivable in classical sequent calculus, then for any propositional variable  $X$  the sequent  $\vdash \neg X, (X \wedge \Gamma), \Delta$  is derivable.*

*Proof.* We proceed by induction on the number of clauses in  $\Gamma$ . If  $\Gamma$  is empty, this is a simple weakening that introduces the clause  $\neg X$ . Otherwise, pick a clause  $c$  in  $\Gamma$  and define  $\Gamma = c, \Gamma'$ . By induction, suppose that  $\vdash \neg X, c, (X \wedge \Gamma), \Delta$  can be derived. Then we derive:

$$\frac{\frac{\vdash \neg X, X, (X \wedge \Gamma), \Delta}{\vdash \neg X, X, (X \wedge \Gamma), \Delta} [\text{id}] \quad \vdash \neg X, c, (X \wedge \Gamma), \Delta}{\vdash \neg X, (X \wedge c), (X \wedge \Gamma), \Delta} [\wedge R]$$

which proves the expected result.  $\square$

This lemma is actually a translation of the unary version of the deduction rule of  $\mathcal{L}_{\vee\wedge}$ , and our alternative soundness proof is an iteration of it:

**Proposition 11 (soundness again).** *Let  $\Gamma$  be a disjunctive normal form. If  $\Vdash \Gamma$  is provable in  $\mathcal{L}_{\vee\wedge}$  then  $\vdash \Gamma$  is provable in classical sequent calculus.*

*Proof.* We proceed by induction on the proof of  $\Gamma$  in  $\mathcal{L}_{\vee\wedge}$ . Consider the derivation

$$\frac{\Vdash \Gamma_1, \Delta \quad \dots \quad \Vdash \Gamma_n, \Delta}{\Vdash (\neg X_1 \wedge \dots \wedge \neg X_n), \Gamma'}$$

By induction hypothesis we know that  $\vdash \Gamma_i, \Delta_i$  is derivable for each  $i$ , therefore using lemma 10 we can derive  $\vdash \neg X_i, (X_i \wedge \Gamma_i), \Delta$ , which can be weakened into  $\vdash \neg X_i, \Gamma'$  since the formula  $(X_i \wedge \Gamma_i), \Delta$  is a subset of  $\Gamma'$ . Then by induction on the arity  $n$  we derive

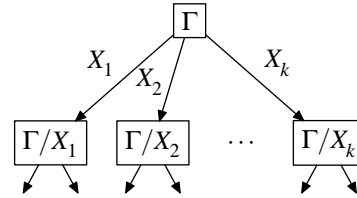
$$\frac{\frac{\vdash \top, \Gamma' [\top R] \quad \vdash \neg X_1, \Gamma' [\wedge R]}{\vdash \neg X_1, \Gamma' [\wedge R]} \quad \vdots}{\vdash (\neg X_1 \wedge \dots \wedge \neg X_{n-1}), \Gamma' [\wedge R]} \quad \vdash \neg X_n, \Gamma' [\wedge R]}{\vdash (\neg X_1 \wedge \dots \wedge \neg X_n), \Gamma' [\wedge R]}$$

which leads to the expected conclusion.  $\square$

As said, none of these tautologies is intuitionistically valid, except in the particular case where they contain the trivial clause  $\top$ , therefore any behaviour they may specify is fundamentally concerned with global control.

## 4.2. Charts and combinators

Proofs in the logic  $\mathcal{L}_{\vee\wedge}$  can be interpreted as strategies in a counter-example game: in a node that proves  $\Gamma$ , the conclusion contains a distinguished totally negative clause  $\neg X_1 \wedge \dots \wedge \neg X_k$ . If we play this clause and the opponent refutes it, he does so by providing a proof of an  $X_i$ , so we can deduce that  $\Gamma$  is equivalent to  $\Gamma/X_i$  and the proof may go down this branch. We can materialize this with a graphical notation that we call *control charts*:



where the label  $X_i$  on an edge represents the passing of a proof of  $X_i$ . The nullary rule is then interpreted as a simple leaf:

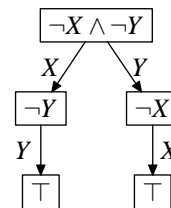


### 4.2.1. Example: a twofold excluded-middle

At each step, the important clause is the distinguished purely negative one, so we write only this one. Taking for instance the complete DNF for variables  $X$  and  $Y$  we have

$$\Gamma = \{\neg X \wedge \neg Y, X \wedge \neg Y, \neg X \wedge Y, X \wedge Y\}$$

The unique proof of this formula corresponds to the following control chart:



The point of this new notation is that it can also be understood as describing a combinator  $\mathbf{C}_\Gamma$ :

$$\begin{aligned}\mathbf{C}_\Gamma * \vec{f} \cdot \pi &\succ f_{\neg X \wedge \neg Y} * \alpha_X \cdot \alpha_Y \cdot \pi \\ \alpha_X * a \cdot \pi' &\succ f_{X \wedge \neg Y} * a \cdot \alpha_{XY} \cdot \pi \\ \alpha_Y * b \cdot \pi' &\succ f_{\neg X \wedge Y} * b \cdot \alpha_{YX} \cdot \pi \\ \alpha_{XY} * c \cdot \pi' &\succ f_{X \wedge Y} * a \cdot c \cdot \pi \\ \alpha_{YX} * d \cdot \pi' &\succ f_{X \wedge Y} * d \cdot b \cdot \pi\end{aligned}$$

The  $\alpha$ s above correspond to edges in the chart, while the  $f$ s correspond to nodes. This definition is rather informal because it hides information: the  $\alpha$ s are not constants since they are related to a particular instance of  $\mathbf{C}_\Gamma$ , moreover they must remember the argument vector  $\vec{f}$  and the initial stack  $\pi$ , as well as an  $a$  or a  $b$  in the case of  $\alpha_{XY}$  and  $\alpha_{YX}$ .

Another, more subtle point, is that these  $a$  and  $b$  may contain occurrences of  $\alpha_X$  and  $\alpha_Y$  respectively and raise them again after  $\alpha_{XY}$  or  $\alpha_{YX}$  has been triggered. Though this behaviour is type-theoretically correct, we can do better and prevent such non-linear behaviour by rebinding the exceptions on the fly:

$$\begin{aligned}\alpha_X * a \cdot \pi' &\succ f_{X \wedge \neg Y} * (\kappa \alpha_X \cdot a) \cdot \alpha_{XY} \cdot \pi \\ \alpha_Y * b \cdot \pi' &\succ f_{\neg X \wedge Y} * (\kappa \alpha_Y \cdot b) \cdot \alpha_{YX} \cdot \pi \\ \alpha_{XY} * c \cdot \pi' &\succ f_{X \wedge Y} * (\kappa \alpha_X \cdot a) \cdot (\kappa \alpha_{XY} \cdot c) \cdot \pi \\ \alpha_{YX} * d \cdot \pi' &\succ f_{X \wedge Y} * (\kappa \alpha_{YX} \cdot d) \cdot (\kappa \alpha_Y \cdot b) \cdot \pi\end{aligned}$$

We observe that this is actually closer to exception handling as found in functional languages, where exceptions are caught only once.

#### 4.2.2. General definition

We are left now with the mission to define, for any chart, the associated combinator and to prove its correctness.

Given a chart  $\mathcal{C}$  over  $\Gamma$ , starting with clause  $c = \neg X_1 \wedge \dots \wedge \neg X_k$  and with immediate sub-charts  $\mathcal{C}_{X_1}, \dots, \mathcal{C}_{X_k}$ , we define  $\mathbf{C}_\mathcal{C}$  inductively:

$$\mathbf{C}_\mathcal{C} * \vec{f} \cdot \pi \succ f_c * \alpha_{X_1} \cdots \alpha_{X_k} \cdot \pi \quad (1)$$

$$\alpha_{X_i} * v \cdot \pi' \succ \mathbf{C}_{\mathcal{C}_{X_i}} * \vec{f}^i \cdot \pi \quad (2)$$

where the  $\alpha_{X_i}$  are fresh symbols, and where  $\vec{f}^i$  is indexed on the clauses of  $\Gamma/X_i$  and defined as

$$f'_c = f_c \quad \text{if } c \in \Gamma \quad (3)$$

$$f'_c = f_{X_i \wedge c}(\kappa x.v[x/\alpha_{X_i}]) \quad \text{if } X_i \wedge c \in \Gamma \quad (4)$$

assuming, in the second case, that the type of  $f_{X_i \wedge c}$  has the literal  $X_i$  in first position.

In a way, the  $\kappa x.v[x/\alpha_{X_i}]$  represents what has been learned from raising the exception  $\alpha_{X_i}$ , and we see that this exception will never be raised again, since  $\alpha_{X_i}$  is no longer free in the right-hand side of (2).

In the particular case where  $c$  is the trivial clause  $\top$ , the definition downs to

$$\mathbf{C}_\mathcal{C} * \vec{f} \cdot \pi \succ f_\top * \pi$$

so the combinator is a pure projection, indeed an intuitionistic construct.

Fresh symbols are needed at step (1) because at step (4) we have to be sure that what we bind was actually created at the corresponding step (1). This is crucial in the correctness argument below.

#### 4.3. Correctness

So we have a class of combinators with clearly defined reduction rules, but we still have to prove that they actually realize the type they are intended to implement. The dynamic binding, while making the combinator's definition valid, imposes a restriction on the validity of realization.

**Definition 3 (sequentiality).** *An observable set  $\perp$  is called sequential if it is not empty, closed under reduction and validates the substitution property: for any context  $e$  and any term  $t$ , if  $e[t] \in \perp$  and if  $e[x]$  does not reduce into an  $x * \pi$ , then  $e[u] \in \perp$  for all terms  $u$ .*

**Theorem 12.** *Let  $\Gamma$  be a tautology in disjunctive normal form and  $\mathcal{C}$  be a chart over  $\Gamma$ . For any sequential  $\perp$ , the combinator  $\mathbf{C}_\mathcal{C}$  realizes the type  $\Gamma$ .*

*Proof.* We proceed by induction on the height of the chart  $\mathcal{C}$ . When  $\mathcal{C}$  is a leaf, we have  $\Gamma = \top, \Delta$ , which is the type  $Z, (\Delta \rightarrow Z) \rightarrow Z$ . As said, the definition of  $\mathbf{C}_\mathcal{C}$  gives  $\mathbf{C}_\mathcal{C} * \vec{f} \cdot \pi \succ f_\top * \pi$  so for any  $\perp$  and any value for  $[Z]$ , if  $f_\top \models Z$  then for any  $\pi$  in  $[Z]$  we have  $\mathbf{C}_\mathcal{C} * \vec{f} \cdot \pi \succ f_\top * \pi \in \perp$ , thus  $\mathbf{C}_\mathcal{C} \models \Gamma$ .

Suppose  $\mathcal{C}$  starts with a negative clause  $n = \neg X_1 \wedge \dots \wedge \neg X_k$  and let  $\perp$  be a sequential set of observables. Assume for each variable  $X$  a valuation  $[X] \subseteq \Pi$ , and call  $Z$  the variable used as the return type. For each clause  $c$  in  $\Gamma$ , let  $f_c$  be a term that realizes  $c \rightarrow Z$ . By definition we have

$$\mathbf{C}_\mathcal{C} * \vec{f} \cdot \pi \succ f_n * \alpha_{X_1} \cdots \alpha_{X_k} \cdot \pi$$

so if we prove that the right member is in  $\perp$  we can conclude  $\mathbf{C}_\mathcal{C} \models \Gamma$ .

If the reduction of  $f_n * \vec{\alpha} \cdot \pi$  never places any  $\alpha_{X_i}$  in head position,  $f_n * \vec{\alpha} \cdot \pi$  will also ignore  $\vec{f}$  for any family of terms, so it holds in particular if  $t_X \models \neg X$  (the family is indexed on the variables in  $n$ ), in which case  $f_n * \vec{\alpha} \cdot \pi$  is in  $\perp$ . Then by the substitution property we deduce that  $f_n * \vec{\alpha} \cdot \pi$  is in  $\perp$ , since none of the  $|\neg X|$  is empty (because  $\perp \neq \emptyset$ ).

Otherwise there exists a propositional variable  $X$ , a multi-hole term context  $v_X[\ ]$  and a stack  $\pi'$  such that we have

$$f_n * \vec{\alpha} \cdot \pi \succ \alpha_X * v_X[\alpha_X] \cdot \pi' \succ \mathbf{C}_{\mathcal{C}_X} * \vec{f}^i \cdot \pi$$

using the notations of section 4.2.2, thus proving that the third executable is in  $\perp$  is enough to conclude. By induction hypothesis, the combinator  $C_{e_X}$  realizes  $\Gamma/X$ , so we have to prove that  $f'_c$  realizes  $c \rightarrow Z$  for each  $c$  in  $\Gamma/X$ . If  $c$  is a clause of  $\Gamma$ , we have  $f'_c = f_c$  and  $f_c \models c \rightarrow Z$  by hypothesis. Otherwise  $X \wedge c$  is a clause of  $\Gamma$  and  $f'_c = f_{X \wedge c}(\kappa x.v_X[x])$ . By hypothesis, we know that  $f_{X \wedge c}$  realizes  $X \rightarrow c \rightarrow Z$ , so we can conclude if  $\kappa x.v_X[x]$  realizes  $X$ .

Here is the key argument. Let us write  $n = X \wedge m$ , assuming the first argument to  $f_n$  has type  $X$ , and look closely at the reduction that produces the context  $v_X[\ ]$ . Let  $\pi_X$  be a stack in  $[X]$ . Obviously the term  $k_{\pi_X}$  realizes  $\neg X$ , so  $f_n k_{\pi_X}$  realizes  $m \rightarrow Z$ , so we have these two convergent reductions, where  $\vec{\alpha}$  is any family indexed over  $m$  such that  $\beta_L \models L$  for each literal  $L$  in  $m$ :

$$\begin{aligned} f_n * k_{\pi_X} \cdot \vec{\beta} \cdot \pi \succ k_{\pi_X} * v_X[k_{\pi_X}] \cdot \pi' \succ v_X[k_{\pi_X}] * \pi_X \\ \kappa x.v_X[x] * \pi_X \succ v_X[k_{\pi_X}] * \pi_X \end{aligned}$$

Observe that since  $\alpha_X$  is fresh when passed to  $f_n$ , the  $\alpha_X$ s in  $\alpha_X v_X[\alpha_X]$  are exactly all the residuals of that fresh  $\alpha_X$ , therefore the first reduction holds. We have  $f_n \models X \rightarrow m \rightarrow Z$ ,  $k_{\pi_X} \models X$  and  $\vec{\beta} \models m$  and the stack  $\pi$  is in  $[Z]$ , therefore the first executable is in  $\perp$ , and since  $\perp$  is closed under both reduction and anti-reduction,  $\kappa x.v_X[x] * \pi_X$  is also in  $\perp$ , thus  $\kappa x.v_X[x]$  realizes  $X$  as required.  $\square$

The import of this correctness result is that one can safely extend the  $\lambda\kappa$ -calculus with the family  $C_e$  and assign type  $\Gamma$  to  $C_e$  when  $\mathcal{C}$  is a chart over  $\Gamma$ , while keeping the coherence of the whole system, as we will prove now.

#### 4.4. Computational consistency

To this end, we extend the type system to predicate calculus. Without introducing formally all the material for this purpose, let us consider the language  $\{0, 1\}$  and define the type for boolean  $x$  as

$$B(x) = \forall X(X0 \rightarrow X1 \rightarrow Xx).$$

Then we have the following specification:

**Proposition 13.** *Let  $t$  be a term typable as  $B(x)$  for some  $x \in \{0, 1\}$ . For any couple of variables  $(u_0, u_1)$  the executable  $t * u_0 \cdot u_1 \cdot \varepsilon$  reduces into  $u_x * \varepsilon$ .*

*Proof.* Let  $u_0$  and  $u_1$  be two variables not appearing in  $t$ . Define  $\perp$  as the closure of  $\{u_x * \varepsilon\}$  by anti-reduction, which is clearly a sequential observable set. Since  $\vdash t : B(x)$  is derivable by hypothesis, adequacy proves that  $t \models B(x)$ . Define  $Xx$  to be  $\{\varepsilon\}$  and  $X(1-x)$  to be empty. Then  $u_x \models Xx$  and  $u_{1-x} \models X(1-x)$ , so  $t * u_0 \cdot u_1 \cdot \varepsilon$  is in  $\perp$ , which means that it reduces into  $u_x * \varepsilon$ .  $\square$

This proposition proves that every inhabitant of type  $B(0)$ , in any extended typing system that respects sequential observables, will behave as the first projection (and similarly for  $B(1)$ ). This fact applies to the particular system where we add our combinators.

### 5. A possible syntax

To get an idea of the meaning of the combinators  $C_e$  as programming constructs, we sketch a syntax for them in an ML-like language. On the model of constructs like the **try body with**  $Exn\ x \rightarrow handler$  of exceptions, let us write

```

sync  catch  $C_1^1 x_1, \dots, C_{p_1}^1 x_{p_1}$   throw  $T_1^1, \dots, T_{n_1}^1$   do  $t_1$ 
                                              $\vdots$ 
catch  $C_1^k x_1, \dots, C_{p_k}^k x_{p_k}$   throw  $T_1^k, \dots, T_{n_k}^k$   do  $t_k$ 

```

as a syntactic sugar around  $C_e$  applied to terms  $t_1, \dots, t_k$ . All the  $T_j^i$  and  $C_j^i$  belong to the same set of names, indexed by the variables of  $\Gamma$ :

- $T_j^i$  is understood as an exception, i.e. an object of negative type that one throws with value  $v$  by writing **raise**  $(Tv)$ , or simply  $(Tv)$ ,
- $C_j^i$  is understood as a co-exception, i.e. an object that receives a value of a positive type thrown by an associated exception of the same name  $C_j^i$  and binds it in  $t_i$  to the variable  $x_j$ .

The corresponding DNF is

$$\Gamma = \bigvee_{i=1}^k \left( \bigwedge_{j=1}^{n_k} \neg T_j^i \wedge \bigwedge_{j=1}^{p_k} C_j^i \right)$$

The corresponding chart is constructed inductively by picking the first completely negative clause (that catches nothing) and using the quotient construction. Of course, this might not succeed if  $\Gamma$  is not a tautology, and this is part of the type-checking of this structure. If it does type-check, we know from the correctness result that no exception can escape.

In a programming setting, using names in place of the propositional variables is mandatory since pure type inference cannot discover the whole structure, in particular because different propositional variables may be assigned the same type in a particular instantiation.

#### 5.1. Back to the example

Returning to the example of the twofold excluded middle (as seen in section 4.2.1), the syntax

```

sync                                     throw  $A, B$   do  $t_1$ 
catch  $Ax$                                 throw  $B$     do  $t_2$ 
catch  $Bx$                                 throw  $A$     do  $t_3$ 
catch  $Ax, Bx$                              do  $t_4$ 

```

will then represent the term

$$C_e(\lambda A \lambda B.t_1)(\lambda x \lambda B.t_2)(\lambda y \lambda A.t_3)(\lambda x \lambda y.t_4)$$

Note that with this particular tautology, there is only one possible control chart, so the order in which clauses are written is purely cosmetic.

In the simple case of the excluded middle, the syntax boils down to

```

sync  throw Exn  do body
        catch Exn x do handler

```

which is not too far from the usual **try** . . . **with** presentation, except that we are dealing here with local exceptions and that we provide strong typing.

## 6. Conclusions

We have presented here an analysis of disjunctive normal forms leading to the synthesis of a new family of multi-exception handlers which we can present also in a reasonable programming-style syntax. Free with the method, based on running Curry-Howard backwards and using Krivine’s realizability, comes the means of asserting the correctness of these combinators. The realizability approach shows really strong here in allowing us to deal with dynamic binding and to smoothly extend the typing system.

The combinators we have constructed are all based on the idea of using only negative exceptions. But, DNF always have a purely positive clause as well, not just a purely negative one. Building on this idea, it is possible to develop a completely symmetric world of co-exception based combinators, or even to mix styles, and this still needs to be explored.

Another question is whether the analogy between control charts and winning strategies can be made rigorous and whether there is a connection to Kreisel’s celebrated “no-counter-example” interpretation or other game-based explanations of classical truth.

Finally, another challenging question is whether one can do the same analysis in a call-by-value scenario and get new, clean and abstract control forms that one can prove to be correct and that would actually matter to the programmer. We think it is possible.

## References

- [1] G. Cousineau and M. Mauny. *The Functional Approach to Programming with Caml*. Cambridge University Press, 1998.
- [2] V. Danos, J.-B. Joinet, and H. Schellinx. A new deconstructive logic: Linear logic. *Journal of Symbolic Logic*, 62:755–807, 1996.
- [3] V. Danos and J.-L. Krivine. Disjunctive tautologies as synchronisation schemes. In P. Clote and H. Schwichtenberg, editors, *Proceedings of CSL’00*, number 1862 in Lecture Notes in Computer Science, pages 292–301, Fischbachau, 2000. Springer Verlag.
- [4] J.-Y. Girard. A new constructive logic: Classical logic. *Mathematical Structures in Computer Science.*, 1992.
- [5] T. G. Griffin. A formulae-as-types notion of control. In *17th Symposium on Principles of Programming Languages*, pages 47–58. ACM, Jan. 1990.
- [6] J.-L. Krivine. Typed lambda-calculus in classical Zermelo-Frænkel set theory. *Archive in Mathematical Logic*, 40(3):189–205, 2001.
- [7] J.-L. Krivine. Dependent choice, ‘quote’ and the clock. *Theoretical Computer Science*, to appear.
- [8] C.-H. L. Ong and C. A. Stewart. A Curry-Howard foundation for functional computation with control. In *Proceeding of POPL’97*, 1997.
- [9] M. Parigot. Strong normalization for second-order lambda-mu calculus. In *Proceedings of LICS’93*, 1993.