Disjunctive Normal Forms and Local Exceptions

Emmanuel Beffara
Université Paris 7, Équipe PPS
2 place Jussieu
75251 Paris Cedex 05, France
beffara@pps.jussieu.fr

Vincent Danos
Université Paris 7, Équipe PPS
2 place Jussieu
75251 Paris Cedex 05, France
danos@pps.jussieu.fr

Abstract

All classical λ-terms typable with disjunctive normal forms are shown to share a common computational behavior: they implement a local exception handling mechanism whose exact workings depend on the tautology. Equivalent and more efficient control combinators are described through a specialized sequent calculus and shown to be correct.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Functional Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*control structures*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*control primitives*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*lambda calculus and related systems*

General Terms

Theory, Languages

Keywords

Control structures, disjunctive normal forms, classical realizability

1 Introduction

We show in this paper that to each *disjunctive normal form* Γ corresponds a computational behavior shared by all terms of type Γ. An indication that disjunctive normal forms (DNFs) might be inhabited by interesting computational behaviors comes from previous work by the second author and Krivine [4] where some *disjunctive* tautologies are set in correspondence with a family of synchronization schemes. As a consequence of the intuitionistic disjunction property, none of the non trivial formulas of that class is intuitionistically valid. DNFs are actually the next simplest class with this anti-intuitionistic feature, so it seems natural to investigate these as well.

Nevertheless, one could fear that such a vast set of types as DNFs would be too loose to actually provide an interesting specification problem. But 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 behaviors, by actually extending our basic language, a call-by-name typed λ-calculus with control, with primitive instructions directly implementing these. Proving these are correct is the second contribution. We use Krivine's classical realizability (as documented in his recent papers [9, 10]) to do this.

The new computation rules are, we believe, simple enough to understand, even though they use a tamed form of 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 develop a proof-theoretic description via a specialized sequent calculus, as well as 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 [2] notation for exception handling.

It must be made clear that the control structures we consider here are not exceptions in the ML sense of the word. For one thing, all the structures we manipulate are local, in the sense that they are defined at the point where they are used and that a given exception is always associated to one specific handler. Besides, the ambient language being call-by-name, control is of quite a different and simpler nature than it is in ordinary call-by-value 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.

Another point which might need some clarification is the following: of course *any* propositional formula is classically equivalent to a DNF, but this does not mean we are giving all formulas a computational interpretation! What this says is that the equivalence between generic propositional formulas and DNFs has to be a compilation into our multi-exception handling combinators. We intend to explore the matter further.

ICFP'03, August 25–29, 2003, Uppsala, Sweden.

2 Preliminaries

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

2.1 The Calculus

Our computational framework is $\lambda\kappa$-calculus, a variant on Felleisen's λC-calculus, with a deterministic call-by-name evaluation strategy. We define Λ, the set of terms (or $\lambda\kappa$-terms), and Π, the set of stacks, as well as the set $\Lambda \times \Pi$ of executables, by

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

Stacks should be understood as evaluation contexts in call-by-name strategy: an executable $t * t_1 \cdots t_n$ is seen as the term t applied to arguments $t_1 \cdots t_n$, with the next reduction step occurring at the position of t. The evaluation relation $\succ$ over executables is defined as the reflexive transitive closure of the following set of rules:

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

The "push" and "pop" rules implement usual β-reduction while "save" and "restore" provide the control mechanism. The term $\kappa x.t$ captures the current evaluation context π as a term k_π, and applying the latter restores this context. Binders λx and κx are thus dual in the sense that one substitutes terms while the other substitutes stacks, in the form of the terms k_π 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:

$$\frac{}{\Gamma, x : A \vdash x : A}\,[\text{axiom}]$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}\,[\to\text{intro}] \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B}\,[\to\text{elim}]$$

$$\frac{\Gamma \vdash t : A \qquad X \notin FV(\Gamma)}{\Gamma \vdash t : \forall X A}\,[\forall\text{intro}] \qquad \frac{\Gamma \vdash t : \forall X A}{\Gamma \vdash t : A[B/X]}\,[\forall\text{elim}]$$

These first five rules give a standard presentation, known as *natural deduction*, for second order propositional intuitionistic logic. The typing rule for the κ binder is

$$\frac{\Gamma, x : A \to B \vdash t : A}{\Gamma \vdash \kappa x.t : A}\,[\text{Peirce}]$$

With this sixth rule, known as *Peirce's law*, we get one possible natural deduction presentation of second order propositional classical logic, as it was first noted by Griffin [6] in the context of Scheme with a call/cc operator.

Note that we do not provide any typing rule for continuations k_π. A reason for this is that it would require giving types to stacks. This would be possible but it would lead to unnecessary complication here. Another deeper reason is that typable terms are seen as proofs of their type, and this interpretation would not make sense anymore when writing continuations explicitly. From a computational point of view, typable terms are programs that should be able to be executed in any context, and this forbids them to contain explicit ones.

As usual we define the absurdity as $\bot := \forall X\, X$ and the negation as $\neg A := A \to \bot$. All types considered later on live actually in a fragment of this system that corresponds to simple types augmented with the $\bot$ constant, so types are implicitly considered universally quantified in every variable.

2.3 Realizability

Realizability builds polarized models of the propositional logic above by associating truth values to each formula. Positive truth values are sets of terms while negative ones are sets of stacks. These models are parametrized by a given *observation*: let $\perp\!\!\!\perp \subseteq \Lambda \times \Pi$ be a set of executables closed by anti-reduction (i.e. if $e \succ e'$ and $e' \in \perp\!\!\!\perp$ then $e \in \perp\!\!\!\perp$). The executables in this set are called the *observables* and they are said to realize the absurd. In the definitions below, we assume such a $\perp\!\!\!\perp$ has been chosen.

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

We associate truth values to the types of our system. For a given type A, the positive value $|A|$ will be, intuitively, the set of terms that behave well as members of this type. The corresponding negative value $[A]$, orthogonal to $|A|$, will contain the contexts, i.e. the stacks, where terms of type A behave well.

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

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

where $e[\mathcal{Z}/X]$ is the environment e where value $\mathcal{Z}$ has been assigned to X. These definitions correspond to the semantic meaning of the types. 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 positive truth value $|A|_e = [A]_e^{\perp\!\!\!\perp}$, i.e. the set of terms orthogonal to $[A]_e$, and we say that a term t *realizes* a type A (in environment e), which we write $t \vDash_e A$, if t is in the interpretation of A. We write this $t \vDash A$ when A has no free variable.

The value of some particular types is worth explaining. For the absurd $\bot = \forall X\, X$, the negative value is the whole set of stacks, i.e. $[\bot] = \Pi$; the associated positive value $|\bot|$ is thus the set of terms t such that for any stack π the executable $t * \pi$ is in $\perp\!\!\!\perp$, that is terms who reduce into $\perp\!\!\!\perp$ whatever the context. It is also interesting to describe the values of negated types: the negative value $[\neg X] = [X \to \bot]$ turns out to be the positive value $|X|$ concatenated with Π. Subsequently, if π is a stack in $[X]$ and t is a term in $|X|$, then $t * \pi$ is in $\perp\!\!\!\perp$ by definition, and anti-reduction implies that $k_\pi * t \cdot \pi'$ is also

in $\perp\!\!\!\perp$ for any π', which means that $\pi \in [X]$ implies $k_\pi \Vdash \neg X$. In turn, a consequence is that $k_\pi t$ realizes $\perp$, and actually it is more or less an executable on its own.

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

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

This theorem is standard, but we include its proof anyway, for coherence and as an illustration of the kind of reasoning we will use later on.

PROOF. We proceed by induction on the typing derivation. Call Γ the typing environment $x_1 : A_1, \ldots, x_n : A_n$ 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 $\overline{t}$ the substituted term $t[t_1/x_1, \ldots, t_n/x_n]$.

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

application: Let t and u be terms such that $\Gamma \vdash t : A \to B$ and $\Gamma \vdash u : A$ are derivable. By induction we have $\overline{t} \in |A \to B|_e$ and $\overline{u} \in |A|_e$, so $\overline{u} \cdot \pi$ is in $|A \to B|_e \cdot [B]_e = [A \to B]_e$, therefore $\overline{t} * \overline{u} \cdot \pi$ is in $\perp\!\!\!\perp$, and so is $\overline{tu} * \pi$ since $\perp\!\!\!\perp$ 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 $\overline{t}[u/x] * \pi$ is in $\perp\!\!\!\perp$, and so is $\overline{\lambda x.t} * u \cdot \pi$ by anti-reduction, so we can actually deduce that $\overline{\lambda x.t}$ is orthogonal to $|A|_e \cdot [B]_e = [A \to B]_e$.

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

quantification: Write $B = \forall X\, A$ and suppose that $\Gamma \vdash t : A$ is derivable. From the definition of $[\forall X A]_e$ we deduce the existence of a stack set $\mathcal{Z}$ for which $[A]_{e[\mathcal{Z}/X]}$ contains π. 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 $t_i \in |A_i|_{e[\mathcal{Z}/X]}$, and therefore $\overline{t} * \pi$ is in $\perp\!\!\!\perp$ 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 $[A]_{e[\mathcal{Z}/X]}$ we have $t * \pi \in \perp\!\!\!\perp$. 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. □

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

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

The idea of realizability is derived from Girard's concept of reducibility candidate, and it can actually be used to prove normalization results. Once the adequacy lemma is established, these proofs derive easily, even in our context where the calculus is enriched with global control.

Adequacy is also the key argument in our specification proofs. The general idea is to define $\perp\!\!\!\perp$ as the closure by anti-reduction of a given executable e. That a set of terms X and a set of stacks Z are orthogonal values in the resulting model then means that applying a term in X to a stack in Z gives an executable that reduces into e.

As an example, let us consider the following specification for Church booleans. For each $b \in \{0,1\}$, let us define the type for boolean b as

$$B_b = \forall X_0 \forall X_1 (X_0 \to X_1 \to X_b)$$

Then we have the following specification:

PROPOSITION 3. *Let t be a term typable as B_b for some $b \in \{0,1\}$. For any couple of terms (u_0, u_1) and any stack π, the executable $t * u_0 \cdot u_1 \cdot \pi$ reduces into $u_b * \pi$.*

PROOF. Let u_0 and u_1 be two terms and π be a stack. Define $\perp\!\!\!\perp$ as the closure of $\{u_b * \pi\}$ by anti-reduction. Since $\vdash t : B_b$ is derivable by hypothesis, adequacy proves that $t \Vdash B_b$. As a consequence, for any environment $e : Var \to \mathcal{P}(\Pi)$, i.e. for any value of $e(X_0)$ and $e(X_1)$, we have $t \Vdash_e X_0 \to X_1 \to X_b$. Define $e(X_b)$ as $\{\pi\}$ and $e(X_{1-b})$ as the empty set. Then $u_b \Vdash X_b$ by definition of $\perp\!\!\!\perp$ and trivially $u_{1-b} \Vdash X_{1-b}$, so the stack $u_0 \cdot u_1 \cdot \pi$ is in $[X_0 \to X_1 \to X_b]_e$, and subsequently $t * u_0 \cdot u_1 \cdot \pi$ is in $\perp\!\!\!\perp$, which means that it reduces into $u_b * \pi$. □

Of course, this result could be obtained by a direct proof, using subject reduction arguments. However, the proof above, though voluntarily detailed, is very simple. The pattern is always the same: carefully choose the observation $\perp\!\!\!\perp$ and the value of type variables (thanks to universal quantification), and let the adequacy do the work. We will use the same statement and nearly the same proof in proposition 12 as a consistency argument for the calculus enriched with control combinators.

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 set $\mathcal{P}_{\text{fin}}(Lit)$ of finite parts of *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 $\to$ as the only connector, so we must define how DNFs are converted into types. For any type τ and any ordered clause $\{L_1, \ldots, L_k\}$ we define:

$$\{L_1, \ldots, L_k\} \to \tau \;:=\; L_1 \to \cdots \to L_k \to \tau$$

The ordering we use will be either clear from the context or indifferent. Likewise, given a formula $\Gamma = \{c_1, \ldots, c_n\}$ and a fresh variable

Z, we interpret Γ as a type by defining

$$\Gamma := \forall Z\,(c_1 \to Z) \to \cdots \to (c_n \to Z) \to Z$$

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

3.1 A First Specification

Our purpose now is to identify a computational behavior 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. Let $\Gamma = c_1 \vee \cdots \vee c_n$ be a disjunctive normal form. A section of Γ is an element of the product $c_1 \times \cdots \times c_n$.

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 4. *A disjunctive normal form Γ is a tautology if and only if every section of Γ contains two opposite literals.*

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

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

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. Let Γ be a DNF tautology, let t be a term of type Γ and $\vec{f}$ be a sequence of arguments, with $c \in \Gamma$. What t does is to pass each f_c a freshly created set of exceptions, indexed by the literals in c. If *all* of the f_c raise one of the exceptions they were given, then t selects two of these with opposite types X and $\neg X$ and runs their arguments one against the other. That there always must be two opposite exceptions is precisely what the proposition above says, and therefore the arguments have compatible types.

Note, in the statement of the theorem below, that there is no mention of exceptions. What we assume is that the terms f_c apply one of their arguments, whatever it is. The intuition about exceptions is only a way of saying, and it will be formalized only in the next section. In the theorem, we write $\vec{f}$ to represent a family of terms indexed over Γ and f_c for the element of index c in this family, similarly we write $\vec{u}$ to represent any family of terms indexed over the literals in a given clause of Γ and u_L for members of such a family.

THEOREM 5. *Let Γ be a tautology in disjunctive normal form and let t be a $\lambda\kappa$-term of type Γ. Let π be a stack and $\vec{f}$ a family of terms indexed by the clauses of Γ. Suppose there exists a section σ of Γ and a family of terms v_c and stacks π_c such that for each clause c,*

$$\sigma(c) \in \neg Var \Rightarrow \forall \vec{u}\ \exists \pi' \qquad f_c * \vec{u} \cdot \pi \succ u_{\sigma(c)} * v_c \cdot \pi'$$
$$\sigma(c) \in Var \Rightarrow \forall \vec{u} \qquad f_c * \vec{u} \cdot \pi \succ u_{\sigma(c)} * \pi_c$$

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

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

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

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

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 Γ into a type and X ranges over variables occurring in Γ.

Let c be a clause in Γ^-. For any family (u_L) of terms such that $u_L \vDash \neg L$ for each literal $L \in c$, by hypothesis, $f_c * \vec{u} \cdot \pi$ reduces into $u_{\sigma(c)} * v_c \cdot \pi'$ for some π'. Here $\sigma(c)$ is a negative literal $\neg X$, and the term v_c realizes X since for each element $\pi_{c'}$ in $[X]$ we have $\sigma(c) = \neg\sigma(c')$ and so $v_c * \pi_{c'} \in \perp\!\!\!\perp$, therefore $u_{\sigma(c)} * v_c \cdot \pi'$ is in $\perp\!\!\!\perp$, so is $f_c * \vec{u} \cdot \cdot \pi$ by anti-reduction, and so $f_c \vDash c \to Z$.

Similarly, let c' be a clause in Γ^+. For any family $\vec{u}$ of terms such that $u_L \vDash L$ for each clause $L \in c'$, by hypothesis $f_{c'} * \vec{u} \cdot \pi$ reduces into $a_{\sigma(c')} * \pi_{c'}$ where $\sigma(c')$ is some variable X. This executable is in $\perp\!\!\!\perp$ since $\pi_{c'} \in [\sigma(c')]$, therefore $f_{c'} \vDash c' \to Z$.

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

3.2 Sharper Specifications

The specification extracted above is actually quite shallow. There is an important restriction, namely that the values v_c and the stacks π_c that the processes pass when raising exceptions may not contain any occurrence of the functions' arguments (the u_L in the quantifications). We can do better by relaxing the conditions on the arguments, i.e. by assuming that there exists a family of parametrized terms $v_c[\,]$ and parametrized stacks $\pi_c[\,]$ such that, for each clause c,

$$\sigma(c) \in \neg Var \Rightarrow \forall \vec{u}\ \exists \pi' \qquad f_c * \vec{u} \cdot \pi \succ u_{\sigma(c)} * v_c[\vec{u}] \cdot \pi'$$
$$\sigma(c) \in Var \Rightarrow \forall \vec{u} \qquad f_c * \vec{u} \cdot \pi \succ u_{\sigma(c)} * \pi_c[\vec{u}]$$

and proving that the reduction of $t * \vec{f} \cdot \pi$ leads to $v_c[\vec{a}] * \pi_{c'}[\vec{b}]$ for some families of terms $\vec{a}$ and $\vec{b}$. Proving just this is mainly the same as proving theorem 5. However, proving something about the terms in the families $\vec{a}$ and $\vec{b}$ is harder.

As an illustration, let us consider he particular case of the excluded middle, i.e. $\Gamma = \neg X \vee X$. Its translation as a type, as explained above, is $(\neg X \to Y) \to (X \to Y) \to Y$. Here the following development result holds:

PROPOSITION 6. *Let t be a term of type $\neg X \vee X$. Let π be a stack and let f and g be two terms. Suppose there is a stack π_g and a*

family of contexts $(v_i[])_{0\leq i\leq n}$ *such that*

$$\begin{array}{ll} \forall u\ \exists \pi' & f*u\cdot\pi \succ u*v_0[u]\cdot\pi' \\ \forall u & g*u\cdot\pi \succ u*\pi_g \\ \forall i<n\ \forall u\ \exists \pi' & v_i[u]*\pi_g \succ u*v_{i+1}[u]\cdot\pi' \end{array}$$

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

PROOF. Define $\perp\!\!\!\perp$ as the closure by anti-reduction of

$$\{v_n[u]*\pi_g \mid u\in\Lambda\}$$

and instantiate the propositional variables as

$$[X] := \{\pi_g\} \quad \text{and} \quad [Y] := \{\pi\}.$$

Then obviously for any term u the we have $v_n[u]\Vdash X$.

Let i be an integer with $0\leq i\leq n-1$. Assume that for any term u in $|\neg X|$ the term $v_{i+1}[u]$ realizes X. Then for any such u and any stack π', the executable $u*v_{i+1}[u]\cdot\pi'$ is in $\perp\!\!\!\perp$, therefore $v_i[u]*\pi_g$ is also in $\perp\!\!\!\perp$, by anti-reduction using the third hypothesis. As a consequence, for any u, $u\Vdash\neg X$ implies $v_i[u]\Vdash X$. By recurrence, we thus have this property for the context $v_0[]$.

For any $u\Vdash\neg X$ and any stack π', $u*v_0[u]\cdot\pi'$ is in $\perp\!\!\!\perp$ as we just proved, so by anti-reduction $f*u\cdot\pi$ is also in $\perp\!\!\!\perp$, therefore f realizes $\neg X\to Y$. Besides, one trivially proves that g realizes $X\to Y$, and t realizes $(\neg X\to Y)\to(X\to Y)\to Y$ by the adequacy lemma, so $t*f\cdot g\cdot\pi$ is in $\perp\!\!\!\perp$, which means that it reduces into $v_n[u]*\pi_g$ for some term u. □

This allows values passed to exceptions to contain occurrences of exceptions (however without lifting the constraint on the stacks). The specification means that the reduction of $t*f\cdot g\cdot\pi$ leads to something equivalent to $v_0[k_{\pi_g}]$, i.e. that the terms the combinator puts inside the contexts $v_i[]$ behave like a continuation k_{π_g}. This can be understood as the fact that the exception X can be raised several times and that it will always be caught the same way by g.

Lifting also the constraint on stacks is harder, in particular it seems to impose some restrictions on the allowed observations, though we did not get a proof of this. These results could probably be extended to the general case, but the task seems notationally daunting.

4 Synthesizing Combinators

Of course all DNF tautologies are provable in our system, and therefore one can find λκ-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:

variables	$X\in Var$
literals	$L ::= X \mid \neg X$
clauses	$c ::= \top \mid L\wedge\ldots\wedge L$
formulas	$\Gamma ::= \bot \mid c,\ldots,c$
sequents	$\Vdash\Gamma$

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

$$\frac{\Vdash\Gamma_1,\Delta \qquad \cdots \qquad \Vdash\Gamma_n,\Delta}{\Vdash(\neg X_1\wedge\cdots\wedge\neg X_n),(X_1\wedge\Gamma_1),\ldots,(X_n\wedge\Gamma_n),\Delta}$$

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

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

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 7. *Every tautology in disjunctive normal form has a clause with only negative literals.*

PROOF. Let Γ be a DNF. Suppose that Γ has no such clause, then each clause has at least one positive literal, which defines a section with only positive literals. From proposition 4, we conclude that Γ is not a tautology. □

We also need the following notion of quotient:

Definition 2. Let Γ be a disjunctive normal form and X be a variable. The quotient of Γ 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 Γ under the assumption that variable X is true. Indeed we have the following consistency result:

PROPOSITION 8. *Let* Γ *be a disjunctive normal form and* X *be a variable. If* Γ *is a tautology, so is* Γ/X.

PROOF. By construction, Γ/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 Γ is a tautology, $v'(\Gamma)$ is true, so there is a clause c in Γ such that $v'(c)=\top$. Since $v'(X)=\top$, c does not contain $\neg X$, so $c\setminus\{X\}$ is a clause of Γ/X, and $v(c\setminus\{X\})=\top$, so v validates Γ/X. □

PROPOSITION 9 (COMPLETENESS). *Let* Γ *be a disjunctive normal form. If* Γ *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 Δ the sequent $\Vdash$

Γ, Δ is derivable in $\mathcal{L}_{\vee\wedge}$. We proceed by induction on the number of variables in Γ: if there is no variable, Γ is reduced to one trivial clause $\top$, and $\Vdash \top, \Delta$ is derived using the nullary rule. Otherwise, lemma 7 proves that Γ has a totally negative clause, so we can write

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

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

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

The union of these two formulas is actually the quotient Γ/X_i, and from proposition 8 we know that the formula Γ_i, Δ_i is a tautology. Since it contains strictly fewer variables than Γ, the induction hypothesis proves that the sequent $\Vdash \Gamma_i, \Delta_i, \Gamma', \Delta$ is derivable. Besides, from this definition, obviously Δ_i is a subset of Γ', 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 Γ', we can write

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

which concludes the proof. □

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

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

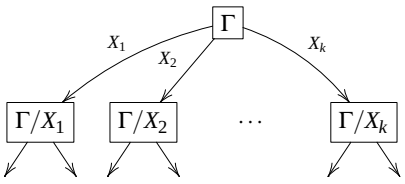
$$\frac{\Vdash \Gamma_1, \Delta \qquad \cdots \qquad \Vdash \Gamma_n, \Delta}{\Vdash (\neg X_1 \wedge \cdots \wedge \neg X_n), (X_1 \wedge \Gamma_1), \ldots, (X_n \wedge \Gamma_n), \Delta}$$

for some $n \geq 1$, so σ 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 Γ_i, Δ, which contains opposite literals by induction hypothesis. □

As said, none of these tautologies is intuitionistically valid, except in the particular case where they contain the trivial clause $\top$, therefore any behavior 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 Γ, the conclusion contains a distinguished totally negative clause $\neg X_1 \wedge \cdots \wedge \neg X_k$. 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 Γ is equivalent to Γ/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:

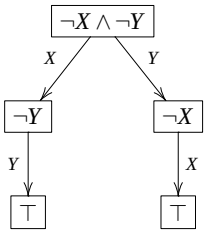
As illustrated below, it suffices to write the distinguished totally negative clause in each node, then the charts are read down from the top. If the root node contains a non-trivial clause $\neg X_1 \wedge \cdots \wedge \neg X_k$, then play it. If the opponent refutes the literal $\neg X_i$ with a value a_i of type X_i, then follow the edge labeled X_i and remember the value a_i. By construction, when reaching a node labeled $\top$, the opponent cannot refute.

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, according to the informal description above:



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

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

The family $\vec{\alpha}$ above corresponds to the edges in the chart, while $\vec{f}$ corresponds to the nodes. This definition is rather informal because it hides information: the terms α_c 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 π, as well as an a or a b in the case of α_{XY} and α_{YX}. For instance, α_{XY} should be written like $\alpha_{XY,\Gamma,\vec{f},\pi,a}$ but we avoid this notation for readability.

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

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

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

The use of such rebinding is not purely syntactical, as it corresponds to what the specification in proposition 6 meant: an exception $\alpha_X a$ is always caught the same way, by calling the function $f_{X\wedge\neg Y}$ with a as argument on the stack π, and if this leads to $a*\pi_X$ then applying α_X to another value a' will also lead to $a'*\pi_X$, so replacing α_X by k_{π_X} will behave the same way. This is the argument for the introduction of $\kappa\alpha_X.a$, as we will see in the proof of theorem 11 below.

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 Γ, starting with clause $c=\neg X_1\wedge\cdots\wedge\neg X_k$ and with immediate sub-charts $\mathcal{C}_{X_1},\ldots,\mathcal{C}_{X_k}$, we define $\mathbf{C}_\mathcal{C}$ inductively:

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

$$\alpha_{X_i}*v\cdot\pi'\succ\mathbf{C}_{\mathcal{C}_{X_i}}*\vec{f}'\cdot\pi \tag{2}$$

where the α_{X_i} are new language constructs (and not variables, for instance), and where $\vec{f}'$ is indexed on the clauses of Γ/X_i and defined as

$$f'_c=f_c \qquad \text{if } c\in\Gamma \tag{3}$$

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

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

Note that the notation α_{X_i} is not formally enough, because the new rules imply that these objects have a memory of the chart $\mathcal{C}$ and the terms f_c. To be precise, we would need to put all required information in the indices, however this would lead to a heavy notation that we avoid by making them implicit.

In a way, the $\kappa x.v[x/\alpha_{X_i}]$ represents what has been learned from raising the exception α_{X_i}, and we see that this exception will never be raised again, since α_{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. An observable set $\perp\!\!\!\perp$ 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 $e[t]\in\perp\!\!\!\perp$ and if $e[x]$ does not reduce into an $x*\pi$, then $e[u]\in\perp\!\!\!\perp$ for all terms u.

These conditions arise rather naturally in the correctness proof. Besides, the observables used in various specification theorems are all of this form. Therefore, these specifications will still hold in our enriched calculus. We examine below the simple example of Church booleans. Another example, that of computational consistency in the presence of arithmetic constructs, is given in related work of the first author [1].

THEOREM 11. *Let Γ be a tautology in disjunctive normal form and $\mathcal{C}$ be a chart over Γ. For any sequential $\perp\!\!\!\perp$, the combinator $\mathbf{C}_\mathcal{C}$ realizes the type Γ.*

PROOF. We proceed by induction on the height of the chart $\mathcal{C}$. When $\mathcal{C}$ is a leaf, we have $\Gamma=\top,c_1,\ldots,c_n$, which is the type $Z\to(c_1\to Z)\to\cdots\to(c_n\to Z)\to Z$. 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 $\perp\!\!\!\perp$ and any value for $[Z]$, if $f_\top\Vdash Z$ then for any π in $[Z]$ we have $\mathbf{C}_\mathcal{C}*\vec{f}\cdot\pi\succ f_\top*\pi\in\perp\!\!\!\perp$, thus $\mathbf{C}_\mathcal{C}\Vdash\Gamma$.

Suppose $\mathcal{C}$ starts with a negative clause $n=\neg X_1\wedge\cdots\wedge\neg X_k$ and let $\perp\!\!\!\perp$ 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 Γ, let f_c be a term that realizes $c\to Z$. 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 $\perp\!\!\!\perp$ we can conclude that $\mathbf{C}_\mathcal{C}$ realizes Γ.

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

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

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

using the notations of section 4.2.2, thus proving that the third executable is in $\perp\!\!\!\perp$ is enough to conclude. By induction hypothesis, the combinator $\mathbf{C}_{\mathcal{C}_X}$ realizes Γ/X, so we have to prove that f'_c realizes $c\to Z$ for each c in Γ/X. If c is a clause of Γ, we have $f'_c=f_c$ and $f_c\Vdash c\to Z$ by hypothesis. Otherwise $X\wedge c$ is a clause of Γ and $f'_c=f_{X\wedge c}(\kappa x.v_X[x])$. By hypothesis, we know that $f_{X\wedge c}$ realizes $X\to c\to Z$, 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 π_X be a stack in $[X]$. Obviously the term k_{π_X} realizes $\neg X$, so $f_nk_{\pi_X}$ realizes $m\to Z$, so we have these two convergent reductions, where $\vec{\alpha}$ is any family indexed over m such that $\beta_L\Vdash L$ for each literal L in m:

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

Observe that since α_X is fresh when passed to f_n, the instances of α_X in $\alpha_Xv_X[\alpha_X]$ are exactly all the residuals of that fresh α_X, there-

fore the first reduction holds. We have $f_n \Vdash X \to m \to Z$, $k_{\pi_X} \Vdash X$ and $\vec{\beta} \Vdash m$ and the stack π is in $[Z]$, therefore the first executable is in $\perp\!\!\!\perp$, and since $\perp\!\!\!\perp$ is closed under both reduction and anti-reduction, $\kappa x.v_X[x] * \pi_X$ is also in $\perp\!\!\!\perp$, thus $\kappa x.v_X[x]$ realizes X as required. □

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

4.4 Computational Consistency

Showing that new combinators integrate well in the tying system is good thing, and showing that the calculus itself still makes sense is also useful. To this end, we show that some non-trivial specification result, namely the one for Church booleans exposed in proposition 3, still holds in the enriched calculus. For each $b \in \{0,1\}$, we defined the type boolean b as

$$B_b = \forall X_0 \forall X_1 (X_0 \to X_1 \to X_b)$$

Then we have the following specification:

PROPOSITION 12. *Let t be a term typable as B_b for some $b \in \{0,1\}$. For any couple of variables (u_0,u_1) the executable $t * u_0 \cdot u_1 \cdot \varepsilon$ reduces into $u_b * \varepsilon$.*

PROOF. Let u_0 and u_1 be two variables not appearing in t. Define $\perp\!\!\!\perp$ as the closure of $\{u_b * \varepsilon\}$ by anti-reduction, which is clearly a sequential observable set. Since $\vdash t : B_b$ is derivable by hypothesis, adequacy proves that $t \Vdash B_b$. Define $[X_b]$ to be $\{\varepsilon\}$ and $[X_{1-b}]$ to be empty. Then $u_b \Vdash X_b$ and $u_{1-b} \Vdash X_{1-b}$, so $t * u_0 \cdot u_1 \cdot \varepsilon$ is in $\perp\!\!\!\perp$, which means that it reduces into $u_b * \varepsilon$. □

This proposition, as compared to its previous statement, deals with variables as arguments and requires an empty stack. The reason for this is that the closure by anti-reduction of $\{u_b * \varepsilon\}$ is then a sequential observation. Stating a general result would require to define $\perp\!\!\!\perp$ as the smallest sequential observation that contains $u_b * \pi$ (in the notations of proposition 3). The conclusion would be that $t * u_0 \cdot u_1 \cdot \pi$ reduces not into $u_b * \pi$, but into something observationally equivalent, namely a reduction of it, potentially with a substitution performed on terms that never reach the head.

However, by reasoning about the stability of variable substitution with respect to the reduction relation, we can get back to the original specification. We will not formalize this, because it is beyond the point of our remark, and because it would require some amount of work, in particular the introduction of stack variables.

5 A Possible Syntax

To get an idea of the meaning of the combinators $\mathbf{C}_\mathcal{C}$ 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* → *handler* of exceptions, let us write

$$\begin{array}{llll} \textbf{sync} & \textbf{catch}\, C_1^1 x_1, \ldots, C_{p_1}^1 x_{p_1} & \textbf{throw}\, T_1^1, \ldots, T_{n_1}^1 & \textbf{do}\, t_1 \\ & \vdots & & \\ & \textbf{catch}\, C_1^k x_1, \ldots, C_{p_k}^k x_{p_k} & \textbf{throw}\, T_1^k, \ldots, T_{n_k}^k & \textbf{do}\, t_k \end{array}$$

as a syntactic sugar around $\mathbf{C}_\mathcal{C}$ applied to terms $t_1, \ldots, t_k$. All the T_j^i and C_j^i belong to the same set of names, indexed by the variables of Γ:

- T_j^i is understood as an exception, i.e. an object of negative type that one throws with value v by writing **raise** $(T\,v)$, or simply $(T\,v)$,
- $C_j^i x_j$ 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

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

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 Γ 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

$$\begin{array}{llll} \textbf{sync} & & \textbf{throw}\, A, B & \textbf{do}\, t_1 \\ & \textbf{catch}\, Ax & \textbf{throw}\, B & \textbf{do}\, t_2 \\ & \textbf{catch}\, By & \textbf{throw}\, A & \textbf{do}\, t_3 \\ & \textbf{catch}\, Ax, By & & \textbf{do}\, t_4 \end{array}$$

will then represent the term

$$\mathbf{C}_\mathcal{C}(\lambda A \lambda B.t_1)(\lambda x \lambda B.t_2)(\lambda y \lambda A.t_3)(\lambda x \lambda y.t_4).$$

The semantics of this construct will be to execute t_1 and return its return value, unless it raises one of the exceptions. If the execution of t_1 leads to **raise** (Aa), then exception A is caught; the value a is sent to all the cases that are able to catch A (here t_2 and t_4) and those that throw A are discarded (here t_3, and of course t_1). The result is that the execution continues with

$$\begin{array}{llll} \textbf{sync} & \textbf{throw}\, B & \textbf{do}\; t_2[a/x] \\ & \textbf{catch}\, By & \textbf{do}\; t_4[a/x] \end{array}$$

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

As shown in the reduction above, in the simple case of the excluded middle, the syntax boils down to

$$\begin{array}{lll} \textbf{sync} & \textbf{throw}\, Exn & \textbf{do}\, body \\ & \textbf{catch}\, Exn\, x & \textbf{do}\, handler \end{array}$$

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

6 Conclusions

Information flows mostly from programming language design to theory and not in the other direction. The story of control oper-

ators, of how they were discovered to correspond to Peirce's law by Griffin [6] and Felleisen, and only later were put on the logic workbench [5, 6, 12, 13, 3], makes this absolutely clear.

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 a controlled form of 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 famous "no-counter-example" interpretation or other game-based explanations of classical truth.

The theory of realizability shows to smoothly extend to more sophisticated calculi, with constants and data types [1]. This suggests that our results will still hold in settings closer to real programming languages, the first example being that of PCF with control. On the way towards realistic programming settings, extensions of λ-calculus with other forms of control, like partial continuations [7], catch/throw [11, 8] could be studied with our techniques.

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 be more meaningful for an actual programming language. We think it is possible.

7 References

[1] E. Beffara. Realizability with constants. Workshop on Formal Methods and Security, Nanjing, China, 2003.

[2] G. Cousineau and M. Mauny. *The Functional Approach to Programming with Caml*. Cambridge University Press, 1998.

[3] V. Danos, J.-B. Joinet, and H. Schellinx. A new deconstructive logic: Linear logic. *Journal of Symbolic Logic*, 62:755–807, 1996.

[4] 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.

[5] J.-Y. Girard. A new constructive logic: Classical logic. *Mathematical Structures in Computer Science.*, 1992.

[6] T. G. Griffin. A formulae-as-types notion of control. In *17th Symposium on Principles of Programming Languages*, pages 47–58. ACM, Jan. 1990.

[7] Y. Kameyama. A type-theoretic study on partial continuations. In J. van Leeuwen, O. Watanabe, M. Hagiya, P. D. Mosses, and T. Ito, editors, *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics*, volume 1872 of *Lecture Notes in Computer Science*, pages 489–504. Springer Verlag, 2000.

[8] Y. Kameyama and M. Sato. Strong normalizability of the nondeterministic catch/throw calculi. *Theoretical Computer Science*, 272(1–2):223–245, 2002.

[9] J.-L. Krivine. Typed lambda-calculus in classical Zermelo-Frænkel set theory. *Archive in Mathematical Logic*, 40(3):189–205, 2001.

[10] J.-L. Krivine. Dependent choice, 'quote' and the clock. *Theoretical Computer Science*, to appear.

[11] H. Nakano. A constructive formalization of the catch and throw mechanism. In *Symposium on Logic in Computer Science (LICS'92)*, pages 82–89, Santa Cruz, California, 1992.

[12] C.-H. L. Ong and C. A. Stewart. A Curry-Howard foundation for functional computation with control. In *Proceeding of POPL'97*, 1997.

[13] M. Parigot. Strong normalization for second-order lambda-mu calculus. In *Proceedings of LICS'93*, 1993.