Course Notes
"An Introduction to Proof Equivalence"

ESSLLI 2023

Classes 9-10: 11/08.

Matteo Acclavio & Paolo Pistone

Today we are going to talk about two things. First, we will discuss what we know (and especially what we do not know) about proof equivalence in presence of first-order and second order quantifiers. Second, we will discuss how the problem of proof equivalence is related to some important questions in the study of programming languages. This connection is particularly relevant for the case of second-order logic.

1 Proof Equivalence and Program Equivalence

I guess many of you have been following Giulio Guerrieri's course and thus you should now have a grasp of the Curry-Howard correspondence. In very rough terms, it is a correspondence which makes it possible to see proofs in natural deduction as functional programs, and normalization of proofs as execution of programs. More precisely, a formula of intuitionistic logic is read as a *type*, that is, a set of programs having a somehow similar behavior, and a proof of that formula yields then a program having the associated type. For instance, the formula $p \to (p \to p)$ corresponds to the type of all programs sending two inputs of type p into an input of type p. The two derivations that we have seen some days ago, namely:

$$\Pi_1 \quad = \quad \dfrac{\dfrac{\overset{0}{p}}{p \to p}\,\to\mathrm{I}(1)}{p \to (p \to p)}\,\to\mathrm{I}(0) \qquad\qquad \Pi_2 \quad = \quad \dfrac{\dfrac{\overset{0}{p}}{p \to p}\,\to\mathrm{I}(0)}{p \to (p \to p)}\,\to\mathrm{I}(1)$$

correspond to two programs of type $p \to (p \to p)$: the first program takes the two inputs x, y of type p and gives back the first one; the second program takes the two inputs and gives back the second one. For those of you who are now familiar with the λ-calculus, these two programs can be written as

$$P_1 := \lambda x.\lambda y.x \qquad\qquad P_2 := \lambda x.\lambda y.y$$

In this way, from the study of proofs and their representation by means of formal derivations, we pass to the study of *algorithms* and their implementation by means of *programs*, that is, formal entities (e.g. lines of code) written in a certain language. λ-calculus is one example of formal language. In fact, when considering programs, there are two different senses in which we can reason about their equivalence:

- a *coarser* sense: intuitively, a program, when executed, implements some form of *function* between datatypes. For example, a program that executes natural number addition and a program that checks the primality of a natural number implement functions

$$\texttt{ADDITION} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$
$$\texttt{ISPRIME} : \mathbb{N} \longrightarrow \{\texttt{TRUE}, \texttt{FALSE}\}$$

  However, there may exist different programs that implement *the same* function. For example, the following two programs both implement the addition function:

```
int add1(int x, int y) {
    for (i=0; i< x, i++){
    x=x+1;
    }
    return x;
}
```

```
int add2(int x, int y) {
    for (i=0; i< y, i++){
    y=y+1;
    }
    return y;
}
```

  Primality can be checked by a program that iteratively checks that n is not divisible by any $p < n$. Observe that this program runs in exponential time. Indeed, the existence of a polytime algorithm to check primality remained open for many years, and was solved only in 2002 via the so-called AKS algorithm. *Program equivalence* is, informally, the property enjoyed by programs that, when executed, implement the same function. Observe that program equivalence is an extensional relation, and in particular it *may break complexity* properties: the exponential time primality check is equivalent to the polytime program AKS.

- a *finer* sense: an algorithm is a procedure which is usually presented informally, or via some pseudo-code. The same algorithm can be implemented via programs written in different languages. Often, the same algorithm can also be implemented by different programs in the same programming language. Typically, if your algorithms includes two independent `for` loops, it may lead to different equivalent formulations, as illustrated below:

```
int i, a[100], b[100];
for (i = 0; i < 100; i++) {
    a[i] = 1;
    b[i] = 2;
}
```

```
int i, a[100], b[100];
for (i = 0; i < 100; i++) {
    a[i] = 1;
}
for (i = 0; i < 100; i++) {
    b[i] = 2;
}
```

  Observe that this is roughly very similar to the permutation problem for the sequent calculus. In general, algorithmic equivalence should *not break complexity*: as soon as we see complexity as a property of algorithms, two programs with different complexity should implement different algorithms.

The difference between program equivalence and algorithmic equivalence resembles somehow the one between *global* and *local* notions of proof equivalence. Yet, there is an important difference between these two: being extensional, program equivalence admits a precise and well-understood definition, while algorithmic equivalence remains today, much like the notion of proof equivalence, a rather elusive one. In the following, we will thus mostly focus on program equivalence.

In Wednesday's class we discuss the typed Böhm theorem, which asserts that normalization-equivalence is a maximal non-trivial equational theory. The core of the argument was showing that two different normal forms Σ_1, Σ_2 in $\mathbf{NJ}_{\to,\wedge}$ are *separable*: one can construct a *context* $\mathtt{C}[-]$ (intuitively, a derivation with a "hole") such that $\mathtt{C}[\Sigma_1]$ reduces to Π_1, and $\mathtt{C}[\Sigma_2]$ reduces to Π_2 (where Π_1, Π_2 are the two derivations above. This idea of separability is at the heart of the definition of program equivalence, via the following notion:

Definition 1 (contextual equivalence)**.** *Two programs M, N of the same type A are* contextually equivalent *when for any context $\mathtt{C}[-] : A \to \{\mathtt{TRUE}, \mathtt{FALSE}\}$, $\mathtt{C}[M] = \mathtt{C}[N]$.*

In other words, two programs are contextually equivalent when there is no way to separate them from inside the language.

Usual important properties of contextual equivalence are the following:

1. contextually equivalent programs implement the same functions and viceversa;
2. normalization-equivalent programs are contextually equivalent;
3. contextual equivalence is the maximum non-trivial equivalence on typed programs.

A consequence of 1. is that contextual equivalence may break complexity, that is, identify programs with different execution time/space. Indeed, this may be seen as a feature, rather than a weak point: it means that we can reason on contextual equivalence to justify *program optimization*. This is the replacement of a piece of expensive code, within a larger code, with some piece of more optimized code. When we do this, we would certainly like to be assured that the replacement will not produce bug, and, more generally, that it will not alter the observable behavior of the program. In other words, we want to be assured that the optimized programs will still compute *the same* function as the non-optimized one.

Indeed, program equivalence methods are largely applied precisely to this aim. Another important application is for the verification of language *compilers*: when we write a program P in a high-level language, say, `C` and we run the program, this has first to be transformed into code which is executable for the machine; indeed P passes through a long list of *translations* $P \to P_1 \to P_2 \to \cdots \to P_n$ from the high-level language into languages of lower and lower level. Usually, these translations will also include some optimization tricks, making the code more and more amenable for an efficient execution on the machine. Now, who is assuring us that the final executable code P_n will still compute *the same* function as the original program P? This can indeed be formulated as a program equivalence problem concerning code written in different languages (a bit like the problem of comparing proof equivalence across different proof systems).

To make an example, the `CompCert` project, developed at Inria in France, led to the formal verification of a compiler for the language `C`, via the proof assistant `Coq`. This formal verification is indeed nothing more than an (extremely difficult and long) formal proof that all translations occurring during compilation do preserve contextual equivalence.

Let us go back to proofs, now. Via the Curry-Howard correspondence, and thus the identification of proofs with programs, the notion of contextual equivalence extends to natural deduction derivations in an obvious way. What can be said then about contextual equivalence of proofs, given what we have seen in previous classes?

First, the typed Böhm theorem assures us that:

Theorem 1. *Normalization-equivalence coincides with contextual equivalence in* $\mathbf{NJ}_{\to,\wedge}$.

What about full natural deduction? As we have said, normalization-equivalence does not capture all admissible rule permutations, like those permuting a $\vee$E with an introduction rule. At the same time, we recalled the non-trivial result that the equivalence arising from reductions, expansions and generalized permutations is decidable and coincides with contextual equivalence. Recall, however, that there can be no canonical proof system capturing this equivalence, as it involves complexity-breaking permutations (i.e. permutations that may make the size of derivations explode).

2 Proof Equivalence and Proof Assistants

As you may know, the last decade has seen a rising of interest towards the development of so-called *interactive proof assistants.* These are programming languages that incorporate in the most literal sense the Curry-Howard perspective, since the programs written in these languages are just formal proofs. Such programs can be used, on the one hand, to *check* the correctness of a proof (just by checking if the associated code compiles!), and, on the other hand, to *assist* the mathematician to produce new proofs (recall that, because of the well-known incompleteness and undecidability results for first-order logic and arithmetic, there is hardly any hope to design a programming language capable of proving theorems alone).

We mentioned before the `CompCert` correctness proof, which has been produced and checked with the proof-assistant `Coq`. Many other striking examples can be found in the recent literature in this very dynamic field.

Since these languages essentially rely on the Curry-Howard correspondence between natural deduction (for higher-order logic) and functional programming languages, it should not be surprising that questions related to proof equivalence arise here and there in this field. Here I will just mention a few interesting directions.

- *Language Interoperability*: existing proof assistants rely on different proof systems; for example, `Coq` and `Lean` are based on natural deduction, while e.g. `Abella` relies on sequent calculus. This generates the problem of how to relate code written in one language with the one written in the other, and, notably, how to reason, in a mathematically sound way, about proofs represented by code written in different languages. Notably, there exist so-called *proof-checker* languages, like e.g. `Dedukti`, which are though to provide a sort of common environment in which proofs, as described in different formal languages, can be encoded.
- *Automatic Transport of Proofs*: in many situation it is convenient to state and prove results about a certain class of mathematical structures that is convenient to formally reason about, and then transport these results to other structures, for instance ones that are computationally more efficient, but possibly less convenient to reason about. For example, one might prefer to reason on natural numbers in unary notation, but to translate the resulting proofs and programs based on the binary notation. The underlying principle here is that, given two *isomorphic* formulae $A \equiv B$, any proof Π of some formula $\phi[A]$ containing A should translate into a proof Π' of the formula $\phi[B]$. However, realizing this apparently natural principle in languages like `Coq` may be very difficult in practice [5], and relies on the application to formal proofs of *parametricity* conditions of the kind we are going to discuss later today. More recently [11], it has been proposed to apply a yet stronger principle called *univalence*, which, in very rough terms, justifies the replacement of isomorphic formulas within a proof. However, a clear constructive and proof-theoretic understanding of this principle has not been reached yet.

3 Second-Order Quantification, a.k.a. Polymorphism

The other thing I wanted to discuss today is quantification, from the viewpoint of proof equivalence. Importantly, with quantifiers we enter a much wilder area, as we will see. However, the Curry-Howard correspondence with programs will be of help here.

First, as you all probably know, quantifications in logic may be of many different types (first-order, second-order, etc.). Today we will focus on *propositional second-order quantification*, that is, quantification on propositions, like in e.g. "all propositions are either true or false", or "any proposition implies itself". For a discussion of proof equivalence in presence of first-order quantification, see e.g. [3, 7].

We will focus on intuitionistic logic. This is for simplicity, but also for another reason: propositional second-order quantification in propositional classical logic is *definable* via

$$\forall p.A := A[\mathbf{T}/p] \wedge A[\mathbf{F}/p] \qquad\qquad \exists p.A := A[\mathbf{T}/p] \vee A[\mathbf{F}/p].$$

This implies in particular that provability in, say, LK2, is decidable. Instead, in intuitionistic logic this reduction is far from possible; in particular, while provability in LJ is decidable, provability in LJ2 is undecidable (a result that was first established in the 50s by Löb).

System F The language of second-order propositional logic is obtained by simply adding universal and existential quantification:

$$A ::= p \mid \cdots \mid \forall p.A \mid \exists p.A$$

The calculus **NJ2** is defined by adding rules for universal and existential quantification

$$\cfrac{\begin{array}{c}\Gamma\\ \vdots\\ A\end{array}}{\forall p.A}\ \forall\mathrm{I} \qquad\qquad \cfrac{A[B/p]}{\exists p.A}\ \exists\mathrm{I}$$

$$\cfrac{\forall p.A}{A[B/p]}\ \forall\mathrm{E} \qquad\qquad \cfrac{\exists p.A \qquad \begin{array}{c}\overset{i}{A}\quad \Gamma\\ \vdots\\ C\end{array}}{C}\ \exists\mathrm{E}(i)$$

In the rule $\forall$I we must ask that the variable X does not occur *free* in any undischarged assumption Γ. Similarly, in $\exists$E we must ask that X does not occur in C nor in any other undischarged assumption Γ.

The rules $\exists$I and $\forall$E make use of a *witness formula* B. This is responsible for an important fact, namely the failure of the sub-formula property, even for normal derivations. There is nothing to do with it, second order logic has no sub-formula property, period.

As in the propositional case, the study of **NJ2** is usually restricted to the *negative fragment* $\mathbf{NJ2}_{\to,\wedge,\forall}$, or even just to $\mathbf{NJ}_{\to,\forall}$, a fragment that has become known as *System F*. On the one hand, you may see that the rule $\exists$E looks very similar to the rule $\vee$E, so you may expect this rule to suffer from the same defects (read, rule permutations!). On the other hand, this fragment is as expressive as full **NJ2**, via the so-called *Russell-Prawitz encodings*:

$$\begin{aligned}(A \wedge B)^* &:= \forall p.(A \to B \to p) \to p\\ (A \vee B)^* &:= \forall p.(A \to p) \to (B \to p) \to p\\ (\exists q.A)^* &= \forall p.(\forall q.A \to p) \to p.\end{aligned}$$

The study of System F is tightly related to the Curry-Howard correspondence. Indeed, proofs in this system correspond to what computer scientists call *polymorphic* programs. For instance, the following proof

$$\cfrac{\cfrac{\overset{i}{p}}{p \to p}\ {\to}\mathrm{I}(i)}{\forall p.p \to p}\ \forall\mathrm{I}$$

corresponds to a program that, *for every possible type* T, yields a program from T to T corresponding to the identity function. A more general example is the `Map` program that, given arbitrary types A, B, a list $L = [a_1, \dots, a_n]$ of elements of type A and a function $f : A \to B$, produces a list

$$\mathtt{Map}(A, B, L, f) = [f(a_1), \dots, f(a_n)]$$

made of the corresponding elements of type B. Given a suitable encoding of the type of lists as a formula $\mathtt{List}[p]$ of System F, the polymorphic program `Map` can be encoded as a proof of the second-order formula

$$\forall p.\forall q.\mathtt{List}[p] \wedge (p \to q) \to \mathtt{List}[q].$$

Polymorphism is a feature that is found in many programming languages (ranging from functional programming languages like `Haskell` or `OCaml`, to more mainstream object-oriented languages like e.g. `Java`). It allows the programmer to define code that can be re-used in a very elegant way in infinitely many different context, by suitably re-arranging the types.

However, just like studying the proof theory second-order logic is in general very difficult (due to the failure of several structural properties, as well as to incompleteness), studying polymorphic programs and their equivalence conditions may also be difficult.

Proof- and Program-Equivalence in System F As in **NJ**, also for System F we can define reductions and expansions, which are very simple:

$$\cfrac{\cfrac{\begin{array}{c}\Pi\\ A\end{array}}{\forall p.A}\ \forall\mathrm{I}}{A[B/p]}\ \forall\mathrm{E} \quad \rightsquigarrow \quad \begin{array}{c}\Pi[B/p]\\ A[B/p]\end{array}$$

$$\overset{i}{\forall p.A} \qquad \rightsquigarrow \qquad \dfrac{\dfrac{\overset{i}{\forall p.A}}{A}\,\forall\mathrm{E}}{\forall p.A}\,\forall\mathrm{I}$$

Moreover, we have a normalization result analogous to **NJ**:

Theorem 2. *For any derivation* Π *of System F, there exists a* unique *normal derivation* $\mathrm{nf}(\Pi)$ *such that* $\Pi \rightsquigarrow^* \Pi'$.

Theorem 2 has the same statement as the corresponding theorem of **NJ**. Yet, it is much more difficult to prove, due to the absence of the sub-formula property. At the same time, thanks to Theorem 2, normalization-equivalence in System F is well-defined and decidable (even if not checkable in polynomial time: the complexity of computing a normal form in System F goes much beyond all primitive recursive functions!).

However, normalization-equivalence is surprisingly *weak*, and indeed rather unsatisfying in practice. Notably, in the practice of polymorphic programming people very often introduce *stronger* notions of equivalence for polymorphic programs. In particular, there is no analogous result to the type Böhm theorem for System F, since contextual equivalence, the maximum non-trivial notion of equivalence, is strictly stronger than normalization-equivalence. In particular, contextual equivalence, unlike normalization-equivalence, is even undecidable.

Here we enter a rather intricate and technical domain, but the look for stronger notions of equivalence can, at least to some extent, be explained and justified from the logical perspective of proof equivalence.

New Rule Permutations from Parametricity What is a proof of a universally quantified formula like $\forall p.p \to p$, where p ranges over the domain of all propositions? The formula says that for any possible choice of a proposition A, it is true that A implies A. A natural answer would thus seem to be that a proof of $\forall p.p \to p$ should be some method producing, for each choice of a proposition A, a proof of $A \to A$. Under the Curry-Howard view, this intuitively means a method producing, for each proposition (or type) A, a function from A to A itself.

Certainly any given proposition comes with its own ways of proving $A \to A$ (or its own class of functions from A to A). Yet, as is clear by inspecting the rules for quantifiers, proving $\forall p.p \to p$ does *not* amount at listing different proofs of $A \to A$, for any possible A; instead, it amounts at proving that $p \to p$ holds where p is a variable standing for any possible proposition; in this way, for any instantiation of p as A, we can obtain a proof of $A \to A$ in a uniform way. In more operational terms, this means having what is called a *parametric* function from a variable type p to itself: any instantiation of p with a type A yields a function from A to A, but all such functions will behave "in the same way" [10, 1, 2].

It has been argued that the interpretation of second-order proofs as parametric functions provides a possible way-out from traditional philosophical arguments agains the so-called "impredicativity" of second order logic, see [4, 6]. This is an interesting debate, but we will not enter it today.

Under the parametric view, there seems to be not many ways of constructing a proof of $p \to p$: such a proof must produce a function from p to p, where p indicates a proposition or type we know nothing about. The only actual choice is provided by the identity function, corresponding to the natural deduction seen above. Indeed, a purely semantic reasoning based on the idea of parametricity leads to conclude that the derivation above provides the *unique* way of proving $\forall p.p \to p$.

We will now see how this line of reasoning can be used to conclude that different derivations must denote the same proof, that is, to justify the appeal to rule permutations of a somewhat *new* kind.

For instance, consider the two derivations below:

$$\dfrac{\overset{0}{B \to C} \qquad \dfrac{\dfrac{\overset{1}{\forall p.p \to p}}{B \to B}\,\forall\mathrm{E} \qquad \overset{2}{B}}{B}\,{\to}\mathrm{E}}{C}\,{\to}\mathrm{E} \qquad\qquad \dfrac{\dfrac{\overset{0}{\forall p.p \to p}}{C \to C}\,\forall\mathrm{E} \qquad \dfrac{\overset{1}{B \to C} \qquad \overset{2}{B}}{C}\,{\to}\mathrm{E}}{C}\,{\to}\mathrm{E} \qquad (1)$$

Both derivations are constructed out of one occurrence of ($\forall$E) and two occurrences of ($\to$E); however, these rules not only occur in different order, but the two occurrences of ($\forall$E) employ distinct instantiations of the variable p (as B on the left and as C on the right). So why should we regard these derivations as denoting the same proof?

Suppose, as we did above, that a proof of $\forall p.p \to p$, since parametric, can only be one which corresponds, functionally, to the identity function. Under this assumption, however we replace the undischarged assumption with label n in any of the derivations in (1), this will produce (after a few normalization steps) the same proof. This fact is best appreciated when the derivations are decorated with proof terms (see also [12], pp. 16-17):

$$\dfrac{h : B\to C \qquad \dfrac{\dfrac{f:\forall p.p\to p}{fB:B\to B}\,\forall\mathrm{E} \qquad b:B}{fBb:B}\,{\to}\mathrm{E}}{h(fBb):C}\,{\to}\mathrm{E} \qquad\qquad \dfrac{\dfrac{f:\forall p.p\to p}{fC:C\to C}\,\forall\mathrm{E} \qquad \dfrac{h:B\to C \qquad b:B}{hb:C}\,{\to}\mathrm{E}}{fC(hb):C}\,{\to}\mathrm{E} \tag{2}$$

Parametricity warrants that fB and fC must be the identity functions $1_B : B \to B$ and $1_C : C \to C$, and thus the two derivations encode (for any f, h, b, B and C) the same proof of C:

$$h(fBb) = h(1_B b) = hb = 1_C(hb) = fC(hb) \tag{3}$$

Parametricity can also be used to justify permutations of rules involving the existential quantifier. For example, how many proofs do there exist of the proposition $\exists p.p$? Intuitively, there are infinitely many, since for any proof Π of some proposition A, we can obtain a different derivation as below.

$$\dfrac{\begin{matrix}\Pi\\ A\end{matrix}}{\exists p.p}\,\exists\mathrm{I} \tag{4}$$

Nevertheless, we will now argue, by reasoning in a somehow *extensional* way, that two derivations as below should denote the same proof:

$$\dfrac{\dfrac{A\to B \qquad \begin{matrix}\Pi\\ A\end{matrix}}{B}\,{\to}\mathrm{E}}{\exists p.p}\,\exists\mathrm{I} \quad \equiv \quad \dfrac{\begin{matrix}\Pi\\ A\end{matrix}}{\exists p.p}\,\exists\mathrm{I} \tag{5}$$

The extensional assumption we make is the following: if two proofs Π, Σ of some formula A are not the same, then there must be some way of separating them; more formally, there must be a (quantifier-free) formula C with two distinct proofs Θ_1, Θ_2, and a derivation Θ of C with assumption A such that Θ composed with Π normalizes to Θ_1, while Θ composed with Σ normalizes to Θ_2:

$$\begin{matrix}\Pi\\ A\\ \Theta\\ C\end{matrix} \quad \rightsquigarrow \quad \begin{matrix}\Theta_1\\ C\end{matrix} \quad \not\equiv \quad \begin{matrix}\Theta_2\\ C\end{matrix} \quad \leftsquigarrow \quad \begin{matrix}\Sigma\\ A\\ \Theta\\ C\end{matrix} \tag{6}$$

Under this assumption, the argument goes like this: suppose there exists a proof $\Sigma : (\exists p.p) \vdash C$ (notice that p cannot occur in C) separating the two derivations in (5) in the sense just explained.

Σ can easily be converted into a proof of $\forall p.p \to C$ (basic logic exercise), hence into a parametric proof Σ' of $p \vdash C$, i.e. a parametric function from the "unknown" type p to a "known" type C, where the latter in no way depends on p. One can then be convinced that this function can in no way use its input of type p, that is Σ' cannot use the assumption p (and thus Σ cannot use the assumption $\exists p.p$). We deduce then that composing Σ with some derivation Λ of $\exists p.p$ has the effect of deleting Λ, and thus Σ cannot be separating:

$$\begin{matrix}\dfrac{\dfrac{A\to B \qquad \begin{matrix}\Pi\\ A\end{matrix}}{B}\,{\to}\mathrm{E}}{\exists p.p}\,\exists\mathrm{I}\\ \Sigma\\ C\end{matrix} \quad \equiv \quad \begin{matrix}\Sigma\\ C\end{matrix} \quad \equiv \quad \begin{matrix}\dfrac{\begin{matrix}\Pi\\ A\end{matrix}}{\exists p.p}\,\exists\mathrm{I}\\ \Sigma\\ C\end{matrix} \tag{7}$$

Put in simple words, the two derivations in (5) must denote the same proof since any way of possibly distinguishing them should arise from some parametric function, but no parametric function can do the job.

Using the permutation above, it is not difficult to show that any two derivations of $\exists p.p$ denote the same proof: given two derivations as below

$$\dfrac{\begin{matrix}\Pi_1\\ A\end{matrix}}{\exists p.p}\,\exists\mathrm{I} \qquad\qquad \dfrac{\begin{matrix}\Pi_2\\ B\end{matrix}}{\exists p.p}\,\exists\mathrm{I} \tag{8}$$

we can reason as illustrated below:

$$\dfrac{\begin{matrix}\Pi_2\\ B\end{matrix}}{\exists p.p}\,\exists\mathrm{I} \quad \overset{\text{(reduction)}}{\equiv} \quad \dfrac{\dfrac{\dfrac{\begin{matrix}\Pi_2\\ B\end{matrix}}{A\to B}\,{\to}\mathrm{I} \qquad \begin{matrix}\Pi_1\\ A\end{matrix}}{B}\,{\to}\mathrm{E}}{\exists p.p}\,\exists\mathrm{I} \quad \overset{(5)}{\equiv} \quad \dfrac{\begin{matrix}\Pi_1\\ A\end{matrix}}{\exists p.p}\,\exists\mathrm{I} \tag{9}$$

Remark 1. *The permutations arising from parametricity can be related to the permutations for* $\vee E$ *in propositional* **NJ**. *Recall that disjunction* $A \vee B$ *can be* defined *in System F by the formula* $\forall p.(A \to p) \to (B \to p) \to p$. *This gives rise to a translation of second-order intuitionistic logic with disjunction into its disjunction-free fragment [9] which not only preserves provability but also translates reductions steps into reduction steps. However, the picture changes when permutative conversions are added as reduction steps: as discussed at length in [13, 8], these conversions do not translate into reductions in second-order logic, but into a sequence of reductions, expansions as well as conversions similar to* (2)*, where the latter must be justified using some form of parametricity.*

References

[1] E. Stewart Bainbridge, Peter J. Freyd, Andre Scedrov, and Philip J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70:35–64, 1990.

[2] Claudio Hermida, Uday S. Reddy, and Edmund P. Robinson. Logical relations and parametricity - A Reynolds programme for category theory and programming languages. In *Proceedings of the Workshop on Algebra, Coalgebra and Topology (WACT 2013)*, volume 303 of *Electronic Notes in Theoretical Computer Science*, pages 149–180. Elsevier, 2014.

[3] Dominic J. D. Hughes. Unification nets: Canonical proof net quantifiers. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 540–549, New York, NY, USA, 2018. Association for Computing Machinery. URL: `https://doi.org/10.1145/3209108.3209159`, `doi:10.1145/3209108.3209159`.

[4] Giuseppe Longo and Thomas Fruchart. Carnap's remarks on impredicative definitions and the genericity theorem. In *Logic, Methodology and Philosophy of Science: Logic in Florence*. Kluwer, 1997.

[5] Nicolas Magaud and Yves Bertot. Changing data structures in type theory: A study of natural numbers. In *Proceedings TYPES 2000*, volume 2277 of *Lecture Notes in Computer Science*, pages 181–196. Springer-Verlag, 2000.

[6] Paolo Pistone. Polymorphism and the obstinate circularity of second order logic, a victims' tale. *The Bulletin of Symbolic Logic*, 24(1):1–52, 2018.

[7] Paolo Pistone. A new conjecture about identity of proofos. To appear in Perspectives on Deduction, Synthèse Library, Studies in Epistemology, Logic, Methodology and Philosophy of Science, 2023. Available at `https://arxiv.org/abs/2110.02630`.

[8] Paolo Pistone, Luca Tranchini, and Mattia Petrolo. The naturality of natural deduction (II). On atomic polymorphism and generalized propositional connectives. *Studia Logica*, 110:545–592, 2022.

[9] Dag Prawitz. *Natural deduction, a proof-theoretical study.* Almqvist & Wiskell, 1965.

[10] John C. Reynolds. Types, abstraction and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing '83*, pages 513–523. North-Holland, 1983.

[11] Nicolas Tabareau, Eric Tanter, and Matthieu Sezeau. Equivalences for free: univalent parametricity for effective transport. *Proceedings of the ACM on Programming Languages*, 2(ICFP):92:1–29, 2018.

[12] Luca Tranchini and Paolo Pistone. Intensional harmony as isomorphism. to appear in Thomas Piecha and Kai Wehmeier (eds.), Peter Schroeder-Heister on Proof-Theoretic Semantics, Outstanding Contributions to Logic, Springer; available at `http://logica.uniroma3.it/pistone/Harmony.pdf`, 2021.

[13] Luca Tranchini, Paolo Pistone, and Mattia Petrolo. The naturality of natural deduction. *Studia Logica*, 107(1):195–231, 2019.