

Coding dependent elimination schemes using Σ -types

SYLVAIN LEBRESNE

*Preuves, Programmes et Systèmes (PPS), CNRS UMR 7126, Université Paris 7
Projet LogiCal, LIX, École Polytechnique
(e-mail: sylvain.lebresne@pps.jussieu.fr)*

Abstract

We present a coding of the dependent elimination scheme of an inductive definition from its non-dependent elimination scheme using Σ -types. This coding leads us to introduce commutation and η -rules on the elimination rules.

1 Introduction

Inductive data types are introduced in functional programming languages as a standard way to define algebraic data types. They are generally introduced as a bunch of constants. For example, in Gödel's system T (see, for example, (Girard *et al.*, 1989)), the data type of natural numbers is defined from the type nat , the constant 0 of type nat , the constant $Succ$ of type $nat \rightarrow nat$, and the recursor scheme (or elimination scheme) R of type

$$A \rightarrow (nat \rightarrow A \rightarrow A) \rightarrow nat \rightarrow A$$

for any type A . This recursor allows us to define functions by recursion over natural numbers. In the context of type theory, the need for a new elimination scheme arises with the introduction of dependent types. This scheme has type

$$A(0) \rightarrow \Pi x: nat. (A(x) \rightarrow A(Succ(x))) \rightarrow \Pi n: nat. A(n)$$

Although its type is more accurate, it is computationally equivalent to the non-dependent scheme. This new scheme allows us to define more functions, such as constructing a dependent list of size n . But above all, via the Curry-Howard correspondance, it allows us to do proofs over natural numbers (note that the type of this new recursor is exactly the induction principle on a predicate A).

In a type system, the possibility of defining new inductive data types (see for example inductive definitions in the Calculus of Inductive Constructions (Paulin-Mohring, 1996)) gives a generic, powerful and convenient way to define data types. A counterpart of this genericity is the difficulty of studying inductive definitions from the theoretical point of view. But certain data types contain the essential difficulties of these inductive definitions. That is the case of Well-ordering types, that can be used to encode strictly positive inductive definitions ((Dybjer, 1997),

(Nordström *et al.*, 1990)). As well, the inductive definition of equality is enough to recover the full strength of dependent inductive data types (Paulin-Mohring, 1996) (*i.e.* inductive definitions of predicates).

The aim of this paper is similar. It shows how to define the dependent elimination scheme of an inductive definition from the non-dependent one using only the dependent elimination scheme of Σ -types¹. However, we will see that our encoding needs the addition of commutation and η -rules to ensure typing.

In section 2, we present the system used in the remaining of the paper. In section 3, we describe our coding of the dependent elimination scheme for a small class of inductive data types. Finally, in section 4, we show that we can generalise this coding to a large class of inductive definitions.

2 Framework

The coding we propose takes place in a typed lambda calculus with dependent types. It is close to Martin-Löf's Type Theory (Martin-Löf, 1984) in a presentation where terms and types are mixed. It provides native Σ -types, inductive types and a fixpoint construction. This system is described in Figure 1. We assume known standard background about λ -calculus.

Some remarks :

- We will use the standard notation $A \rightarrow B$ as a shortcut for $\Pi x:A. B$ when $x \notin FV(B)$.
- Inductive definitions will be introduced in the next section by means of new constants, a constant being a tuple (c, A, R) composed of a name c , a type A and a set of reduction rules R . Each constant introduced will be implicitly added to the context of constants Δ .
- The equality sign, that appears in the conversion rule (*conv*), refers to the congruence generated by all the reduction rules of the system, including reduction rules declared in Δ .
- The system we present contains a typed fixpoint operator similar to that of PCF (Plotkin, 1977). In this paper, we do not care about guarded conditions that are usually introduced to ensure normalisation (see for example (Paulin-Mohring, 1996)). Hence, this system does not enjoy the strong normalisation property. However, all use of the fixpoint operator corresponds to structural recursion and thus should be accepted by any reasonable notion of guarded conditions.

3 Description of the coding for some data types

In this section, we describe our coding of the dependent elimination scheme.

¹ For convenience and without loss of generality, in the remaining of this paper, Σ -type will be embedded directly in our system, and hence not introduced as an inductive definitions

3.1 The type of booleans

Let us extend the global context of constants Δ with the following definitions :

$$\begin{aligned} \mathit{bool} & : \mathit{Prop} \\ \mathit{true} & : \mathit{bool} \\ \mathit{false} & : \mathit{bool} \\ \mathit{bool}^{\mathit{ndp}} & : \Pi P:\mathit{Prop}. P \rightarrow P \rightarrow \mathit{bool} \rightarrow P \end{aligned}$$

Framework

Syntax:

$$\begin{aligned} u, v, A, B ::= & \quad x \mid c \mid \mathit{Prop} \mid \mathit{Type} \mid \mathit{fix}_x(u) \\ & \quad \mid \Pi x:A. B \mid \lambda x:A. u \mid u v \\ & \quad \mid \Sigma x:A. B \mid \langle u, v \rangle \mid \pi_1 u \mid \pi_2 u \end{aligned}$$

(x stands for a variable name taken from an infinite set of names;
 c stands for a constant. Such constants will be introduced later.)

Judgements:

Contexts are ordered lists of pairs (x, A) where x is a variable name and A a term. Constants are defined in an implicit global context Δ .

We distinguish two kinds of judgements :

- $\Gamma \vdash$ says “the context Γ is well-formed”.
- $\Gamma \vdash u : v$ says “in context Γ , u has type v ”.

These judgements are recursively defined by the typing rules given below.

Typing rules:

(In the following, s , $s1$ and $s2$ stand for Prop or Type , $DV(\Gamma)$ stand for Declared Variables in Γ and \max is defined from the order $\mathit{Prop} < \mathit{Type}$.)

$$\begin{array}{c} \frac{}{\boxed{\vdash} \sqcup_{wf}} \\ \frac{\Gamma \vdash (x:A) \in \Gamma}{\Gamma \vdash x:A} \mathit{var} \\ \frac{}{\Gamma \vdash \mathit{Prop} : \mathit{Type}} \mathit{Prop} \\ \frac{\Gamma \vdash u : \Pi x:A. B \quad \Gamma \vdash v : A}{\Gamma \vdash u v : B\{x := v\}} \mathit{app} \\ \frac{\Gamma \vdash u : A \quad \Gamma \vdash v : B\{x := u\}}{\Gamma \vdash \langle u, v \rangle : \Sigma x : A. B.} \mathit{pair} \\ \frac{\Gamma \vdash u : \Sigma x:A. B}{\Gamma \vdash \pi_1(u) : A} \mathit{proj}_1 \\ \frac{\Gamma, (f : X) \vdash t : X}{\Gamma \vdash \mathit{fix}_f(t) : X} \mathit{fix} \end{array} \qquad \begin{array}{c} \frac{\Gamma \vdash A : s \quad x \notin DV(\Gamma)}{\Gamma, (x:A) \vdash} \mathit{var}_{wf} \\ \frac{\Gamma \vdash (c, A, _) \in \Delta}{\Gamma \vdash c : A} \mathit{const} \\ \frac{\Gamma \vdash \Pi x:A. B : s \quad \Gamma, (x:A) \vdash u : B}{\Gamma \vdash \lambda x:A. u : \Pi x:A. B} \mathit{lamb} \\ \frac{\Gamma \vdash A : s1 \quad \Gamma, (x:A) \vdash B : s2}{\Gamma \vdash \Pi x:A. B : \max(s1, s2)} \mathit{prod} \\ \frac{\Gamma \vdash A : s1 \quad \Gamma, (x:A) \vdash B : s2}{\Gamma \vdash \Sigma x:A. B : \max(s1, s2)} \mathit{sig} \\ \frac{\Gamma \vdash u : \Sigma x:A. B}{\Gamma \vdash \pi_2(u) : B\{x := \pi_1(u)\}} \mathit{proj}_2 \\ \frac{\Gamma \vdash u : T \quad \Gamma \vdash T' : s \quad T = T'}{\Gamma \vdash u : T'} \mathit{conv} \end{array}$$

Reduction rules:

$$\begin{aligned} (\lambda x:A. t) u & \triangleright_{\beta} t\{x := u\} \\ \mathit{fix}_f(t) & \triangleright_{\mathit{fix}} t\{f := \mathit{fix}_f(t)\} \\ \pi_1 \langle u, v \rangle & \triangleright_{\pi_1} u \\ \pi_2 \langle u, v \rangle & \triangleright_{\pi_2} v \end{aligned}$$

Fig. 1. The framework

and reductions rules :

$$\begin{aligned} & \text{bool}^{ndp} P x y \text{ true} \triangleright_{\text{true}} x \\ & \text{bool}^{ndp} P x y \text{ false} \triangleright_{\text{false}} y \end{aligned}$$

where bool^{ndp} is the non-dependent² elimination scheme of bool .

We now want to define the corresponding dependent elimination scheme, i.e. a term

$$\text{bool}^{dep} : \Pi P:(\text{bool} \rightarrow \text{Prop}). (P \text{ true}) \rightarrow (P \text{ false}) \rightarrow \Pi b:\text{bool}. P b$$

ensuring the same reduction rules as bool^{ndp} .

The intuition behind our coding is to box the type dependency of the eliminated predicate in a dependent pair; use the non-dependent elimination scheme (to preserve reduction); and then unbox the type dependency using the second projection on the obtained pair.

So, we define bool^{dep} by

$$\text{bool}^{dep} \triangleq \lambda P:(\text{bool} \rightarrow \text{Prop}). \lambda x:(P \text{ true}). \lambda y:(P \text{ false}). \lambda b:\text{bool}. \pi_2(M_{P,x,y,b})$$

where

$$M_{P,x,y,b} \triangleq \text{bool}^{ndp} (\Sigma x:\text{bool}. P x) \langle \text{true}, x \rangle \langle \text{false}, y \rangle b$$

It follows immediately that

$$\text{bool}^{dep} : \Pi P:(\text{bool} \rightarrow \text{Prop}). (P \text{ true}) \rightarrow (P \text{ false}) \rightarrow \Pi b:\text{bool}. P(\pi_1(M_{P,x,y,b}))$$

Observationally, given the definition of bool^{dep} and the reduction rules of bool^{ndp} , whatever the boolean b we eliminate is, the first projection of $M_{P,x,y,b}$ will always be the boolean b itself. However, $\pi_1(M_{P,x,y,b})$ is not reducible if b is a variable and hence $\pi_1(M_{P,x,y,b})$ and b are not convertible. So, in order to have $\pi_1(M_{P,x,y,b}) = b$, we add to Δ two new reduction rules for bool^{ndp} . The first one is a commutation rule³ between the first projection and the non-dependent scheme and the second one is an η -rule for this scheme :

$$\begin{aligned} \pi_1(\text{bool}^{ndp} (\Sigma x:\text{bool}. P) x y b) & \triangleright_{b_c} \text{bool}^{ndp} \text{bool} (\pi_1 x) (\pi_1 y) b \\ \text{bool}^{ndp} \text{bool} \text{ true} \text{ false} b & \triangleright_{b_\eta} b \end{aligned}$$

Provided these two rules, we can check that $\pi_1(M_{P,x,y,b}) \triangleright b$:

$$\begin{aligned} \pi_1(M_{P,x,y,b}) & \triangleq \pi_1(\text{bool}^{ndp} (\Sigma x:\text{bool}. P x) \langle \text{true}, x \rangle \langle \text{false}, y \rangle b) \\ & \triangleright_{b_c} \text{bool}^{ndp} \text{bool} (\pi_1 \langle \text{true}, x \rangle) (\pi_1 \langle \text{false}, y \rangle) b \\ & \triangleright_{\pi_1}^* \text{bool}^{ndp} \text{bool} \text{ true} \text{ false} b \\ & \triangleright_{b_\eta} b \end{aligned}$$

and hence, by conversion,

$$\text{bool}^{dep} : \Pi P:(\text{bool} \rightarrow \text{Prop}). (P \text{ true}) \rightarrow (P \text{ false}) \rightarrow \Pi b:\text{bool}. P b$$

² Of course, we do not want to add the dependent elimination scheme, as its definition is the purpose of our coding.

³ Note that this rule is close to the commuting conversions (or permutative conversions) used for \forall and \exists to ensure cut elimination in natural deduction (Prawitz, 1965).

Finally, we can easily check that $bool^{ndp}$ has the same reductions than $bool^{dep}$. For example, for the first rule, we have :

$$\begin{aligned} bool^{dep} P x y true &\triangleq \pi_2(bool^{ndp} (\Sigma z:bool. P z) \langle true, x \rangle \langle false, y \rangle true) \\ &\triangleright_{true} \pi_2 \langle true, x \rangle \\ &\triangleright_{\pi_2} x \end{aligned}$$

3.2 Enumerated types

Let us consider the family $(\mathcal{S}_i)_{i \in \mathbb{N}}$ of the enumerated type where \mathcal{S}_i is the enumerated type with i constructors (c_1^i, \dots, c_i^i) defined by the constants :

$$\begin{aligned} \mathcal{S}_i &: Prop \\ c_1^i &: \mathcal{S}_i \\ &\vdots \\ c_i^i &: \mathcal{S}_i \\ \mathcal{S}_i^{ndp} &: \Pi P:Prop. \underbrace{P \rightarrow \dots \rightarrow P}_{i \text{ times}} \rightarrow \mathcal{S}_i \rightarrow P \end{aligned}$$

and i reduction rules :

$$\begin{aligned} \mathcal{S}_i^{ndp} P x_1 \dots x_i c_1^i &\triangleright x_1 \\ &\vdots \\ \mathcal{S}_i^{ndp} P x_1 \dots x_i c_i^i &\triangleright x_i \end{aligned}$$

We define the dependent elimination scheme \mathcal{S}_i^{dep} of \mathcal{S}_i by :

$$\begin{aligned} \mathcal{S}_i^{dep} &\triangleq \lambda P:(\mathcal{S}_i \rightarrow Prop). \lambda x_1:(P c_1^i). \dots \lambda x_i:(P c_i^i). \lambda s:\mathcal{S}_i. \\ &\pi_2(\mathcal{S}_i^{ndp} (\Sigma x:\mathcal{S}_i. P x) \langle c_1^i, x_1 \rangle \dots \langle c_i^i, x_i \rangle s) \end{aligned}$$

and we add the following two reduction rules for \mathcal{S}_i^{ndp} :

$$\begin{aligned} \pi_1(\mathcal{S}_i^{ndp} (\Sigma x:\mathcal{S}_i. P) x_1 \dots x_i s) &\triangleright \mathcal{S}_i^{ndp} \mathcal{S}_i (\pi_1 x_1) \dots (\pi_1 x_i) s \\ \mathcal{S}_i^{ndp} \mathcal{S}_i c_1^i \dots c_i^i s &\triangleright s \end{aligned}$$

We can easily check that the scheme

$$\mathcal{S}_i^{dep} : \Pi P:(\mathcal{S}_i \rightarrow Prop). (P c_1^i) \rightarrow \dots \rightarrow (P c_i^i) \rightarrow \Pi s:\mathcal{S}_i. P s$$

has the same reductions as \mathcal{S}_i^{ndp} . Hence, it is a correct coding of the dependent elimination scheme of \mathcal{S}_i .

Let us note that the data type \emptyset (*empty*) with no constructors and *unit* with only one constructor \top of type *unit* are respectively \mathcal{S}_0 and \mathcal{S}_1 , and *bool* is \mathcal{S}_2 .

3.3 The type of natural numbers

We now consider a more complex inductive data type, that of Peano natural numbers. A major difference between the booleans and the natural numbers is the recursivity of the constructor S (successor). If we try to define the dependent elimination

scheme of the natural numbers using their non-dependent elimination scheme with a coding close to the one of the boolean case, we shall have difficulties expressing the commutation rule of the first projection with the non-dependent elimination scheme. Commutation of operators (like π_1) and recursivity don't mix well. So, in order to get rid of this problem, we use the non-dependent case analysis scheme⁴ of the natural numbers to define their dependent case analysis scheme. We then define the dependent elimination scheme using a fixpoint.

The data type of natural numbers is defined by the following constants :

$$\begin{aligned} nat & : Prop \\ 0 & : nat \\ S & : nat \rightarrow nat \\ nat_{case}^{ndp} & : \Pi P:Prop. P \rightarrow (nat \rightarrow P) \rightarrow nat \rightarrow P \end{aligned}$$

where nat_{case}^{ndp} is the non-dependent case analysis scheme of nat which has reductions :

$$\begin{aligned} nat_{case}^{ndp} P x f 0 & \triangleright x \\ nat_{case}^{ndp} P x f (S n) & \triangleright f n \end{aligned}$$

We define the dependent case analysis scheme by :

$$\begin{aligned} nat_{case}^{dep} & \triangleq \lambda P:(nat \rightarrow Prop). \lambda x:(P 0). \lambda f:(\Pi n:nat. P (S n)). \lambda n:nat. \\ & \pi_2(nat_{case}^{ndp} (\Sigma x:nat. P x) \langle 0, x \rangle (\lambda m:nat. \langle S m, f m \rangle)) n \end{aligned}$$

and we add two new reduction rules, a commutation rule of the first projection with the non-dependent case analysis scheme and an η -rule⁵ for this scheme :

$$\begin{aligned} \pi_1(nat_{case}^{ndp} (\Sigma z:nat. P) x (\lambda m:nat. y) n) & \triangleright nat_{case}^{ndp} nat (\pi_1 x) (\lambda m:nat. \pi_1 y) n \\ nat_{case}^{ndp} nat 0 (\lambda m:nat. S m) n & \triangleright n \end{aligned}$$

Then, one could easily check that

$$nat_{case}^{dep} : \Pi P:(nat \rightarrow Prop). (P 0) \rightarrow (\Pi n:nat. P (S n)) \rightarrow \Pi n:nat. P n$$

So, it has the type of the dependent case analysis scheme (which follows directly from the fact that $\pi_1(nat_{case}^{ndp} (\Sigma x:nat. P x) \langle 0, x \rangle (\lambda m:nat. \langle S m, f m \rangle)) n \triangleright n$), and ensures the same reduction rules than nat_{case}^{ndp} .

Finally, as is already known, we can define the dependent elimination scheme using the fixpoint operator, which leads to :

$$\begin{aligned} nat_{rec}^{dep} & \triangleq \lambda P:(nat \rightarrow Prop). \lambda x:(P 0). \lambda f:(\Pi n:nat. (P n) \rightarrow (P (S n))). \\ & fix_{rec}(nat_{case}^{dep} P x (\lambda m:nat. f m (rec m))) \end{aligned}$$

This has the type and reduction of the dependent elimination scheme, i.e. :

$$\begin{aligned} nat_{rec}^{dep} & : \Pi P:(nat \rightarrow Prop). (P 0) \rightarrow (\Pi n:nat. (P n) \rightarrow (P (S n))) \rightarrow (\forall m : nat, P m) \\ & nat_{rec}^{dep} P x f 0 \triangleright x \\ & nat_{rec}^{dep} P x f (S n) \triangleright f n (nat_{rec}^{dep} P x f n) \end{aligned}$$

⁴ Which can trivially be defined using the non-dependent elimination scheme

⁵ If we had the standard η -rule for λ in the system, this rule can be more simply $nat_{case}^{ndp} nat 0 (S) n \triangleright n$

$$\begin{array}{l}
W_{case}^{dep} : \Pi P:(W \rightarrow Prop). \left[\Pi x:A. \Pi w:(B x \rightarrow W). P (node x w) \right] \rightarrow \Pi w:W. P w \\
\triangleq \lambda P:(W \rightarrow Prop). \lambda H:(\Pi x:A. \Pi w:(B x \rightarrow W). P (node x w)). \lambda w:W. \\
\pi_2(W_{case}^{ndp} (\Sigma x:W. P x) (\lambda x:A. \lambda f:(B x \rightarrow W). \langle node x f, H x f \rangle) w) \\
\\
W_{rec}^{dep} : \Pi P:(W \rightarrow Prop). \\
\left[\Pi x:A. \Pi w:(B x \rightarrow W). (\Pi b:(B x). P (w b)) \rightarrow P (node x w) \right] \\
\rightarrow \Pi w:W. P w \\
\triangleq \lambda P:(W \rightarrow Prop). \\
\lambda H:\left[\Pi x:A. \Pi w:(B x \rightarrow W). (\Pi b:(B x). P (w b)) \rightarrow P (node x w) \right]. \\
fix_{rec}(W_{case}^{dep} P (\lambda x:A. \lambda f:(B x \rightarrow W). H x f (\lambda b:(B x). rec (f b))))
\end{array}$$

Fig. 2. Definition of W_{rec}^{dep} .

3.4 Well-ordering types

We now show that the same kind of coding as for the natural numbers works for the well-ordering type. Well-ordering types have been introduced by Martin-Löf (Martin-Löf, 1980) and can be used to represent various inductive data types, a property that we will use later. This data type has two parameters (A and B) introduced as constants to simplify notation. Hence, well-ordering data types, generally noted $Wx:A. B$, are simply noted W hereafter.

It is defined by the constants :

$$\begin{array}{l}
A : Prop \\
B : A \rightarrow Prop \\
W : Prop \\
node : \Pi x:A. (B x \rightarrow W) \rightarrow W \\
W_{case}^{ndp} : \Pi P:Prop. \left[\Pi x:A. (B x \rightarrow W) \rightarrow P \right] \rightarrow W \rightarrow P
\end{array}$$

where W_{case}^{ndp} is the non-dependent case analysis scheme of W . This scheme is given by the following three reduction rules :

$$\begin{array}{l}
W_{case}^{ndp} P H (node x f) \triangleright H x f \\
\pi_1(W_{case}^{ndp} (\Sigma x:W. P) (\lambda x. \lambda f. t) w) \triangleright W_{case}^{ndp} W (\lambda x. \lambda f. \pi_1 t) w \\
W_{case}^{ndp} W (\lambda x. \lambda f. node x f) w \triangleright w
\end{array}$$

The coding of the dependent elimination scheme of W , namely W_{rec}^{dep} , is given in Figure 2. Moreover, we can easily check that it enjoys the expected reduction rule :

$$W_{rec}^{dep} P H (node x f) \triangleright H x f (\lambda b:(B x). W_{rec}^{dep} P H (f b))$$

4 Generalisation

We have shown that we can derive the dependent elimination scheme of the well-ordering type from the non-dependent one. It is well known that this type can be used to represent various inductive data types (see, for example, (Dybjer, 1987)). More specifically, P. Dybjer has shown that, associated with disjunction, product⁶, empty and unit types, the well-ordering type can be used to represent strictly positive inductive definition schemes (Dybjer, 1997). Thus, our coding can be generalised to these inductive definitions.

However, this well-ordering based coding doesn't give the expected induction principle in an intensional type theory (there is a 'junk' in the representations (see, for example, chapter 15 of (Nordström *et al.*, 1990))). Goguen and Luo have shown that, in order to correct this problem, it is sufficient to add some filling-up equality rules (see (Goguen & Luo, 1993)). These rules are forms of η -rule a little bit more general than the ones we introduced to ensure typing in our coding.

In their paper, Goguen and Luo discuss the possibility of orienting their equality rules in order to get reduction rules, but face some difficulties in doing so (one way is not Church-Rosser, the other one break strong normalisation). However, in our coding, we only use the new reduction rules (η and commutation of π_1) to ensure typing and not to compute. So, our coding also work if we take only the equality generated by the new rules and not the rules themselves. Hence, one could directly mix our work with the one of Goguen and Luo by adding only the commutation rules to their rules.

References

- Dybjer, Peter. (1987). Inductively defined sets in Martin-Löf's set theory. Avron, A, & al (eds), *Workshop on general logic*.
- Dybjer, Peter. (1997). Representing inductively defined sets by wellorderings in Martin-Löf's type theory. *Theoretical computer science*, **176**, 329–335.
- Girard, Jean Yves, Lafont, Yves, & Taylor, Paul. (1989). *Proofs and types*. Cambridge University Press.
- Goguen, Healdene, & Luo, Zhaohui. (1993). Inductive data types: Well-ordering types revisited. Huet, G., & Plotkin, G. (eds), *Logical environments*.
- Martin-Löf, Per. (1980). Constructive mathematics and computer programming. *Pages 153–175 of: North-Holland (ed), Logic, methodology and philosophy of science vi*.
- Martin-Löf, Per. (1984). *Intuitionistic type theory*. Studies in Proof Theory : Lecture Notes, vol. 1. Naples: Bibliopolis.
- Nordström, Bengt, Petersson, Kent, & Smith, Jan M. (1990). *Programming in Martin-Löf's type theory*. Oxford University Press.
- Paulin-Mohring, Christine. 1996 (Dec.). *Définitions inductives en théorie des types d'ordre supérieur*. Habilitation à diriger les recherches, Université Claude Bernard Lyon I.
- Plotkin, Gordon. (1977). LCF considered as a programming language. *Theoretical computer science*, **5**, 223–255.

⁶ one can easily check that our coding can be extended without difficulty on disjunction and product types.

Prawitz, Dag. (1965). *Natural deduction: A proof-theoretical study*. Stockholm: Almqvist & Wiksell.