Defining λ-Typed λ-Calculi by Axiomatizing the Typing Relation

Philippe de Groote

INRIA-Lorraine – CRIN – CNRS
Campus Scientifique - B.P. 239
54506 Vandœuvre-lès-Nancy Cedex – FRANCE

Abstract. We present a uniform framework for defining different λ-typed λ-calculi in terms of systems to derive typing judgements, akin to Barendregt's Pure Type Systems [3]. We first introduce a calculus called λ^λ and study its abstract properties. These are, among others, the property of Church-Rosser, the property of subject reduction, and the one of strong normalization. Then we show how to extend λ^λ to obtain an inferential definition. of Nederpelt's Λ [20]. One may also extend λ^λ to get inferential definitions of van Daalen Λ_β [24], and de Bruijn's $\Lambda\Delta$ [9] and we argue that these new inferential definitions are well suited for language-theoretic investigations.

1 Introduction

There is a growing interest in designing generic formal systems that can be used to specify various object logics [16, 18]. These systems, also known as *logical frameworks*, may be used to implement and automate many of the logics that are of interest in computer science. For this reason, they form the basis of generic theorem provers [23].

We think, as the author of [10] does, that λ-typed λ-calculi have an important role to play as the backbones of logical frameworks. λ-Typed λ-calculi are as elegant and simple than they are powerful. They correspond, at the implementation level, to a uniform data-structure allowing syntactic categories and formal languages, as well as logical rules and proofs to be represented. As pointed out in [17], systems such as LF [16] can be seen as higher-level languages that can be compiled into λ-typed λ-calculi. Nevertheless the notion of λ-type did not spread outside the AUTOMATH community.

In this paper, we define λ-typed λ-calculi in terms of systems to derive typing judgements, in the spirit of [3]. We consider, for instance, Nederpelt's Λ [20] whose original definition is strongly algorithmic. Indeed the notion of well-typed expressions is defined by means of an algorithm that tells whether an expression is well-typed or not. Such an algorithmic style of definition is better suited for implementation than it is for language-theoretic investigations. For a theoretical study, the inferential style of definition that we use here (called E-definition in [24]) presents further advantages:

In: *10th Annual Symposium on Theoretical Aspects of Computer Science, STACS'93,*
P. Enjalbert, A. Finkel, and K.W. Wagner (Eds.),
Lecture Notes in Computer Science, Vol. 665, Springer-Verlag (1993), pp. 712-723.

1. it allows proofs to be conducted by induction on the derivations of judgements;
2. it provides a uniform framework to compare the different calculi with each other;
3. it makes easier the comparison between λ-typed λ-calculi and more usual typed λ-calculi such as the ones defined in [3].

The paper is organized as follows. In Section 2, we introduce and discuss the notion of λ-type from a general and somewhat intuitive point of view. In Section 3, and 4, we introduce the λ-typed λ-calculus λ^λ, which is equivalent to the variant of Nederpelt's Λ that we studied in [12]. The definition that we give is an adaptation of the one used by Barendregt in [3]. In Section 5, we studied the abstract properties of λ^λ and we show how the techniques used in [3, 4, 14, 15] may be adapted to the case of λ-typed λ-calculi. In Sections 6 we extend λ^λ by allowing for $\beta\eta$-conversion and we obtain a calculus equivalent to Nederpelt's Λ. In section 7, we suggest how to obtain inferential definitions of van Daalen's Λ_β [24] and de Bruijn's $\Lambda\Delta$ [9]. Finally, we conclude in Section 8.

2 The Notion of λ-Type

The notion of λ-type originated in the frame of the AUTOMATH project [8]. In the earliest versions of AUTOMATH, λ-types were already present, because a unique binding operator was used both for the terms and their types. Then, by a further unification of concepts, de Bruijn designed a language called AUT-SL (a shorthand for AUTOMATH *single line*), which permits a whole AUTOMATH book to be expressed as a single λ-term [6]. This language gave rise to the calculus Λ that Nederpelt introduced in his dissertation and for which he proved strong normalization [20, 21]. Finally, a last achievement, due to de Bruijn again, was to provide the calculus $\Lambda\Delta$ [9] as a generalization of Λ.

The easiest way (but also maybe the most harmful) of understanding the notion of λ-type is based on the following simple observation: although the functional abstractor (λ) and the constructor of dependent types (Π) are usually distinguished, their syntactic features are basically the same; both are universal binding operators. Hence, it is consistent, at least syntactically, to identify them. This identification gives rise to calculi where the types that are assigned to λ-terms are themselves λ-terms.

To see λ-typed λ-calculi as the result of that syntactic identification between λ and Π is harmful for at least two reasons. First of all, it suggests that the λ-typed λ-calculi were designed after more usual typed λ-calculi. This is wrong. AUT-SL, for instance, is more than fifteen years older than LF. Second of all, it makes one feel that λ-typed λ-calculi are based on a syntactic confusion and amount, therefore, to a semantic absurdity. This is not true by any means since the consistency of a calculus like Nederpelt's Λ, for instance, can be established in a proof-theoretic way [20, 24] (see also Section 5, hereafter).

A better way of understanding λ-typed λ-calculi is to forget about other typed λ-calculi and to think of the untyped λ-calculus as a calculus of substi-

tution. Then, λ-typed λ-calculi may be seen as calculi of substitution typed by substitution.[1] Actually, λ-typed λ-calculi are very natural in the sense that the only concepts needed to assign types to terms are the operations of abstraction and application, i.e. the central concepts of the (untyped) λ-calculus.

The terms and the types of a λ-typed λ-calculus obey the same syntax. This feature presents some technical advantages, notably when implementing the calculus. For example, the meta-operation of substituting a term for a variable into a type may be represented within the calculus by a β-redex. This, in turn, allows an explicit typing operator to be defined (see Section 5, Definition 5). We do not claim, however, that λ-typed λ-calculi must be preferred to other calculi. For instance, we do not think that Nederpelt's Λ must, in any case, be preferred to LF. Preference is often only a matter of style. What we believe is that λ-typed λ-calculi must not be forgotten or ignored.

3 Syntax of Raw λ-Expressions

λ-Expressions are built from a countably infinite set of variables $\mathcal{V}$ and a single constant τ.

Definition 1. *The set $\mathcal{E}$ of λ-expressions is defined inductively as follows:*

i. $\tau \in \mathcal{E}$
ii. $x \in \mathcal{V} \Rightarrow x \in \mathcal{E}$
iii. $x \in \mathcal{V}$ *and* $A, B \in \mathcal{E} \Rightarrow (\lambda x\!:\!A.\, B) \in \mathcal{E}$
iv. $A, B \in \mathcal{E} \Rightarrow (A\, B) \in \mathcal{E}$

The constant τ is akin to the constant Type of other calculi [5, 16] or to Barendregt's $*$ [3]. In Nederpelt's Λ, λ-expressions whose head is τ (i.e. expressions of the form $\lambda x_1\!:\!A_1.\, \cdots \lambda x_n\!:\!A_n.\, \tau\, B_1 \cdots B_m$) are not assigned any type. Since we want to give to λ^λ a definition in the spirit of [3], contrary to Nederpelt, we will state the axiom

$$\vdash \tau : \kappa$$

where κ is another constant corresponding to Barendregt's $\Box$. To this end, we introduce the set $\mathcal{K}$ of λ-kinds.

Definition 2. *The set $\mathcal{K}$ of λ-kinds is defined inductively as follows:*

i. $\kappa \in \mathcal{K}$
ii. $x \in \mathcal{V}$ *and* $A \in \mathcal{E}$ *and* $B \in \mathcal{K} \Rightarrow \lambda x\!:\!A.\, B \in \mathcal{K}$

[1] It is worth noting that the notions of reduction considered in the AUTOMATH project (called mini-reductions by de Bruijn in [9]) do not amount to *global* substitutions, like β-reduction does, but to *local* ones. Recently, in [22], Nederpelt pursued further the study of these mini-reductions and provided a comparison with the explicit substitutions of Abadi, Cardelli, Curien, and Lévy [1].

For convenience, we also define the set of pseudo-expressions $\mathcal{P} = \mathcal{E} \cup \mathcal{K}$.

As customary, λ is a binding operator. The scoping rules are the usual ones. In particular, variables occurring free in a λ-expression A remain free in $\lambda x\!:\!A.\, B$. Pseudo-expressions that can be transformed into each other by renaming their bound variables are identified (see [7]).

The equality of λ^λ amounts to the relation of β-conversion ($\twoheadleftarrow\!\!\!\!\twoheadrightarrow_\beta$), which is defined as the reflexive, transitive, symmetric closure of the relation of β-contraction ($\rightarrow_\beta$). The latter is defined on raw pseudo-expressions, as usual.

4 Well-Typedness and Correctness

Type checking is defined according to typing contexts. A typing context is a sequence of declarations $x : A$, where $x \in \mathcal{V}$ and $A \in \mathcal{E}$. Any context Γ, $x : A$ is such that (i) the variable x is not declared in Γ, (ii) all the variables occurring free in the λ-expression A are declared in Γ. $\mathcal{C}$ is the set of typing contexts.

We define the notion of well-typed λ-expression by providing a proof system to derive typing judgements of the shape

$$\Gamma \vdash A : B \tag{1}$$

where Γ is a typing context and A and B are both λ-expressions. While A and B belong to the same syntactic category (they are both in $\mathcal{E}$), we will sometimes, for convenience, refer to A as a term and to B as a type.

Another form of judgement is necessary. The judgements of this second form have also the shape of (1) with the difference that B is no longer a λ-expression but a λ-kind. Strictly speaking, the two forms of judgements are different. Nevertheless, by a slight abuse of language, we identify them and say that A is a well-typed λ-expression in both cases.

Definition 3. *Let $\Gamma \in \mathcal{C}$, $A \in \mathcal{E}$ and $B \in \mathcal{P}$. A typing judgment of λ^λ is an expression of the form*

$$\Gamma \vdash A : B$$

derivable according to the following proof system:

$$\vdash \tau : \kappa \qquad \textit{(constant)}$$

$$\frac{\Gamma \vdash A : B}{\Gamma,\, x : A \vdash x : A} \qquad \textit{(variable)}$$

$$\frac{\Gamma,\, x : A \vdash B : C}{\Gamma \vdash \lambda x\!:\!A.\, B : \lambda x\!:\!A.\, C} \qquad \textit{(abstraction)}$$

$$\frac{\Gamma \vdash A : \lambda x\!:\!C.\, D \quad \Gamma \vdash B : C}{\Gamma \vdash A\, B : D[x\!:=\!B]} \qquad \textit{(application)}$$

$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash C : D}{\Gamma,\, x : C \vdash A : B} \qquad \textit{(weakening)}$$

$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash C : D}{\Gamma \vdash A : C} \quad \text{if } B \twoheadleftarrow\!\!\!\twoheadrightarrow_\beta C \quad \textit{(type conversion)}$$

Given some $A \in \mathcal{E}$ and some $\Gamma \in \mathcal{C}$, one says that the λ-expression A is well-typed according to the context Γ if and only if there exists a pseudo-expression $B \in \mathcal{P}$ such that $\Gamma \vdash A : B$.

Given some $A \in \mathcal{E}$, A is called a correct λ-expression of λ^λ if and only if A is well-typed according to the empty context.

We may now compare the system given in Definition 3 with related systems such as LF [16] or Barendregt's $\lambda\mathbf{P}$ [3]. Two rules of the present system seem to be unusual. The first one is the abstraction rule that introduces an abstractor λ in the right-hand side of the colon (instead of some specific type constructor such as Π). The second one is the weakening rule where no degree restriction is given on C. This means that any well-typed term may be used as a type or, in other words, that there may be chains $\Gamma \vdash A_1 : A_0$, $\Gamma \vdash A_2 : A_1$, ..., $\Gamma \vdash A_n : A_{n-1}$ of arbitrary lengths. This must be contrasted with the other (more usual) calculi where expressions of only three degrees are provided: the *kinds*, the *types*, and the *terms*.

5 The Language Theory of λ^λ

In this section, we review the main properties of λ^λ. Among others, we state the properties of Church-Rosser, of subject reduction, and of strong normalization. These properties, for Nederpelt's Λ, were first established by Nederpelt and van Daalen in their respective PhD theses [20, 24]. The proofs they give, however, are rather difficult because of the algorithmic nature of Nederpelt's original definition (see section 6 below). On the other hand, Definition 3 allows the same properties to be proven by induction on the derivations of the typing judgements. It is then a mere exercise to adapt the proofs that are given in [4, 15]

The first property that we state is the Church-Rosser Theorem for the set of pseudo-expressions.

Proposition 1. (Church-Rosser) *Let $A, B, C \in \mathcal{P}$ be such that $A \twoheadrightarrow_\beta B$ and $A \twoheadrightarrow_\beta C$. Then there exists $D \in \mathcal{P}$ such that $B \twoheadrightarrow_\beta D$ and $C \twoheadrightarrow_\beta D$.*

Proof. The usual Tait–Martin-Löf proof for type free λ-terms [2, pp. 61–62] generalizes easily to raw λ-expressions and λ-kinds. □

The next property of interest is subject reduction. This property turns out to be important in practice. If one sees β-reduction as the process of evaluating a λ-expression, a consequence of subject reduction is that there is no need for any kind of dynamic type checking.

Proposition 2. (Subject reduction) *For all $A, B \in \mathcal{E}$, $C \in \mathcal{P}$ and all $\Gamma \in \mathcal{C}$, if $A \twoheadrightarrow_\beta B$ and $\Gamma \vdash A : C$ then $\Gamma \vdash B : C$.*

Proof. The property can be established by induction on the derivation of $\Gamma \vdash A : C$. *Some technical lemmas are needed. See [4] or [15] for details.* □

The subject reduction and Church-Rosser properties allow one to give a simple characterization of the relation of β-equality between well-typed terms. β-equality is the least equivalence relation containing the relation of β-contraction between well-typed terms. It is defined, more explicitly, as follows.

Definition 4. *Let* $A, A' \in \mathcal{E}$ *and* $\Gamma \in \mathcal{C}$. *We say that* A *and* A' *are* β*-equal with respect to* Γ, *and we write*

$$\Gamma \vdash A =_\beta A'$$

if and only if there exist $A_1, \ldots, A_n \in \mathcal{E}$, $B_1, \ldots, B_n \in \mathcal{P}$ *such that:*

i. $A_1 \equiv A$ *and* $A_n \equiv A'$,
ii. for all $1 \leq i \leq n$, $\Gamma \vdash A_i : B_i$,
iii. for all $1 \leq i < n$, $A_i \to_\beta A_{i+1}$ *or* $A_{i+1} \to_\beta A_i$.

As a corollary of Propositions 1 and 2, we have that $\Gamma \vdash A =_\beta A'$ if and only if $A \twoheadleftarrow\!\!\twoheadrightarrow_\beta A'$, $\Gamma \vdash A : B$, and $\Gamma \vdash A' : B'$, for some $B, B' \in \mathcal{P}$. In connection with this (and with Proposition 4 below), it is worth noting that the type conversion rule of Definition 3 is equivalent to the following type equality rule:

$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash B =_\beta C}{\Gamma \vdash A : C} \qquad (\textit{type equality})$$

Actually, the type equality rule is the intended meaning of the type conversion rule. Nevertheless if we replace the latter by the former, using Definition 4, we would introduce circularity into the definitions. It is possible to circumvent this problem by providing an axiomatization of the equality judgements, in the spirit of Martin-Löf's type theory [19]. This solution, however, would lengthen most of the proofs of the propositions.

Most of the typed λ-calculi that have been studied in the literature, among others the eight systems of Barendregt's λ-cube, are strongly normalizable. This property also holds for λ^λ.

Proposition 3. (Strong normalization) *Let* $A, B \in \mathcal{P}$ *and* $\Gamma \in \mathcal{C}$ *be such that* $\Gamma \vdash A : B$. *Then there is no infinite sequence of* β*-contraction starting in* A.

Proof. As in the case of LF or $\lambda\mathbf{P}$, *strong normalization for* λ^λ *may be derived from strong normalization for the simply typed* λ*-calculus. See [15, 16].* □

Two other properties are related to the λ-expressions that are acting as types.

The peculiarity of λ-typed λ-calculi is that the set of terms is identical to the set of types. We know that this is certainly true at the context-free level of raw λ-expressions. To make sense, however, this identification must also exist for well-typed expressions. In other words, when a well-typed λ-expression is assigned another λ-expression as a type, we expect the latter λ-expression to be also well-typed. This is stated by the following result.

Proposition 4. (Well-typedness of types) *Let $A, B \in \mathcal{E}$ and $\Gamma \in \mathcal{C}$. If $\Gamma \vdash A : B$ then there exists $C \in \mathcal{P}$ such that $\Gamma \vdash B : C$.*

Proof. The proof is by induction on the derivation of $\Gamma \vdash A : B$. The only problematic case is the one of application for which a substitution lemma is needed. □

We also have that the type of an expression is unique up to β-conversion.

Proposition 5. (Unicity of types) *Let $A, B \in \mathcal{E}$, $C \in \mathcal{P}$ and $\Gamma \in \mathcal{C}$. If $\Gamma \vdash A : B$ and $\Gamma \vdash A : C$ then $B \twoheadleftarrow\!\!\twoheadrightarrow_\beta C$.*

Proof. A straightforward induction on the derivation of $\Gamma \vdash A : B$. □

One of the main consequences of the above metatheoretic properties is the decidability of the typing relation of λ^λ. Another application consists in designing other definitions of λ^λ that are more suited to implementation [12] and then proving their equivalence with definition 3. Let us illustrate partially this by introducing Nederpelt's typing operator.

Definition 5. *Nederpelt's typing operator* $\mathsf{type}\colon \mathcal{C} \times \mathcal{E} \rightharpoonup \mathcal{P}$ *is defined inductively according to the following clauses:*

i. $\mathsf{type}_\Gamma[\tau] = \kappa$,
ii. $\mathsf{type}_\Gamma[x] = A \quad$ *if* $x{:}A \in \Gamma$,
iii. $\mathsf{type}_\Gamma[\lambda x : A.\, B] = \lambda x : A.\, \mathsf{type}_{\Gamma, x:A}[B]$,
iv. $\mathsf{type}_\Gamma[A\,B] = \mathsf{type}_\Gamma[A]\,B$.

Except for the first clause, which is proper to our formalism, the above definition corresponds to the one given by Nederpelt's in his thesis.

The connection between this typing operator and the typing relation defined by Definition 3 is expressed by the following property.

Proposition 6. *Let $A \in \mathcal{E}, B \in \mathcal{P}$ and $\Gamma \in \mathcal{C}$ be such that $\Gamma \vdash A : B$. Then $B \twoheadleftarrow\!\!\twoheadrightarrow_\beta \mathsf{type}_\Gamma[A]$.*

Proof. By induction on the derivation of $\Gamma \vdash A : B$. □

It is remarkable that the operator type is defined on the raw expressions. This allows one to also define so-called applicability conditions on the raw expressions. Then the well-typedness of a λ-expression may be checked simply by structural induction except for the case of an application where, in addition, the applicability conditions must be satisfied. For λ^λ, we have that an application $(A\,B)$ is well typed according to a context Γ if and only if

i. A and B are well-typed according to Γ,
ii. there exist $C \in \mathcal{E}$ and $D \in \mathcal{P}$ such that $\mathsf{type}_\Gamma[A] \twoheadrightarrow_\beta \lambda x : C.\, D$, and
iii. $\mathsf{type}_\Gamma[B] \downarrow_\beta C$ (where the relation $\downarrow_\beta$, by definition, indicates the existence of a common reduct).

Clauses ii and iii correspond to the applicability conditions.

6 Nederpelt's Λ

The main difference between λ^λ and Nederpelt's Λ [20, 21], besides the way in which they are defined, is that the equality of Λ is extensional in the sense that it is based on the notion of $\beta\eta$-reduction.

Let us look at the applicability conditions for Λ as defined by Nederpelt in [20]. First the degree of an expression is defined as follows.

Definition 6. *The degree* $\mathsf{deg}_\Gamma[A]$ *of a* λ*-expression* A *according to a context* Γ *is defined inductively according to the following clauses:*

i. $\mathsf{deg}_\Gamma[\tau] = 0$,
ii. $\mathsf{deg}_\Gamma[x] = \mathsf{deg}_\Gamma[A] + 1$ *if* $x{:}A \in \Gamma$,
iii. $\mathsf{deg}_\Gamma[\lambda x{:}A.\, B] = \mathsf{deg}_{\Gamma, x:A}[B]$,
iv. $\mathsf{deg}_\Gamma[A\, B] = \mathsf{deg}_\Gamma[A]$.

Then an iterated version of the typing operator is defined.

Definition 7. *Let* type^n *be defined as follows:*

i. $\mathsf{type}^0_\Gamma[A] = A$,
ii. $\mathsf{type}^{n+1}_\Gamma[A] = \mathsf{type}^n_\Gamma[\mathsf{type}_\Gamma[A]]$.

Then one defines $\mathsf{type}^*_\Gamma[A] = \mathsf{type}^d_\Gamma[A]$, *where* $d = \mathsf{deg}_\Gamma[A]$.

Finally, Nederpelt's applicability conditions are given by the following definition: a λ-expression A is applicable to a λ-expression B in the context γ if and only if there exist $C \in \mathcal{E}$ and $D \in \mathcal{P}$ such that

i. $\mathsf{type}^*_\Gamma[A] \twoheadrightarrow_\beta \lambda x{:}C.\, D$,
ii. $\mathsf{type}_\Gamma[B] \downarrow_{\beta\eta} C$.

The above applicability conditions might seem awkward. Nevertheless they may be explained in term of $\beta\eta$-conversion in a rather clean way.Roughly speaking, applicability conditions say that the *domain* of a functional expression must *match* the *type* of its argument. In the case of λ^λ, the *domain* and the *type* may be computed using the operator type, and the meaning of the word *"match"* is β-conversion.

In the case of Nederpelt's Λ, Clause (ii) above suggests that the meaning of the word *"match"* is $\beta\eta$-conversion. One also has that the *type* of the argument is computed using the same operator type. What is unclear is why an iterated version of the typing operator is used when computing the *domain*. In order to answer this question, let us first consider an example.

Example. Consider the context

$$\Gamma \equiv a : \lambda x{:}\tau.\, \tau,\ b : a,\ c : \tau$$

In this context, the expression b is applicable to the expression c according to Nederpelt's conditions (while, in λ^λ, $b\,c$ is not well-typed with respect to Γ). Indeed we have:

$$\mathsf{type}^*_\Gamma\,[b] = \lambda x{:}\tau.\,\tau \quad \text{and} \quad \mathsf{type}_\Gamma\,[c] = \tau.$$

Let us try now to extend λ^λ in order to derive $\Gamma \vdash b\,c : A$, for some λ-expression A. A possible derivation tree is the following:

$$\dfrac{\dfrac{\Gamma \vdash b : a \qquad \dfrac{\dfrac{\vdots}{\Gamma,\, x:\tau \vdash a : \lambda x{:}\tau.\,\tau} \quad \dfrac{\vdots}{\Gamma,\, x:\tau \vdash x : \tau}}{\Gamma,\, x:\tau \vdash a\,x : \tau}}{\Gamma \vdash \lambda x{:}\tau.\,a\,x : \lambda x{:}\tau.\,\tau}}{(\star)\ \ \Gamma \vdash b : \lambda x{:}\tau.\,a\,x} \qquad \dfrac{\vdots}{\Gamma \vdash c : \tau}}{\Gamma \vdash b\,c : a\,c}$$

The interesting step in the above derivation is $(\star)$. The expression b is assigned a functional type $\lambda x : \tau.\,a\,x$ thanks to the η-expansion $a \leftarrow_\eta \lambda x : \tau.\,a\,x$. This η-expansion is legitimate because the expanded expression, that is $\lambda x{:}\tau.\,a\,x$, is well-typed. Now, the reason why the latter expression is well-typed is because the type of a is functional or, in other words, because $\mathsf{type}^*_\Gamma\,[b]$ is functional.

This example demonstrates that the iterated version of the typing operator is simply related to η-conversion. Therefore, in order to get an inferential definition of Nederpelt's Λ, we must adapt Definition 3 to take η-conversion into account. The first idea consists of simply replacing the type conversion rule by the following one:

$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash C : D}{\Gamma \vdash A : C} \quad \text{if } B \twoheadleftarrow\!\!\!\!\twoheadrightarrow_{\beta\eta} C \tag{2}$$

Unfortunately, this does not work. We have seen that, when dealing with β-conversion only, the type conversion rule is equivalent to a type equality rule. The problem, in the present case, is precisely that rule (2) is not equivalent to the corresponding type equality rule:

$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash B =_{\beta\eta} C : D}{\Gamma \vdash A : C} \tag{3}$$

In fact, (2) and (3) are not equivalent becauses $\beta\eta$-reduction on pseudo-expressions is not Church-Rosser (see [14, 20, 24]). This was first pointed out by Nederpelt in his thesis [20]. The typical counterexample he gives is the following:

$$\lambda x{:}A.\,(\lambda x{:}B.\,C)\,x \to_\eta \lambda x{:}B.\,C \quad \text{and} \quad \lambda x{:}A.\,(\lambda x{:}B.\,C)\,x \to_\beta \lambda x{:}A.\,C$$

where A and B can be any λ-expressions. In particular, A and B could be such that the λ-expressions $\lambda x : B.\,C$ and $\lambda x : A.\,C$ would be well-typed. Therefore Rule (2) allows one to change arbitrarily the domain of a functional expression.

The problem can be circumvented by strengthening the type conversion rule as follows:

$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash B : D \quad \Gamma \vdash C : D}{\Gamma \vdash A : C} \quad \text{if } B \twoheadleftarrow\!\!\!\!\twoheadrightarrow_{\beta\eta} C. \tag{4}$$

In Rule (4), the λ-expressions B and C are explicitly required to have the same type D. As a consequence, when B and C are functional expressions, they must have the same domain. This is clear for λ-expressions of degree 0 because type conversion is not allowed at the level of λ-kinds. Then, for λ-expressions of degree $n > 0$, it can be established by induction.

Definition 3 where the type conversion rule is replaced by Rule (4) corresponds to a new system that we will call $\lambda^{\lambda}_{\beta\eta}$. This system is equivalent to Nederpelt's Λ.

The system $\lambda^{\lambda}_{\beta\eta}$ is well suited for language-theoretic investigations. The metatheoretic results of Section 5 may be adapted. Nevertheless the adaptation is not straightforward. The problem, of course, is the failure of the Church-Rosser property on pseudo-expressions. $\beta\eta$-Reduction on well-typed expressions satisfies the Church-Rosser property, but to prove it is far from easy. In fact, the property was conjectured by Nederpelt in his thesis [20] and proven by van Daalen seven years later [24]. Recently, Geuvers has given a proof of the Church-Rosser property for a large class of Pure Type System with $\beta\eta$-reduction [14]. His techniques may be adapted to $\lambda^{\lambda}_{\beta\eta}$.

7 Van Daalen's Λ_β and de Bruijn's $\Lambda\Delta$

In his thesis [24], van Daalen investigates a subsystem of Nederpelt's Λ that he calls Λ_β. In [9] de Bruijn introduces the calculus $\Lambda\Delta$, which is a generalization of Nederpelt's Λ.

Definition 3 may be extended in order to get inferential definitions of van Daalen's and de Bruijn's calculi: by allowing for a weak form of η-expansion, one obtains a calculus equivalent to Λ_β; by adding a subject expansion rule, one obtains a calculus equivalent to $\Lambda\Delta$. We do not give the precise definitions for the sake of shortness.

8 Conclusions

While the idea of dealing with λ-types emerged more than twenty years ago, the concept of λ-typed λ-calculus has remained proper to the AUTOMATH project and appears to have been somewhat overlooked by the rest of the scientific community. There are two main reasons to this:

1. the identification between terms and types is felt to be purely syntactic and to carry little semantic content,
2. the original definitions of Λ and $\Lambda\Delta$ may be difficult to master because of their strong algorithmic flavor.

This first reason can be considered at the same time as a cause or as a consequence. It is true that no interesting model of λ-typed λ-calculi has been

developed and that without such models it could be hard to have an intuition of what is a λ-type. However, properties such as Church-Rosser and normalization, which can be interpreted as consistency results, show that the notion of λ-type make sense and that, at least, we can construct a term model. Therefore, one may invert the argument. One may say that no interesting model of λ-typed λ-calculi has been developed not because the notion of λ-type is purely syntactic but because it has been overlooked.

The second reason is more pertinent. The first time one is confronted with the original definitions of Λ and $\Lambda\Delta$, which both consist in a type-checking algorithm, one has the feeling of being confronted with some complicated new system. This is because one has to face two problems at the same time. On the one hand, one has to understand the behavior of an algorithm while, on the other hand, one wants to understand abstractly the features of a new calculus. Moreover, to compare Λ and $\Lambda\Delta$ to other systems, with which one is possibly familiar, could be difficult. Hence, sooner or later, one is tempted to draw the wrong conclusion that λ-typed λ-calculi are artificially complicated because of the purely syntactic identification between abstractions and dependent types.

The inferential definitions that we have given in this paper should settle this misunderstanding. In particular, they enlighten the relation existing between λ-typed and other typed λ-calculi. They also explain unusual features of Λ and $\Lambda\Delta$ in terms of well-known notions. For instance, we have seen that the type-iteration that is used in [9, 20, 24] to compute the domain of a functional λ-expression may be explained in term of η-expansion.

References

1. M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.
2. H.P. Barendregt. *The lambda calculus, its syntax and semantics*. North-Holland, revised edition, 1984.
3. H.P. Barendregt. Introduction to Generalised Type Systems. *Journal of Functional Programming*, 1(2):125–154, 1991.
4. H.P. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbai, and T. Maibaum, editors, *Handbook of Logic in Computer Science*. Oxford University Press, 1992.
5. Th. Coquand. Metamathematical investigations of a calculus of constructions. In P. Odifreddi, editor, *Logic and Computer Science*, pages 91–122. Academic Press, 1990.
6. N.G. de Bruijn. AUT-SL, a single line version of AUTOMATH. Technical Report AUT 20, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1971.
7. N.G. de Bruijn. Lambda calculus notations with nameless dummies, a tool for automatic formula manipulation, with an application to the Church-Rosser theorem. *Indigationes Mathematicae*, 34:381–392, 1972.
8. N.G. de Bruijn. A survey of the project Automath. In J.P. Seldin and J.R. Hindley, editors, *to H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, 1980.

9. N.G. de Bruijn. Generalizing automath by means of a lambda-typed lambda-calculus. In *Mathematical Logic and Theoretical Computer Science*, pages 71–92. Lecture Notes in pure and applied Mathematics, 106, Marcel Dekker, New York, 1987.
10. N.G. de Bruijn. A plea for weaker frameworks. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 40–67. Cambridge University Press, 1991.
11. N.G. de Bruijn. Algorithmic definition of λ-typed λ-calculus. In G. Huet and G. Plotkin, editors, *Logical Environments*. Cambridge University Press, 1992.
12. Ph. de Groote. *Définition et Propriétés d'un métacalcul de représentation de théories*. PhD thesis, Université Catholique de Louvain, Unité d'Informatique, 1991.
13. Ph. de Groote. Nederpelt's calculus extended with a notion of context as a logical framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 69–86. Cambridge University Press, 1991.
14. H. Geuvers. The Church-Rosser property for $\beta\eta$-reduction in typed λ-calculi. In *Proceedings of the seventh annual IEEE symposium on logic in computer science*, pages 453–460, 1992.
15. H. Geuvers and M.-J. Nederhof. Modular proof of strong normalization for the calculus of construction. *Journal of Functional Programming*, 1(2):155–189, 1991.
16. R. Harper, F. Honsel, and G. Plotkin. A framework for defining logics. In *Proceedings of the second annual IEEE symposium on logic in computer science*, pages 194–204, 1987.
17. R. Harper and F. Pfenning. A module systems for a programming language based on the lf logical framework. Submitted for publication, 1992.
18. G. Huet and G. Plotkin, editors. *Logical Frameworks*. Cambridge University Press, 1991.
19. P. Martin-Löf. An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73*, pages 73–118. North-Holland, 1975.
20. R.P. Nederpelt. *Strong normalization in a typed lambda calculus with lambda structured types*. PhD thesis, Technische hogeschool Eindhoven, 1973.
21. R.P. Nederpelt. An approach to theorem proving on the basis of a typed lambda-calculus. In *Proceedings of the 5th international conference on automated deduction*, pages 182–194. Lecture Notes in Computer Science, 87, Springer Verlag, 1980.
22. R.P. Nederpelt. *The fine-structure of lambda calculus*. Computing Science Notes. Eindhoven University of Technology, 1992.
23. L.C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.
24. D.T. van Daalen. *The language theory of Automath*. PhD thesis, Technische hogeschool Eindhoven, 1980.