Verified Representations of Landau's "Grundlagen" in $\lambda\delta$ and in the Calculus of Constructions

Ferruccio Guidi
Department of Computer Science and Engineering
University of Bologna
Mura Anteo Zamboni 7, 40127, Bologna, ITALY
e-mail: ferruccio.guidi@unibo.it

Landau's "Grundlagen der Analysis" formalized in the language Aut–QE, represents an early milestone in computer-checked mathematics and is the only non-trivial development finalized in the languages of the Automath family. Here we discuss an implemented procedure producing a faithful representation of the Grundlagen in the Calculus of Constructions, effectively accepted by the proof assistant Coq. The point at issue is distinguishing λ-abstractions from Π-abstractions where the original text uses Automath unified binders, taking care of the cases in which a binder corresponds to both abstractions at one time. It is a fact that some binders can be disambiguated only by validating the Grundlagen in a calculus accepting Aut–QE and the Calculus of Constructions. To this end, we rely on $\lambda\delta$ version 3, a system that the author is proposing here for the first time.

1. INTRODUCTION

Landau's "*Grundlagen der Analysis*" [Lan65], formalized by Jutting [vB79] in the Automath language Aut–QE [vD94a] (henceforth, the QE–GdA), represents an early milestone in computer-checked mathematics and is the only non-trivial development finalized in the languages of the Automath family [NGdV94].

Actually, the use of the QE–GdA as a background for significant formalized mathematics is limited in that Aut–QE and the tools for its management [Wie02] seem incapable to compete with the most recent logical frameworks and with the state-of-the-art proof management systems based on them. For this reason, some authors propose translations of Aut–QE into Pure Type Systems [Bar93, KLN03, Bro11], and give the background for making the QE–GdA accessible to systems like Coq [Coq15], which accepts the Calculus of Constructions [CH88] (henceforth, CC for short). Here we discuss a farther step: an implemented procedure producing the CC–GdA, a representation of the QE–GdA in CC effectively accepted by Coq.

Firstly, we recall some concepts about CC and Aut–QE for convenience.

The terms of both systems are organized in three classes of increasing degree (kinds, types, and elements) comprising two sorts denoting the universes of sets and propositions, references by name, applications, and binding abstractions.

On the one hand, CC has distinct abstractions $\lambda_x N.M$ and $\Pi_x N.M$ for functions and function spaces respectively. On the other hand, Aut–QE is best known for its unified abstraction `[x:N]M` representing both abstractions of CC, as well as for its reversed application `<N>M` that corresponds to the application $(M\ N)$ in CC.

Considering the verification of an Automath text from the CC perspective, unified abstraction yields no logical confusion as long as the verifier can separate λ-

QE–GdA	CC–GdA

`+l`
`@[a:'prop'][b:'prop']`
`imp:=[x:a]b:'prop'`
	
	$(l\_imp \coloneqq \lambda_a \mathrm{Prop}.\lambda_b \mathrm{Prop}.(\Pi_x a.b : \mathrm{Prop}))$

Fig. 1. Using definitions and type-annotated terms in CC.

abstractions and Π-abstractions during type conversion and type inference.

On the contrary, when the author of the text assumes the unpleasant equality $\lambda_x N.M = \Pi_x N.M$, which in Aut–QE is an identity, and which implies $\Pi_x N.S = S$ if S is the sort typing M, then logical confusion is unavoidable since, for instance, a predicate is equated to the formula representing its universal quantification.

Unfortunately, this inconvenience affects the QE–GdA indeed, as we clearly see in the case of the constant `all"l"`, whose name stands for "∀-introduction", defined as the function $\lambda_\sigma \mathrm{Type}.\lambda_p(\Pi_x \sigma.\mathrm{Prop}).p$ that maps the predicate p to itself, instead of mapping it to the formula $\Pi_x \sigma.(p\ x)$ that is the universal quantification of p.

We want to stress that the Π-introduction $p \mapsto \Pi_x \sigma.(p\ x)$ reads in Aut–QE as `p → [x:sigma]<x>p`, exactly like the η-reduction $p \mapsto \lambda_x \sigma.(p\ x)$. For this reason some authors address these transformations with misleading terminology.

Next, we outline the structure of the QE–GdA that, being an Automath book, amounts to a list of (almost 7000) constants declared or defined within a system of sections (known as paragraphs), and introduced in a context of unified binders (known as block openers). So, for the purpose of representing this book in CC, it is convenient to equip the original calculus with definitions. Formally, these are entries of the form $(x \coloneqq N)$ occurring in contexts beside the usual entries of the form $(x : N)$. Needless to say, a specific reduction step is provided for expanding a reference to a defined name (this is known as δ-expansion).

Each defined constant is presented with its type, which shares the block openers of the definiens. In order to express such a definition in the form $(x \coloneqq M)$ while preserving the shared structure of the block openers, we extend CC with type-annotated terms. Formally, these terms have the form $(M : N)$, and their meaning is "M with expected type N". Our representation in CC of definition `imp"l"` (Figure 1), that opens the QE–GdA, shows the intended use of this construction.

We stress that definitions and type-annotated terms do not alter the expressive power of CC. Moreover, these constructions are fully accepted by Coq.

As for the mechanical translation of the QE–GdA into the CC–GdA, this is not an issue once references are resolved, and unified binders are disambiguated.

While static analysis suffices to resolve all references in the QE–GdA, and to disambiguate the majority of its approximately 47000 unified binders, almost 3000 such binders can be disambiguated only by observing their reductional behavior during validation in a calculus that accepts Aut–QE and CC at once.

To this end, we rely on $\lambda\delta$ version 3, a system that the author is proposing in this article for the first time. The former versions of the calculus are in [Gui09b, Gui14].

In particular, the mechanical translation reported in this article works as follows: by static analysis (Section 2), followed by dynamic analysis (Section 3), we build the $\lambda\delta$–GdA: a representation of the QE–GdA in $\lambda\delta$. This is mapped straightforwardly to the CC–GdA, coded in the Gallina specification language. Our final

outcome, a user-level script fully accepted by Coq, is hosted on $\lambda\delta$ Web site at <http://lambdadelta.info/download/grundlagen_2.v>. Our conclusions are in Section 4, where we show some benchmarks and some examples of the translation.

2. STATIC ANALYSIS OF THE "GRUNDLAGEN"

Landau's "Grundlagen der Analysis" [Lan65] contains 301 propositions on the arithmetics of rational, irrational and complex numbers. This theory was digitally specified in the language Aut–QE [vD94a] by Jutting [vB77]. Later, it was recovered from Jutting's original files by Wiedijk, who included it in the latest distribution of his validator for Aut–68 [vB94a] and Aut–QE.

Unfortunately, we have scarce practical information on the specification, and we are still missing Jutting's detailed explanation of the QE–GdA (five volumes titled "A translation of Landau's Grundlagen in AUTOMATH" of which only the cover pages are available). Here is a summary of what we know up to now [Gui09a].

—The *concrete syntax*, found in [Wie99], (see Section 2.1) relies on the next facilities meant to decrease the verbosity of Automath books.

—The *block system* allows to share the formal parameters that several global constants have in common (see Section 2.2). Actually, the discussion on instantiation in [dB91] explains that this is more than a mere facility.

—The *paragraph system*, briefly mentioned in [Zan94] and fully explained in [vB77], allows to reuse identifiers avoiding name collisions (see Section 2.3).

—The *abbreviation system* (*i.e.*, the *shorthand facility* [vD94a]) allows to omit some actual parameters in a reference to a global constant (see Section 2.4).

In any case, some aspects of these facilities seem undocumented and thus remain ambiguous to us. As a reasonable way out, we checked how these ambiguities are solved in the QE–GdA knowing that the specification must be correct as it stands.

Contrary to Aut–QE, the formal system $\lambda\delta$ (see Section 3.1) is an abstract language not supporting any facility in the first place. Therefore, in the first step of our translation, a static analyzer removes the Aut–QE shorthand from the QE–GdA and disambiguates most of its unified binders as we explain in Section 2.5.

The product of this step is a raw version of the $\lambda\delta$–GdA in which the remaining binders are still ambiguous and, thus, need additional processing.

Section 2.1, Section 2.2, Section 2.3, and Section 2.4 present in a single text the most accurate description of Automath concrete syntax and its related facilities.

2.1 Parsing

The grammar recognized by our Automath parser is presented in Figure 2 (our notational conventions for displaying grammars are in Figure 3, the monospaced typeface is just presentational). Properly nested comments are accepted. It should be noted that the Automath grammar evolved through time and has many variants [NGdV94], which we try to capture. Our parser recognizes "`_E`" in place of the original underscored "E", anyway this notation does not appear in the QE–GdA.

We recall that a text written in an Automath language (also known as an Automath *book*), is structured as a sequence of lines, each asserting a statement.

The following kinds of statement are available:

```
(contents)
  <book>    ::= [ <line> ]* <EOF>
  <line>    ::= <section> | <context> | <opener> | <decl> | <def>
  <section> ::= "+" [ "*" ]? <id> | "-" <id> | "--"
  <context> ::= <STAR> | <qid> <STAR>
  <opener>  ::= <id> <DEF> <EB> <E> <term>
             |  <id> <E> <term> <DEF> <EB>
             |  "[" <id> <OF> <term> "]"
  <decl>    ::= <id> <DEF> <PN> <E> <term>
             |  <id> <E> <term> <DEF> <PN>
  <def>     ::= <id> <DEF> [ "~" ]? <term> <E> <term>
             |  <id> <E> <term> <DEF> [ "~" ]? <term>
  <term>    ::= <TYPE> | <PROP>
             |  <qid> [ "(" [ <term> [ "," <term> ]* ]? ")" ]?
             |  "[" <id> <OF> <term> "]" <term>
             |  "<" <term> ">" <term>
  <qid>     ::= <id> [ `"` [ <id> ]? [ <PATH> <id> ]* `"` ]?
  <id>      ::= [ "0"-"9" | "A"-"Z" | "a"-"z" | "_" | "'" | "`" ]+
(presentational variants)
  <STAR>    ::= "*" | "@"
  <DEF>     ::= ":=" | "="
  <EB>      ::= "---" | "'eb'" | "EB"
  <PN>      ::= "???" | "'pn'" | "PN" | "'prim'" | "PRIM"
  <E>       ::= "_E" | "'_E'" | ";" | ":"
  <OF>      ::= ":", ","
  <TYPE>    ::= "'type'" | "TYPE"
  <PROP>    ::= "'prop'" | "PROP"
  <PATH>    ::= "-", "."
  <EOF>     ::= ";" | eof
(spaces and comments)
  <space>   ::= [ space | tab | newline ]+
  <comment> ::= [ "#" | "%" ] [ . ]* [ newline | eof ]
             |  "{" [ . ]* "}"
```

Fig. 2. Automath concrete syntax.

`" "`	the enclosed characters	`|`	choice
`` `"` ``	the character "	`[ ]?`	optional
`space tab newline eof`	special characters	`[ ]*`	zero or more
`-`	any character in the specified range	`[ ]+`	one or more
`.`	any character	`[ ]`	bracketing

Fig. 3. Conventions for displaying the concrete syntax.

—Sectioning-related statement. This statement opens or closes a *paragraph* (a better translation would be a *section* as pointed out in [Wie99]). Automath paragraphs are possibly nested named scopes in which the global constants are declared or defined. It should be noted that a previously closed scope can be reopened. Moreover, in a well-formed Automath book, sections are properly nested so only the last opened section can be closed.

—Block opener. This statement introduces a local declaration in a given context. The semantics of context formation is recalled in Section 2.2.

—Global declaration. This is like the block opener but the declaration is global.
—Global definition. This statement defines an identifier as an abbreviation of a term explicitly typed in a given context. There is a way, never used in the QE–GdA, to inhibit the δ-expansion of the identifier for speeding reduction.

An identifier declared or defined by a statement which is not sectioning-related, is termed a *notion*. Moreover, the notions that are not block openers will be termed *global constants*. Automath terms and types are λ-terms of a single syntactic category comprising two sorts, references, unified typed abstractions, and binary applications. References to global constants may have actual parameters and a section indicator acting as a qualifier (see Section 2.3).

2.2 Context Chains

The block system is a peculiar feature of concrete Automath languages [NGdV94]. In principle, a global constant has a list of formal parameters that are retrieved by following a chain of block openers, each representing a parameter declaration. To this end, every notion has a *context marker* indicating the start of its chain (*i.e.*, its context). The rules for constructing this chain, follow.

—If the notion has an empty marker, then its chain is empty.
—If the notion has a reference marker pointing to a block opener, then its chain contains the chain of the block opener plus the block opener itself.
—If the notion has no marker and the preceding statement is a block opener, then the intended marker is a reference to it.
—If the notion has no marker and the preceding statement is a global declaration or definition, then the intended marker is the one of that statement (recursively).

The intended meaning of "preceding" in the last two rules becomes unclear when the paragraph system is in effect. Given that the QE–GdA becomes incorrect if "preceding" is understood literally, we argue that the block system must be aware of the paragraph system somehow. In particular, "preceding" reasonably means "preceding in the same section or in its parent", but may also mean "preceding in the same section fragment or in its parent" (recall that sections can be closed and reopened, thus a section might be divided in many fragments).

The QE–GdA does not help to solve this ambiguity because Jutting always reopens a section with a statement having an explicit context marker.

2.3 Paragraphs

The proper lines of an Automath text using the paragraph system facility, *i.e.*, the lines that are not sectioning-related, are grouped into possibly nested named sections. Furthermore, a *complete index* is assigned to each such line. This is the list of the sections' names containing that line, sorted according to the outermost-to-innermost order. The paragraph system specification requires a *cover* section (named "`1`" in the QE–GdA) enclosing the entire book, so a complete index is never empty. A section can be reopened at the same nesting level but two sections having the same name can not be nested. Once the index of a line is computed, that line receives a URI based on that index [Gui09a]. Note that the *rule for constants*

stated in the specification of the paragraph system, implies that different proper lines of a well-formed book always receive different URI's.

A reference r in a line l can have a complete index or an *incomplete index* or no index at all. Such a reference is resolved by computing either the position index of the referred local declaration, or the URI of the referred notion.

The original resolution rules from [vB77] Appendix 2, are given next.

—If r has a complete index j, being the concatenation of the component s before the list j_t, and if the line l has the complete index i, being the concatenation of the list i_h before the component s and before the list i_t, then a constant with r's name is looked up in the section whose index is the concatenation of i_h before j. Such a constant must exist and r receives its URI.

—If r has the incomplete index j and if the line l has the complete index i, then a constant with r's name is looked up in the section whose index is the concatenation of i before j. Such a constant must exist and r receives its URI.

—If r has no index, a declaration with r's name is looked up in the local environment of r. If such a declaration exists, r receives its depth index [dB94].

—On the other hand, a constant with r's name is looked up in the sections containing the line l, sorted according to the innermost-to-outermost order. Such a notion must exist and r is resolved by receiving its URI.

—If r is a context marker, then r must refer to a block opener otherwise r must refer to a global constant, or to a declaration in r's local environment.

Our implemented processor generalizes the first two rules by extending the search for a notion that should be in a section, say k, to the sections containing k, sorted according to the innermost-to-outermost order. This mechanism agrees with the forth rule and allows to regard a reference without an index as having the complete index of the line in which it occurs. We remark that a reference without an index is resolved first in its local environment and then in the global environment. This seems to be the originally intended order of precedence because the QE–GdA fails to validate if we reverse this precedence [Wie99].

2.4 Implicit Arguments

The abbreviation system [vD94a] is a facility of some concrete Automath languages including the extension of Aut–QE that Jutting used for the QE–GdA.

This facility works as follows: suppose that a constant c is defined or declared in a context Γ of formal parameters, say $x_1, \ldots, x_n$. Then a reference to c in a subsequent line, say l, generally needs to be applied to n actual parameters and thus appears like $c(t_1, \ldots, t_n)$. Nevertheless, if the context Γ is an initial segment of the context of the notion defined or declared in the line l, where the reference to c appears, this reference is allowed to take less than n actual parameters and $c(t_{m+1}, \ldots, t_n)$ must be interpreted as $c(x_1, \ldots, x_m, t_{m+1}, \ldots, t_n)$. Here we are assuming $m \leq n$, thus all actual parameters may be omitted in some cases.

2.5 Static Disambiguation of Unified Binders

Our static analyzer implements two strategies for disambiguating the binders of the QE–GdA. One strategy is degree-based [Bro11], while the other is position-based.

Note that both strategies rely on the fact that the QE–GdA is in β-normal form.

In the disambiguation process each binder receives a *layer constant*, which is either "Π", or "λ", or else it receives a *layer variable* in case of ambiguity.

—According to the degree-based strategy, we compute the degree of a binder by innermost-to-outermost propagation. Since Aut–QE features three degrees of terms, we argue that a binder of lowest degree (*i.e.*, one) is a "Π", whereas a binder of highest degree (*i.e.*, three) is a "λ". A binder of degree two remains ambiguous. This strategy is not applied to the block-opening binders.

—According to the position-based strategy, a block-opener in the context of a declared constant is a "Π". On the other hand, a block-opener in the context of a defined constant is a "λ", that is β-reduced when that constant is referred to. Moreover, a binder placed along the *spine* of a term representing a type annotation, must be a "Π". The other binders remain ambiguous.

The positioning information is computed by outermost-to-innermost propagation. Therefore, our our static disambiguation procedure is bidirectional.

Finally, our analyzer annotates all binders with their sort (*i.e.*, either "Type", or "Prop") as hints for presenting Π-binders as ∀-binders when their sort is "Prop".

3. DYNAMIC ANALYSIS OF THE "GRUNDLAGEN"

The raw version of the $\lambda\delta$–GdA produced by our static analyzer is a global environment of $\lambda\delta$ version 3, a calculus we introduce in Section 3.1 for the first time.

To the end of managing the ambiguous binders, we allow layer variables (say: ϕ, ψ) in abstractors. Therefore, a typical ambiguous abstraction looks like $\lambda_x^{\phi} W.T$.

The raw $\lambda\delta$–GdA is analyzed by applying a validation procedure that produces a system of constraints on the layer variables (see Section 3.5). Once this system is solved (also by correcting some points of the QE–GdA as we explain in Section 3.6), the proper $\lambda\delta$–GdA, without layer variables, is mapped to the CC–GdA, *i.e.*, our final outcome. This is a single user-level script that can be presented to Coq.

The apparatus for validating in $\lambda\delta$ version 3 derives essentially from [Gui10], which refers to a previous version of the calculus, and consists of a reduction machine (Section 3.2), two controllers (Section 3.3) asserting applicability and convertibility in context, and a validator (Section 3.4) implementing top-level verification.

3.1 The Formal System $\lambda\delta$ Version 3: "To Π ... and Beyond"

The formal system $\lambda\delta$, introduced in [Gui09b], is a typed λ-calculus inspired by Λ_∞ [vB94b], The present version 3, designed for the dynamic analysis of the QE–GdA, extends Λ_∞ with *type inclusion by reduction*. Using this device for the formation of universes, that we deduced from *sort inclusion* [Zan94], our calculus can integrate Λ_∞ with Aut–QE and CC. Although the system still lacks a theoretical study, it is based on a previous calculus [Gui14] whose desirable properties are certified.

The calculus comprises three components: the generative grammar (Definition 1), the transition system (Definition 2), and the validity rules (Definition 3).

Definition 1. We define terms and environments in Figure 4. References occur by name. We assume Barendegt's convention in that the same name does not occur free and bound in the same term or judgment, nor is bound more than once. Layers are integers extended with ∞. For $n < \infty$ we set: $\infty - n = \infty + n = \infty$. ▲

index:	k,n	::= integer	$0 \le k, n < \infty$
name:	u,x	::= identifier	Barendegt's convention is assumed
layer:	e	::= ext. integer	$0 \le e \le \infty$
term:	T,U,V,W	::= $\star k$	sort of index k
		$\mid$ $\$u$	reference to global name u
		$\mid$ $\#x$	reference to local name x
		$\mid$ $\delta_x V.T$	local abbreviation $(x \coloneqq V)$ in T
		$\mid$ $\lambda^e_x W.T$	local abstraction in layer e of $(x : W)$ in T
		$\mid$ $@V.T$	applicative term $(T\ V)$
		$\mid$ $©U.T$	type-annotated term $(T : U)$
local env.:	K,L	::= $\star$	empty environment
		$\mid$ $L.\delta_x V$	local definition $(x \coloneqq V)$ after L
		$\mid$ $L.\lambda^e_x W$	local declaration $(x : W)$ in layer e after L
global env.:	F,G	::= $\star$	empty environment
		$\mid$ $G.\delta_u V$	global definition $(u \coloneqq V)$ after G
		$\mid$ $G.\lambda_u W$	global declaration $(u : W)$ after G

Fig. 4. The abstract syntax of terms and environments.

$$\frac{}{G_1.\lambda_u W.G_2, L \vdash \$u \overset{0}{\twoheadrightarrow}_h \$u}\,\rho\$ \qquad \frac{}{G, L_1.\lambda^e_x W.L_2 \vdash \#x \overset{0}{\twoheadrightarrow}_h \#x}\,\rho\#$$

$$\frac{}{G, L \vdash \star k \overset{n}{\twoheadrightarrow}_h \star h^n(k)}\,s \qquad \frac{G, L \vdash V_1 \overset{0}{\twoheadrightarrow}_h V_2 \quad G, L \vdash T_1 \overset{n}{\twoheadrightarrow}_h T_2}{G, L \vdash @V_1.T_1 \overset{n}{\twoheadrightarrow}_h @V_2.T_2}\,\nu$$

$$\frac{G, L.\delta_x V \vdash T_1 \overset{n}{\twoheadrightarrow}_h T_2}{G, L \vdash \delta_x V.T_1 \overset{n}{\twoheadrightarrow}_h \delta_x V.T_2}\,\sigma \qquad \frac{G, L \vdash W_1 \overset{0}{\twoheadrightarrow}_h W_2 \quad G, L.\lambda^{e-n}_x W_1 \vdash T_1 \overset{n}{\twoheadrightarrow}_h T_2}{G, L \vdash \lambda^e_x W_1.T_1 \overset{n}{\twoheadrightarrow}_h \lambda^{e-n}_x W_2.T_2}\,x$$

$$\frac{G_1, L \vdash V_1 \overset{n}{\twoheadrightarrow}_h V_2}{G_1.\delta_u V_1.G_2, L \vdash \$u \overset{n}{\twoheadrightarrow}_h V_2}\,\delta\$ \qquad \frac{G, L_1 \vdash V_1 \overset{n}{\twoheadrightarrow}_h V_2}{G, L_1.\delta_x V_1.L_2 \vdash \#x \overset{n}{\twoheadrightarrow}_h V_2}\,\delta\#$$

$$\frac{G, L \vdash \delta_x V_2.@V_1.T_1 \overset{n}{\twoheadrightarrow}_h T_2}{G, L \vdash @V_1.\delta_x V_2.T_1 \overset{n}{\twoheadrightarrow}_h T_2}\,\theta \qquad \frac{G, L \vdash \delta_x(©W.V).T_1 \overset{n}{\twoheadrightarrow}_h T_2}{G, L \vdash @V.\lambda^e_x W.T_1 \overset{n}{\twoheadrightarrow}_h T_2}\,\beta$$

$$\frac{G, L \vdash T_1 \overset{n}{\twoheadrightarrow}_h T_2}{G, L \vdash \delta_x V.T_1 \overset{n}{\twoheadrightarrow}_h T_2}\,\zeta \qquad \frac{G, L \vdash T_1 \overset{n}{\twoheadrightarrow}_h T_2}{G, L \vdash \lambda^e_x W.T_1 \overset{n}{\twoheadrightarrow}_h T_2}\,\upsilon$$

$$\frac{G, L \vdash T_1 \overset{n}{\twoheadrightarrow}_h T_2}{G, L \vdash ©U.T_1 \overset{n}{\twoheadrightarrow}_h T_2}\,\epsilon \qquad \frac{G, L \vdash U_1 \overset{n}{\twoheadrightarrow}_h U_2}{G, L \vdash ©U_1.T \overset{n+1}{\twoheadrightarrow}_h U_2}\,e$$

$$\frac{G_1, L \vdash W_1 \overset{n}{\twoheadrightarrow}_h W_2}{G_1.\lambda_u W_1.G_2, L \vdash \$u \overset{n+1}{\twoheadrightarrow}_h W_2}\,l\$ \qquad \frac{G, L_1 \vdash W_1 \overset{n}{\twoheadrightarrow}_h W_2}{G, L_1.\lambda^e_x W_1.L_2 \vdash \#x \overset{n+1}{\twoheadrightarrow}_h W_2}\,l\#$$

Side conditions: $n < e$ for β and $e \le n$ for υ; x not free in T_1 for ζ and υ.

Fig. 5. The rt-transition system.

Definition 2. Given a function h chosen at will as long as $\forall k.\ k < h(k)$, and given an integer n such that $0 \le n < \infty$, the predicate $G, L \vdash T_1 \overset{n}{\twoheadrightarrow}_h T_2$ defined in Figure 5 states that T_1 computes to T_2 in our rt-transition system [vD94b].
Transition occurs in the context of G and L, and with respect to h and n.
The system comprises structural schemes, reduction schemes (the r-transitions), and type inference schemes (the t-transitions). Thus, the *height* n indicates the desired difference of degree between T_1 and T_2 occurring because of the t-transitions.

$$\frac{}{G, L \vdash \star k\ !_h}\,S \qquad \frac{G, L \vdash V\ !_h \quad G, L.\delta_x V \vdash T\ !_h}{G, L \vdash \delta_x V.T\ !_h}\,F \qquad \frac{G, L \vdash W\ !_h \quad G, L.\lambda^e_x W \vdash T\ !_h}{G, L \vdash \lambda^e_x W.T\ !_h}\,C$$

$$\frac{G_1, L \vdash V\ !_h}{G_1.\delta_u V.G_2, L \vdash \$u\ !_h}\,D\$ \qquad \frac{G, L_1 \vdash V\ !_h}{G, L_1.\delta_x V.L_2 \vdash \#x\ !_h}\,D\#$$

$$\frac{G_1, L \vdash W\ !_h}{G_1.\lambda_u W.G_2, L \vdash \$u\ !_h}\,I\$ \qquad \frac{n, G \vdash L_1\ !_h W}{G, L_1.\lambda^e_x W.L_2 \vdash \#x\ !_h}\,I\#$$

$$\frac{G, L \vdash U\ !_h \quad G, L \vdash T\ !_h \quad G, L \vdash U \overset{0}{\twoheadrightarrow}_h U_0 \quad G, L \vdash T \overset{1}{\twoheadrightarrow}_h U_0}{g, L \vdash ©U.T\ !_h}\,T$$

$$\frac{G, L \vdash V\ !_h \quad G, L \vdash T\ !_h \quad G, L \vdash V \overset{1}{\twoheadrightarrow}_h W_0 \quad G, L \vdash T \overset{n}{\twoheadrightarrow}_h \lambda^{e+1} W_0.U_0}{G, L \vdash @V.T\ !_h}\,\mathrm{A}*$$

$$\frac{}{\vdash \star\ !_h}\,S\$ \qquad \frac{\vdash G\ !_h \quad G, \star \vdash V\ !_h}{\vdash G.\delta_u V\ !_h}\,F\$ \qquad \frac{\vdash G\ !_h \quad G, \star \vdash W\ !_h}{\vdash G.\lambda_u W\ !_h}\,C\$$$

Side condition for A$*$: Figure 6(υ) not allowed when reducing the spine of T (forth premise).

Fig. 6. The validity of terms and global environments.

$$\frac{G, L \vdash V\ !_h \quad G, L \vdash T\ !_h \quad G, L \vdash V \overset{1}{\twoheadrightarrow}_h W_0 \quad G, L \vdash T \overset{1}{\twoheadrightarrow}_h \lambda^{e+1} W_0.U_0}{G, L \vdash @V.T\ !_h}\,\mathrm{A1}$$

Side condition for A1: Figure 6(υ) not allowed when reducing the spine of T (forth premise).

Fig. 7. The restricted applicability condition.

Firstly, the structural steps are: $\rho\$$, $\rho\#$, s for $n = 0$ (termed $\rho\star$), ν, σ, and x for $n = 0$ (termed ξ). Secondly, the reduction steps are: β (function application), $\delta\$$ (global expansion), $\delta\#$ (local expansion), ζ (abbreviation removal), θ (commutation of application-abbreviation pair as in [CH00]), ϵ (annotation removal), and our υ (λ^0-abstraction removal) that generalizes *sort inclusion*. Thirdly, the type inference steps are: s for $n > 0$ (sort typing), $l\$$ (global reference typing), $l\#$ (local reference typing), x for $n > 0$ (abstraction typing) and e (annotation typing). ▲

Definition 3. The predicate $G, L \vdash T\ !_h$ defined in Figure 6 states the validity of T (*i.e.*, its syntactical correctness) with respect to h in the context of G and L. In particular, Figure 6(A$*$) states the *extended* applicability condition.
The predicate $\vdash G\ !_h$ defined in Figure 6 states the validity of G w.r.t. h.
A type judgment is defined by setting: $G, L \vdash T :_h U$ iff $G, L \vdash ©U.T\ !_h$. ▲

It is a fact that sort inclusion and β-contraction yield a non-confluent critical pair when applied to the same λ-abstraction. In this respect, Automath's approach [Zan94] is to apply sort inclusion only when β-contraction is not applicable.

On the other hand, the key idea behind $\lambda\delta$ version 3 is to apply sort inclusion and β-contraction to different λ-abstractions. To this end, a λ^e-abstraction is provided for each value of the integer *layer* e in the range $0 \le e \le \infty$. In this setting, sort inclusion applies to λ^0-abstractions in the form $G, L \vdash \lambda^0_x W.T \overset{0}{\twoheadrightarrow}_h T$ (x not free in T) implied from Figure 5(υ), while β-contraction applies to λ^{e+1}-abstractions as we imply from Figure 5(β) when $n = 0$. Thus, a λ^{e+1}-abstraction is a head normal form and as such it appears in Figure 6(A$*$), that states our applicability condition.

The infinitely many λ^e-abstractions are linked by stating that a λ^e-abstraction is canonically typed by a λ^{e-1}-abstraction, as we imply from Figure 5(x) when $n = 1$.

In this respect, λ^∞ is typed by λ^∞ (as one expects), and λ^0 is typed by λ^0.

Nevertheless, the strict monotonicity condition $k < h(k)$ of Definition 2, applied to Figure 5(s) when $n = 1$, yields that the sort of index k is typed by a sort of higher index. Therefore, a term is never typed by itself (as the reader might fear).

As of λ^1, the assumption $G, L \vdash T \overset{1}{\twoheadrightarrow}_h \star k$ (*i.e.*, T is typed by the sort of index k) yields by Figure 5(υ) the conclusion $G, L \vdash \lambda^1_x W.T \overset{1}{\twoheadrightarrow}_h \star k$ (*i.e.*, $\lambda^1_x W.T$ is typed by the same sort). The situation is explained by observing that $\lambda^1_x W.T$ types with $\lambda^0_x W.\star k$ which υ-reduces to $\star k$, and suggests that λ^1 corresponds to Π in CC.

This digression yields with no surprise that λ^2 must stand for λ in CC.

As to λ^0, since it types Π, we agree to term it *hyper* Π, which literally means *beyond* Π. Its role in Aut–QE is that of constructing some *quasi-expressions*.

In principle, υ-contraction can include big universes into small ones, so, eventually, restrictions might apply to the term W of Figure 5(υ) in order to prove strong normalization and to avoid inconsistency. These restrictions will influence the λ^e-abstractions with $0 \le e < \infty$, while the λ^∞-abstractions will not be influenced. In fact, we see that layer 0 cannot be reached by iterated typing from layer ∞.

Thus, $\lambda\delta$ version 3 accounts for the distinction between *instantiation* and *abstraction* [dB91] by which block openers correspond to λ^∞-abstractions, whereas local variable declarations correspond to λ^e-abstractions with $0 < e < \infty$.

However, in order to validate the QE–GdA in CC, block openers must be represented in layer 1 and 2. We will discuss the consequences of this issue in Section 4.

The restriction of $\lambda\delta$ version 3 to layer ∞ is essentially the calculus we presented in [Gui14]. Remarkably, the calculus we present in this article is a minimal-impact extension of that system, which adds just one new rule: υ-contraction.

As of our applicability condition, the extended form stated by Figure 6(A$*$), where n can have any value, is the one of Λ_∞: a calculus featuring more than three degrees of valid terms. However, in the subsystem for representing Aut–QE and CC, where only three degrees of valid terms are available, taking 1 for n suffices. Thus, we obtain the *restricted* applicability condition shown in Figure 7.

In this respect, the calculus we presented in [Gui09b] is essentially $\lambda\delta$ version 3 restricted to layer ∞ and equipped just with the restricted applicability condition.

We observe that the author of the QE–GdA relies on extended applicability to verify the constant `ande2"l-r"`, where a variable for a predicate is unified with its universal quantification four times. This issue is discussed in Section 3.6.

Looking at the premise $G, L \vdash T \overset{n}{\twoheadrightarrow}_h \lambda^{e+1} W_0.U_0$ of Figure 6(A$*$), we see that W_0 (*i.e.*, the expected type of the application argument) may depend on n. However, several side conditions can be set if we desire to avoid this dependence.

Our choice to exclude υ-contractions when deriving the premise, is implied by the use of sort inclusion in [Zan94] and has some advantages. On the one hand, υ-contraction remains fully general. On the other hand, there is no need to check the side condition of υ-contractions during the mechanical derivation of the premise.

3.2 An Overview of the Reduction and Type Machine

Mechanical validation in $\lambda\delta$ is based on the Reduction and Type Machine (RTM).

	term	$n=0$	$n>0$	n	restricted mode
		$d=1$	$d>1$	d	extended mode
01r	$\star h$	stop	$\to$s	dec.	
01x		stop			
02r	$©U.T$	$\to\epsilon$	$\to$e	dec.	
02x		$\to\epsilon$		as is	
03r	$\lambda^e W.T$ before up.	as is	$\to$x	as is	to *after up.*
03x		as is		$d \wedge e$	
04	$\lambda^e W.T$ after up. with empty stack	stop		as is	restarts with $\to\xi$
05	$\lambda^{e+1} W.T$ after up. with $@V$ on stack	$\to\beta$		as is	also if $e+1$ is ϕ
06	$\lambda^0 W.T$ after up. with $@V$ on stack	error			invalid application
07	$\delta_x V.T$	$\to\sigma$		as is	push $\delta_x V$ on env.
08	$@V.T$	$\to\nu$		as is	push $@V$ on stack
09	dangling $\$x$ or $\#x$	error			invalid reference
10	$\$u$ with $\delta_u V$ on env.	stop		as is	restarts with $\to\delta\$$
11	$\#x$ with $\delta_x V$ on env.	$\to\delta\#$		as is	
12	$\$u$ with $\lambda_u W$ on env.	stop	$\to$l\$	dec.	
13	$\#x$ with $\lambda_x^{e+1} W$ on env.	stop	$\to$l#	dec.	also if $e+1$ is ϕ
14	$\#x$ with $\lambda_x^0 W$ on env.	error	$\to$l#	dec.	failure of υ

(1) $\to$s is: $\star k \to \star h(k)$
(2) $\to$x is: $\lambda^e W.T \to \lambda^{e-n} W.T$ for $0 < n$
(3) $\to\xi$ is: push $\lambda_x^e W$ on env.
(4) column n and d: value after step (dec.: decrement if positive, $\wedge$: minimum)
(5) single line: same operation in both modes

Fig. 8. Operational semantics of the RTM.

This is an abstract machine of the K family [Kri07], defined first in [Gui10], that computes the weak head normal forms by rt-reduction according to Figure 5.

The RTM is a *controlled* machine, in that it can stop before global δ-expansions and υ-contractions, and then it can be restarted by the calling controller. In fact, these reductions are better managed on the controller's side (see Section 3.3).

Our verification procedure involves the RTM to test the next conditions:

(1) convertibility for type annotation: premises 3 and 4 of Figure 6(T);
(2) convertibility for application: premise 3 of Figure 6(A$*$) and of Figure 7(A1);
(3) restricted applicability: premise 4 of Figure 7(A1).
(4) extended applicability: premise 4 of Figure 6(A$*$);

In the last case the computation's height n is not known in advance. In the other cases, instead, this height is set to 0 or 1. Thus, the RTM runs in two modes.

—When the the caller specifies a value for n, the RTM runs in the restricted mode, suitable for cases (1) to (3). In this mode the e-step (when applicable) is preferred to ϵ-step. This means that, in case of type inference, we use expected types if we know them. The reader should note the type annotation in Figure 5(β).
Operating on a valid term, the RTM detects an error just upon failure of the side condition for the υ-step, which the machine checks with a lazy policy.
—When the caller does not specify a value for n, the RTM needs an upper bound d for it (∞ is allowed) and runs in the extended mode, suitable for case (4).

In this mode, targeted at extended applicability, the RTM looks for a λ^{e+1}-abstraction by repeatedly increasing the height n of the computation, or else it stops as soon as such an abstraction cannot be found in the range $0 \le n < d$.
We remark that, on valid terms, this problem is decidable even for $d = \infty$.
The RTM increases n just when forced by an l-step, so s-steps, e-steps and x-steps are disabled. Anyway, care is taken to ensure the side condition for $\rightarrow\beta$.
Suppose the RTM finds the β-redex $@V.\lambda^e_x W.T_0$ and computes it. If the height of this computation needs be increased by n because of following l-steps, the redex eventually becomes $@V.\lambda^{e-n}_x W.T_n$ and remains valid only if $n < e$. The RTM ensures this condition by updating its height's upper bound to at most e on encountering a λ^e-abstraction. Thus, we explain the need for the upper bound.

The raw QE−GdA coming from the static analyzer, is verified under the restricted applicability condition and contains abstractions with layer variables. Thus, the RTM is instructed to deal with these variables in the restricted mode.

In particular, when a decision must be made on $\lambda^\phi_x W$ whether to consider $\phi > 0$ or $\phi = 0$ for an unknown ϕ, the first alternative is chosen being the safest. In fact, choosing $\phi = 0$ may lead the RTM to an error condition (either invalid application, or failure of υ-step). It follows that on $\lambda^\phi_x W.T$, the β-step is taken whenever possible, in accordance with the original verification algorithm for Aut−QE [Zan94].

Figure 8 displays, informally but precisely, the operational semantics of the RTM updated for $\lambda\delta$ version 3. The extended mode is included for completeness.

We stress that being a machine of the K family, the RTM avoids ζ-contractions, which are known as *delifting steps* following a well-established terminology.

3.3 An Overview of the Controllers

The RTM's are operated by two controllers: the *comparator* and the *applicator*.

The comparator asserts the convertibility between an expected type U and the inferred type of a term V. To this end, it starts a machine on U with $n = 0$, and a machine on V with $n = 1$. So, both machines run in the restricted mode.

The conversion test occurs by levels, *i.e.*, by repeated comparison of weak head normal forms. The comparison policy follows [Gui10] and is given next.

(1) Two sorts are compared by their index. The arguments stacked by the two machines are not considered since the RTM's run on valid terms.
(2) Two references to local abstractions are compared by their level, *i.e.*, not by their depth, thus these references are not relocated. Information on the level of each reference is provided by the respective RTM. In case of match, we assert the convertibility of the arguments stacked by the two machines.
(3) Two references to global declarations are compared by URI. In case of match, we assert the convertibility of the arguments stacked by the two machines.
(4) Two references to global definitions are compared by URI. In case of match, we test the convertibility of the arguments stacked by the two machines and, if this test fails, we δ-expand both definitions. In case of mismatch, we δ-expand the *older* definition according to an *age* system we shall explain.
(5) A reference to a global definition compared to any other term, is δ-expanded.
(6) An abstraction in layer 0 is υ-contracted by restarting its machine.

(7) Two abstractions are compared by their layer and, in case of match, we assert the convertibility of their arguments. Variable layers are accepted.

(8) A variable layer abstraction compared to any other term is υ-contracted. The earlier definition of this controller [Gui10] was asymmetric in this clause. Following [Zan94] too closely, we contracted just an abstraction coming from the term V compared with a sort coming from U. Moreover, we disabled this clause when leaving the *spine* of V to avoid an evident inconsistency [Gui09a], which disappears by restricting υ-contraction to abstractions in layer 0.

It is a known fact that the QE–GdA is validated faster if we limit δ-expansions during convertibility tests. To this end, the comparator implements *age-controlled* δ-expansions [Zan94]. In particular, a progressively increasing integer $\$u$ is assigned to each constant u after its static analysis. Thus, constants become totally ordered.

The lines of the QE–GdA appear in order of dependence, so a constant u_1 processed before a constant u_2, *i.e.*, satisfying $\$u_1 < \$u_2$, cannot depend on u_2. So, when we compare a reference to u_1 with a reference to u_2, it is safe to δ-expand just the reference to u_2. In this respect, u_2 is the *older* constant of the two.

The second controller, the applicator, tests the applicability of a term T by starting a machine on T with $n = 1$ or with $d = \infty$ depending on the mode desired by the user. The test succeeds if the machine stops on $\lambda_x^{e+1} W_0.U_0$ where the layer may be a variable. In particular, the controller operates thus:

(1) On an abstraction, its layer is tested for positivity.

(2) On a reference to a definition, the machine is restarted, so a δ-expansion occurs.

(3) in the other cases the test fails.

The machine is not restarted on λ^0-abstractions, thus υ-contractions are avoided as the side condition for applicability specifies. See Figure 6(A$*$) and Figure 7(A1).

3.4 An Overview of the Validator

Indeed, the validator is the simpler component of our verification system. To some extent, it follows [Gui10] but, remarkably, we replace the canonical type synthesizer with a procedure to assert the validity relation $\vdash G\,!_h$ of Figure 6.

As a result, the overall validation of the $\lambda\delta$–GdA becomes 1% faster on average (the type synthesizer is still available, thus we can compare the two approaches).

Our point is that computing the canonical type of a term, is more expensive than just asserting its existence. Contrary to the rules of canonical typing, the rules of validity are relocation-free, *i.e.*, they do not involve *lifting*, following a well-established terminology. Thus, in the end, we achieve the long awaited fully relocation-free verification process we advocated in [Gui09a].

Looking at Figure 6(A$*$) and Figure 7(A1), the validator asserts the applicability of T to V by calling the applicator on T, which provides a RTM, say M, ready on $\lambda^{e+1} W_0.U_0$. Secondly, it calls the comparator on W_0 and on V, using M in its current state for W_0. On the contrary, the RTM for V has an initial state.

3.5 Dynamic Disambiguation of Unified Binders

When layer variables are allowed, the discussed validation procedure produces a set of constraints on such variables. These constraints are of four kinds.

```
 all"l"
-all:=p:'prop'
+all:=[x:sigma]<x>p:'prop'
```

```
 imp"l-r"
-imp:=b:'prop'
+imp:=[x:a]<x>b:'prop'
```

```
 ande2"l-r"
-b@[a1:and(a,b)]
-ande2:=<ande1(a,b,a1)>ande2"l"(a,b,a1):<ande1(a,b,a1)>b
+b@[a1:and(a,imp(a,b))]
+ande2:=<ande1(a,imp(a,b),a1)>ande2"l"(a,imp(a,b),a1):<ande1(a,imp(a,b),a1)>b
```

```
 some"l"
-some:=not(non(p)):'prop'
+none:=all(sigma,non(p)):'prop'
+some:=not(none(p)):'prop'
```

∓`non` replaced by `none` in 5 "block openers" and in the constants:
`th1"l-some"`, `th3"l-some"` (two times), `th5"l-some"`, `empty"l-e-st"`, `t5"l-e-st-isset"` (two times), `t13"l-e-st-eq-landau-n-327"`, and `t38"l-e-st-eq-landau-n-327"`

Marks: − (old text), + (new text), ∓ (multiple replacement)

Fig. 9. Static corrections to the QE–GdA.

(1) The constraint $\phi - n = \psi$ is generated by the RTM on an x-step from the abstraction $\lambda^{\phi}W.T$ to the abstraction $\lambda^{\psi}W.T$. See line 03r of Figure 8.
(2) The constraint $\phi = \psi$ is generated by the comparator matching two abstractions $\lambda^{\phi}W_1.T_1$ and $\lambda^{\psi}W_2.T_2$ See Clause (7) of the comparator in Section 3.3.
(3) The constraint $\phi = 0$ is generated by the comparator υ-reducing the abstraction $\lambda^{\phi}W.T$. See Clause (8) of the comparator in Section 3.3.
(4) The constraint $\phi > 0$ is generated by the RTM when it β-reduces $\lambda^{\phi}W.T$. Moreover, it is generated when asserting the applicability of a term with functional structure $\lambda^{\phi}W_0.U_0$ See Clause (1) of the applicator in Section 3.3.

Whenever a constraint is issued, the system of known constraints is reduced by repeated substitution. Thus, inconsistencies are discovered as soon as possible.

Notably, a consequence of static disambiguation (see Section 2.5), is that the unified binders of the $\lambda\delta$–GdA can be disambiguated constant by constant.

In fact, all variables remaining in a constant u after static analysis are determined after u is validated, *i.e.*, without checking the subsequent references to u.

This means that layer variables can be reused after each constant is validated, and that the system of constraints can be kept small during validation.

We would like to stress that both static disambiguation strategies must be used in order to achieve this result, *i.e.*, they disambiguate distinct sets of binders.

3.6 A Posteriori Static Corrections

As is stands originally, the QE–GdA fails to validate in $\lambda\delta$ and in CC, since two constants require formal η-reduction on Π (*i.e.*, the inferred type $\lambda^1_x W.(@x.T)$ must reduce to the expected type T). They are `t2"l-some"` and `th2"l-r-imp"`.

Strategy	First effective use
Validation-based	`imp"l"`
Position-based (on openers)	`imp"l"`
Position-based (on constants)	`t5"l-some"`
Degree-based	`t9"l-e-st-eq-landau-n-rt-rp-r-c-v9"`

Fig. 10. Effective use of disambiguation strategies.

Step	Amount
β-contraction	907865
δ-expansion (on variables)	451799
δ-expansion (on constants)	418357

Fig. 11. Main reduction steps of the RTM.

Nevertheless, these reductions can be avoided by following a suggestion due to van Daalen and reported in [vB77], by which we apply a ∀-introduction to a predicate symbol in two constants: `all"l"` and `imp"l-r"` (see Figure 9).

With these corrections, our dynamic analysis of the QE–GdA reports 20 inconsistencies in its layer constraint system. Four of these are located in the constant `ande2"l-r"` and state that sort inclusion is required on Π (*i.e.*, υ-contraction is required in λ^1) four times. We highlight the problem in [Gui09a], noting that `ande2"l-r"` requires the *pure* type inference rule for function application [dB91], corresponding to the extended applicability condition. See Figure 6(A∗).

In [Gui14] we note that, sometimes, extended (*i.e.*, Λ_∞-like) applicability reduces to restricted (*i.e.*, CC-like) applicability by applying λ^{e+1}-introductions.

In the case of the QE–GdA, we invoke `imp"l-r"` to apply a ∀-introduction to a predicate `b` passed as a type and occurring four times (see Figure 9).

After the correction, just 12 inconsistencies remain. The first one is located in the constant `some"l"`, where the predicate `non(p)` is passed as a proposition.

We cannot apply a ∀-introduction to `non(p)` in its definition since some constants use it a predicate indeed (they are: `somei"l"`, `t1"l-some"`, `t2"l-some"`, `th1"l-some"`, `t3"l-some"`, `t4"l-some"`, and `th2"l-some"`). So, we introduce a new constant `none` for the ∀-quantified `non(p)` and we replace `non` with `none` where required (see Figure 9). Note that `none` is indeed the counterpart of `some`.

This correction solves all inconsistencies. In the end, we add 21 ∀-introductions to the original QE–GdA to obtain the CC–GdA. This result agrees with the statement that the QE–GdA validates in the PTS of [Bro11] just by formal η-expansion.

4. CONCLUSION AND FUTURE WORK

In [Gui09a] we describe the $\lambda\delta$–GdA: a presentation of the QE–GdA into the formal system $\lambda\delta$ version 2, experimentally equipped with *sort inclusion* to this end.

In this paper we take a step further by describing the CC–GdA: a presentation of the $\lambda\delta$–GdA in CC. The point at issue is assigning a *layer* to Automath's unified binders, *i.e.*, separating λ-abstractions and Π-abstractions. For this purpose, we replace $\lambda\delta$ version 2, a calculus having a single binder, with $\lambda\delta$ version 3 first introduced in Section 3.1: a system featuring infinite (actually, $\omega + 1$) binders,

Input	System	Execution (seconds)	Task
QE–GdA	Helena 0.8.2	01.02 to 01.05	disambiguation, validation in $\lambda\delta$
CC–GdA	Coq 8.4.3 (no VM)	24.26 to 24.43	type checking in CC
	Coq 8.4.3 (with VM)	run was halted after 60 minutes	

All systems are implemented in the Objective Caml programming language

Fig. 12. Time of one run (min. and max. on 31 runs).

properly managing sort inclusion through υ-contraction.

In Section 2.5 and Section 3.5 we discuss the three strategies we implemented for assigning such layers to binders. The reader should note that each strategy is effective, in that it considers binders ignored by the other strategies. We show in Figure 10 the first constant of the QE–GdA on which each strategy is effective.

Our analysis reveals that some unified binders correspond to λ-abstractions and to Π-abstractions at the same time. Brown [Bro11] has an *ad hoc* automated procedure to solve this situation by formally η-expanding these inconsistent binders.

On the contrary, our approach is to apply these expansions ($\forall$-introductions, from the logical standpoint) by hand on the QE–GdA as we show in Section 3.6.

Helena, our implemented processor for $\lambda\delta$ version 3, verifies the QE–GdA by operating the amount of β-contractions and of δ-expansions shown in Figure 11. The δ-expansions on variables come from the β-reductum of Figure 5(β).

On our hardware, a 3 GHz Intel processor (1.3 MHz bus, 12 MB cache) with 10K rpm (3 Gb/s SATA) hard drives, we measured the execution times of Figure 12 concerning Helena (processing the QE–GdA), and Coq (processing the CC–GdA).

We stress that the CC–GdA is a faithful presentation of the corrected QE–GdA in that the QE–GdA is $\alpha\delta\eta$-equivalent to the CC–GdA, once abstractions and function types are replaced by the corresponding unified binding constructions.

Automath η-equivalence solves the incompatibilities between Aut–QE and CC, δ-equivalence is introduced for convenience, and α-equivalence is necessary because of different naming conventions in the QE–GdA and in Coq.

Figure 13 shows the first ten constants of the Grundlagen as they appear in the QE–GdA and in the CC–GdA. The reader sees both definitions and axioms.

Our work shows that CC is an upper-bound system for validating the CC–GdA, but we may ask whether a subsystem can be used as well. The mechanical inspection of the abstractions occurring in the $\lambda\delta$–GdA shows the situation of Figure 14, from which we argue that validation really needs the full power of CC, including the impredicative Π's of category $(\Box, \star)$ according to the classification in [Bar93].

On the other hand, if we follow Automath's perspective and we present block openers as λ^∞-abstractions, we see that the $\lambda\delta$–GdA is valid in $\Lambda_\infty + \lambda P$.

This system does not allow impredicative constructions [Gui09b], and quoting [KLN01], is located "in the middle of the right side of the Refined Barendregt Cube, exactly in between λC and λP." In fact we note that the λ^∞-abstractions of category $(\Box, \star)$ are much less expressive than the corresponding Π-abstractions.

As of now, the CC–GdA is a user-level script consisting of a flat sequence of lines, each declaring or defining a constant of the QE–GdA in the syntax of CC.

Original presentation in the QE–GdA (Aut–QE concrete syntax)

```
+l
@[a:'prop'][b:'prop']
imp:=[x:a]b:'prop'
[a1:a][i:imp(a,b)]
mp:=<a1>i:b
a@refimp:=[x:a]x:imp(a,a)
b@[c:'prop'][i:imp(a,b)][j:imp(b,c)]
trimp:=[x:a]<<x>i>j:imp(a,c)
@con:='prim':'prop'
a@not:=imp(con):'prop'
wel:=not(not(a)):'prop'
[a1:a]
weli:=[x:not(a)]<a1>x:wel(a)
a@[w:wel(a)]
et:='prim':a
a@[c1:con]
cone:=et([x:not(a)]c1):a
```

Corresponding presentation in the CC–GdA (Coq concrete syntax)

```
Definition l_imp := (fun (a:Prop) => (fun (b:Prop) =>
                    ((forall (x:a), b) : Prop))).

Definition l_mp := (fun (a:Prop) => (fun (b:Prop) =>
                   (fun (a1:a) => (fun (i:l_imp a b) =>
                   (i a1 : b))))).

Definition l_refimp := (fun (a:Prop) =>
                       ((fun (x:a) => x) : l_imp a a)).

Definition l_trimp := (fun (a:Prop) => (fun (b:Prop) => (fun (c:Prop) =>
                      (fun (i:l_imp a b) => (fun (j:l_imp b c) =>
                      ((fun (x:a) => j (i x)) : l_imp a c)))))).

Axiom l_con : Prop.

Definition l_not := (fun (a:Prop) =>
                    (l_imp a l_con : Prop)).

Definition l_wel := (fun (a:Prop) =>
                    (l_not (l_not a) : Prop)).

Definition l_weli := (fun (a:Prop) => (fun (a1:a) =>
                     ((fun (x:l_not a) => x a1) : l_wel a))).

Axiom l_et : (forall (a:Prop), (forall (w:l_wel a), a)).

Definition l_cone := (fun (a:Prop) => (fun (c1:l_con) =>
                     (l_et a (fun (x:l_not a) => c1) : a))).
```

Fig. 13. First ten constants of the Grundlagen.

In order to be usable as a background for formalized mathematics, this script must be improved by making the original structure of the Grundlagen explicit.

abstraction	λC	CC			
unrestricted	$(\Box, \Box)$	$(\Box_T, \Box_T)$	$(\Box_T, \Box_P)$	$(\Box_P, \Box_T)$	$(\Box_P, \Box_P)$
	$(\Box, \star)$	$(\Box_T, \mathrm{Type})$	$(\Box_T, \mathrm{Prop})$	$(\Box_P, \mathrm{Type})$	$(\Box_P, \mathrm{Prop})$
	$(\star, \Box)$	$(\mathrm{Type}, \Box_T)$	$(\mathrm{Type}, \Box_P)$	$(\mathrm{Prop}, \Box_T)$	$(\mathrm{Prop}, \Box_P)$
	$(\star, \star)$	$(\mathrm{Type}, \mathrm{Type})$	$(\mathrm{Type}, \mathrm{Prop})$	$(\mathrm{Prop}, \mathrm{Type})$	$(\mathrm{Prop}, \mathrm{Prop})$
restricted	$(\star, \Box)$		$(\mathrm{Type}, \Box_P)$		$(\mathrm{Prop}, \Box_P)$
	$(\star, \star)$	$(\mathrm{Type}, \mathrm{Type})$	$(\mathrm{Type}, \mathrm{Prop})$	$(\mathrm{Prop}, \mathrm{Type})$	$(\mathrm{Prop}, \mathrm{Prop})$

Note: we assume $(\mathrm{Type} : \Box_T)$ and $(\mathrm{Prop} : \Box_P)$.

Fig. 14. The categories of abstractions in the CC–GdA.

In particular, we would like to see definitions and propositions typeset with domain-specific mathematical notation. Proofs should appear in a domain-specific language as well, and the whole matter should be organized in different files respecting the system of chapters and sections that we see in [Lan65].

Such a step will require to operate manually of the $\lambda\delta$–GdA with the help of a dedicated technology supervising crucial aspects of the work. For instance, defining and inserting notations, or applying semantics-preserving changes.

The QE–GdA, the $\lambda\delta$–GdA and the CC–GdA, as well as Helena 0.8.2, are available at $\lambda\delta$ Web site: <http://lambdadelta.info/implementation.html#v2>.

ACKNOWLEDGMENTS

I wish to thank the anonymous referee for valuable suggestions that helped me to improve this text and to achieve a clearer understanding of $\lambda\delta$ version 3.

Moreover, I wish to dedicate this work to K. Barr for having patiently awaited my e-mail replies while I was working on the formal system $\lambda\delta$ over the years.

References

[Bar93] H.P. Barendregt. Lambda Calculi with Types. *Osborne Handbooks of Logic in Computer Science*, 2:117–309, 1993.

[Bro11] C.E. Brown. Faithful Reproductions of the Automath Landau Formalization. Typescript note, 2011.

[CH88] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2-3):95–120, March 1988.

[CH00] P.L. Curien and H. Herbelin. The duality of computation. In *5th ACM SIGPLAN int. conf. on Functional programming (ICFP '00)*, volume 35/9 of *ACM SIGPLAN Notices*, pages 233–243, New York, USA, Sept 2000. ACM Press.

[Coq15] Coq development team. *The Coq Proof Assistant Reference Manual: release 8.4pl6*. INRIA, Orsay, France, April 2015.

[dB91] N.G. de Bruijn. A plea for weaker frameworks. In *Logical Frameworks*, pages 40–67. Cambridge University Press, Cambridge, UK, 1991.

[dB94] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Selected Papers on Automath [NGdV94]*, pages 375–388. North-Holland Pub. Co., Amsterdam, The Netherlands, 1994.

[Gui09a] F. Guidi. Landau's "Grundlagen der Analysis" from Automath to lambda-delta. Technical Report UBLCS 2009-16, University of Bologna, Bologna, Italy, September 2009.

[Gui09b] F. Guidi. The Formal System $\lambda\delta$. *Transactions on Computational Logic*, 11(1):5:1–5:37, online appendix 1–11, November 2009.

[Gui10] F. Guidi. An Efficient Validation Procedure for the Formal System $\lambda\delta$. In F. Ferreira, H. Guerra, E. Mayordomo, and J. Rasga, editors, *Local Proceedings of 6th Conference on Computability in Europe (CiE 2010)*, pages 204–213. Centre for Applied Mathematics and Information Technology, Department of Mathematics, University of Azores, Ponta Delgada, Portugal, July 2010.

[Gui14] F. Guidi. The Formal System $\lambda\delta$ Revised, Stage A: Extending the Applicability Condition. CoRR identifier 1411.0154, November 2014. Submitted to ACM ToCL (available at <http://lambdadelta.info/>).

[KLN01] F. Kamareddine, T. Laan, and R. Nederpelt. Refining the Barendregt Cube using Parameters. In H. Kuchen and K. Ueda, editors, *Functional and Logic Programming, 5th International Symposium (FLOPS 2001)*, volume 2024 of *Lecture Notes in Computer Science*, pages 375–389, Berlin, Germany, March 2001. Springer.

[KLN03] F. Kamareddine, T. Laan, and R. Nederpelt. De Bruijn's Automath and Pure Type Systems. In F. Kamareddine, editor, *Thirty Five Years of Automating Mathematics*, volume 28 of *Kluwer Applied Logic series*, pages 71–123. Kluwer Academic Publishers, Hingham, MA, USA, November 2003.

[Kri07] J.-L. Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3):199–207, September 2007.

[Lan65] E.G.H.Y. Landau. *Grundlagen der Analysis*. Chelsea Pub. Co., New York, USA, 1965.

[NGdV94] R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer, editors. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*, Amsterdam, The Netherlands, 1994. North-Holland Pub. Co.

[vB77] L.S. van Benthem Jutting. Checking Landau's "Grundlagen" in the AUTOMATH System. Ph.D. thesis, Eindhoven University of Technology, 1977.

[vB79] L.S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the automath system*, volume 83 of *Mathematical Centre Tracts*. Mathematisch Centrum, Amsterdam, The Netherlands, 1979.

[vB94a] L.S. van Benthem Jutting. Description of AUT-68. In *Selected Papers on Automath [NGdV94]*, pages 251–273. North-Holland Pub. Co., Amsterdam, The Netherlands, 1994.

[vB94b] L.S. van Benthem Jutting. The language theory of λ_∞, a typed λ-calculus where terms are types. In *Selected Papers on Automath [NGdV94]*, pages 655–683. North-Holland Pub. Co., Amsterdam, The Netherlands, 1994.

[vD94a] D.T. van Daalen. A Description of Automath and Some Aspects of its Language Theory. In *Selected Papers on Automath [NGdV94]*, pages 101–126. North-Holland Pub. Co., Amsterdam, The Netherlands, 1994.

[vD94b] D.T. van Daalen. The language theory of Automath. In *Selected Papers on Automath [NGdV94]*, pages 163–200 and 303–312 and 493–653. North-Holland Pub. Co., Amsterdam, The Netherlands, 1994.

[Wie99] F. Wiedijk. A Nice and Accurate Checker for the Mathematical Language Automath. Documentation of the AUT checker, version 4.1, 1999.

[Wie02] F. Wiedijk. A new implementation of Automath. *Journal of Automated Reasoning*, 29(3-4):365–387, 2002.

[Zan94] I. Zandleven. A Verifying Program for Automath. In *Selected Papers on Automath [NGdV94]*, pages 783–804. North-Holland Pub. Co., Amsterdam, The Netherlands, 1994.