A Verified Translation of Landau's "Grundlagen" from Automath into a Pure Type System, via $\lambda\delta$

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 version of the "Grundlagen" in the language of Pure Type Systems, 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 are disambiguated only by validating the "Grundlagen" against a type system accepting Aut–QE. To this end, we rely on $\lambda\delta$ "Version 3", a calculus that the author is developing within the HELM working group.

1. INTRODUCTION

Landau's "*Grundlagen der Analysis*" [Lan65], formalized by Jutting [vB77] in the Automath language Aut–QE [vD94a] (henceforth, the u–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 u–GdA as a background for significant formalized mathematics, is limited by the fact 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 (PMS's) based on them.

Anyway, some authors proposed translations of Aut–QE into Pure Type Systems (PTS's) [Bar93, KLN03, Bro11], and studied the possibility of making the u–GdA theoretically accessible to PTS-accepting systems like Coq [Coq14].

Here we take a step further: we discuss an implemented procedure producing a PTS-based version of the u–GdA, the $\lambda\Pi$–GdA, effectively accepted by Coq.

The validation of a $\lambda\Pi$–GdA for the PMS Matita [ARST11] is work in progress.

The concrete syntax of the typical Automath language is best known for its unified binding construction `[x:N]M`, as well as for its reversed application `<N>M`. In a PTS, the first one corresponds either to the abstraction $\lambda_x N.M$, or to the function type $\Pi_x N.M$, while the second one corresponds to the application $(M\ N)$.

We shall see that our translation is faithful in that the u–GdA is $\alpha\delta\eta$-equivalent to the $\lambda\Pi$–GdA, once abstractions and function types are replaced by the corresponding unified binding constructions. We stress that Automath's η-equivalence is `M` $=$ `[x:N]<x>M`, which in the PTS world, may correspond either to η-equivalence, or to Π-introduction/elimination (when `M` is a function and `[x:N]` is $\Pi_x N$).

In particular, η-equivalence solves the incompatibilities between Aut–QE and the PTS. On the other hand, δ-equivalence is introduced for convenience. Finally, α-equivalence is due to different naming conventions in the u–GdA and in Coq.

A main issue of the translation is that of deciding if a given unified binder corresponds to a λ-binder, or to a Π-binder. While static analysis suffices to disambiguate the majority of the approximately 47000 unified binders of the u–GdA, almost 3000 such binders can be disambiguated only by observing their behavior during the validation of the u–GdA against a type system that accepts Aut–QE.

To this end, we rely on $\lambda\delta$ "Version 3", a calculus outlined in this article, that the author is developing in the context of the HELM working group [APS+03].

Being an Automath "book", the u–GdA 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").

Moreover, the terms of Aut–QE are organized in three classes (kinds, types, and elements) comprising two sorts, references, applications, and unified bindings.

Therefore, once references are solved, and unified binders are disambiguated, translating the u–GdA into a PTS with constants and type casts, is not an issue.

In particular, by static analysis (Section 2), followed by dynamic analysis (Section 3), we build a $\lambda\delta$ version of the u–GdA, to be termed the $\lambda\delta$–GdA.

This is mapped straightforwardly to a user-level script, the $\lambda\Pi$–GdA, which is fully accepted by Coq. Our conclusions are in Section 4.

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 u–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 [Gui09].

—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 u–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

```
(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. 1. 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. 2. Conventions for displaying the concrete syntax

our translation, a static analyzer removes the Aut–QE shorthand from the u–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.

2.1 Parsing

The grammar recognized by our Automath parser is presented in Figure 1 (our notational conventions for displaying grammars are in Figure 2). Properly nested comments are accepted. It should be noted that the Automath grammar evolved

trough time and has many variants [NGdV94], which we try to capture. However, our parser recognizes "`_E`" in place of the original "underscored E", but this is not a problem since this notation does not appear in the u–GdA.

It is important to recall the structure of a formal specification in the language Aut–QE. 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:

—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 u–GdA, to inhibit the δ-expansion of the defined identifier in order to speed 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 every concrete Automath language [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 "Grundlagen" 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 u–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 u–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 [Gui09]. 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 u–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 the u–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 u–GdA. One strategy is degree-based [Bro11], while the other is position-based.

Note that both strategies rely on the fact that the u–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 $\forall$-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 an environment G of $\lambda\delta$ "Version 3" (see Section 3.1) in which a primitive constant p of type W in the context L corresponds to the entry $\lambda_p(L.W)$, while a defined constant p with body V of type W in the context L corresponds to the entry $\delta_p(L.©W.V)$.

Here $L.T$ is the term formed by concatenating the entries of L before T.

Note that this encoding improves the one of [Gui09], where we map the defined constant p to the entry $\delta_p(©(L.W).(L.V))$ (*i.e.*, the context L is not shared).

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

natural number	i, j, k, p	starting at 0
natural number or ∞	e	starting at 0
term	T, U, V, W	$::= \star k \mid \#i \mid \$p \mid \delta V.T \mid \lambda^e W.T \mid @V.T \mid ©W.T$
local environment	K, L	$::= \star \mid L.\delta V \mid L.\lambda^e W$
global environment	F, G	$::= \star \mid G.\delta V \mid G.\lambda W$

Fig. 3. Terms and environments.

solved (also by correcting some points of the u–GdA as we explain in Section 3.6), the "proper" $\lambda\delta$–GdA, without layer variables, is mapped to the $\lambda\Pi$–GdA, *i.e.*, our final outcome. This is a single user-level script that can be effectively presented to the PMS Coq for validation in the Calculus of Constructions [CH88].

The apparatus for validating in $\lambda\delta$ "Version 3" derives essentially from [Gui10a], which refers to a previous version of the calculus, and consists of a reduction machine (Section 3.2), a "comparator" (Section 3.3) asserting convertibility in context, and a "validator" (Section 3.4) implementing the top-level validation algorithm.

We want to keep the focus of this article on the "Grundlagen", so our description of $\lambda\delta$ "Version 3" and of this machinery will contain minimal formalism.

3.1 An Overview of $\lambda\delta$ Version 3: "To Π ... and Beyond"

The formal system $\lambda\delta$ is a typed λ-calculus originally conceived by the author as an extension of Λ_∞ [vB94b]. The system is evolving constantly as updated versions are released from time to time. The present "Version 3", implemented for the dynamic analysis of the u–GdA, still lacks a theoretical study. Nevertheless, it should be a conservative extension of the previous "Version 2a" [Gui14], whose desirable properties are certified. In particular, it supports the formation of universes through "type inclusion by reduction" (inspired by the "sort inclusion" of [Zan94]).

Looking at [Gui14], we annotate λ-abstractions with an integer "layer" e in the range $0 \le e \le \infty$, and we agree that a λ^e-abstraction is typed by a λ^{e-1}-abstraction. Moreover we restrict β-contractions to λ^{e+1}-abstractions, and we allow the equivalent of ζ-contraction for λ^0-abstractions. Finally we set $\infty - 1 = \infty + 1 = \infty$.

As an additional feature, we add environments with constants referred by level.

A formal exposition of the language is not in the scope of the article. We just introduce its syntax and its key rules, keeping formalism at the bare minimum.

Definition 1. Terms and environments are defined in Figure 3. $\star k$ is the sort of index k, $\#i$ is the reference to the variable introduced at depth i [dB94] (so i is a "de Bruijn index"), $\$p$ is the reference to the constant introduced at level p, $\delta V.T$ is the abbreviation "let $\#0 = V$ in T", $\lambda^e W.T$ is the abstraction "$(\#0 : W) \mapsto T$" in layer e, $@V.T$ is the application "$T(V)$", and $©W.T$ is the type annotation "$(T : W)$". $\star$ is the empty environment, $L.\delta V$ is L with the variable definition "let $\#0 = V$", and $L.\lambda^e W$ is L with the variable declaration "$(\#0 : W)$" in layer e. $G.\delta V$ is G with the constant definition "let $\$|G| = V$", and $G.\lambda W$ is G with the constant declaration "$(\$|G| : W)$". Here $|G|$ is the number of entries in G. ▲

Definition 2. We update the rules of [Gui14] named "Figure 13(β)" (transitions), "Figure 18(bind)" (iterated static type assignment), and "Figure 22(appl)" (strati-

$$\frac{}{G, L \vdash @V.\lambda^{e+1}W.T \to \delta(©W.V).T}\,\beta \qquad \frac{\uparrow^{\langle 0,1\rangle} T_2 = T_1}{G, L \vdash \lambda^0 W.T_1 \to T_2}\,\upsilon$$

$$\frac{G, L.\delta/\lambda^e W \vdash T \bullet_h^{*(n)} U}{G, L \vdash \delta/\lambda^e W.T \bullet_h^{*(n)} \delta/\lambda^{e-1} W.U}\,\text{bind}$$

$$\frac{G, L \vdash V\,!_{h,g} \quad G, L \vdash T\,!_{h,g} \quad G, L \vdash V \bullet^* \Rightarrow_{h,g}^{*(1)} W_0 \quad G, L \vdash T \bullet^* \Rightarrow_{h,g}^{*(n)} \lambda^{e+1} W_0.U_0}{G, L \vdash @V.T\,!_{h,g}}\,\text{appl}$$

Fig. 4. Key reduction and validation rules.

fied validity), with the rules of Figure 4. Moreover, we introduce the υ-contraction of Figure 4(υ), producing Zandleven-style "type inclusion" [Zan94]. ▲

A main observation concerns the inhabitation of sorts in the next situation. Under the assumption $\Gamma, (x : W) \vdash M : T : s$ where s is a sort, in a PTS we may have $\Gamma \vdash \lambda_x W.M : \Pi_x W.T : s$. On the other hand, assuming $G, L.\lambda_x^2 W \vdash M \bullet T$ and $G, L.\lambda_x^1 W \vdash T \bullet s$, in $\lambda\delta$ "Version 3" we have $G, L \vdash \lambda_x^2 W.M \bullet \lambda_x^1 W.T \bullet \lambda_x^0 W.s$ where $\lambda_x^2 W.M$ corresponds to $\lambda_x W.M$, and $\lambda_x^1 W.T$ corresponds to $\Pi_x W.T$, and $G, L \vdash \lambda_x^0 W.s \to s$ by υ-contraction. In this respect, $\lambda_x^0 W$ is an "hyper-Π" (*i.e.*, a "beyond-Π") serving as a constructor of some "quasi-expressions" of Aut–QE.

Another observation concerns the typable applications. Under the assumption $\Gamma \vdash M : \Pi_x W.T$, in a PTS we may have $\Gamma \vdash M(N) : [x{\leftarrow}N]\,T$. On the other hand, under the corresponding assumption $G, L \vdash M \bullet \lambda_x^1 W.T$, in $\lambda\delta$ "Version 3" we have $G, L \vdash @N.M \bullet @N.\lambda_x^1 W.T$ where $G, L \vdash @N.\lambda_x^1 W.T \to \delta_x(©W.N).T$ by β-contraction, and then $\delta_x(©W.N).T$ computes to $[x{\leftarrow}N]\,T$ by δ-expansion followed by ϵ-contraction (*i.e.*, elimination of type annotations). In this respect, we see that β-contraction on λ^1, *i.e.*, Π-contraction in the PTS world, is essential in $\lambda\delta$.

The reader should observe that β-contraction and υ-contraction act on different abstractions. In this way, $\lambda\delta$ "Version 3" avoids a critical pair that would not be confluent. On the other hand, Aut–QE clearly supports "sort inclusion" (*i.e.*, υ-contraction) on λ^1 [vD94a]. As a result, the predicate $\lambda_x S.P$ and its universal quantification $\forall_x S.P$ have the same type `'prop'` in the u–GdA. To us, such a position brings unnecessary complications into the logical framework.

Moreover, $\lambda^{e+1} W_0.U_0$ is a weak head normal form (WHNF), whereas $\lambda^0 W_0.U_0$ might be υ-reduced. Thus, we justify the forth premise of Figure 4(appl).

We wish to add that υ-contraction can include big universes into small ones, so, eventually, restrictions might apply to the term W of Figure 4(υ) in order to prove strong normalization and to avoid inconsistency. Because of Figure 4(bind), such restrictions will influence the term $\lambda^e W.T$ with $0 \le e < \infty$, while the terms $\lambda^\infty W.T$ will not be influenced (*i.e.*, layer 0 cannot be reached by iterated typing from layer ∞). In this respect, $\lambda\delta$ "Version 3" accounts for the distinction between "instantiation" and "abstraction" given in [dB91]. "Block openers" (*i.e.*, unrestricted abstractions) `[x:W]` actually correspond to $\lambda_x^\infty W$, whereas local variable declarations (*i.e.*, restricted abstractions) `[x:W]` correspond to $\lambda_x^e W$ with $0 < e < \infty$.

However, to the end of validating the u–GdA in a PTS, we saw in Section 2.5 that our static analyzer translates a "block opener" into either a λ^1, or a λ^2. Thus, we must rely on the capability of the PTS to provide for big universes.

The previous observations should convince the reader that $\lambda\delta$ "Version 3" is feasible for validating Aut–QE-based theories, as well as some PTS-based theories.

A key point to us is that $\lambda\delta$ "Version 3" is a minimal-impact extension of $\lambda\delta$ "Version 2a" with constants, since it adds just one reduction rule to the framework.

3.2 An Overview of the Reduction and Type Machine

The key ingredient of mechanical validation in $\lambda\delta$ "Version 3" is the Reduction and Type Machine (RTM), an abstract machine of the "K" family [Kri07] that computes the WHNF's by "rt-reduction" [vD94b] for $\lambda\delta$ (see [Gui09, Gui10a]).

In addition to ordinary reduction steps, mainly β-contractions, δ-expansions, and, notably, υ-contractions, the RTM can take the next type-inference steps [Gui14].

—The step s: from a sort to the next sort in the type hierarchy.

—The step l: from a reference to a declaration λW, to its declared type W.

—The step e: from a type annotation $©U.T$ to its declared type U.

—The step x: from a declaration $\lambda^e W.T$ to $\lambda^{e-1} W.T$ (not in $\lambda\delta$ "Version 2").

In this respect, and contrary to the usual approach, the RTM performs all the reduction and all the type inference that is required by the validation procedure.

We motivate this position by noting that abstract machines can be very efficient.

Furthermore, the RTM is a "controlled" machine, in that it can stop on references to defined variables or constants (which are not WHNF's), and then it can carry on its computation with a δ-expansion, when restarted by the calling controller.

In this way, the policy for managing δ-expansions, which is crucial for the validation of the "Grundlagen", is entirely on the controller's side (see Section 3.3).

The RTM maintains an integer parameter n that the caller can specify or not, when the machine is started. Consequently, the RTM runs in two modes.

—The mode C (convertibility). When n is specified (typical values are: either 0, or 1), the RTM applies exactly n type-inference steps before stopping on a WHNF. Note that the RTM operates on valid terms, so it does not meet dangling references and either s-steps, l-steps, or x-steps always apply as necessary.
Moreover, on type annotations, the e-step (when applicable) is preferred to ϵ-contraction (*i.e.*, $G, L \vdash ©U.T \to T$). This means that, in case of type inference, we use declared types if we know them. The reader should note the type annotation in the β-reductum of Figure 4(β), and in the first paragraph of Section 3.
In this mode, the "stop before δ-expansion" is enabled.

—The mode A (applicability). When n is not specified, the RTM disables s-steps, e-steps, x-steps, and the "stop before δ-expansion". In this situation, considering the fourth premise of Figure 4(appl), the RTM started on T, stops on $\lambda^{e+1} W_0.U_0$.
It is important to stress that the term $@V.T$ is not valid in a PTS, if this computation requires more than one type-inference step. In this respect $\lambda\delta$ features an extended "applicability condition" agreeing with its multi-layer architecture.
The reader should note that uncontrolled s-steps and x-steps cause infinite computations, while e-steps make the RTM miss the $\lambda^{e+1} W_0.U_0$ in some cases.
For instance, consider the term $@x.f$ in the context $\star.\lambda_x W.\lambda_f(©\star.(\lambda^1 W.U))$.
To reach the λ^1 after the l-step on f, the ϵ-step must be preferred to the e-step.

These modes are related to the situations in which the RTM is involved. Either two synchronized machines are started when a convertibility condition is checked, or one machine is started when an applicability condition is checked. The reader may refer to the conclusion of [Gui14], for a discussion on "applicability condition".

As layer variables may occur in the $\lambda\delta$–GdA, the RTM operates in two scenarios.

(1) When layer variables are allowed, *i.e.*, when validating the raw $\lambda\delta$–GdA, every abstraction is a WHNF in mode C, so the RTM stops on a υ-redex. Thus, in this case, the management of type inclusion is on the controller's side, as it is in the original validation algorithm implemented for Aut–QE [Zan94].
On the other hand, in mode A (*i.e.*, when the RTM must not be synchronized), υ-contraction is enabled. Yet, an abstraction with a variable layer is a WHNF.

(2) When layer variables are not allowed, *i.e.*, when validating the proper $\lambda\delta$–GdA, the RTM runs how it should and υ-contraction is enabled in mode C as well.

We stress that the RTM avoids ζ-contractions in the sense of [Gui14], which are known as "delifting steps" following a well-established terminology.

3.3 An Overview of the Comparator

The comparator asserts the convertibility between a declared 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 $n = 1$. This means that both machines run in mode C.

The conversion test occurs by levels, *i.e.*, by repeated comparison of WHNF's.

In particular, the comparison policy follows [Gui10a] 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 "younger" definition according to an "age" system we shall explain.

(5) A reference to a definition compared to any other term, is δ-expanded.

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

(7) When layer variables are allowed, and we are comparing an abstraction (coming from V) with a sort (coming from U), then we proceed by υ-reducing the abstraction. Note that this clause makes the comparator not symmetrical.
In [Gui10a] we disable this clause when leaving the "spine" of V to avoid inconsistency [Gui09]. However, by restricting υ-contraction to abstractions in layer 0 (see Section 3.1), we should ensure consistency despite this workaround.

It is a known fact that the "Grundlagen" can be validated in a reasonable amount of time only if we limit δ-expansions during convertibility checks. In this respect, the first slow constant seems to be `t29"l-e-st-eq-landau-n-428"` (which validates in more than 12 minutes on a 3 GHz Intel processor). While the next slow constant seems to be `t2"l-e-st-eq-landau-n-430"` (which takes more than 21 minutes).

To avoid these delays, our system implements "age-controlled" δ-expansions [Zan94]. In particular, a progressively increasing natural number $\#p$ is assigned to each constant p after its static analysis. Thus, constants become totally ordered.

The lines of the u–GdA appear in order of dependence, so a constant p_1 processed before a constant p_2, *i.e.*, satisfying $\#p_1 < \#p_2$, cannot depend on p_2. This means that when we compare a reference to p_1 with a reference to p_2, it is safe to δ-expand just the reference to p_2. In this respect, p_2 is the "younger" constant of the two).

On the other hand, Matita uses "height-controlled" δ-expansions [ARST09], differing in that the same "height can be assigned to independent constants.

3.4 An Overview of the Validator

Indeed, the validator is the simpler component of our validation system. To some extent, it follows [Gui10a] but, remarkably, we replace the canonical type synthesizer with a procedure to assert the validity relation of [Gui14], with the "applicability rule" updated to Figure 4(appl) in order to comply with $\lambda\delta$ "Version 3".

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 [Gui10a], the rules of validity [Gui14] 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 validation process we advocated in [Gui09].

Looking again at Figure 4(appl), the validator asserts the applicability of T to V as follows. Firstly, it starts a RTM, say m, on T with the parameter n unspecified, which should stop on the abstraction $\lambda^{e+1}W_0.U_0$. Secondly, it calls the comparator (Section 3.3) on W_0 (as the declared type) and on V, using for W_0 the machine m in its current state. On the contrary, the machine for V has an initial state.

3.5 Dynamic Disambiguation of Unified Binders

The dynamic analysis of the "raw" $\lambda\delta$–GdA relies on the validation procedure we discussed in the previous sections. When layer variables are allowed, this procedure produces a sequence of constraints on such variables. They are of four kinds.

(1) The constraint $\phi - 1 = \psi$ is generated by the RTM on a "step x" from the abstraction $\lambda^{\phi}W.T$ to the abstraction $\lambda^{\psi}W.T$ (see Section 3.2).

(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 (6) of Section 3.3).

(3) The constraint $\phi = 0$ is generated by the RTM υ-reducing the abstraction $\lambda^{\phi}W.T$ as required by the comparator (see Clause (7) of Section 3.3).

(4) The constraint $\phi > 0$ is generated by the RTM β-reducing the abstraction $\lambda^{\phi}W.T$. Moreover, it is generated by the validator asserting the applicability of a term T with functional structure $\lambda^{\phi}W_0.U_0$ (see Section 3.4).

```
 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. 5. Static corrections to the u–GdA.

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 p after static analysis are determined after p is validated, *i.e.*, without checking the subsequent references to p.

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 in its original form, the u–GdA fails to validate in $\lambda\delta$ and in a PTS, since two constants require 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"`.

Nevertheless, these reductions can be avoided by following a suggestion due to van Daalen and reported in [vB77], by which we apply a $\forall$-introduction to a predicate symbol in two constants: `all"l"` and `imp"l-r"` (see Figure 5).

With these corrections, our dynamic analysis of the u–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 [Gui09], noting that `ande2"l-r"` requires the "pure" type inference rule for function application [dB91], corresponding to the "extended" applicability condition (see Section 3.2, mode A).

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

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. 6. Effective use of disambiguation strategies.

In the case of the u–GdA, we invoke `imp"l-r"` to apply a ∀-introduction to a predicate `b` passed as a type and occurring four times (see Figure 5).

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 5). Note that `none` is indeed the counterpart of `some`.

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

4. CONCLUSION AND FUTURE WORK

In [Gui09] we describe the λδ–GdA: a translation of the u–GdA into the formal system λδ "Version 2", experimentally equipped with "sort inclusion" to this end.

In this paper we take a step further by describing the λΠ–GdA: a translation of the λδ–GdA into a PTS. The point at issue is assigning a "layer" to Automath's unified binders, *i.e.*, distinguishing λ-abstractions from Π-abstractions.

For this purpose, we replace λδ "Version 2", a calculus having a single binder, with λδ "Version 3" outlined in Section 3.1: a system featuring infinite (actually, $\omega + 1$) binders, 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 6 the first constant of the u–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 has an "ad hoc" automated procedure [Bro11] to solve this situation by formally η-expanding these inconsistent binders.

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

Helena, our implemented processor for λδ "Version 3", validates the corrected u–GdA by operating the amount of β-contractions and of δ-expansions shown in Figure 7. The δ-expansions on variables come from the β-reductum of Figure 4(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 8 concerning Helena (processing the u–GdA), and Coq (processing the λΠ–GdA).

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

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

Fig. 7. Main reduction steps of the RTM.

Input	System	Execution (seconds)	Task
u–GdA	Helena 0.8.2	01.02 to 01.05	disambiguation, validation in $\lambda\delta$
$\lambda\Pi$–GdA	Coq 8.4.3 (no VM)	25.80 to 26.16	validation in λC
	Coq 8.4.3 (with VM)	run was halted after 60 minutes	
	Matita 0.99.2	work in progress	validation in λC

All systems are implemented in the Objective Caml programming language

Fig. 8. Time of one run (min. and max. on 31 runs).

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.

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.

As to improving the proofs, we are in favor of using our "procedural reconstruction" [Gui10b], a technology we implemented for a former version of Matita.

The corrected u–GdA, the $\lambda\delta$–GdA and the $\lambda\Pi$–GdA, as well as our processor for $\lambda\delta$ "Version 3", are available at $\lambda\delta$ Web site: <http://lambdadelta.info/>.

ACKNOWLEDGMENTS

I wish to thank the HELM working group for constant efforts in favoring the development of the formal system $\lambda\delta$ by meeting the related requirements on Matita.

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

[APS+03] A. Asperti, L. Padovani, C. Sacerdoti Coen, F. Guidi, and I. Schena. Mathematical Knowledge Management in HELM. *Annals of Mathematics and Artificial Intelligence*, 38(1):27–46, May 2003.

[ARST09] A. Asperti, W. Ricciotti, C. Sacerdoti Coen, and E. Tassi. A compact kernel for the calculus of inductive constructions. *SĀDHANĀ Academy Proceedings in Engineering Sciences*, 34(1):71–144, February 2009. Special Issue on Interactive Theorem Proving and Verification.

[ARST11] A. Asperti, W. Ricciotti, C. Sacerdoti Coen, and E. Tassi. The Matita Interactive Theorem Prover. In N. Bjørner and V. Sofronie-Stokkermans,

editors, *Proceedings of the 23rd International Conference on Automated Deduction (CADE-2011)*, volume 6803 of *Lecture Notes in Computer Science*, pages 64–69, Berlin, Germany, 2011. Springer.

[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.

[Coq14] Coq development team. *The Coq Proof Assistant Reference Manual: release 8.4pl5*. INRIA, Orsay, France, October 2014.

[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.

[Gui09] F. Guidi. Landau's "Grundlagen der Analysis" from Automath to lambda-delta. Technical Report UBLCS 2009-16, University of Bologna, Bologna, Italy, September 2009.

[Gui10a] 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.

[Gui10b] F. Guidi. Procedural Representation of CIC Proof Terms. *Journal of Automated Reasoning*, 44(1-2):53–78, February 2010. Special Issue on Programming Languages and Mechanized Mathematics Systems.

[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.

[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.

[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.