An Efficient Validation Procedure for the Formal System $\lambda\delta$

Ferruccio Guidi*

Department of Computer Science
Mura Anteo Zamboni 7, 40127 Bologna, ITALY.
fguidi@cs.unibo.it

Abstract We take the opportunity of presenting a rigorous validation procedure for a version of the system $\lambda\delta$, to propose some improvements on the PTS-oriented type synthesizers based on the Constructive Engine.

1 Introduction

The formal system $\lambda\delta$ [9,8] is a typed λ-calculus inspired by Λ_∞ [13] (a formal language of the Automath family), that we are developing in the context of the HELM project [1]. As a framework for encoding formal mathematics, our calculus defines a structure, that we call an *environment*, in which a mathematical theory can be represented. The syntactic correctness of such a structure, that we call the *validity* of an environment, is also defined and is decidable. Not surprisingly, this notion of validity is related to the fact that the $\lambda\delta$-terms appearing in an environment are typable. This is to say that validity is related to type inference.

In this paper, we are concerned with an efficient procedure for the validation of a $\lambda\delta$-environment. To this aim, we observe that even if $\lambda\delta$ is not a Pure Type System (PTS) [3], it resembles a PTS enough to address the problem of efficient type inference as in a PTS. So, we are naturally led to base our procedure on a suitable version of the Constructive Engine [11] as implemented by the latest type synthesizers [2,4] for the Calculus of Inductive Constructions (CIC).

Generally speaking, the type inference algorithm of this engine synthesizes a type incrementally by recurring on the structure of the given term. This means that the algorithm operates on a closure, *i.e.*, a context and a possibly open term that refers to it. Moreover, it requires two operations that are crucial with respect to its efficiency: asserting the convertibility of two types, and asserting the applicability of two terms. In the latest versions of the engine, convertibility is asserted with an efficient algorithm that operates on two types closed in a common context, using a fast reduction machine (typically, a variant of the K machine [5]) for computing the weak head normal form (w.h.n.f.) of a term.

On the other hand, we see some inefficiency as to asserting applicability in that the w.h.n.f. of the type of the function is extracted from the reduction

* Partially supported by the Strategic Project DAMA (Dimostrazione Assistita per la Matematica e l'Apprendimento) of the University of Bologna.

machine that computed it in order to obtain the expected type of the function argument. Normally, the extraction of a w.h.n.f. resulting from the computation of a reduction machine requires a decompilation of the machine, during which the contents of the machine registers undergo time-consuming operations such as substitutions and relocations. As a possible way out, it was proposed to delay such substitutions and relocations by adopting explicit substitutions [4]. Nevertheless, $\lambda\delta$ would require an unplanned extension to support explicit substitutions, so this solution seems inconvenient to us in the present case.

Schematically, the solution we are proposing in this article is as follows. Firstly, we allow the convertibility checker to operate on two closures rather than on two terms closed in a common context. Secondly, we use reduction engines rather than closures throughout the type synthesizer and the convertibility checker. By so doing, we can assert applicability without extracting the w.h.n.f. of the type of the function from the reduction machine that computed it.

In particular, our solution avoids the distinction made in [2,4] between a context and an environment of a reduction machine, which makes the machine less efficient when reducing a term that refers to the context with respect to reducing a term that refers just to the machine environment.

In this article, we propose a complete procedure that includes a reduction machine, a convertibility controller, a type synthesizer and a validity controller. Contrary to [8], such procedure is specified rigorously, but no proof of its correctness is given. In our opinion, such a proof would be carried out most naturally by representing a state of the reduction machine with a $\lambda\delta$-closure, but we believe that the calculus needs further development, both on the practical and on the theoretical side, before this representation is possible. The problem being to support a term construction for representing closures. Nevertheless, we implemented this procedure as a part of the HELM software and we have evidence (see [8]) that it correctly validates a naive translation of the "*Grundlagen der Analysis*" [12], *i.e.*, the only development finalized in a language of the Automath family.

For the reader's convenience, we end this section including a summary of $\lambda\delta$ consisting of its syntax (Figure 1), its reduction rules (Figure 2), its type assignment rules (Figure 7), and its validity rules (Figure 4) taken essentially from [8]. In these figures, $\uparrow^i$ is the "*relocation function*", $|L_2|$ is the "*length*" of L_2 *i.e.*, the number of binders in L_2, while $x \notin G_2$ means that there is no binder named x in G_2. The judgement $G, L \vdash U_1 \leftrightarrow^* U_2$ asserts the convertibility of the terms U_1 and U_2 in the environments G and L. Moreover, the type assignment depends on the parameter h, which is a function from the natural numbers to themselves that can be chosen at will as long as $l < h(l)$ for every l. The available space forces us to delegate any comment on the calculus to [9,8], where we explain it and motivate it at our best. We just want to remark the improvements of the system $\chi\lambda\delta$ "*brg*"[8], that we consider here, over its stable version [9].

In $\chi\lambda\delta$ "*brg*"we removed the level indication from the environment's bottom sort. We also removed the application and the type annotation from the environment's constructors. Moreover we split the environment into a local part and a global part, providing a differentiated way for accessing the binders in each part, *i.e.*, by position and by name. We also stress that the reduction schemes

are substitution-free. More interestingly, we added the "*pure*" type assignment rule for function application, Figure 7(pure), motivated in [6]. This rule enables functions accepting arguments whose degree is greater than 3 according to de Bruijn's hierarchy, so it is not applicable in a PTS. Nevertheless, following the approach of Vera [14] (the first validator for Automath), our procedure handles this rule in the reduction machine, so the type synthesizer is not affected.

2 The Reduction Machine

In this section we give a formal presentation of the reduction and typing machine ($\lambda\delta$-RTM) [8] for computing the higher-order deep w.h.n.f. in $\chi\lambda\delta$ "*brg*".

To this aim, we recall that a term is in w.h.n.f. of *depth* or *level* n, when it has the form $\lambda W_1.\cdots.\lambda W_n.U$ where $n \geq 0$ and U is in w.h.n.f.. Then we define recursively an *iterated type* of a term T as a type of T or an iterated type of a type of T. Moreover, we define a *higher-order* w.h.n.f. of T as a w.h.n.f. of an iterated type of T (both concepts apply to any typed λ-calculus). These definitions allow to reduce the applicability condition of $\lambda\delta$ to that of a PTS in that a term of $\chi\lambda\delta$ "*brg*" has a function type when it admits a higher-order w.h.n.f. that is a λ-abstraction. The same concept applies to a PTS and is stated thus: a term of a PTS has a function type when it admits a higher-order w.h.n.f. that is a Π-abstraction. This observation makes it possible to use, for $\chi\lambda\delta$ "*brg*", a type synthesis algorithm for a PTS even if the "*pure*" type rule is in effect.

Essentially, the RTM (Figure 5) adapts the KN machine [5] to $\chi\lambda\delta$ "*brg*" without implementing the support for the evaluation of the stack contents, but implementing the support for computing higher-order normal forms.

In particular, the RTM is thought for reducing a term in the context of a validation algorithm and thus it presupposes an external controller that takes charge of evaluating the stack contents when necessary, whereas the KN machine is thought for the stand-alone computation of the normal form of a term.

The following is a description of the RTM that refers to Figure 5.

Structure of the machine. The RTM (M) is organized as a KN machine and has five registers: an accumulator (a) holding the depth of the w.h.n.f. to be computed; a read-only global environment (G) used to resolve global references; a local environment (E) of closures, in which we distinguish three kinds of entries (in particular a λ^a entry corresponds to a $V(a+1)$ entry of the KN machine); a stack (S) of closures storing function arguments; the code (T) to be reduced.

As regards the local environment, we stress that the δ-entries correspond to the "non-special" environment entries of the KN machine, while the χ-entries are introduced to support the exclusion binder featured by $\chi\lambda\delta$ "*brg*"[9].

Construction and projection. As the KN machine, the RTM does not require a dedicated encoding for the terms (contrary to the Symbolic Machine of [7]), so the terms must not be compiled or decompiled when the registers are initialized or read by the external controller of the machine.

Transition rules. The transitions (T1), (T2), (T4), (T6), (T7), and mainly (T13) come from the KN machine; the transitions (T5), (T8) and (T10) are

typical in the machines of the K family for the calculi featuring explicit type annotations, abbreviations, and references to a global environment (see for instance [2]); the transitions (T3) and (T9) are specific to $\chi\lambda\delta$ "*brg*" and handle the exclusion binder following the pattern of (T2) and (T8) respectively; the transitions from (T14) to (T20) are invoked by the functions $\mathcal{P}$, $\mathcal{D}$ and $\mathcal{R}$ explained below.

The transitions (T11) and (T12) enable the computation of a higher-order w.h.n.f. when the "standard" w.h.n.f. does not start with a sort or with a λ-abstraction (to assert the applicability condition). In particular, it follows from Figure 7 that if a typed term T has the form $X.\#i$ (where X denotes a term segment) and if $\#i$ refers to a λ-abstraction of type W, then $U \equiv X.\uparrow^{i+1}W$ is a type for T. Now suppose that the RTM is started on T and has already scanned the segment X of it, so that $\#i$ is in the code register, and suppose that in this situation a higher-order w.h.n.f. of T is required, then the RTM must compute a w.h.n.f. of U. As the segment X has been scanned already, what remains to do, is to continue the computation with W in the code register, and that is what the transition (T12) does. Finally we note that the transitions (T11) and (T12) follow the pattern of the transitions (T10) and (T4) respectively.

Transition functions. The RTM can be started in different modes, each enabling a subset of transitions, and we represent such modes with functions returning the final state of a computation started in the corresponding mode.

The convertibility controller (Section 3) applies $\mathcal{H}$ (head) and $\mathcal{X}$ (expand) to obtain the w.h.n.f. of a type; the type synthesizer (Section 4) applies $\mathcal{F}$ (function) to obtain the (possibly higher-order) w.h.n.f. of a type to assert the applicability condition; this controller also invokes $\mathcal{R}$ (resolve) to perform a look-up operation on the local environment; both controllers apply $\mathcal{P}$ (push) to place a binder in the environment, and $\mathcal{D}$ (drop) to deconstruct the contents of the code register. The reader should note that a λ-entry can be pushed on the local environment only if the stack is empty. Unfortunately, the available space does not allow us to present the final states of the RTM explicitly.

3 The Convertibility Controller

Our algorithm follows the pattern of the latest PTS-oriented convertibility controllers [2,4] and is presented in Figure 6 by means of syntax-oriented rules with non-commutable premises (note the semicolon after them). The main judgement is $M_1 \Leftrightarrow^{\mathrm{si}} M_2$, introduced by the rule "ac" (are convertible), which asserts the convertibility of the terms $\mathcal{C}(M_1)$ and $\mathcal{C}(M_2)$ applied to the arguments on the respective machine stacks and closed in the respective machine environments. A key feature of our convertibility controller is that it operates on two machines, rather than on two terms closed in a common context, like the one of [2].

Note that two references to local declarations are compared by de Bruijn's level rather than de Bruijn's index (rule "lref") so to avoid the need to relocate these references in a common context before the comparison. Also note that age-based global δ-expansion is supported as in [2,14] (rules "def-$\geq$" and "def-$<$").

We stress that the accumulators (Section 2) of the machines given to the controller should not differ (two w.h.n.f.s of different depths are not convertible). Moreover the controller maintains this invariant throughout execution.

The controller can assert two kinds of convertibility: standard (*i.e.*, predicate $M_1 \Leftrightarrow M_2$) and up to *sort inclusion* (*i.e.*, predicate $M_1 \Leftrightarrow^{\mathrm{si}} M_2$). In the latter case, the machine M_2 must represent a synthesized type and the machine M_1 must represent the corresponding expected type. This means that the predicate $M_1 \Leftrightarrow M_2$ is symmetric, while the predicate $M_1 \Leftrightarrow^{\mathrm{si}} M_2$ is not.

Sort inclusion is a subtyping mechanism that is missing in $\chi\lambda\delta$ "*brg*"because of its incompatibility with β-reduction, but that is needed to process the examples on which our validation procedure was tested (see Section 1). Our support for this feature follows the approach of the system Vera [14] in that the λ-abstractions forming the "spine" [5] of a synthesized type that are not β-reduced in the convertibility tests, are eligible for the reduction in Figure 3.

4 The Type Synthesizer

In principle, our synthesizer implements a standard algorithm [2,4], presented in Figure 8 by means of syntax-oriented rules with non-commutable premises.

The main judgement is $h \vdash M ::^{\mathrm{si}} U$ asserting that the term U is the type inferred for the term $\mathcal{C}(M)$ applied to the arguments on M's stack, with respect to the sort hierarchy parameter h (see Section 1).

The distinguishing feature of our controller concerns the validation of the applicability condition, performed by the last two premises of the rule "appl".

Once we understand that the term T inhabits a function type, by finding the abstraction $\lambda W_2.U_2$ as the result of a (possibly deep) w.h.n.f., we compare the type W_2 and type W_1 of the argument V, by passing them directly to the convertibility controller with their respective machines. We stress that we can perform this operation without decompiling the machine resulting from the w.h.n.f. computation because our convertibility controller operates on machines and can accept two machines (these features are crucial here).

These two machines may have different local environments in general, but their accumulators do not differ because the computation of a (deep) w.h.n.f. does not involve the transition (T13) of Figure 5, which is the only one that modifies the machine accumulator. This is to say that the invariant under which the convertibility controller works (Section 3) is maintained.

In our opinion, it should be possible to improve this synthesizer so to avoid the relocation function invoked by the rules "ldecl" and "ldef".

5 The Validity Controller

Our controller simply validates all entries of a global environment by calling itself recursively. Its rules are presented in Figure 9 following the style of Figure 6 and Figure 8. The main judgement is $h \vdash^{\mathrm{si}} G$ and asserts the validity of the global environment G with respect to the sort hierarchy parameter h (Section 1).

We implemented this controller as a part of the HELM software [1] and we tested it on a two-steps naive mechanical translation of the "*Grundlagen der Analysis*" [12] into $\chi\lambda\delta$ "*brg*"[8]. Figure 10 shows some statistical data about the performance of our implementation, which includes both the validator and the translation. Unfortunately, the only competing validator for the "*Grundlagen*" is written in C rather than in Caml, so a comparison would not be fare.

References

1. 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.
2. 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.
3. H.P. Barendregt. Lambda Calculi with Types. Osborne Handbooks of Logic in Computer Science, 2:117–309, 1993.
4. B. Barras. Auto-validation d'un système de preuves avec familles inductives. Thése de doctorat, spécialité informatique fondamentale, Université Paris 7, Paris, France, November 1989.
5. P. Crégut. Strongly reducing variants of the Krivine abstract machine. Higher-Order and Symbolic Computation, 20(3):209–230, September 2007.
6. N.G. de Bruijn. A plea for weaker frameworks. In Logical Frameworks, pages 40–67. Cambridge University Press, Cambridge, UK, 1991.
7. B. Grégoire. Compilation des termes de preuves: un (nouveau) mariage entre Coq et Ocaml. Thése de doctorat, spécialité informatique, Université Paris 7, École Polytechnique, France, December 2003.
8. F. Guidi. Landau's "Grundlagen der Analysis" from Automath to lambda-delta. Technical Report UBLCS 2009-16, University of Bologna, Bologna, Italy, September 2009.
9. F. Guidi. The Formal System $\lambda\delta$. Transactions on Computational Logic, 11(1):Article No. 5, October 2009.
10. 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.
11. G. Huet. The Constructive Engine. In R. Narasimhan, editor, A Perspective in Theoretical Computer Science, volume 16 of Series in Computer Science, pages 38–69. World Scientific Publishing, Singapore, 1989. Commemorative volume for Gift Siromoney.
12. 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.
13. L.S. van Benthem Jutting. The language theory of λ_∞, a typed λ-calculus where terms are types. In Selected Papers on Automath, pages 655–683. North-Holland Pub. Co., Amsterdam, The Netherlands, 1994.
14. I. Zandleven. A Verifying Program for Automath. In Selected Papers on Automath, pages 783–804. North-Holland Pub. Co., Amsterdam, The Netherlands, 1994.

i,l,x	::=	natural numbers	starting at 0
T,U,V,W	::=	terms	
	$*l$		sort of level l
	$\#i$		reference to the i-th local binder
	$\$x$		reference to the global binder x
	$\langle U\rangle.T$		annotation of T with its type U
	$(V).T$		application of T to the argument V
	$\lambda W.T$		local abstraction over the type W in T
	$\delta V.T$		local abbreviation of V in T
	$\chi.T$		local binder exclusion in T
L	::=	local environment	
	$*$		environment bottom
	$L.\lambda W$		declaration of type W
	$L.\delta V$		abbreviation of V
	$L.\chi$		binder exclusion
G	::=	global environment	
	$*$		environment bottom
	$G.\lambda_x W$		declaration of type W
	$G.\delta_x V$		abbreviation of V

Figure 1. Abstract syntax of $\chi\lambda\delta$ "*brg*".

scheme	**environment**	**redex**	**reduct**	
β-contraction	$G, L \vdash$	$(V).\lambda W.T$	$\to \delta V.T$	
local δ-expansion	$G, L_1.\delta V.L_2 \vdash$	$\#i$	$\to \uparrow^{i+1} V$	if $i = \lvert L_2\rvert$
global δ-expansion	$G_1.\delta_x V.G_2, L \vdash$	$\$x$	$\to V$	if $x \notin G_2$
ζ-contraction for δ	$G, L \vdash$	$\delta V.\uparrow^1 T$	$\to T$	
ζ-contraction for χ	$G, L \vdash$	$\chi.\uparrow^1 T$	$\to T$	
υ-swap for δ	$G, L \vdash$	$(V_1).\delta V_2.T$	$\to \delta V_2.(\uparrow^1 V_1).T$	
υ-swap for χ	$G, L \vdash$	$(V_1).\chi.T$	$\to \chi.(\uparrow^1 V_1).T$	
τ-contraction	$G, L \vdash$	$\langle U\rangle.T$	$\to T$	

Figure 2. Reduction steps of $\chi\lambda\delta$ "*brg*".

scheme	**environment**	**redex**	**reduct**
sort inclusion	$G, L \vdash$	$\lambda W.*l$	$\to *l$

Figure 3. Sort inclusion for $\chi\lambda\delta$ "*brg*"in the style of the system Vera.

$$\frac{}{\mathrm{wf}_h(*)}\ \text{sort} \qquad \frac{\mathrm{wf}_h(G) \quad G, * \vdash_h W : V}{\mathrm{wf}_h(G.\lambda W)}\ \text{abst} \qquad \frac{\mathrm{wf}_h(G) \quad G, * \vdash_h V : W}{\mathrm{wf}_h(G.\delta V)}\ \text{abbr}$$

Figure 4. Validity rules of $\chi\lambda\delta$ "*brg*".

Structure:

$$
\begin{array}{rll}
M & ::= & (a, G, E, S, T) \quad \text{machine state} \\
a, b & ::= & \text{natural number} \quad \text{de Bruijn's level} \\
G & ::= & \text{see Figure 1} \quad \text{global environment} \\
E, F & ::= & * \mid E.\lambda^a C \mid E.\delta C \mid E.\chi \quad \text{local environment} \\
S & ::= & * \mid S.C \quad \text{stack} \\
T, U, V, W & ::= & \text{see Figure 1} \quad \text{code} \\
C & ::= & (E, T) \quad \text{closure} \\
x & ::= & \text{see Figure 1} \quad \text{global name}
\end{array}
$$

Construction:

$$
\begin{array}{rl}
\mathcal{I}(G, T) \equiv (0, G, *, *, T) & \text{initial state} \\
(a, G, E, S, T) \leftarrow V \equiv (a, G, E, *, V) & \text{update code} \\
(a, G, E, S, T) \leftarrow (F, V) \equiv (a, G, F, *, V) & \text{update closure}
\end{array}
$$

Transition rules:

$$
\begin{array}{rcll}
(a, G, E.\lambda^b C, S, \#(i+1)) & \rightarrow_r & (a, G, E, S, \#i) & \text{(T1)} \\
(a, G, E.\delta C, S, \#(i+1)) & \rightarrow_r & (a, G, E, S, \#i) & \text{(T2)} \\
(a, G, E.\chi, S, \#(i+1)) & \rightarrow_r & (a, G, E, S, \#i) & \text{(T3)} \\
(a, G, E.\delta(F, V), S, \#0) & \rightarrow & (a, G, F, S, V) & \text{(T4)} \\
(a, G, E, S, \langle U \rangle.T) & \rightarrow_\tau & (a, G, E, S, T) & \text{(T5)} \\
(a, G, E, S, (V).T) & \rightarrow & (a, G, E, S.(E, V), T) & \text{(T6)} \\
(a, G, E, S.(E, V), \lambda W.T) & \rightarrow & (a, G, E.\delta(E, V), S, T) & \text{(T7)} \\
(a, G, E, S, \delta V.T) & \rightarrow_s & (a, G, E.\delta(E, V), S, T) & \text{(T8)} \\
(a, G, E, S, \chi.T) & \rightarrow_s & (a, G, E.\chi, S, T) & \text{(T9)} \\
(a, G_1.\delta_x V.G_2, E, S, \$x) & \rightarrow_x & (a, G_1.\delta_x V.G_2, E, S, V) & \text{(T10)} \\
(a, G_1.\lambda_x W.G_2, E, S, \$x) & \rightarrow_t & (a, G_1.\lambda_x W.G_2, E, S, W) & \text{(T11)} \\
(a, G, E.\lambda^b(F, W), S, \#0) & \rightarrow_t & (a, G, F, S, W) & \text{(T12)} \\
(a, G, E, *, \lambda W.T) & \rightarrow_p & (a+1, G, E.\lambda^a(E, W), S, T) & \text{(T13)} \\
(a, G, E, S, (V).T) & \rightarrow_a & (a, G, E, S, T) & \text{(T14)} \\
(a, G, E, S, \langle U \rangle.T) & \rightarrow_d & (a, G, E, S, U) & \text{(T15)} \\
(a, G, E, S, (V).T) & \rightarrow_d & (a, G, E, S, V) & \text{(T16)} \\
(a, G, E, S, \lambda W.T) & \rightarrow_d & (a, G, E, S, W) & \text{(T17)} \\
(a, G, E, S, \delta V.T) & \rightarrow_d & (a, G, E, S, V) & \text{(T18)} \\
(a, G, E.\delta(F, V), S, \#0) & \rightarrow_c & (a, G, E.\delta(F, V), S, F.\delta V) & \text{(T19)} \\
(a, G, E.\lambda^a(F, W), S, \#0) & \rightarrow_c & (a, G, E.\lambda^a(F, W), S, F.\lambda W) & \text{(T20)}
\end{array}
$$

Transition functions:

$\mathcal{H}(M) \equiv$ final state from M with all rules except: x, t, p, a, d, c
$\mathcal{F}(M) \equiv$ final state from M with all rules except: p, a, d,c
$\mathcal{P}(M) \equiv$ one transition τ, s, p or a from M, or else the identity
$\mathcal{D}(M) \equiv$ one transition d from M, or else the identity
$\mathcal{X}(M) \equiv$ one transition x from M, or else the identity
$\mathcal{R}(M) \equiv$ final state from M with the rules: r and c

Projection:

$$
\begin{array}{rll}
\mathcal{G}(a, G_1.\lambda_x W.G_2, E, S, \$x) \equiv G_1.\lambda_x W & & \text{get global abstraction} \\
\mathcal{G}(a, G_1.\delta_x V.G_2, E, S, \$x) \equiv G_1.\delta_x V & & \text{get global abbreviation} \\
\mathcal{L}(a, G, E.\lambda^b C, S, \#0) \equiv b & & \text{get reference level} \\
\mathcal{S}(a, G, E, S, T) \equiv S & & \text{get stack} \\
\mathcal{C}(a, G, E, S, T) \equiv T & & \text{get code}
\end{array}
$$

Figure 5. The reduction and typing machine for $\chi\lambda\delta$ "*brg*".

$$\frac{\mathcal{C}(M_1) = \mathcal{C}(M_2) = *l}{M_1 \Leftrightarrow^{\mathrm{si}}_{\mathrm{whnf}} M_2}\ \mathrm{sort} \qquad \frac{\begin{array}{c}\mathcal{C}(M_1) = \lambda W_1.T_1; \quad \mathcal{C}(M_2) = \lambda W_2.T_2;\\ \mathcal{D}(M_1) \Leftrightarrow \mathcal{D}(M_2); \quad \mathcal{P}(M_1) \Leftrightarrow^{\mathrm{si}} \mathcal{P}(M_2)\end{array}}{M_1 \Leftrightarrow^{\mathrm{si}}_{\mathrm{whnf}} M_2}\ \mathrm{abst}$$

$$\frac{\mathcal{C}(M_1) = \mathcal{C}(M_2) = \#0; \quad \mathcal{L}(M_1) = \mathcal{L}(M_2); \quad M_1 \Leftrightarrow_{\mathrm{args}} M_2}{M_1 \Leftrightarrow^{\mathrm{si}}_{\mathrm{whnf}} M_2}\ \mathrm{lref}$$

$$\frac{\mathcal{C}(M_1) = \mathcal{C}(M_2) = \$x; \quad \mathcal{G}(M_1) = G.\lambda_x W; \quad M_1 \Leftrightarrow_{\mathrm{args}} M_2}{M_1 \Leftrightarrow^{\mathrm{si}}_{\mathrm{whnf}} M_2}\ \mathrm{decl}$$

$$\frac{\mathcal{C}(M_1) = \mathcal{C}(M_2) = \$x; \quad \mathcal{G}(M_1) = G.\delta_x V; \quad M_1 \Leftrightarrow_{\mathrm{args}} M_2}{M_1 \Leftrightarrow^{\mathrm{si}}_{\mathrm{whnf}} M_2}\ \mathrm{def}$$

$$\frac{\begin{array}{l}\mathcal{C}(M_1) = \$x_1; \quad \mathcal{C}(M_2) = \$x_2; \quad \mathcal{G}(M_1) = G_1.\delta_{x_1} V_1; \quad \mathcal{G}(M_2) = G_2.\delta_{x_2} V_2;\\ |G_1| \geq |G_2|; \qquad \mathcal{H}(\mathcal{X}(M_1)) \Leftrightarrow^{\mathrm{si}}_{\mathrm{whnf}} M_2\end{array}}{M_1 \Leftrightarrow^{\mathrm{si}}_{\mathrm{whnf}} M_2}\ \mathrm{def}{-}{\geq}$$

$$\frac{\begin{array}{l}\mathcal{C}(M_1) = \$x_1; \quad \mathcal{C}(M_2) = \$x_2; \quad \mathcal{G}(M_1) = G_1.\delta_{x_1} V_1; \quad \mathcal{G}(M_2) = G_2.\delta_{x_2} V_2;\\ |G_1| < |G_2|; \qquad M_1 \Leftrightarrow^{\mathrm{si}}_{\mathrm{whnf}} \mathcal{H}(\mathcal{X}(M_2))\end{array}}{M_1 \Leftrightarrow^{\mathrm{si}}_{\mathrm{whnf}} M_2}\ \mathrm{def}{-}{<}$$

$$\frac{\mathcal{C}(M_1) = \$x; \quad \mathcal{G}(M_1) = G.\delta_x V; \quad \mathcal{H}(\mathcal{X}(M_1)) \Leftrightarrow^{\mathrm{si}}_{\mathrm{whnf}} M_2}{M_1 \Leftrightarrow^{\mathrm{si}}_{\mathrm{whnf}} M_2}\ \mathrm{def}{-}\mathrm{sx}$$

$$\frac{\mathcal{C}(M_2) = \$x; \quad \mathcal{G}(M_2) = G.\delta_x V; \quad M_1 \Leftrightarrow^{\mathrm{si}}_{\mathrm{whnf}} \mathcal{H}(\mathcal{X}(M_2))}{M_1 \Leftrightarrow^{\mathrm{si}}_{\mathrm{whnf}} M_2}\ \mathrm{def}{-}\mathrm{dx}$$

$$\frac{\mathcal{C}(M_1) = *l; \quad \mathcal{C}(M_2) = \lambda W_2.T_2; \quad \mathcal{P}(M_1 \leftarrow \lambda W_2.*l) \Leftrightarrow^{\mathrm{si}} \mathcal{P}(M_2)}{M_1 \Leftrightarrow^{\mathrm{si}}_{\mathrm{whnf}} M_2}\ \mathrm{si}$$

$$\frac{}{(M_1, *) \Leftrightarrow^{\mathrm{rec}}_{\mathrm{args}} (M_2, *)}\ \mathrm{null} \qquad \frac{(M_1, S_1) \Leftrightarrow^{\mathrm{rec}}_{\mathrm{args}} (M_2, S_2); \quad (M_1 \leftarrow C_1) \Leftrightarrow (M_2 \leftarrow C_2)}{(M_1, S_1.C_1) \Leftrightarrow^{\mathrm{rec}}_{\mathrm{args}} (M_2, S_2.C_2)}\ \mathrm{cons}$$

$$\frac{(M_1, \mathcal{S}(M_1)) \Leftrightarrow^{\mathrm{rec}}_{\mathrm{args}} (M_2, \mathcal{S}(M_1))}{M_1 \Leftrightarrow_{\mathrm{args}} M_2}\ \mathrm{acs} \qquad \frac{\mathcal{H}(M_1) \Leftrightarrow^{\mathrm{si}}_{\mathrm{whnf}} \mathcal{H}(M_2)}{M_1 \Leftrightarrow^{\mathrm{si}} M_2}\ \mathrm{ac}$$

The rules are listed in decreasing order of precedence.
The label "si" (sort inclusion) is uniformly optional in all rules except for rule "si".

Figure 6. The convertibility controller for $\chi\lambda\delta$ "*brg*".

$$\frac{G_1, * \vdash_h V : W \quad x \notin G_2}{G_1.\delta_x V.G_2, L \vdash_h \$x : W}\ \mathrm{g}{-}\mathrm{def} \qquad \frac{G_1.* \vdash_h W : V \quad x \notin G_2}{G_1.\lambda_x W.G_2, L \vdash_h \$x : W}\ \mathrm{g}{-}\mathrm{decl}$$

$$\frac{G, L_1 \vdash_h V : W \quad i = |L_2|}{G, L_1.\delta V.L_2 \vdash_h \#i : \uparrow^{i+1} W}\ \mathrm{l}{-}\mathrm{def} \qquad \frac{G, L_1 \vdash_h W : V \quad i = |L_2|}{G, L_1.\lambda W.L_2 \vdash_h \#i : \uparrow^{i+1} W}\ \mathrm{l}{-}\mathrm{decl}$$

$$\frac{}{G, L \vdash_h *l : *h(l)}\ \mathrm{sort} \qquad \frac{G, L \vdash_h T : U \quad G, L \vdash_h U : V}{G, L \vdash_h \langle U \rangle.T : \langle V \rangle.U}\ \mathrm{cast} \qquad \frac{G, L.\chi \vdash_h T : U}{G, L \vdash_h \chi.T : \chi.U}\ \mathrm{void}$$

$$\frac{G, L \vdash_h V : W \quad G, L.\delta V \vdash_h T : U}{G, L \vdash_h \delta V.T : \delta V.U}\ \mathrm{abbr} \qquad \frac{G, L \vdash_h W : V \quad G, L.\lambda W \vdash_h T : U}{G, L \vdash_h \lambda W.T : \lambda W.U}\ \mathrm{abst}$$

$$\frac{G, L \vdash_h V : W \quad G, L \vdash_h T : \lambda W.U}{G, L \vdash_h (V).T : (V).\lambda W.U}\ \mathrm{appl} \qquad \frac{G, L \vdash_h T : U \quad G, L \vdash_h (V).U : W}{G, L \vdash_h (V).T : (V).U}\ \mathrm{pure}$$

$$\frac{G, L \vdash_h U_2 : V \quad G, L \vdash_h T : U_1 \quad G, L \vdash U_1 \leftrightarrow^* U_2}{G, L \vdash_h T : U_2}\ \mathrm{conv}$$

Figure 7. Type assignment rules of $\chi\lambda\delta$ "*brg*".

$$\frac{\mathcal{C}(M) = \$x; \quad \mathcal{G}(M) = G.\lambda_x W}{h \vdash M ::^{\text{si}} W}\ \text{gdecl} \qquad \frac{\mathcal{C}(M) = \$x; \quad \mathcal{G}(M) = G.\delta_x\langle W\rangle.V}{h \vdash M ::^{\text{si}} W}\ \text{gdef}$$

$$\frac{\mathcal{C}(M) = \#i; \quad \mathcal{C}(\mathcal{R}(M)) = L.\lambda W}{h \vdash M ::^{\text{si}} \uparrow^{i+1} W}\ \text{ldecl} \qquad \frac{\mathcal{C}(M) = \#i; \quad \mathcal{C}(\mathcal{R}(M)) = L.\delta\langle W\rangle.V}{h \vdash M ::^{\text{si}} \uparrow^{i+1} W}\ \text{ldef}$$

$$\frac{\mathcal{C}(M) = *l}{h \vdash M ::^{\text{si}} *h(l)}\ \text{sort} \qquad \frac{\mathcal{C}(M) = \lambda W.T; \quad h \vdash \mathcal{D}(M) ::^{\text{si}} V \quad h \vdash \mathcal{P}(M) ::^{\text{si}} U}{h \vdash M ::^{\text{si}} \lambda W.U}\ \text{abst}$$

$$\frac{\mathcal{C}(M) = \delta V.\langle U\rangle.T; \quad h \vdash \mathcal{D}(M) ::^{\text{si}} W \quad h \vdash \mathcal{P}(M) ::^{\text{si}} U}{h \vdash M ::^{\text{si}} \delta V.U}\ \text{abbr1}$$

$$\frac{\mathcal{C}(M) = \delta V.T; \quad h \vdash \mathcal{D}(M) ::^{\text{si}} W \quad h \vdash \mathcal{P}(M \leftarrow \delta\langle W\rangle.V.T) ::^{\text{si}} U}{h \vdash M ::^{\text{si}} \delta V.U}\ \text{abbr2}$$

$$\frac{\mathcal{C}(M) = \chi.T; \quad h \vdash \mathcal{P}(M) ::^{\text{si}} U}{h \vdash M ::^{\text{si}} \chi.U}\ \text{void}$$

$$\frac{\begin{array}{l}\mathcal{C}(M) = (V).T; \quad h \vdash \mathcal{D}(M) ::^{\text{si}} W_1; \quad h \vdash \mathcal{P}(M) ::^{\text{si}} U_1; \\ \mathcal{C}(\mathcal{F}(\mathcal{P}(M))) = \lambda W_2.U_2; \qquad\qquad \mathcal{D}(\mathcal{F}(\mathcal{P}(M))) \Leftrightarrow^{\text{si}} (M \leftarrow W_1)\end{array}}{h \vdash M ::^{\text{si}} (V).U_1}\ \text{appl}$$

$$\frac{\mathcal{C}(M) = \langle W\rangle.T; \quad h \vdash \mathcal{D}(M) ::^{\text{si}} V; \quad h \vdash \mathcal{P}(M) ::^{\text{si}} U; \quad \mathcal{D}(M) \Leftrightarrow^{\text{si}} (M \leftarrow U)}{h \vdash M ::^{\text{si}} W}\ \text{cast}$$

The rules are listed in decreasing order of precedence.
The label "si" (sort inclusion) is uniformly optional in all rules.
The rule "appl" is better implemented by sharing the result of $\mathcal{F}(\mathcal{P}(M))$.

Figure 8. The type synthesizer for $\chi\lambda\delta$ "*brg*".

$$\frac{h \vdash G_1 \parallel^{\text{si}} G_2}{h \vdash^{\text{si}} G_1}\ \text{wf} \qquad \frac{}{h \vdash * \parallel^{\text{si}} *}\ \textit{null} \qquad \frac{h \vdash G_1 \parallel^{\text{si}} G_2; \quad h \vdash \mathcal{I}(G_2, \langle W\rangle.V) ::^{\text{si}} W}{h \vdash G_1.\delta_x\langle W\rangle.V \parallel^{\text{si}} G_2.\delta_x\langle W\rangle.V}\ \text{abbr1}$$

$$\frac{h \vdash G_1 \parallel^{\text{si}} G_2; \quad h \vdash \mathcal{I}(G_2, V) ::^{\text{si}} W}{h \vdash G_1.\delta_x V \parallel^{\text{si}} G_2.\delta_x\langle W\rangle.V}\ \text{abbr2} \qquad \frac{h \vdash G_1 \parallel^{\text{si}} G_2; \quad h \vdash \mathcal{I}(G_2, W) ::^{\text{si}} V}{h \vdash G_1.\lambda_x W \parallel^{\text{si}} G_2.\lambda_x W}\ \text{abst}$$

The rules are listed in decreasing order of precedence.
The label "si" (sort inclusion) is uniformly optional in all rules.

Figure 9. The validity controller for $\chi\lambda\delta$ "*brg*".

Size of the "*Grundlagen*"	
Language	*Int. complexity*
Aut − QE	319706
intermediate	754578
$\lambda\delta$ "*brg*"	998232

Performance of the validator		
Phase	*Run time fraction*	*Run time*
parsing	10%	0.7s
translation	25%	1.7s
validation	65%	4.4s

Relocated data	
terms	295202
int. complexity	1252256
the relocations are due	
to the "l-decl" type rule	

Reductions			
β	1034626	τ	17166
local δ	494271	transition (T11)	0
global δ	17166	transition (T12)	1
υ	2040476	sort inclusion	904

The "intrinsic complexity" [10] approximates the number of nodes in the tree representation of the data thought as a single λ-term.
The validator was run on the HELM server (2 × AMD Athlon MP 1800+, 1.53 GHz, 256 KB L2 cache) and operated for 6.8s on average.

Figure 10. Some statistics on our validation of the "*Grundlagen*".