A Validator for the Formal System $\lambda\delta$

Ferruccio Guidi

University of Bologna, Italy

fguidi@cs.unibo.it

February 8, 2010

Overview

- The formal system $\lambda\delta$ is a typed λ-calculus under development in the context of the HELM project at the University of Bologna.
- As an expected feature, $\lambda\delta$ should serve as a modular framework flexible enough to encode Mathematics in a realistic manner.
- To verify this feature, we encoded a non-trivial mathematical theory into $\lambda\delta$ by implementing a computer-assisted translation.
- Given that the resulting λ-terms must be valid against $\lambda\delta$'s type system, we were naturally led to implement a validator as well.
- An XML representation of the λ-terms can be generated following HELM approach to long-term storage of mathematical contents.
- Our validator is implemented in the Caml programming language for a better integration with the rest of the HELM software.

In this talk

- We present the variant of $\lambda\delta$ recognized by the validator.
- We discuss the implementation of the validation procedure.
- We present the translation of the theory we encoded into $\lambda\delta$.

$\lambda\delta$ "*brg*" ("*basic, relative, global*")

- $\lambda\delta$ is an evolving framework, whose development process will eventually give rise to a family of languages.
- $\lambda\delta$ "*brg*" is very close to the official variant in print on ToCL.
- No applications, type casts and level indicators in environments.
- The environment is split and has a component accessed by name.
- We added the "*pure*" type assignment rule for applications.

The abstract syntax of $\lambda\delta$ "*brg*"

Natural number: i,l,x (corresponding data-type: $\mathbb{N}$)

Term: $T,U,V,W ::= *l \mid \#i \mid \$x \mid \langle U\rangle.T \mid (V).T \mid \lambda W.T \mid \delta V.T \mid \chi.T$

Local environment: $E ::= * \mid E.\lambda W \mid E.\delta V \mid E.\chi$

Global environment: $\mathcal{G} ::= * \mid \mathcal{G}.\lambda_x W \mid \mathcal{G}.\delta_x V$

The reduction steps of $\lambda\delta$ "*brg*"

$\mathcal{G},\ E \vdash (V).\lambda W.T \to_\beta \delta V.T$	$\mathcal{G},\ E \vdash \langle U\rangle.T \to_\tau T$
$\mathcal{G},\ E_1.\delta V.E_2 \vdash \#i \to_\delta \uparrow^{i+1} V$ if $i = \lvert E_2\rvert$	$\mathcal{G}_1.\delta_x V.\mathcal{G}_2,\ E \vdash \$x \to_\delta V$ if $x \notin \mathcal{G}_2$
$\mathcal{G},\ E \vdash \delta V.\uparrow^1 T \to_\zeta T$	$\mathcal{G},\ E \vdash \chi.\uparrow^1 T \to_\zeta T$
$\mathcal{G},\ E \vdash (V_1).\delta V_2.T \to_\upsilon \delta V_2.(\uparrow^1 V_1).T$	$\mathcal{G},\ E \vdash (V_1).\chi.T \to_\upsilon \chi.(\uparrow^1 V_1).T$

$\uparrow^i$ is the "*relocation function*". $\lvert E_2\rvert$ is the number of binders in E_2.

$x \notin \mathcal{G}_2$ means that there is no global binder named x in $\mathcal{G}_2$.

The fundamental judgements of $\lambda\delta$ "*brg*"

- $h : \mathbb{N} \longrightarrow \mathbb{N}$ is any function satisfying $h(l) > l$ for each l.
- Conversion: $\mathcal{G}, E \vdash U_1 \leftrightarrow^* U_2$ (U_1 and U_2 are convertible).
- Type assignment: $\mathcal{G}, E \vdash_h T : U$ (T has type U).
- Correctness: $\mathrm{wf}_h(\mathcal{G})$ ($\mathcal{G}$ is well formed).

The type assignment rules of $\lambda\delta$ "*brg*"

$$\frac{\mathcal{G}_1, * \vdash_h V : W \quad x \notin \mathcal{G}_2}{\mathcal{G}_1.\delta_x V.\mathcal{G}_2, E \vdash_h \$x : W}\ \mathrm{g{-}def} \qquad \frac{\mathcal{G}_1.* \vdash_h W : V \quad x \notin \mathcal{G}_2}{\mathcal{G}_1.\lambda_x W.E_2, E \vdash_h \$x : W}\ \mathrm{g{-}decl}$$

$$\frac{\mathcal{G}, E_1 \vdash_h V : W \quad i = |E_2|}{\mathcal{G}, E_1.\delta V.E_2 \vdash_h \#i : \uparrow^{i+1} W}\ \mathrm{l{-}def} \qquad \frac{\mathcal{G}, E_1 \vdash_h W : V \quad i = |E_2|}{\mathcal{G}, E_1.\lambda W.E_2 \vdash_h \#i : \uparrow^{i+1} W}\ \mathrm{l{-}decl}$$

The type assignment rules of $\lambda\delta$ "*brg*" (continued)

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

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

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

$$\frac{\mathcal{G}, E \vdash_h U_2 : V \quad \mathcal{G}, E \vdash_h T : U_1 \quad \mathcal{G}, E \vdash U_1 \leftrightarrow^* U_2}{\mathcal{G}, E \vdash_h T : U_2}\ \text{conv}$$

The correctness rules of $\lambda\delta$ "*brg*"

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

Highlights of our validation procedure

- Our validation procedure follows the general pattern implemented in many λ-calculus-based proof assistants, like Coq and Matita.
- We have a type checker, a convertibility checker and a domain retrieval function (in other systems this computes a w.h.n.f.).

1. The data-types for terms and environments allow to annotate each term and each environment entry with non-logical information.
2. The reduction apparatus is based on Krivine machines (KAMs), and the type checker uses them in place of local environments.
3. The application of some type inference rules (namely, *purity* and *sort inclusion*) is delegated to the reduction apparatus.
4. The reduction apparatus works without relocations (the functions *lift*, *delift*), in particular we do not need the *unwind* function.

Non-logical information

```
type id     = string
type attr   = Mark of int (* node marker *)
            | Name of id  (* name *)
            | Apix of int (* abs./alt. position index *)
type attrs = attr list    (* attributes *)
```

- `Mark`: persistent numeric information (line numbers, pointers). This attribute is not used now, but we plan to use it in the future.

- `Name`: persistent literal information (variables names, sort names) used for presentational purposes. NB Names are non-logical in $\lambda\delta$.

- `Apix`: transient numeric information used by the reduction apparatus to avoid δ-expansions and remarkably ζ-contractions.

- The persistent information is included in the long-term persistence format of the validated data (to be explained in the following).

Terms and local environments

```
type uri  = NUri.uri
type bind = Void
          | Abst of term                   (* domain *)
          | Abbr of term                   (* body *)
and  term = Sort of attrs * int            (* hierarchy index *)
          | LRef of attrs * int            (* position index *)
          | GRef of attrs * uri            (* reference *)
          | Cast of attrs * term * term (* type, member *)
          | Appl of attrs * term * term (* argument, function *)
          | Bind of attrs * bind * term (* binder, scope *)
type lenv = Null                           (* bottom *)
          | Cons of lenv * lenv * attrs * bind
```

- Our URI scheme is `ld` and we use the HELM URI module.
- A local environment entry (`Cons`) has a secondary closure used by the reduction apparatus (set to `Null` when not needed).

Global environment

```
type 'term bind   = Abst of 'term (* declaration: domain *)
                  | Abbr of 'term (* definition: body *)
type 'term entity = attrs * uri * 'term bind
val set_entity: term entity -> term entity
val get_entity: uri -> term entity
```

- The global environment entry (**`entity`**) is parametric over **`term`** to reuse it with different languages of the $\lambda\delta$ family (we have two now).
- The global environment is a table of entities hashed on their URIs, but is not a cache now. This limitation will be eventually solved.
- The environment must contain valid entities and the incoming entities receive an "*apix*" consisting of their ordinal entry number.
- This information, termed the *age*, is used like Matita's *height* to prevent useless global δ-expansions during the convertibility checks.

Pseudo-reductions

Local reference typing: $\mathcal{G},\ E_1.\lambda W.E_2 \vdash \#i \longrightarrow \uparrow^{i+1} W$ if $i = |E_2|$

Global reference typing: $\mathcal{G}_1.\lambda_x W.\mathcal{G}_2,\ E \vdash \$x \longrightarrow W$ if $x \notin \mathcal{G}_2$

Sort inclusion: $\mathcal{G},\ E \vdash \lambda W.{*l} \longrightarrow {*l}$

- The reduction apparatus implements these steps just as part of the type-checking algorithm. These reductions break $\lambda\delta$'s meta-theory.
- Reference typing is enabled in the domain retrieval function to implement the "*pure*" type inference rule, as we will explain.
- Sort inclusion raises $\lambda\delta$'s expressive power from $\lambda\!\rightarrow$ to λP and is needed to process the examples on which the validator was tested.
- Its implementation follows the Vera system: when the convertibility of two types is checked, sort inclusion is applied to the inferred type when no other step applies to it (some restrictions are needed).

The reduction apparatus

```
type status = {
   delta: bool; (* global delta-expansion *)
   rt: bool;    (* reference typing *)
   si: bool     (* sort inclusion *)
}
```

- `status` is an aggregate of flags passed to all functions concerned to reduction. These flags enable or disable some reduction steps.
- Only the `si` flag can be set by the user (via a command line option), the other flags are controlled by the reduction functions.
- Both the domain retrieval and the convertibility check require to apply a chain of reduction steps to some given terms. The literature suggests that w.h.n.f.'s are good break points for these chains.
- The main task of the reduction apparatus is to compute w.h.n.f.'s.

Computing the w.h.n.f.

```
type kam = {
   e: lenv;                    (* environment *)
   s: (lenv * term) list;  (* stack *)
   d: int                      (* depth *)
}
val empty: kam                               (* initial value *)
val get: kam -> int -> bind                  (* read entry by index *)
val push: kam -> attrs -> bind -> kam (* add entry on top *)
```

- The w.h.n.f. of a term is computed by a KAM extended to cope with $\lambda\delta$. The term is not stored in a KAM register, but it could.

- We provide access to the KAM environment (**get**, **push**) because we want to use it as a reduction and type inference environment. In particular we want our KAM to compute *deep* w.h.n.f.'s.

- The environment can be accessed only if the stack (**s**) is empty.

Computing the w.h.n.f. (continued)

- We count the incoming abstraction entries in the `d` register and each incoming abstraction entry receives an "*apix*" with its depth.
- The KAM computes the $\beta\delta\upsilon\tau$-w.h.n.f. of a given term, but it can stop on global δ-redexes by disabling the `delta` flag.
- The KAM stops on references to abstractions (both local and global) only if reference typing is disabled via the `rt` flag.
- When the KAM stops on a reference (both local and global), the "*apix*" of the referred binder is returned (it always exists).
- An error is produced if a reference to a χ binder is encountered, even if the w.h.n.f. exists anyway, but this is not a limitation.
- Our KAM is not parametric over the evaluation strategy and the call-by-need optimization is not implemented at the moment.

Domain retrieval

```
val xwhd: status -> kam -> term -> kam * term
```

- $(V).T$ is typable in $\mathcal{G}$ and E if (V) matches a λW_1 retrieved in T, E or $\mathcal{G}$. Moreover W_1 and the type W_2 of V must be convertible.

- λW_1 must start the w.h.n.f. of the type U of T, or else, if the "*pure*" rule is in effect, the iterated types of U must be considered.

- This search eventually comes to an end since it involves just a finite number of iterated types of T, which are strongly normalizable.

- The w.h.n.f. of the first iterated type of T that may contain λW_1, is given by our KAM run on U with `delta` and `rt` enabled.

- Formally, reference typing is similar to δ-expansion so the KAM does not need to perform any relocation when computing it.

The convertibility check

```
val are_convertible: status -> kam -> term -> kam -> term -> bool
   (* arguments: expected type, inferred type *)
```

- We can process two terms closed in two environments, given the invariant that they contain the same number of abstractions.
- The check applies to types and is not symmetric if sort inclusion is in effect. The arguments are an expected type and an inferred type.
- The terms are reduced in parallel and compared each time a (deep) w.h.n.f. is reached. The KAMs are run with **`delta`** and **`rt`** disabled.
- The α-convertibility check is not attempted before the full convertibility check as in Matita. Moreover the terms must be valid.
- Two sorts are compared by level but the KAM stacks are not compared because a sort after an application is always invalid.

The convertibility check (continued)

- Two local references are compared by "*apix*" (instead of by index) so they need not to be relocated (delifted) before the comparison.
- Two global references to abstractions are compared by "*apix*" (age). This is the same as comparing them by name.
- Two global references to abbreviations are compared by "*apix*". If they differ, the abbreviation with the greatest age is expanded.
- Two abstractions are matched by comparing their domains and then their scopes, after pushing the domains in the respective KAMs.
- Depending on the `si` flag, sort inclusion is attempted as a last resort before asserting that the compared terms are not convertible.
- `si` is disabled when matching the KAM stacks and the domains of abstractions, otherwise some non-normalizing terms (Ω) are valid.

Sort hierarchy management

```
type graph = string * (int -> int)        (* sort hierarchy *)
val graph_of_string: string -> graph      (* graph constructor *)
val string_of_graph: graph -> string      (* graph name *)
val apply: graph -> int -> int            (* graph look up *)
val set_sorts: string list -> int -> int (* sort registration *)
val get_sort: int -> string               (* sort look up *)
```

- The sort hierarchy parameter (h) has predefined values, denoted by strings, that ensure its strict monotonicity (`graph_of_string`).
- The recognized values are “Zn” with $n > 0$, meaning $h(l) \equiv l + n$.
- This parameter is set from the command line and defaults to “Z2”.
- The parameter is applied using the function `apply`.
- We can assign names to sorts for presentational purposes (`set_sorts`, `get_sort`), which are stored in a hash table.

Computing the canonical type

```
val type_of: (term -> 'a) -> status -> graph -> kam -> term -> 'a
```

- This function is better implemented using the CPS paradigm.
- The local environment (the KAM) contains just valid items.
- Every δV is annotated with the inferred type of V before entering the environment, becoming $\delta\langle W\rangle.V$, so V is typed only once.

```
(* m: kam, u: type of the function, w: type of the argument *)
let assert_applicability st m u w = match xwhd st m u with
   | _, Sort _                    -> error ...
   | mu, Bind (_, Abst u, _) ->
       if are_convertible st mu u m w then () else error ...
   | _                            -> assert false
```

- `mu` and `u` go from `xwhd` to `are_convertible` as they are.
- Passing two KAMs to `are_convertible` is crucial here.

Validation

```
val validate: si:bool -> graph -> entity -> entity
```

Testing the validator

- We can translate Jutting's formal specification of Landau's "*Grundlagen der Analysis*" from Aut – QE into $\lambda\delta$ "*brg*" and we can validate it enabling sort inclusion (maybe not necessary).
- Two steps: we build an intermediate representation where the syntactic shorthand is removed, and we encode this into $\lambda\delta$ "*brg*".
- The validator implements both steps of the translation as well.
- The intermediate language is a version $\lambda\delta$ still under design.
- The validator has a "*multi-kernel*" architecture and will be able to validate this version of $\lambda\delta$ (and hopefully others) in the future.

Long-term persistence of the global environment

- We can produce an XML representation of each `entity`.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ENTITY SYSTEM "http://helm.cs.unibo.it/lambda-delta/xml/ld.dtd">
<ENTITY hierarchy="Z2" options="si">
   <ABBR uri="ld:/brg/grundlagen/l/not.ld" name="not" mark="6">
      <Cast>
         <Abst name="a">
           <Sort position="1" name="Prop"/>
         </Abst>
         <Sort position="1" name="Prop"/>
      </Cast>
      <Abst name="a">
         <Sort position="1" name="Prop"/>
      </Abst>
      <Appl>
         <GRef uri="ld:/brg/grundlagen/l/con.ld" name="con"/>
      </Appl>
      <Appl>
         <LRef position="0" name="a"/>
      </Appl>
      <GRef uri="ld:/brg/grundlagen/l/imp.ld" name="imp"/>
   </ABBR>
</ENTITY>
```

Some statistical data

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	23%	1.7s
validation	67%	4.9s

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	global r.t.	0
global δ	17166	local r.t.	1
υ	2040476	s.i.	904

- The "*intrinsic complexity*" approximates the number of nodes.

  The validator was run on a 2×AMD Athlon MP 1800+, 1.53 GHz.

  The ζ-contractions, avoided by the validator, would be: 3694769.

Thank you