ELPI: fast, Embeddable, λProlog Interpreter

Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, Enrico Tassi

ELPI: fast, Embeddable, λProlog Interpreter

Cvetan Dunchev,[1] Ferruccio Guidi,[1] Claudio Sacerdoti Coen,[1] Enrico Tassi[2]

[1] Department of Computer Science, University of Bologna, name.surname@unibo.it
[2] Inria Sophia-Antipolis, name.surname@inria.fr

Abstract. We present a new interpreter for λProlog that runs consistently faster than the byte code compiled by Teyjus, that is believed to be the best available implementation of λProlog. The key insight is the identification of a fragment of the language, which we call reduction-free fragment ($\mathcal{L}_\lambda^\beta$), that occurs quite naturally in λProlog programs and that admits constant time reduction and unification rules.

1 Introduction

λProlog is a logic programming language based on an intuitionistic fragment of Church's Simple Theory of Types. An extensive introduction to the language with examples can be found in [9]. Teyjus [10,7] is a compiler for λProlog that is considered to be the fastest implementation of the language. The main difference with respect to Prolog is that λProlog manipulates λ-tree expressions, i.e. syntax containing binders. Therefore, the natural application of λProlog is meta-programming (see [11] for an interesting discussion), including: automatic generation of programs from specifications; animation of operational semantics; program transformations and implementation of type checking algorithms.

Via the Curry-Howard isomorphism a type-checker is a proof-checker, the main component of an interactive theorem prover (ITP). Indeed the motivation of our interest in λProlog is that we are looking for the best language to implement the so called *elaborator* component of an ITP. The elaborator is used to type check the terms input by the user. Such data, for conciseness reasons, is typically incomplete and the ITP is expected to infer what is missing. The possibility to extend Coq's built-in elaborator with user provided "logic programs" (in the form of Canonical Structures [4,1] or Type Classes [12]) to help it infer the missing data turned out to be a key ingredient in successful formalizations like [3]. Embedding a λProlog interpreter in an ITP would enable the elaborator and its extensions to be expressed in the same, high level, language. A crucial requisite for this plan to be realistic is the efficiency of the λProlog interpreter.

In this paper we introduce ELPI, a fast λProlog interpreter written in OCaml that can be easily embedded in OCaml softwares, like Coq. In particular we focus on the insight that makes ELPI fast when dealing with binders by identifying a reduction-free fragment ($\mathcal{L}_\lambda^\beta$) of λProlog that, if implemented correctly, admits constant-time unification and reduction operations. We analyze the role of β-reduction in Section 2 and higher order unification in Section 3; we discuss bound names representations in Section 4; we define $\mathcal{L}_\lambda^\beta$ in Section 5 and we assess the results in Section 6.

2 The two roles of β-reduction in λProlog

Example 1 implements type-checking and reduction for λ-terms represented in λ-tree syntax. For instance, the object-level encoding of $(\lambda x.xx)$ is the term `(lam (x\ app x x))` of type $\mathcal{T}$. The syntax `(x\ F)` denotes the λ-abstraction of λProlog, that binds `x` in `F`; `lam` is the constructor for object-level abstraction, that builds a term of type $\mathcal{T}$ from a function of type $\mathcal{T} \to \mathcal{T}$; `app` takes two terms of type $\mathcal{T}$ and builds their object-level application of type $\mathcal{T}$. Following the tradition of Prolog, capitals letters denote unification variables.

The second clause for the `of` predicate shows a recurrent pattern in λProlog: in order to analyze an higher order term, one needs to recurse under a binder. This is achieved combining the forall quantifier, written `pi x\ G`, with logical implication `H => I`. The operational semantics implements the standard introduction rules of implication and the universal quantifier: the forall quantifier declares a new local constant `x`, meant to be fresh in the entire program; logical implication temporarily augments the program with the new axiom `H` about `x`.

In Example 1, line 4, the functional (sub-)term `F` is applied to the fresh constant `x`. Being `F` a function, the β-redex `(F x)`, once reduced, denotes the body of our object-level function where the bound variable is replaced by the fresh constant `x`. The implication is used to assume `A` to be the type of `x`, in order to prove that the body of the abstraction has type `B` and therefore the whole abstraction has type `(arr A B)` (i.e. $A \to B$). Note that, unlike in the standard presentation of the typing rules, we do not need to manipulate an explicit context Γ to type the free variables. Instead the assumptions of the form `(of x A)` are just added to the program's set of clauses, and λProlog takes care of dropping them when `x` goes out of scope. Example: if the initial goal is `(of (lam (w\ app w w)) T)` by applying the second clause we assign `(arr A B)` to `T` and generate a new goal `(of (app c c) B)` (where `c` is the fresh constant substituted for `w`) to be solved with the extra clause `(of c A)` at disposal.

In the type-checking example, the meta-level β-reduction is only employed to inspect a term under a binder by replacing the bound name with a fresh constant. The reduction example in line 6 shows instead a radically different pattern: in order to implement object-level substitution — and thus object-level β-reduction — we use the meta-level β-reduction. E.g. if `F` is `(w\ app w w)` then `(F N)` reduces to `(app N N)`. Note that in this case β-reduction is fully general, because it replaces a name with a general term, not constrained to be a fresh constant. This distinction is crucial in the definition of $\mathcal{L}_\lambda^\beta$ in Section 5.

```
1 of (app M N) B :-
2   of M (arr A B), of N A.
3 of (lam F) (arr A B) :-
4   pi x\ of x A => of (F x) B.
5 cbn (lam F) (lam F).
6 cbn (app (lam F) N) M :- cbn (F N) M.
7 cbn (app M N) R :-
8   cbn M (lam F), cbn (app (lam F) N) R.
```

Example 1: Type checker and Weak CBN for simply typed λ-calculus.

3 Higher Order unification

Higher order (HO) unification admits no most general unifiers (MGUs), forcing implementations to enumerate all solutions or delay the flexible-rigid and the flexible-flexible problems. Moreover, the presence of binders requires a way to avoid captures, i.e. to check that unification variables are instantiated with terms containing only bound variables in their scope.

To cope with the absence of MGUs, Dale Miller identified in [8] a well-behaved fragment ($\mathcal{L}_\lambda$) of higher-order unification that admits MGUs and that is stable under λProlog resolution. The restriction defining $\mathcal{L}_\lambda$ is that unification variables can only be applied to (distinct) variables (i.e. not arbitrary terms) that are not already in the scope of the variable. Such fragment can effectively serve as a primitive for a programming language and indeed Teyjus 2.0 is built around this fragment: no attempt to enumerate all possible unifiers is performed, and unification problems falling outside $\mathcal{L}_\lambda$ are just delayed. Many interesting λProlog programs can be rewritten to fall in the fragment. For example, we can make `cbn` of Example 1 stay in $\mathcal{L}_\lambda$ by replacing line 6 (that contains the offending `(F N)` term) with the following code:

```
1 cbn (app (lam F) N) M :- subst F N B, cbn B M.
2 subst F N B :- pi x\ copy x N => copy (F x) B.
3 copy (lam F1) (lam F2) :- pi x\ copy x x => copy (F1 x) (F2 x).
4 copy (app M1 N1) (app M2 N2) :- copy M1 M2, copy N1 N2.
```

The idea of `subst` is that the term `F` is recursively copied in the following way: each bound variable is copied in itself but for the top one that is replaced by `N`. The interested reader can find a longer discussion about `copy` in [9, page 199]. The `of` program falls naturally in $\mathcal{L}_\lambda$, since `F` is only applied to the fresh variable `x` (all unification variables in a λProlog program are implicitly existentially bound in front of the clause, so `F` does not see `x`). The same holds for `copy`.

In λProlog unification takes place under a mixed prefix of $\forall$ and $\exists$ quantifiers. Their order determines if a unification variable (an existential) can be assigned to a term that contain a universally quantified variable. E.g. $\forall x, \exists Y, Y = x$ is provable while $\exists Y, \forall x, Y = x$ is not. An implementation can keep track of the scoping constraints using *levels*. When a clause's head is unified with the goal in a context of length n, the universally quantified variables of the clause are instantiated to unification variables X^n where the level n records that X has only visibility of the initial prefix of length n of the context. If later a fresh constant is added by the `pi` rule, the constant occupies position $n+1$ (its level is $n+1$) and it will not be allowed to occur in instances of the variable X^n. From now on we will write levels in superscript.

If we run the program `(of (lam f\lam w\app f w) T`^{0}`)`, after two steps the goal becomes `(of (app c`^{1} `d`^{2}`) B`^{0}`)`. Concretely, Teyjus replaces the bound names `f` and `w` with the level-annotated fresh constants c^1 and d^2 performing the β-reductions. As a crucial optimization [7] Teyjus implements reductions in a lazy way using an explicit substitution calculus. The reader can find this example developed in full details at page 6, where we demonstrate how substitutions of bound names by fresh level-annotated constants can be avoided in $\mathcal{L}^\beta_\lambda$.

4 Bound variables

The last missing ingredient to define $\mathcal{L}_\lambda^\beta$ and explain why it can be implemented efficiently is to see how systems that manipulate λ-terms accommodate α-equivalence. Bound variables are not represented by using real names, but canonical "names" (hence α-equivalence becomes syntactic equality). De Bruijn introduced two, dual, naming schemas for λ-terms in [2]: depth indexes (DBI) and levels (DBL). In the former, that is the most widely adopted one, a variable is named n if its binder is found by crossing n binders going in the direction of the root. In the latter a variable named n is bound by the n-th binder one encounters in the path from the root to the variable. Below we write the term $\lambda x.(\lambda y.\lambda z.f\ x\ y\ z)\ x$ and its reduct in the two notations:

$$\text{Indexes: } \lambda x.(\lambda y.\lambda z.f\ x_2\ y_1\ z_0)\ x_0 \to_\beta \lambda x.\lambda z.f\ x_1\ x_1\ z_0$$
$$\text{Levels: } \lambda x.(\lambda y.\lambda z.f\ x_0\ y_1\ z_2)\ x_0 \to_\beta \lambda x.\lambda z.f\ x_0\ x_0\ z_1$$

In both notations when a binder is removed and the corresponding variable substituted some "renaming" (called lifting) is performed. Teyjus follows a third approach that mixes the two, using indexes for variables bound in the terms, and levels for variables bound in the context. The advantage is that no lifting is required when moving a term under additional binders. However, an expensive substitution of a level for an index is required to push a binder to the context.

In ELPI we crucially chose DBL because of the following three properties:

DBL1 x_i in Γ keeps the same name x_i in any extended context Γ, Δ
DBL2 the variables bound by Γ in a β-redex keep their name in the reduct
DBL3 when a binder is pushed to the context, the bound occurrences keep their name: no lifting is required to move from $\Gamma \vdash \forall x_i, p(x_i)$ to $\Gamma, x_i \vdash p(x_i)$

Another way to put it is that variables already pushed in the context are treated *exactly as constants*, and that the two notions of level — De Bruijn's and the position in the context introduced in Section 3 — coincide.

5 The reduction-free fragment $\mathcal{L}_\lambda^\beta$

λProlog is a truly higher order language: even clauses can be passed around, unified, etc. Nevertheless this plays no role here, so we exclude formulas from the syntax of terms. Therefore, our terms are defined just by:

$$t ::= x_i \mid X^j \mid \lambda x_i.t \mid t\ t$$

Since variables follow the DBL representation, we do not have a case for constants like `app` or `lam`, that are represented as x_i for some negative i. Since the level of a variable completely identifies it, when we write $x_i \dots x_{i+k}$ we mean k distinct bound (i.e. $i \geq 0$) variables. The superscript j annotates unification variables with their visibility range ($0 \leq j$, since all global constants are in range). A variable X^j has visibility of all names strictly smaller than j. E.g. X^1 has visibility only of $\{\dots, x_{-1}, x_0\}$, and X^3 has visibility of $\{\dots, x_{-1}, x_0, x_1, x_2\}$. Technically, when following the De Bruijn convention, we could just write $\lambda x_i.t$ as $\lambda.t$. We keep writing the name x_i to ease reading.

Definition 1 ($\mathcal{L}_\lambda^\beta$) *A term is in the reduction-free fragment $\mathcal{L}_\lambda^\beta$ iff every occurrence of a unification variable X^j is applied to $x_j \dots x_{j+k-1}$ for $k >= 0$.*

We allow $k = 0$ to accept variables that are not applied. A consequence of the definition is that if a term is in $\mathcal{L}_\lambda^\beta$ then all occurrences of applications of unification variables can be instantiated with a term closed in an initial segment of the λProlog context seen as a ordered list. Examples: $X^2\ x_2\ x_3$ and X^2 are in the fragment; $X^2\ x_3$ and $X^2\ x_3\ x_2$ are not; $X^2\ x_2\ x_3$ can be instantiated with any term closed in $\{\dots, x_0, x_1, x_2, x_3\}$.

Observe that the programs in Example 1 (when `cbn` is rewritten to be in the pattern fragment as in Section 3) are in $\mathcal{L}_\lambda^\beta$. Also, every Prolog program is in $\mathcal{L}_\lambda^\beta$. As we will see in Section 6, a type-checker for a dependently typed language and evaluator based on a reduction machine are also naturally in $\mathcal{L}_\lambda^\beta$, showing that, in practice, the fragment is quite expressive.

Property 1 (Decidability of HO unification) *Being $\mathcal{L}_\lambda^\beta$ included in the pattern-fragment $\mathcal{L}_\lambda$, higher order unification is decidable for $\mathcal{L}_\lambda^\beta$.*

The most interesting property of $\mathcal{L}_\lambda^\beta$, which also justify its name, is:

Property 2 (Constant time head β-reduction) *Let σ be a* valid *substitution for existentially quantified variables. Then the first $k-1$ head reductions of $(X^j\ x_j \dots x_{j+k-1})\sigma$ can be computed in constant time.*

A valid substitution assigns to X^j a term t of the right type (as in simply typed λ-calculus) and such that the free variables of t are all visible by X^j (all x_i are such that $i < j$). Therefore $X^j\sigma = \lambda x_j. \dots \lambda x_{j+n}.t$ for some n. Then

$$(X^j\ x_j \dots x_{j+k-1})\sigma = \begin{cases} t\ x_{j+n+1} \dots x_{j+k-1} \text{ if } n+1 < k \\ \lambda x_{j+k}. \dots \lambda x_{j+n}.t \quad \text{otherwise} \end{cases} \tag{1}$$

Thanks to property **DBL2**, Equation 1 is *syntactical*: no lifting of t is required. Hence the β-reductions triggered by the substitution of X^j take constant time.

Property 3 (Constant time unification) *A unification problem of the form $X^j\ x_j \dots x_{j+k-1} \equiv t$ can be frequently solved in constant time.*

The unification problem $X^j\ x_j \dots x_{j+k-1} \equiv t$ can always be rewritten as two simpler problems: $X^j \equiv \lambda x_j. \dots \lambda x_{j+k-1}.Y^{j+k}$ and $Y^{j+k} \equiv t$ for a fresh Y. The former is a trivial assignment that requires no check. The latter can be implemented in constant time iff no occur-check is needed for X and if the level of the highest free variable in t can be recovered in $O(1)$ and is smaller than $j+k$. The recovery can be economically implemented caching the maximum level in the term, that is something often pre-computed on the input term in linear time. Avoiding useless occur-check is a typical optimization of the Warren Abstract Machine (WAM), e.g. when X occurs linearly in the head of a clause. These

properties enable us to implement the operational semantics of `pi` in constant time for terms in $\mathcal{L}_\lambda^\beta$.

We detail an example. The first column gathers the fresh constants and extra clauses. The second one shows the current goal(s) and the program clause that is used to back chain.

Context	Goals and refreshed program clause
	`of (lam x0\lam x1\app x0 x1) T0`
	`of (lam F0) (arr A0 B0) :- pi x0\ of x0 A0 => of (F0 x0) B0`
`x0;(of x0 A0)`	`of (lam x1\app x0 x1) B0`
	`of (lam G1) (arr C1 D1) :- pi x1\ of x1 C1 => of (G1 x1) D1`
`x0;(of x0 A0)`	`of (app x0 x1) D0`
`x1;(of x1 C0)`	`of (app M2 N2) S2 :- of M2 (arr R2 S2), of N2 R2`
`x0;(of x0 A0)`	`of x0 (arr R2 S0), of x1 R2`
`x1;(of x1 C0)`	`of x0 A0 , of x1 C0`

After the first step we obtain `F0:= x0\lam x1\app x0 x1`; `T0:= arr A0 B0`; the extra clause about $\mathtt{x}_0$ in the context and a new subgoal. Thanks to property **DBL3**, $\mathtt{x}_0$ has been pushed to the context in constant time. Note that the redex `(F0 x0)` is in $\mathcal{L}_\lambda^\beta$ and thanks to Equation 1 head normalizes in constant time to `(lam x1\app x0 x1)`. The same phenomenon arises in the second step, where we obtain `G1:= x1\app x0 x1` and we generate the redex `(G1 x1)`. Unification variables are refreshed in the context under with the clause is used, e.g. `C` is placed at level 1 initially, but in consequence to a unification step they may be *pruned* when occurring in a term assigned to a lower level unification variable. Example: unifying `B0` with `(arr C1 D1)` prunes `C` and `D` to level 0.

The choice of using DBL for bound variables is both an advantage and a complication here. Clauses containing no bound variables, like `(of x0 A0)`, require no processing thanks to **DBL1**: they can be indexed as they are, since the name $\mathtt{x}_0$ is stable. The drawback is that clauses with bound variables, like the one used in the first two back chains, need to be lifted: the first time the bound variable is named $\mathtt{x}_0$, while the second time $\mathtt{x}_1$. Luckily, this renaming, because of property **DBL1**, can be performed in constant time using the very same machinery one uses to refresh the unification variables. E.g. when the WAM unifies the head of a clauses it assigns fresh stack cells: the clause is not really refreshed and the stack pointer is simply incremented. One can represent the locally bound variable as an extra unification variable, and initialize, when `pi` is crossed, the corresponding stack cell to the first $\mathtt{x}_i$ free in the context.

Stability of $\mathcal{L}_\lambda^\beta$. Unlike $\mathcal{L}_\lambda$, $\mathcal{L}_\lambda^\beta$ is not stable under λProlog resolution: a clause that contains only terms in $\mathcal{L}_\lambda^\beta$ may generate terms outside the fragment. Therefore an implementation must handle both terms in $\mathcal{L}_\lambda^\beta$, with their efficient computation rules, and terms outside the fragment. Our limited experience so far, however, is that several programs initially written in the fragment remains in the fragment during computation, or they can be slightly modified to achieve that property.

6 Assessment and conclusions

We assess ELPI on a set of synthetic benchmarks and a real application. Synthetic benchmarks are divided into three groups: first order programs from the Aquarius test suite (crypto-multiplication, μ-puzzle, generalized eight queens problem and the Einstein's zebra puzzle); higher order programs falling in $\mathcal{L}_\lambda^\beta$; and an higher order program falling outside $\mathcal{L}_\lambda^\beta$ taken from the test suite of Teyjus normalizing expressions in the SKI calculus.

The programs in $\mathcal{L}_\lambda^\beta$ are respectively type checking lambda terms using the `of` program of Example 1 and reducing expressions like 5^5 in the syntax of Church numerals using a call by value/name (CBV/CBN) strategy. The typeof test was specifically conceived to measure the cost of moving under binders: the type checked terms, projections, are mainly made of `lam` nodes.

Test	ELPI		Teyjus		ELPI/Teyjus	
	time (s)	space (Kb)	time (s)	space (Kb)	time	space
crypto-mult	3.48	27,632	6.59	18,048	0.52	1.53
μ-puzzle	1.82	5,684	3.62	50,076	0.50	0.11
queens	1.41	108,324	2.02	69,968	0.69	1.54
zebra	0.85	7,008	1.89	8,412	0.44	0.83
typeof	0.27	8,872	5.64	239,892	0.04	0.03
reduce_cbv	0.15	7,248	11.11	57,404	0.01	0.12
reduce_cbn	0.33	8,968	0.81	102,896	0.40	0.08
SKI	1.32	15,472	2.68	8,896	0.49	2.73

The data in the table shows that ELPI shines on programs in $\mathcal{L}_\lambda^\beta$, and compares well outside it. The alternating performance Teyjus on the reduction tests has to be attributed to the explicit substitutions (ES) machinery [7] when employed to cross binders: by its very nature ES fit well a lazy reduction strategy like CBN (even if some space overhead is visible). On the contrary ES are counterproductive in the CBV case since the program, by fully traversing the redex argument, systematically pushes the suspended substitution to the leaves of the term, completely defeating the purpose of the entire machinery (i.e. if the substitution has to be performed, there is no gain in delaying it). If one makes Teyjus artificially push explicit substitutions in the CBN case too, he halves memory consumption but degrades the performances by 10 seconds, confirming the time we see in the CBV case is dominated by the overhead of ES. By avoiding substitution when crossing binders ELPI is not only faster, but also more predictable performance wise: as one expects CBV is faster than CBN in computing the normal form of 5^5 since it avoids duplicating non-normal terms.

The real application we present is a checker for the formal system $\lambda\delta$ [5,6] Such checker is able to validate the proof terms of the formalization of Landau's "Grundlagen" [13] done in Automath. The reference checker for $\lambda\delta$, named Helena, has been implemented in OCaml. Our λProlog implementation follows it closely, and naturally falls in $\mathcal{L}_\lambda^\beta$. Nevertheless, the λProlog code is much simpler than the corresponding OCaml code and consists of just 50 clauses.

The "Grundlagen" is a theory comprising definitions and proofs for a total of 6911 items (circa 8MB of data). Teyjus seems to have a fixed maximum heap

size of 256MB that in turn limits it to the verification of the first 2615 items. In the table we compare pre-processing (Pre) time like parsing, compilation or elaboration, and verification (Ver). We compare ELPI with Helena, Teyjus, and Coq. The Coq system implements a type checker for a λ-calculus strictly more expressive than $\lambda\delta$, hence can check the proof terms directly but surely incurs in some overhead. We use its timings as a reference for the order of magnitude between the performance of ELPI and the ones of a state-of-the-art ITP. When applicable we compare softwares compiled to native code or interpreted.

Time (s) for 2615 items only			
	ELPI	Teyjus	ELPI/Teyjus
Pre	2.55	49.57	0.05
Ver	3.06	203.36	0.02
RAM (Mb)	91,628	1,072,092	0.09

Time (s) for all 6911 items					
Task	Helena		ELPI	Coq	
	interp.	comp.	interp.	interp.	comp.
Pre	2.42	0.41	9.04	49.28	8.83
Ver	4.40	0.33	13.90	7.21	1.19

Our conclusion is that $\mathcal{L}_\lambda^\beta$ admits a very efficient implementation and is large enough to express realistic programs like a type checker for a dependently typed λ-calculus. ELPI is under active development at http://lpcic.gforge.inria.fr.

References

1. A. Asperti, W. Ricciotti, C. Sacerdoti Coen, and E. Tassi. Hints in unification. In *TPHOLs*, pages 84–98, 2009.
2. 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*, pages 375–388. North-Holland, 1994.
3. G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Le Roux, A. Mahboubi, R. O'Connor, S. Ould Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A Machine-Checked Proof of the Odd Order Theorem. In *ITP 2013*, volume 7998 of *LNCS*, pages 163–179, July 2013.
4. G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. *J. Funct. Program.*, 23(4):357–401, 2013.
5. F. Guidi. The Formal System $\lambda\delta$. *ToCL*, 11(1):5:1–5:37, November 2009.
6. F. Guidi. A Verified Translation of Landau's Grundlagen" from Automath into a Pure Type System, via $\lambda\delta$, 2015. Submitted to JFR. http://lambdadelta.info/.
7. C. Liang, G. Nadathur, and X. Qi. Choices in representation and reduction strategies for lambda terms in intensional contexts. *JAR*, 33(2):89–132, 2004.
8. D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1:253–281, 1991.
9. D. Miller and G. Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012.
10. G. Nadathur and D. J. Mitchell. Teyjus - A compiler and abstract machine based implementation of λprolog. In *CADE-16*, pages 287–291, 1999.
11. O. Ridoux. *Lambda-Prolog de A a Z... ou presque*. Habilitation à diriger des recherches, Université de Rennes 1, 1998.
12. M. Sozeau and N. Oury. First-class type classes. In *TPHOLs*, volume 5170 of *LNCS*, pages 278–293, 2008.
13. L.S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the automath system*, volume 83 of *Mathematical Centre Tracts*. 1979.