Implementing Type Theory in Higher Order Constraint Logic Programming

Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi

[1] Department of Computer Science and Engineering, University of Bologna
ferruccio.guidi@unibo.it
[2] Department of Computer Science and Engineering, University of Bologna
claudio.sacerdoticoen@unibo.it
[3] Inria Sophia-Antipolis, Enrico.Tassi@inria.fr

Abstract

In this paper we are interested in high-level programming languages to implement the core components of an interactive theorem prover for a dependently typed language: the kernel — responsible for type-checking closed terms — and the elaborator — that manipulates terms with holes or, equivalently, partial proof terms.

In the first part of the paper we confirm that λProlog, the language developed by Miller and Nadathur since the 80s, is extremely suitable for implementing the kernel, even when efficient techniques like reduction machines are employed.

In the second part of the paper we turn our attention to the elaborator and we observe that the eager generative semantics inherited by Prolog makes it impossible to reason by induction over terms containing metavariables. We also conclude that the minimal extension to λProlog that allows to do so is the possibility to delay inductive predicates over flexible terms, turning them into (set of) constraints to be propagated according to user provided constraint propagation rules.

Therefore we propose extensions to λProlog to declare and manipulate higher order constraints, and we implement the proposed extensions in the ELPI system. Our test case is the implementation of an elaborator for a type theory as a CLP extension to a kernel written in plain λProlog.

1 Introduction

A pragmatic reconstruction of λProlog. In [5] Belleannée et. al. propose a pragmatic reconstruction of λProlog [26, 28, 27], the Higher Order Logic Programming (HOLP) language introduced by Dale Miller and Gopalan Nadathur in the '80s. Their conclusion is that λProlog can be characterised as the minimal extension of Prolog that allows to program by structural induction on λ-terms. According to their reconstruction, in order to achieve that goal, Prolog needs to be first augmented with λ-abstractions in the term syntax; then types are added to drive full higher-order unification; then universal quantification in goals and η-equivalence are required to express relations between λ-abstractions and their bodies; and finally implication in goals is needed to allow for definitions of predicates by structural induction.

By means of λ-abstractions in terms, λProlog can easily encode all kind of binders without the need to take care of binding representation, α-conversion, renaming and instantiation. Structural induction over syntax with binders is also made trivial by combining universal quantification and implication following the very same pattern used in Logical Frameworks like LF (also called λP). Indeed, LF endowed with an HOLP semantics, is just a sub-language of λProlog.

The "hello world" example of λProlog is therefore the following two lines program to compute the simple type of a λ-expression:

```
kind term,typ type.             type arr typ -> typ -> typ.
type app term -> term -> term.  type lam typ -> (term -> term) -> term.

type of term -> typ -> o.
of (app M N) B :- of M (arr A B), of N A.
of (lam A F) (arr A B) :- pi x\ of x A => of (F x) B.
```

λProlog for proof-checking. According to the Curry-Howard isomorphism, the program above can also be interpreted as a proof-checker for minimal propositional logic. By escalating the encoded λ-calculus to more complex terms and types, it is possible to obtain a proof-checker for a richer logic, like the Calculus of Inductive Constructions (CIC) that, up to some variations, is the common logic shared by the interactive theorem provers (ITPs) Coq, Matita and Lean.

All the ITPs mentioned above are implemented following basically the same architecture. At the core of the system there is the *kernel*, that is the trusted code base (together with the compiler and run-time of the programming language the system is written on). The kernel just implements the type-checker together with all the judgements required for type-checking, namely: well-formation of contexts and environments, substitution, reduction, convertibility. The last three judgements are necessary because the type system has dependent types and therefore types need to be compared up to computation.

Realistic implementations of kernels employ complex techniques to improve the performance, like reduction machines to speed up reduction and heuristics to completely avoid reduction during conversion when possible. Is λProlog suitable to implement such techniques? The question is not totally trivial because, for example, reduction machines typically rely on the representation of terms via De Bruijn indexes.

We already gave a positive answer in [13] where we describe ELPI, a fast λProlog interpreter, and implement a type-checker for a dependently type lambda calculus. ELPI is able to run such type checker on the Automath's proof terms of Landau's Grundlagen reasonably fast (5 times slower than a corresponding OCaml implementation). In Section 2 we confirm the answer again by implementing a kernel for a generic Pure Type System (PTS) that supports cumulativity between sorts. We then instantiate the PTS to the one of Matita and we modularly add to the kernel support for globally defined constants and primitive inductive types, following the variant of CIC of Matita. Finally, we embed ELPI in Matita and divert all calls to the λProlog kernel in order to test it on the arithmetic library of the system.

From proof-checking to interactive proving. The kernel is ultimately responsible for guaranteeing that a proof built using an ITP is fully sound. However, in practice the user never interacts with the kernel and the remaining parts of the system do not depend on the behaviour of the kernel. Where the real intelligence of the system lies is instead in the second layer, called *elaborator* or *refiner* [11, 4]. In Section 3 we recall what an elaborator does and we explain why the state of the art is not satisfactory, which motivated at the very beginning our interest in using HOLP languages to implement elaborators

In Section 4 we pose again the same question we posed for the kernel. Is λProlog suitable as a very high-level programming language to implement an elaborator? If not, what shall be added? We conclude that λProlog is not suitable and, combining and extending existing ideas in the literature, we introduce the first Constraint Programming extension of λProlog. We also implemented the language extensions in the ELPI system.

In Section 5 we work towards an implementation of an elaborator for CIC written in ELPI, obtained extending modularly a kernel presented in Section 2. The implementation is the first major use case for the language. The final Section 6 contains comparisons with related work and future works.

2 A modular kernel for CIC

We now scale the type-checker for simply typed λ-calculus of Section 1 to a type-checker for a generic PTS that also allows cumulativity between universes and dependent products that are covariant in the second argument. Then we instantiate it to obtain the predicative universal fragment of Luo's ECC [21] and the PTS of CIC, and then we modularly extend the type-checker to the whole CIC by adding inductive types as well.

Term representation and type-checking rules We start identifying syntactically types and terms, adding a new constructor for sorts of the PTS and by refining `arr` to the dependently typed product. `@univ` is a macro (in the syntax of the ELPI interpreter [13]) that will be instantiated in the file that implements the PTS, for example with the type of integers.

```
kind term type.
type sort @univ -> term.        type arr term -> (term -> term) -> term.
type app term -> term -> term.  type lam term -> (term -> term) -> term.
```

We introduce the typing rules for the new constructors and we refine the ones for abstraction and application. In particular, the types inferred and expected for the argument of an application are meant to be compared up to β-reduction and cumulativity of universes. The `sub` predicate, that will be instantiated later, implements this check. The `match_sort` and the `match_arr` predicates, also to be instantiated later, are used to check if the weak head normal form of the first argument is respectively a sort or a dependent product. In both cases the sub-terms are returned by the predicate. For example, `match_arr (arr nat x \ x) A F` is meant to instantiate `A` with `nat` and `F` with `x \ x`. Finally, the `succ` and `max` predicates are meant to be later instantiated with the corresponding rules of a PTS.

```
type of term -> term -> o.

of (sort I) (sort J) :- succ I J.

of (app M N) BN :- of M TM, match_arr TM A1 Bx, of N A2, sub A2 A1, BN = Bx N.

of (lam A F) (arr A B) :-
 of A SA, match_sort SA (sort _),
 (pi x\ of x A => of (F x) (B x)), of (arr A B) _.

of (arr A Bx) (sort K) :-
 of A TA, match_sort TA I,
 (pi x\ of x A => of (Bx x) (TB x), match_sort (TB x) J),
 max I J K.
```

The rules above have the merit of being syntax-directed, never requiring backtracking and always inferring the most general type, in the sense of Luo [21].

Reduction and conversion rules: a trivial but inefficient implementation. To implement `sub`, `match_sort` and `math_arr` we first implement weak head reduction in one step, predicate `whd1`, and then its transitive closure `whd*`. The former is trivial because we reuse the β-reduction of λProlog for capture avoiding substitution. The predicates `sub` is then defined by levels: to compare two terms, we reduce both to their weak head normal forms via whd* and then compare the two heads. If they match, the comparison is called recursively on the subterms.

```
type whd1,whd* term -> term -> prop.

whd1 (app M N) R :- whd* M (lam _ F), R = F N.
whd* A B :- whd1 A A1, !, whd* A1 B.
whd* X X.

type match_sort term -> @univ -> prop.
match_sort T I :- whd* T (sort I).

type match_arr term -> term -> (term -> term) -> prop.
match_arr T A F :- whd* T (arr A F).

type conv,conv-whd term -> term -> prop.

conv A B :- whd* A A1, whd* B B1, conv-whd A1 B1, !.

% fast path + axiom rule for sorts and bound variables
conv-whd X X :- !.
% congruence rules
conv-whd (app M1 N1) (app M2 N2) :- conv M1 M2, conv N1 N2.
conv-whd (lam A1 F1) (lam A2 F2) :- conv A1 A2, pi x\ conv (F1 x) (F2 x).
conv-whd (arr A1 F1) (arr A2 F2) :- conv A1 A2, pi x\ conv (F1 x) (F2 x).

type sub,sub-whd term -> term -> prop.

sub A B :- whd* A A1, whd* B B1, sub-whd A1 B1, !.

sub-whd A B :- conv A B, !.
sub-whd (sort I) (sort J) :- lt I J.
sub-whd (arr A1 F1) (arr A2 F2) :- conv A1 A2, pi x\ sub (F1 x) (F2 x).
```

Defining the PTS To obtain the kernel, the programmer just needs to accumulate together the λProlog file that implements the type-checking rules, the one that deals with reduction and conversion, and a final file containing the definition of a particular PTS. The latter file just needs to implement the succ, max and lt predicates over universes.

For example, the following lines define the predicative hierarchy of Luo's ECC, using the integer i to represent the universe Type_i. The macro directive is recognised by the ELPI interpreter that transparently replaces all occurrences of @univ by int.

```
macro @univ :- int.
max N M M :- N =< M.     lt I J :- I < J.
max N M N :- M < N.      succ  I J :- J is I + 1.
```

Reduction and conversion rules: an efficient implementation via a reduction machine for call-by-need. The implementation of weak head reduction given before is well-known to be very inefficient, because the body of the abstraction of a β-redex is immediately and completely traversed when the redex is fired, unless the reduction of λProlog is implemented via explicit substitutions like in Teyjus. Moreover, the strategy implemented by whd* is call-by-name, that is inefficient per se.

To speed up reduction, we now provide a different implementation based on the reduction machine for call-by-need called Wadsworth Abstract Machine [1]. The machine is a modified Krivine Abstract Machine (KAM [20]) that records in the machine environment both terms and their normal forms, the latter being computed lazily by a new invocation of the reduction machine that is fired only when the normal form is required for the first time, i.e. when the variable bound to that value occurs in head position during reduction.

In standard implementations, variables are represented by De Bruijn indexes and machine environments are stacks, indexed by the variables. Since we do not use and do not want to use indexes, we take a different approach. The machine environment is represented by a set of `(val N T V NF)` assumptions that we dynamically add to the λProlog environment using logical implication. Their meaning is that to the variable `N` we associate a value `V` of type `T` and its normal form `NF`. The latter is initialised with a fresh metavariable when the binding for `N` is destroyed by a β-reduction, and it is filled in when the value is requested for the first time. To compute the term to be aligned to `NF` we first start a new machine on `V` and then we decode the final machine state, an operation that we call *unwind*. We call a variable `N` for which a `val N _ _ _` assumption is present *val-bound*.

Fetching the normal form of `x` from the environment via a call to `val x _ _ NF` amounts to fetching a clause from the current program, that is performed efficiently in any reasonable implementation of λProlog thanks to clause indexing. ELPI, as most Prolog engines, indexes clauses on the head predicate *and its first argument*: since `x` is a fresh constant there is little risk of having collisions. Even if the indexing ignores the predicate argument the cost of the lookup is at worst $O(n)$ where n is the size of the reduction machine environment. In line with what one would obtain by representing the environment as a linked list and variables with De Bruijn indexes.

An assumption A introduced in λProlog via $A \Rightarrow B$ is only visible in B. Therefore, once a variable is bound via an assumption A of the form `(val x t n nf)`, we need either 1) to be sure that the rest of the computations that requires `x` is performed in B; or 2) to reintroduce syntactically the binding again around every term that escapes the scope of the assumption.

We force 1) by coding the reduction machine in Continuation Passing Style (CPS): the `whd1` predicate takes in input the head and stack of the machine together with a continuation `K`, and apply `K` to the new machine head and stack. Moreover, it also passes to `K` the list of val-bound variables. The `whd*` predicate is also implemented in CPS style, and it is responsible for composing together the lists of val-bound variables.

Such list of variables is used when reduction is over to end the CPS style by performing 1). For this purpose we introduce the term constructor for local abbreviations `abbr` (also called `let .. in` in several functional programming languages). For example the `whd_unwind t NF` predicate computes the normal form `NF` of `t` by calling `whd*` on `t` and passing a continuation that unwinds the final machine state by introducing the explicit binder `abbr ty val n \ ..` for each val-bound variable `n`. The signature and type-checking rule for `abbr` follow:

```
type abbr term -> term -> (term -> term) -> term.  % local definition (let-in)

of (abbr TY TE F) (abbr TY TE G) :-
 of TE TY', sub TY' TY, pi x \ of x TY => val x TY TE NF => of (F x) (G x).
```

The code of the reduction machine is the following:

```
type whd1 term -> list term -> (list var -> term -> list term -> prop) -> prop.

% KAM-like rules in CPS style
whd1 (app M N) S K :- K [] M [N|S].
whd1 (lam T F1) [N|NS] K :- pi x \ val x T N NF => K [x] (F1 x) NS.
whd1 X S K :- val X _ N NF, if (var NF) (whd_unwind N NF), K [] NF S.

% reflexive, transitive closure
whd* T1 S1 K :- whd1 T1 S1
 (vl1 \ t1 \ s1 \ whd* t1 s1
  (vl2 \ t2 \ s2 \ sigma VL \ append vl1 vl2 VL, K VL t2 s2)), !.
whd* T1 S1 K :- K [] T1 S1.

% Whd followed by machine unwinding.
type whd_unwind term -> term -> prop.
whd_unwind N NF :-
 whd* N [] (l \ t \ s \ sigma TS \ unwind_stack s t TS, put_abbr l TS NF).

% unwind_stack takes an head and a stack and decodes them to a term
unwind_stack [] T T.
unwind_stack [X|XS] T O :- unwind_stack XS (app T X) O.

% put_abbr takes a list of variables and a term and wraps the latter
% with local definitions for the variables in the list
put_abbr [] NF NF.
put_abbr [X|XS] I (abbr T N K) :- val X T N _, put_abbr XS I (K X).
```

The predicate `match_sort` is implemented trivially. The one for `match_arr` is slightly more delicate: the input `T` is put in weak head normal form by `whd*`; the continuation retrieves the term and checks that it is a dependent product, projecting out the two subterms; the subterms may contain variables bound in the environment; therefore, before returning them, it is necessary to close them using technique 2) above by means of `abbr`s via `put_abbr`, like `whd_unwind` does.

```
type match_sort term -> @univ -> prop.
match_sort T I :- whd* T [] (l \ t \ s \ t = sort I, s = []).

type match_arr term -> term -> (term -> term) -> prop.
match_arr T A F :-
 whd* T [] (l \ t \ s \ sigma A' \sigma F' \
  s = [], t = arr A' F', put_abbr l A' A,
  pi x \ put_abbr l (F' x) (F x)).
```

Efficient reduction is not sufficient to obtain a quick term comparison routine. In particular both Matita and Coq implement the same heuristic: in place of putting the two terms to be compared in weak head normal form immediately, the two terms are lazily reduced on demand during comparison, that at every step tries α-conversion (i.e. λProlog equality) and congruence rules to avoid unnecessary reductions.

In order to perform weak head reductions lazily, we obtain `conv` and `sub` as two instances of a `comp` comparison predicate that works on two reduction machine statuses. The third argument is `eq` if the intended comparison is `conv` while `leq` for `sub`.

```
type conv term -> term -> prop.
type sub term -> term -> prop.
type comp term -> list term -> eq_or_leq ->  term -> list term -> prop.

conv T1 T2 :- comp T1 [] eq T2 [].
sub T1 T2 :- comp T1 [] leq T2 [].

% fast path + axiom rules
comp T1 S1 _ T1 S1 :- !.
comp (sort I) [] leq (sort J) [] :- lt I J.
% congruence + fast path rule
comp X S1 _ X S2 :- map S1 S2 conv, !.
% congruence rules
comp (lam T1 F1) [] _ (lam T2 F2) [] :- conv T1 T2, pi x \ conv (F1 x) (F2 x).
comp (arr T1 F1) [] D (arr T2 F2) [] :- conv T1 T2, pi x \ comp (F1 x) [] D (F2 x) [].
% reduction rules
comp T1 S1 D T2 S2 :- whd1 T1 S1 (_ \ t1 \ s1 \ comp t1 s1 D T2 S2), !.
comp T1 S1 D T2 S2 :- whd1 T2 S2 (_ \ t2 \ s2 \ comp T1 S1 D t2 s2), !.
```

Extensions with global definitions, declarations, primitive inductive types, case analysis and structural recursive definitions. We implemented all the extensions mentioned in a modular way. To activate one extension, it is sufficient to accumulate in the kernel a λProlog file that adds new constructors to the `term` and to the reduction machine stack type, and the relative clauses to the `whd1`, `conv` and `of` predicates.

We omit the implementation of the extensions in the paper. All together, they provide a kernel that is functionally equivalent to the one of Matita. The reduction strategies implemented differ though, the one of Matita being more complex than call-by-need. We plan in the future to synchronise the two strategies in order to compare the performance of the two kernels.

To test the kernel, we branched it to Matita, feeding it with every term that is type-checked by Matita when checking the arithmetic library of the system. According to our measurements the kernel presented in the paper is 15 times slower than the interpreted version of Matita (to be fair versus ELPI that is an interpreter). The trade off is that the code is more elegant, much simpler and shorter (e.g. 91 lines saved just from the lifting of De Bruijn indexes). A previous implementation of a type-checker for Automath was only 5 times slower [13], suggesting the possibility to optimize the code further.

3 The elaborator component of today's ITPs

An elaborator takes in input a *partial term* and optionally a type, and returns the closest term similar to the input one such that the term has the expected type. Both the input and output terms are partial in the sense that subterms can be omitted and replaced with named holes to be later filled in or, in logic programming terminology, with *existentially quantified metavariables*. For example, the partial term $\lambda x : T.f\ (X\ x)\ Y$ where T, X, Y are quantified outside the term represents the λ-abstraction over x of a type yet to be determined of the application of f to two arguments, both to be yet determined and such that x can appear free only in the first. Elaborating the term versus the expected type $\mathbb{N} \rightarrow \mathbb{B}$ will instantiate T with $\mathbb{N}$ and verify that f is a binary function returning a boolean.

The importance of the elaborator is twofold. On the first hand, it allows to interpret the terms that are input by the user, usually by means of a user-friendly syntax where information can be omitted, mathematical notation is used and abused, subtyping is assumed even if elements of the first type can only be coerced to elements of the second by inserting a function call in the elaborated term. A better elaborator therefore gives to the user the feeling of a more intelligent and user friendly system. On the other hand, via Curry-Howard, a partial term is a partial proof and an ITP is all about instantiating holes in partial proofs with new partial terms to advance in the proof. The elaborator is thus the mechanism that takes a partial sub-proof and makes it fit in the global one to progress on a particular proof obligation. In other words, all tactics of the ITP ultimately produce partial proof terms that are elaborated. The more advanced is the elaborator, the simpler the code implementing tactics can be.

Implementing an elaborator: state of the art. The elaborators of the majority of the provers mentioned above are all implemented according to the same schema: the syntax of terms is augmented with explicitly substituted existential variables and the judgements of the kernel are re-implemented from scratch, generalising them to take into account metavariables and elaboration.

In particular the elaborator code works on two new data types: one for partial terms and one, called *metasenv*, that assigns to metavariables a typing judgement (a sequent) and, eventually, an assignment. E.g. a metasenv containing `x:nat, y:bool` $\vdash$ `X x y : nat` declares `X` to be an hole to be instantiated only with terms of type `nat` in the context `x:nat, y:bool`.

All algorithms manipulating terms are extended to partial terms. For example, conversion (the `sub` predicates in the running example) becomes narrowing, i.e. higher order unification under a mixed prefix [25] in presence of rewriting rules (and cumulativity of universes as subtyping). Unification requires metavariable instantiation, that is implemented lazily for efficiency. Type checking `of` is generalised to elaboration, for example by replacing all calls to conversion with calls to narrowing and by threading around the metasenv.

The state-of-the-art approach is sub-optimal in many ways:

programming platform weakness Much of this new code has very little to do with the prover or the implemented logic: in particular code that deals with binders (α-conversion, capture avoiding substitution, renaming) and code that deals with existentially quantified metavariables (explicit substitution management, name capturing instantiation).

intricacy of algorithms Such code is far from being trivial, since it tackles problems that, like higher order unification, are only semi-decidable. For efficiency reasons a lot of incomplete heuristics are implemented to speed up the system and reduce backtracking. The heuristics are quite ad-hoc and they interact with one another in unpredictable ways. Because they are hidden in the code, the whole system becomes unpredictable to the user.

code duplication Given such complexity in the elaborator, and the safety requirements of interactive provers, the kernel of the system is kept simple by making it unaware of partial terms. As a consequence a lot of code is duplicated, and the elaborator ends up being a very complicated *twin brother of the kernel* (Huet's terminology).

twins' disagreement Worse than that, such twin components need to agree on ground terms. Typically a proof term is incrementally built by the elaborator: starting from a metavariable that has the type of the conjecture the proof commands make progress by instantiating it with partial terms. Once there are no unresolved metavariables left, the ground term is checked, again and in its totality, by the kernel.

extensibility of the elaborator Finally, the elaborator is the brain of the system, but it is oblivious of the pragmatic ways to use the knowledge in the prover library, e.g. to automatically fill in gaps [18, 3], to coerce data from one type to another [22] or to enrich data to resolve mathematical abuse of notation [9]. Therefore systems provide ad-hoc extension points to increase the capabilities of the elaborator. The languages to write this code are typically high-level, declarative, and try to hide the intricacies of bound variables, metavariables, etc. to the user. The global algorithm is therefore split in multiple languages, defying the hope for static analysis and documentation of the elaborator.

The proposed approach: the semi-shallow embedding. The motivation of our research is to improve over the latter issues by identifying an high-level (logic) programming language suitable for the implementation of elaborators. In particular:

1. The programming language takes care of the representation of *bindings and metavariables* in the spirit of semi-shallow embedding [12], solving the **programming platform weakness** issue. It also improves on **extensibility** by allowing both the core implementors and the users to work on the same, high-level code. Finally the **intricacy of elaboration** is mitigated.

2. The programming language features a primitive and powerful notion of extensibility: programs are organised into clauses, and new clauses can be freely added. In this way the

rules of the kernel need not to be re-implemented in the elaborator. On the contrary, they are extended to cover partial terms, solving the **code duplication** issue. Also, the **twins' disagreement** problem becomes less severe, since most code is shared, and **extensibility of the elaborator** become less ad-hoc: the user simply declares new clauses.

3. Finally the programming language has a clean semantics, making it easy to prove that the extensions to the kernel only accept partial terms that are, once completed, well-typed according to the core set of rules of the kernel. This completely solves the **twins' disagreement** problem, making it possible to merge the kernel and the elaborator.

We envisage such language to be a logic one for two reasons: first we hope to reuse or at least extend λProlog; second we observe that another component of each ITP, the one implementing proof commands, can take real advantage from a programming language where backtracking is built-in, in particular to write proof commands performing proof search.

The Higher Order Abstract Syntax approach identifies the object language binders and the meta-language ones, obtaining α-conversion and β-reduction for free. The semi-shallow embedding [12] we expect to use in our language *identifies the metavariables of the object language with the metavariables of λProlog*. Such metavariables already come in λProlog with automatic instantiation, context management and forms of higher order unification.

At a first sight, the runtime of λProlog seems to already provide metavariables and all related operations required for the semi-shallow embedding. So does the technique work out of the box? Unfortunately not quite, as we will see in the next section.

4 λProlog meets partial terms

We already know from [5] that λProlog is the minimal extension of Prolog that allows to implement inductive predicates over syntax containing binders. Does it work when applied to data that is meant to contain existentially quantified metavariables too?

From generative semantics to constraints. Consider the λ-term `(lam a\ P a)` that encodes a partial proof of $\forall A, A \Rightarrow A$. If we run the following query, the computation diverges:

```
?- of (lam T a\ P a) (arr (sort I) a\ arr a _\ a)
```

The rule for λ-abstraction applies fine, but generates the problematic goal `of P (arr x _\ x)` for some fresh `x`. Then, the type checking rule for application, which head is `of (app M N) BN`, applies indefinitely.

The `of` predicate inherits from Prolog a generative semantics: when called recursively on a flexible input, it enumerates all instances trying to find the ones that are well-typed. Even when the computation does not diverge, the behaviour obtained is not the one expected from an elaborator for an *interactive* prover: the elaborator is not meant to fill in the term, unless the choice is obliged. On the contrary, it should leave the metavariable not instantiated and should *remember* (in some kind of metasenv) the need for verifying if the typing judgement holds later on, when the metavariable gets instantiated. In the example above, type-checking `(lam T a\ P a)` forces the system to remember that term of type `(arr x _\ x)` has to be provided (in a context where `of x (sort I)` holds), that in turn corresponds to the proof obligation $A : \mathrm{sort}\ \mathrm{I} \vdash A \Rightarrow A$.

As we mentioned earlier the sometimes undesired generative semantics is inherited from Prolog. Nevertheless, all modern Prolog engines provide a `var/1` built-in to test/guard predicates against flexible input, provide one/many variants of `delay/2` to suspend a goal until the "input" becomes rigid, and provide modes declarations to both statically/dynamically detect problematic goals and to (semi) automatically suspend them. For example, by using the `delay` pack of SWI-Prolog [31], the goal `plus(X,1,Y)` is delayed until either `X` or `Y` are instantiated. Delayed goals can be thought as *constraints* over the metavariables occurring in them. In the example above, the programmer is imposing a constraint between `X` and `Y`.

These mechanisms, however, have never been standardised. Some of them break the clean declarative semantics of λProlog, others respect it. λProlog does not provide any of these facilities, or at least, does not expose them to the programmer. For example, the Teyjus system delays unification problems outside L_λ and implements a complex machinery to wake up delayed goals that re-enter the fragment, but does not expose any primitive to delay other kind of goals.

In the light of all these considerations we extend the ELPI λProlog interpreter with a `delay` directive that we can use as follows:

```
delay (of X T) on X.
```

The interpreter now delays goals of the form `of X T` whenever `X` is flexible. Instead of diverging, the running example now terminates leaving the following (suspended) goal unsolved:

```
of x (sort I) ?- of (P x) (arr x _\ x)
```

Such suspended goal is to be seen as a typing *constraint* on assignments to `P`: when `P` gets instantiated by a term `t`, the goal `of (t x) (arr x _\ x)` is resumed and `(t x)` is checked to be of type `(arr x _\ x)`. In turn such check can either:

terminate successfully if `t` has the right type and is ground, e.g. `t = x\lam x w\ w`. If such assignment for `P` comes from a proof command, then it corresponds to a proof step that closes the goal.

fail rejecting the proposed assignment for `P`, e.g. `t = x\lam x w\ x`. This corresponds to an erroneous proof step.

advance and generate one or more new constraints if `t` is partial e.g. `t = x\lam x w\Q x w`. If `t` is generated by a proof command, then this corresponds to a proof step that opens one or more new goals.

The three scenarios above perfectly match the desired behavior of an elaborator. In addition to that, the set of delayed goals implicitly kept by the language interpreter represents faithfully the metasenv data structure, relieving the programmer to manage it himself, threading it through all typing rules and managing explicitly backtracking and goal resumption. Moreover, the automatic resumption of typing constraints makes it impossible to obtain an ill-typed term by instantiation: the code is correct by construction. Lastly, it lets one selectively disable the undesired generative semantics of predicates like `of`, `lt`, ...

Constraint propagation as meta-theorems on ground terms. In many situations, the constraints that accumulate over time are not independent, and sometimes sets of constraints can be rewritten in order to simplify them, or detect their unsatisfiability and backtrack. For example, if our metavariable `P` does not occur linearly, one can end up with two distinct typing constraints on it:

```
of x (sort I) ?- of (P x) (arr x _\ x)
of z (sort I) ?- of (P z) (arr (T z) y\ S z y)
```

If the object language features uniqueness of typing, as it is the case for CIC, one surely wants to get rid of the second constraint, and force `T = a\ a` and `S = a\ b\ a`. This way, when

`P` gets instantiated *only one goal* is resumed (to type check the assigned term). Alternatively, if the two typing constraints clash, one wants the elaborator to backtrack immediately.

Languages (or libraries for existing languages) that allow constraints declaration usually come with ad-hoc *constraint solvers* that take care of propagating a specific class of constraints (arithmetic, finite domain, etc. . .), and are then called *Constraint Programming* (CP) languages. For example, the set $\{0 \leq N \leq 4, 2 \leq N \leq 5\}$ can be rewritten into $\{2 \leq N \leq 4\}$ without changing the set of ground solutions.

Most CPs do not allow the user to define new constraints and propagation rules in user space: only constraints belonging to well-known categories can effectively be used. In our case the constraints originate from the meta-theory of the object language we are implementing, hence the programmer must be able to declare new kind of constraints and specify how constraints are to rewritten according to meta-theorems about the object language (e.g. uniqueness of typing).

Languages to describe sets of rules to manipulate a set of constraints have been studied in the literature, and the most well-known one is certainly CHR (Constraint Handling Rules [14]). CHR is a first-order language in which the user declares predicates and then gives a set of rewriting rules of the form $S_1 \setminus S_2 \mid G \iff S_3$ whose declarative semantics is that $\bigwedge S_1 \wedge G \Rightarrow (\bigwedge S_2 \iff \bigwedge S_3)$ and whose operational semantics is to match the current set of constraints against both S_1 and S_2 and, if the clause G holds, replace the constraints in S_2 with the ones in S_3. All syntactic components can be omitted: G defaults to `true` and the sets S_1, S_2, S_3 to the empty set. The $\iff$ symbol is also omitted when S_3 is, and $\mid$ is omitted when G is. The semantics of the language is completed by a strategy that fixes the order in which propagation rules are fired and in what cases propagation rules can be backtracked.

Adding to a λProlog with a `delay` construct (to declare constraints) a CHR-like language to declare constraint propagation rules enables the following applications:

forward reasoning during search When propagation rules are theorems over the ground instances, then propagation respects the declarative, proof theoretic semantics of λProlog. In particular, constraint propagation is just a controlled form of forward reasoning, while SLD resolution only does backward reasoning.

code optimisations given by the meta-theory of the object language. In the running example, uniqueness of typing is key to keep the implicit metasenv linear in the number of missing sub-proofs. In turn, this greatly reduces the number of checks to be performed when a metavariable is instantiated, and it allows to directly present to the user the set of constraints as the frontier of the proof search.

turn checking into inference By replacing the predicates `<`, `max`, `succ` with generic order constraints `leq`, `ltn` one gets an elaborator that works with floating sorts [19], rather than fixed integers, and maintains an acyclic graph of constraints linking these. More generically, we can turn type checking into type inference and elaboration.

In the light of all these considerations we extend the ELPI λProlog interpreter with a higher order constraint propagation language inspired by CHR.

The following propagation rule is valid code, and implements uniqueness of typing:

```
constraint of {
  rule (G1 ?- of X T1) \ (G2 ?- of Y T2) > X ~ Y
       | (equiv G1 G2) <=> (G1 => conv T1 T2).
}
```

Here `G1 ?- of X T1` is a constraint (delayed goal) required to be present, `G2 ?- of Y T2` is another one we want to remove. The type of `G1` and `G2` is `list o`, i.e. it is a first class representation of the λProlog context (that augments the original program). The guard `equiv G1 G2` is a user defined λProlog program checking that the two contexts are equivalent (i.e. contain the same items, disregarding order and multiplicity). `G1 => conv T1 T2` (syntactic sugar for `H1 => ... => HN => conv T1 T2` where `G1 = [H1, ..., HN]`) is the new goal generated by the rule. The alignment expressions, `> X ~ Y`, asserts `X` and `Y` are the same metavariable and finds a bijection between the eigenvariables (introduced by `pi` rules) in the two goals.

We say that CHR rules run at the *meta-level*, i.e. they have access to the syntax of goals and to their context. In particular, all flexible terms in a constraint are replaced by fresh constants before being inspected by a CHR rule (we call this operation freezing). Therefore, the propagation rules can inspect the goal, compare frozen metavariables, detect frozen flexible terms etc. without triggering the instantiation of the metavariable. As a comparison, detecting if a term is flexible cannot be done reliably in λProlog. An interpreter for the language can take the meta-level metaphor literally: we implemented constraint propagation by starting a new interpreter that takes in input the guard of the propagation rule and a reflection of the syntax of the goals of the interpreter below. In principle, we could even delay goals in the meta-level and propagate them using a meta-meta-level.

The bijection described by the alignment expression is key to manipulate in the same guard (or combine in the new goal) terms living in different contexts.

As an implementation detail, heads are *matched* against the constraints by the interpreter, and only when matching and alignment succeed, freezing takes place, the meta-interpreter is started and the guard is run.

When the meta-interpreter halts successfully on the guard, the new goal is defrost and it is immediately scheduled for execution in the interpreter. Defrosting restores the metavariables originally present in the constraints, and it also creates a new metavariable (in the interpreter) for each new metavariable (in the meta-interpreter) that was generated by executing the guard.

For example, if we take the two typing constraints on `P` above and we freeze them, we obtain:

```
of x (sort c0) ?- of (c1 x) (arr x _\ x)
of z (sort c0) ?- of (c1 z) (arr (c2 z) y\ c3 z y)
```

Note that equal metavariables are frozen with the same fresh constant. Eigenvariables are left untouched by freezing, it is the work of the alignment phase to make sense of them. In this simple case there is only one possible bijection between the two (singleton) sets of eigenvariables: $x \mapsto z$.

The following query is run in the meta-interpreter to validate the guard of the rule.

```
?- equiv [of z (sort c0)] [of z (sort c0)]
```

Finally the following goal is generated and scheduled for the next SLD resolution step.

```
of z (sort I) => conv (arr z _\ z) (arr (T z) y\ S z y)
```

The new goal eventually assigns `T` and `S`, and its success or failure is semantically equivalent to the satisfiability of the second constraint on `P`, that is removed.

Automatic alignment of goals. Matching of λProlog goals is a delicate matter: eigenvariables are fresh names, not fixed constants, and their semantics is quite different [16]. In particular, matching goal H against goal G amounts to solving this equation written using the nabla quantifier of [16]: $\nabla x_1 \ldots x_n, \mathcal{H} = \nabla y_1 \ldots y_m, \mathcal{G}$.

In the general case it is closely related to equivariant unification that is known to be a NP hard [8] problem. While being an elegant high-level language, CHR is not efficient. It is estimated be on average 100 times slower than conventional programming languages. If, by handling higher order constraints, we make its core operation, matching, even more expensive, we are unlikely to obtain a working system.

Providentially the use case that drove our design seems to be on a lucky spot: all relevant names are visible by the key of the delayed goal. Indeed, in the Curry-Howard homomorphism, a metavariable represents a sequent, and its proof (and type) can only use variables in scope of the metavariable. The design choice we make is to forbid the ∇ quantifier in the patterns of CHR rules, and implicitly consider as many ∇ quantifications as eigenvariables visible to the key variable of the constraint being matched by the CHR head. This way we can impose the trivial bijection between the names in the pattern and the names in the constraint. Also, all patterns use distinct metavariables, so all injections are trivial. In other words, names are only dealt with in the alignment. We support two form of alignments: one "automatic" based on the L_λ invariant and one "manual" for programs that go outside that fragment.

In L_λ the variables visible by each constraint key are distinct, and equating the keys gives a bijection between the names. If a CHR rue has n patterns, and each corresponding goal has m eigenvariables, then the alignment is computed in $O(m^n)$

The manual alignment injects all eigenvariables to the disjoint union of the eigenvariables visible by each constraint key. In other words the terms bound by the CHR rule's heads share no names: it is up to the programmer to eventually relate the names. In this case we also let the programmer access the list of terms to which the metavariable is applied with the following syntax: `(?? K L as X)`. Here `K` is the (frozen) metavariable, `L` is the list of terms to which it is applied and `X` is just `K` applied to `L`. Because we are at the meta-level, this kind of inspection of the syntax is again perfectly reasonable and allowed. We put this feature to good use in the next section.

Meta level: elegant but overkill. The operational semantics of the rule $S_1 \setminus S_2 \mid G \iff S_3$ replaces S_2 with S_3. Therefore its soundness only requires $\bigwedge S_1 \wedge G \Rightarrow (\bigwedge S_2 \Leftarrow \bigwedge S_3)$. When it is not the case that also $\bigwedge S_1 \wedge G \Rightarrow (\bigwedge S_2 \Rightarrow \bigwedge S_3)$, the rewriting rule is not complete or, equivalently, the user is only specifying and heuristic that potentially throws away solutions.

Heuristics are something the user surely wants to do when extending the elaborator. Our running example is the following emblematic conversion (or, better, narrowing) problem

```
conv (app carrier_of R) integers
```

In CIC a term can pack together types, terms and properties. In the example above `R` of type `group` is to be instantiated with a record that packs together the group carrier, the group operations and their properties. In this setting one can clearly see that the generative mode of λProlog corresponds to proof search, i.e. the blind enumeration of λ-terms to build (uninteresting) groups over the integers. Of course the user is likely to have already built, in his/her library, the standard ring of integers, and he would like to instruct the system with the heuristics that pick for `R` that group. The code to do so is the first propagation rule below:

```
delay (conv (app carrier_of X) Y) on X.
constraint conv {
  rule \ (conv (carrier_of X) integers) <=> (X = integer_group).
  rule \ (conv (carrier_of X) (intersection A B)) <=>
    (conv (carrier_of GA) A, conv (carrier_of GB) B,
     X = intersection_group GA GB).
}
```

The second rule is recursive and re-phrases the theorem saying that the intersection of two groups is a group: in order to find a group `X` whose carrier is the intersection of the two sets `A` and `B`, the heuristic suggests to recursively discover two groups `GA` and `GB` of carrier `A` and `B`, and to instantiate `X` with the group intersection of `GA` and `GB`.

Remark how little we use of CHR in this rule: only one head, no alignment, no guard (i.e. no need to freeze). Still, these simple rules faithfully model the extension to the elaborator of Coq that proved to be key to outstanding formalisations like the Odd Order Theorem [18]. Such formalisation declares a 26 hundreds of rules like these ones. It goes without saying that the efficiency of such rules is critical.

To achieve efficiency, in this simple scenario we would like to avoid delaying the goal, matching it with a propagation rule, execute the guard at the meta-level, etc. To do so, we will expose in the language the low level primitives that are used in the interpreter to implement constraint generation and propagation. The idea is that, in the restricted scenario above, the user will be able to directly write the propagation rule in the interpreter.

The first primitive is called `mode` and lets one tag arguments of a predicate as input ones.

```
mode (of i o).
```

The result is that arguments marked as input are *matched* against the corresponding arguments in the goal, exactly as the CHR engine matches the heads of a rule against constraints (without instantiating any metavariable occurring in the constraints). This let us write programs like the following one without risking that the first two rules "generate" the input. Of course the right hand side of the clauses can instantiate metavariables.

```
of (lam T F) T :- ...
of (app M N) T :- ...
of X T :- var X, $delay (of X T) [X].
```

The last rule is the actual implementation of the global directive `delay .. on ..` we described before. Note that this way the condition to identify goals to delay can be made arbitrarily sophisticated. Also remark that the mode declaration for `of` is not equivalent to place a clause like `(of X T :- var A, !, ...)` on top, since hypothetical clauses may be placed by the runtime before this one, and they can be generative as well.

The syntax `(?? K L as X)` seen in 4 is also available, and lets one access the arguments of a flexible term. This lets one extend the conversion predicate (unification in the elaborator) to cover terms outside L_λ. An emblematic example is the following one:

```
rule (G ?- conv (?? F [nat_of_ord x]) T)
  | ((pi y\ copy (nat_of_ord x) y => copy T (T' y)),
     (pi w\ copy (T' x) (T' w))) <=> (F = T')
```

where `x` never occurs in `T` alone, only inside the `nat_of_ord` context: `T'` is `T` where all occurrences of `nat_of_ord x` are bound. Such example occurs very frequently in the library of big operators of Coq [6], where theorems about iterated operations over finite domains are provided. For example the $\sum_{i<7}(F\ i)$ expression hides the `nat_of_ord` injection around i (a term of the type of I_7, a finite subset of the integers of cardinality 7). The impossibility to extend the elaborator of Coq with any heuristic to solve the goal above makes the use of these lemmas painful, since one may have to provide `F` by hand, instead of having it automatically inferred.

In Section 5 we use the low level primitives `mode`, `delay` and `var` to implement our elaborator. In some cases, we combine `mode` and `delay` just to implement the high-level `delay` construct. In some other cases we use `mode` and pattern matching over the syntax of goals to implement heuristics without triggering delay.

In all cases, the low level primitives above can be understood in term of the high-level language provided before. In particular, semantically `mode` already acts as the request of delaying the goal when certain inputs are flexible. Then, in the goals that match metavariables, one can either confirm the intention of delaying (via `$delay`), or he can provide a propagation rule by immediately issuing a new query to be executed by the interpreter. In the future we plan to study the elaboration of the syntax based on the low level primitives to the high-level syntax and its clean declarative semantics.

ELPI = λProlog + CHR. We implemented the language described in the previous sections in an efficient interpreter, written in OCaml, that we called ELPI (Embedded Lambda Prolog Interpreter) and that is open source and downloadable from the Web. In addition to the new programming constructs that deal with constraints, ELPI slightly differs from the version of λProlog implemented in Teyjus in a few minor aspects.

First, in ELPI all types and type and sort declarations can be omitted, whereas they are mandatory in Teyjus. Originally, and according to [5], types were necessary to implement Huet's algorithm for higher order unification. However, for several years now the unification algorithm of Teyjus only solves equations in the pattern fragment L_λ discovered by Miller [24], which admits most general unifiers and reduce unification to a decidable problem. All other equations are automatically delayed by Teyjus to be fired only when the equation is instantiated to one in the pattern fragment. At the level of the implementation, the code of ELPI that implements this delay is shared with the one to delay arbitrary predicates.

Second, the module system of Teyjus is not implemented. Only the `accumulate` directive is honored for backward compatibility and with a different semantics. In ELPI we provide instead explicit existentially quantified local constants whose scope spans over a set of program clauses. This mechanism gives in a simple way predicate and constructor hiding that are provided differently by the module system of Teyjus.

Last, in a few corner cases, the parsing of an expression by Teyjus is influenced by the types. In particular, types are used to disambiguate between lists of elements and a singleton list of conjunctions. The syntax is disambiguated in a different way in ELPI.

Despite the differences above, we tried very hard to maintain backward compatibility with Teyjus and its standard library. Indeed, we are able to execute in ELPI all the code from Teyjus that we collected from the Web, up to a very few minor changes to the source code. Last, ELPI is a pure interpreter written in OCaml. Embedding it into larger applications like Coq or Matita is easy (no external program to run, no compilation chain).

5 Towards an elaborator for CIC

We are developing an elaborator for CIC as a modular extension of the kernel described in Sect. 2. The elaborator mimics as close as possible the behaviour of the one of Matita 0.99.1 [4], with the exception of the handling of universe constraints that follows Coq [19]. For the time being, however, the elaborator can only instantiate metavariables that appear in the term, lacking the possibility of entirely modifying the input term, for example introducing type casts. To overcome this limitation, it is no longer possible to reuse the kernel type-checking rules as they are now: the type-checking judgement itself must give in output a new term, that is the elaboration of the term in input. All the rules of the kernel would just copy the input term in output verbatim, leaving to additional rules in the elaborator the introduction of type casts.

In place of following the algorithm of Matita and Coq, an alternative very promising choice would have been to mimic the elaborator described in [23] and already presented via typing rules that yield a set of higher order unification constraints to be solved later. In the future, the choice to be closer to Matita will allow us to easily compare the performances of the elaborator written in ELPI with the one of Matita written in OCaml, in order to further optimise the ELPI interpreter.

To implement the elaborator, we need to consider all the predicates defined by induction over the shape of terms, and either turn them into constraints to be propagated, or immediately suggest solutions.

Type-checking: the `of` predicate. Using a mode declaration plus `$delay`, we delay type-checking a metavariable, turning it into a proof obligation, i.e. a sequent of the form `G ?- of X t` where `G` holds type declarations for variables (`of x t`) or value assignments (`val x t v nf`).

```
mode (of i o).
of (?? as K) T :- !, $delay (of K T) K.
```

We then declare one constraint propagation rule that corresponds to a special case of uniqueness of typing in the case of dependently typed languages: one of the two obligations is *canonical*, i.e. it is of the form `G ?- of (X x1 .. xn) t` where all the `xi` are distinct variables or, equivalently, when `X x1 .. xn` is in the pattern fragment L_λ discovered by Miller.

The restriction may seems severe, but, once a canonical typing constraint enters the set of delayed goals, it is simple to reduce the whole set to just the canonical one plus additional conversion constraints. Moreover, the first time a metavariable is generated in an interactive prover like Matita its typing constraint is canonical by construction. Therefore, it is customary both in theory [17] and implementation [4] to always keep only canonical constraints, that are collected in a set called *metavariable environment* (metasenv) and that are the only proof obligations presented to the user.

The propagation works as follow: let `G1 ?- of (X x1 .. xn) u1` be the canonical sequent and `G2 ?- of (X t1 .. tn) u2` be the one to be simplified. First we compute the type of `x1` in `G1` and of `t1` in `G2`, imposing as a new constraint their convertibility in the union of `G1` and `G2`. Then we proceed recursively over `x2 .. xn` and `t2 .. tn` after assigning `t1` to `x1` via `val` in `G1`. When the two lists become empty, we generate a final constraint to check if `u1` is convertible with `u2`.

To detect canonicity, we use the ad-hoc extension predicate `name` of ELPI that holds iff the argument is a universal variable. Recall tat the syntax `G => L` when `G` is a list of propositions is equivalent in ELPI to assuming the conjunction of the predicates in `G`.

```
constraint of val {
  rule (G1 ?- of (?? X1 L1) T1) \ (G2 ?- of (?? X2 L2) T2) > X1 ~ X2
    | (is_canonical L1, compat G1 L1 T1 G2 L2 T2 L3) <=> L3.
}

is_canonical [].
is_canonical [X|XS] :- name X, not (mem X XS), is_canonical XS.

% (G2 ?- T2) is the canonical one
compat G2 [] T2 G1 [] T1 H :- append G1 G2 G12, H = (G12 => conv T2 T1).

compat G2 [X2|XS2] T2 G1 [X1|XS1] T1 K :-
 H1 = (G1 => of X1 U1),
 H2 = (G2 => of X2 U2),
 append G1 G2 G12, H3 = (G12 => conv U1 U2),
 compat G1 XS1 T1 [val X2 U2 X1 _NF|G2] XS2 T2 K2,
 K = (H1, H2, H3, K2).
```

Last, remark that the alignment mode selected only checks that `X1` and `X2` are the same metavariable and makes all eigenvariables distinct.

Universe constraints: the `lt`, `succ`, `max` predicates. All three predicates must be delayed when at least one of the arguments is flexible. However, satisfiability of the constraints is a necessary requirement for logical consistency. In case of violation of the constraints, it is indeed easy to encode a form of Russel's paradox.

In order to verify satisfiability, we could devise a set of propagation rules for constraints over integers induced by the last three predicates. However, to preserve the logical consistency of the system, it is not necessary to keep the total, discrete order of integers. On the contrary, it is more flexible for the user to assume a generic partially ordered set `(univt,lteq)`, and to relax the successor relation to being strictly greater and the max to a generic upper bound. Satisfiability of the set is now equivalent to the absence of a strict cycle, i.e. to the possibility to derive `ltn U U` for some universe `U`.

Detecting an inconsistency from a set of constraints expressed using strict and lax inequalities is such an easy job for CHR that we basically just had to modify the "hello world" example for CHR given on Wikipedia, that simplifies constraints over lax inequalities only. Each constraint propagation rule corresponds to an instance of a characterizing property of the order, i.e. reflexivity, transivity and antisimmetry of `leq`, transitivity and antiriflexivity of `ltn`, and finally inconsistency of `ltn X Y` with both `ltn Y X` and `leq Y X`. We only show in the following code a few propagation rules.

```
kind univt type.
macro @univ :- univt.

lt    I J :- ltn I J.
succ  I J :- ltn I J.           % succ relaxed to <
max N M X :- leq N X, leq M X.  % max relaxed to any upper bound

... /* boilerplate code to delay leq and ltn over flexible terms */

constraint leq ltn {
  % incompatibility and irreflexivity
  rule (leq X Y) (ltn Y X) <=> false.  rule (ltn X X) <=> false.
  % reflexivity, antisymmetry, ...
  rule \ (leq X X). rule (leq X Y) \ (leq Y X) <=> (Y = X). ...
}
```

Reduction: the `whd1`, `whd*` predicates. The predicate `whd*`, defined in terms of `whd1`, computes the normal form of the input. It is used by `match_sort` and `match_arr` to verify if the normal form has a given shape. Moreover, it calls itself recursively to compute the normal form of arguments according to the call-by-need strategy.

The typing system we are implementing cannot distinguish between a term and its normal form. Therefore, when computing the normal form of a flexible input `X`, it is meaningless to delay a goal stating that `Y` is the normal form of `X`, because typing does not distinguish them. Therefore, we just return `X` as the normal form of `X` by letting `whd1` fail over flexible terms.

```
mode (whd1 i i o).
whd1 ?? _ _ :- fail.
```

The consequence on the strategy is that, under certain alternations of reductions and instantiations of metavariables, the implemented strategy will be intermediate between call-by-name and call-by-need, recomputing the normal form of arguments that were metavariables at the time of their first use, with no consequences on typability.

On `match_sort` and `match_arr` there is no consequence: the metavariable is immediately assigned the wanted shape, meaning that we have decided to instantiate the metavariable immediately with its normal form.

Term comparison: the `sub`, `conv` predicates. Conversion when at least one of the arguments is a flexible term amounts to higher order narrowing. Instead of delaying the goal, an heuristic already used in Matita immediately rewrites the constraint by solving the unification problem. The heuristic always favours projections to mimic (in Huet's terminology). Further heuristics are used in Matita to avoid projecting over flexible arguments, to make unification more predictable to the user. These heuristics are clearly incomplete, discarding all solutions but one and sometimes failing to find a solution when it exists. However, they are more general than the ones implemented in Coq and, from a practical perspective, they guess most of the time the unifier that is expected by the user when interacting with the ITP.

The following code shows the clauses that deal with a flexible term (on the left-hand side) to be unified with a rigid one on the right-hand side. A simmetric set of rules is required for the dual case, and the code can obviously be unified with minor effort. Additional rules, omitted here, are required to deal with the flexible-flexible case.

```
mode (comp i i i i i).

% T1 :- lam TYA F  + beta step
comp (?? as T1) [A|AS] M T2 L2 :-
 of A TYA, T1 = lam TYA F, pi x \ val x TYA A _NF => comp (F x) AS M T2 L2.

% PROJECTION
comp (?? as V1) [] M T2 S2 :- val X _ _ _, V1 = X, comp V1 [] M T2 S2, !.

% MIMIC
% non empty stack = application
comp (?? as V1) [] _ T2 S2 :-
 append S21 [S2l] S2,
 V1 = app X Y, comp X [] eq T2 S21, comp Y [] eq S2l [].
% regular mimic rules
comp (?? as V1) [] M T2 [] :- mcomp V1 M T2. % mcomp used generatively
% variables and constants
comp (?? as V1) [] M T2 [] :- V1 = T2.

% REDUCTION (same rule as the kernel)
comp (?? as V1) [] D T2 S2 :- whd1 T2 S2 (_ \ t2 \ s2 \ comp V1 [] D t2 s2).

% FAIL
comp ?? [] _ _ _ :- !, fail. % to avoid the fast path of the kernel

% MIMIC RULES (same rules as the kernel)
mcomp (sort I) eq (sort I) :- !.
mcomp (sort J) leq (sort I) :- !, leq J I.
mcomp (app A1 B1) _ (app A2 B2) :- !, comp A1 [] eq A2 [], comp B1 [] eq B2 [].
mcomp (lam TY1 F1) _ (lam TY2 F2) :-
 !, comp TY1 [] eq TY2 [], (pi x \ comp (F1 x) [] eq (F2 x) []).
mcomp (arr TY1 F1) M (arr TY2 F2) :-
 !, comp TY1 [] eq TY2 [], (pi x \ comp (F1 x) [] M (F2 x) []).
```

The code assumes all metavariable occurrences to be in L_λ. Otherwise, calls like `V1 = app X Y` in the code may be automatically delayed by ELPI, thus breaking the algorithm.

Several tests already pass, but we are currently still developing the full set of rules that reflect the algorithm developed by Matita. Once completed and made functional equivalent, we plan to test the elaborator written in λProlog against the original one for debugging and performance analysis.

6 Conclusions and related works

In this paper we validate λProlog as a good language to implement the type checker (kernel) of a fully fledged type theory like the Calculus of Inductive Constructions. In order to implement an elaborator, i.e. a type checker for partial terms, we extend the λProlog with constructs to turn goals into constraints and we add a CHR like language to manipulate the set of constraints. We implement the proposed extensions in the ELPI system and use them to extend, in a fully modular way, the kernel into an elaborator.

To our knowledge ELPI is the first λProlog implementation extended with first class constraints. The situation is very different for Prolog, where all mainstream implementations integrate some constraint solvers. Moreover, the CHR language is typically compiled to Prolog, hence Prolog systems can quite easily provide a CHR package. It is for example the case for SWI Prolog [31] and SICStus Prolog [2]. Providing to the Prolog language primitives of the meta-level, like matching, has also been tried before. For example the Picat language, based on the B-Prolog [32] engine, lets the user chose, for each clause, if the head has to be matched or unified with the goal.

While basing the implementation of the kernel of an interactive prover on a logical framework (LF) eventually animated by SLD resolution is not new, little has been done to reuse the same technology for the elaborator component. For example MMT [30] "meta" system lets one define kernel rules in LF, but resorts to the Scala language for the elaborator. Isabelle lets one describe the axioms of a logic in an LF framework and exposes the higher order unification algorithm to the higher layers of the system, like the proof language one. Incidentally the dominant logic implemented in Isabelle is HOL, that lacks dependent types, hence needs no term comparison algorithm based on narrowing. As a consequence, no need to extend such algorithm to encompass more rewriting is perceived, and no way to extend such algorithm is given to the user. The Dudukti [7] language lets one describe the axioms of a logic in LF modulo equational theories. Unfortunately, at the time of writing, the system lacks an elaborator.

Describing the elaborator in terms of constraints was a choice first made in Agda version 2 [29] and more recently in Lean [11]. Both system implement they own, ad-hoc, constraint solver, the former system in Haskell, while the latter in C++. The approach we follow in the paper is to provide a programming platform where constraints and their propagation rules are first class, to ease documentation, experimentation and extension of the implemented system.

A big advantage of writing the code in λProlog is the possibility of using the Abella [15] theorem prover to mechanically check its correctness. In particular, Abella can be used to prove the soundness of all constraint propagation rules. For example, for an early prototype of the elaborator presented in this paper, the typing constraint propagation rule implementing uniqueness of typing was proved correct in Abella. However, the current version of Abella reasons on the big step operational semantics of λProlog programs without cuts. Because delaying a goal means interrupting execution in the middle, the small step version of the operational semantics would be the appropriate tool to use instead. For example, in the aforementioned proof of uniqueness of typing, we had to assume a certain invariant on the λProlog context that is the context of the delayed typing sequent. In order to know that the invariant really holds during computation, we need to prove that, if it holds initially, it holds after any number of small execution steps. Currently, this statement cannot even be stated in Abella. In the future, we would like to re-design Abella around a small step operational semantics to overcome this limitation and also to be able to reason over cuts.

We plan to compare in a disciplined way the performance of the elaborator of Matita with the one presented in this paper, to precisely size the trade-off between the programming abstractions offered by the HOCLP language we have implemented and their runtime cost.

As another future work, we hope in Abella to fully prove correct, although not complete, the elaborator component, so to call into question the standard design of interactive provers: what is the point of having a kernel component if the elaborator is proved to only generate terms accepted by it?

References

[1] Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. Distilling abstract machines. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 363–376, 2014.

[2] Johan Andersson, Stefan Andersson, Kent Boortz, Mats Carlsson, Hans Nilsson, Thomas Sjöland, and Johan Widen. Sicstus prolog user"s manual. Technical report, 1993.

[3] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. Hints in unification. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, pages 84–98, 2009.

[4] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. A bi-directional refinement algorithm for the calculus of (co)inductive constructions. *Logical Methods in Computer Science*, 8(1), 2012.

[5] Catherine Belleannée, Pascal Brisset, and Olivier Ridoux. A pragmatic reconstruction of lambda-prolog. In *Journal of Logic Programming*, 1998.

[6] Yves Bertot, Georges Gonthier, Sidi Ould Biha, and Ioana Pasca. Canonical Big Operators. In *Theorem Proving in Higher Order Logics*, volume 5170/2008 of *LNCS*, Montreal, Canada, August 2008.

[7] Mathieu Boespflug, Quentin Carbonneaux, and Olivier Hermant. The lambda-Pi-calculus Modulo as a Universal Proof Language. In *Proceedings of the Second International Workshop on Proof Exchange for Theorem Proving*, volume 878 of *CEUR Workshop Proceedings*, pages 28–43, Manchester, UK, June 2012.

[8] James Cheney. *The Complexity of Equivariant Unification*, pages 332–344. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[9] Claudio Sacerdoti Coen and Enrico Tassi. Nonuniform coercions via unification hints. In *Proceedings Types for Proofs and Programs, Revised Selected Papers, TYPES 2009, Aussois, France, 12-15th May 2009.*, pages 16–29, 2009.

[10] Leonardo Mendonça de Moura, Jeremy Avigad, Soonho Kong, and Cody Roux. Elaboration in dependent type theory. *CoRR*, abs/1505.04324, 2015.

[11] Leonardo Mendonça de Moura, Jeremy Avigad, Soonho Kong, and Cody Roux. Elaboration in dependent type theory. *CoRR*, abs/1505.04324, 2015.

[12] Cvetan Dunchev, Claudio Sacerdoti Coen, and Enrico Tassi. Implementing hol in an higher order logic programming language. In *Proceedings of the Eleventh Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, LFMTP '16, pages 4:1–4:10, New York, NY, USA, 2016. ACM.

[13] Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. ELPI: fast, Embeddable, λProlog Interpreter. In *Proceedings of LPAR*, Suva, Fiji, November 2015.

[14] Thom Frhwirth. *Constraint Handling Rules*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[15] Andrew Gacek. The Abella interactive theorem prover (system description). In *Proceedings of IJCAR 2008*, volume 5195 of *Lecture Notes in Artificial Intelligence*, pages 154–161. Springer, August 2008.

[16] Andrew Gacek, Dale Miller, and Gopalan Nadathur. Nominal abstraction. *Information and Computation*, 209(1):48 – 73, 2011.

[17] Herman Geuvers and Gueorgui I. Jojgov. Open proofs and open terms: A basis for interactive logic. In *Computer Science Logic, 16th International Workshop, CSL 2002, 11th Annual Conference of the EACSL, Edinburgh, Scotland, UK, September 22-25, 2002, Proceedings*, pages 537–552, 2002.

[18] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A Machine-Checked Proof of the Odd Order Theorem. In *ITP 2013, 4th Conference on Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 163–179, Rennes, France, July 2013. Springer.

[19] Hugo Herbelin. Type inference with algebraic universes in the calculus of inductive constructions. manuscript, available at http://pauillac.inria.fr/~herbelin/articles/univalgcci.pdf, 2005.

[20] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher Order Symbol. Comput.*, 20(3):199–207, September 2007.

[21] Zhaohui Luo. Ecc, an extended calculus of constructions. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pages 386–395, 1989.

[22] Zhaohui Luo. Coercive subtyping in type theory. In *Computer Science Logic, 10th International Workshop, CSL '96, Annual Conference of the EACSL, Utrecht, The Netherlands, September 21-27, 1996, Selected Papers*, pages 276–296, 1996.

[23] Francesco Mazzoli and Andreas Abel. Type checking throug unification. *CoRR*, abs/1609.09709, 2016.

[24] Dale Miller. *A logic programming language with lambda-abstraction, function variables, and simple unification*, pages 253–281. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991.

[25] Dale Miller. Unification under a mixed prefix. *J. Symb. Comput.*, 14(4):321–358, 1992.

[26] Dale Miller and Gopalan Nadathur. Higher-order logic programming. In *3rd Int. Conf. Logic Programming*, volume 225 of *LNCS*, pages 448 – 462. Springer-Verlan, 1986.

[27] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic.* Cambridge University Press, New York, NY, USA, 1st edition, 2012.

[28] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51, 1991.

[29] Ulf Norell. Dependently typed programming in agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming*, AFP'08, pages 230–266, Berlin, Heidelberg, 2009. Springer-Verlag.

[30] F. Rabe. The MMT API: A Generic MKM System. In J. Carette, D. Aspinall, C. Lange, P. Sojka, and W. Windsteiger, editors, *Intelligent Computer Mathematics*, pages 339–343. Springer, 2013.

[31] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

[32] Neng-fa Zhou. The language features and architecture of b-prolog. *Theory Pract. Log. Program.*, 12(1-2):189–218, January 2012.