Implementing Type Theory in Higher Order Constraint Logic Programming

Ferruccio Guidi, Claudio Sacerdoti Coen, Enrico Tassi

Under consideration for publication in Math. Struct. in Comp. Science

Implementing Type Theory in Higher Order Constraint Logic Programming

Ferruccio Guidi[†] and Claudio Sacerdoti Coen[†] and Enrico Tassi[‡]

[†] *Department of Computer Science and Engineering, University of Bologna*

ferruccio.guidi@unibo.it, claudio.sacerdoticoen@unibo.it

[‡] *Université Côte d'Azur, Inria*

Enrico.Tassi@inria.fr

Received 6 November 2018

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 open terms, that is terms containing unresolved unification variables.
In this paper we confirm that λProlog, the language developed by Miller and Nadathur since the 80s, is extremely suitable for implementing the kernel. Indeed we easily obtain a type checker for the Calculus of Inductive Constructions (CIC). Even more, we do so in an incremental way by escalating a checker for a pure type system to the full CIC.
We then turn our attention to the elaborator with the objective to obtain a simple implementation thanks to the features of the programming language. In particular we want to use λProlog's unification variables to model the object language ones. In this way scope-checking, carrying of assignments and occur-checking are handled by the programming language.
We observe that the eager generative semantics inherited from Prolog clashes with this plan. We propose an extension to λProlog that allows to control the generative semantics, suspend goals over flexible terms turning them into constraints, and finally manipulate these constraints at the meta-meta level via Constraint Handling Rules.
We implement the proposed language extension in the ELPI system and we discuss how it can be used to extend the kernel into an elaborator for CIC.

1. Introduction

1.1. *A pragmatic reconstruction of λProlog*

In (Belleanne et al., 1998) Belleannée et. al. propose a pragmatic reconstruction of λProlog (Miller and Nadathur, 1986; Miller et al., 1991; Miller and Nadathur, 2012), the Higher Order Logic Programming (HOLP) language introduced by Dale Miller and Gopalan Nadathur in the '80s. Their opinion is that λProlog can be characterized as the minimal extension of Prolog that allows to program by structural induction on λterms. 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.

The "hello world" example of λProlog is therefore the following two lines program to compute the simple type of a closed 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.
```

1.2. *λProlog for proof-checking*

According to the Curry-Howard isomorphism, the type-checker 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. For example, it is possible to implement a type-checker for the Calculus of Inductive Constructions (CIC) (Paulin-Mohring, 1996; Werner, 1994; Barras, 1999) that, up to some variations, is the common type-theory/logic shared by the interactive theorem provers (ITPs) Coq (Coq development team, 2017), Matita (Asperti et al., 2011) and Lean (de Moura et al., 2015b).

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 in). The kernel just implements the type-checker together with all the judgments required for type-checking, namely: well-formation of contexts and environments, substitution, reduction, convertibility. The last three judgments are necessary because the notion of equality of the type system incorporates some rewriting rules. In the specific case of CIC rewriting amounts to not only β-reduction, it also includes δ-reduction (definition unfolding) and ι-reduction (fixpoint unfolding and pattern matching evaluation).

1.3. *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 behavior of the kernel. Where the real intelligence of the system lies is instead in the second layer, called *elaborator* or *refiner* (de Moura et al., 2015a; Asperti et al., 2012).

In Section 2 we recall what an elaborator does and we explain why the state of the art of the implementation of elaborators is not satisfactory, which motivated at the very beginning our interest in using HOLP languages to implement elaborators.

2. The elaborator component of today's ITPs

An elaborator takes as input a *partial term*, and optionally a type, and returns the closest term similar to the input 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. In logic programming terminology these holes are unification variables, or more precisely *existentially quantified metavariables*.

For example consider the partial term $\lambda x : T.f\ (X\ x)\ Y$ where T, X, Y are existentially quantified outside the term. This term represents the λ-abstraction of $f\ (X\ x)\ Y$ over the variable x of one type T yet to be determined. The function symbol f is applied to two arguments, both to be determined, and such that x can appear free only in the first one. Elaborating the term versus the expected type $\mathbb{N} \to \mathbb{B}$ instantiates T with $\mathbb{N}$ and verifies that f is a binary function returning a boolean.

The importance of the elaborator is twofold. On the one hand, it is in charge of interpreting the terms that are input by the user in a user-friendly syntax. The user is typically allowed to omit some piece of information; use (and sometimes abuse) mathematical notation; assume proper subtyping even if the formal system only allows to explicitly cast elements of a type to another. A good elaborator therefore gives to the user the feeling of a more intelligent and friendly system.

On the other hand, the elaborator is the mechanism that takes a partial proof and fits it as a sub-proof of another partial proof, in order to make progress on a particular proof obligation. This role of the elaborator arises from the fact that, 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 the proof. In other words, all tactics of the ITP ultimately produce partial proof terms that are elaborated: The more advanced the elaborator is, the simpler the code implementing tactics can be.

2.1. *Implementing an elaborator: state of the art*

The elaborators of the majority of interactive provers are implemented according to the following schema: the syntax of terms is augmented with existential variables (also called metavariables) and the judgments of the kernel are re-implemented from scratch, generalizing 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 *metas-env*, that assigns to metavariables a type judgment (a sequent) and, eventually, an assignment. E.g. a metas-env containing `x:nat, y:bool` ⊢ `X x y : nat` declares `X` to be a 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 becomes narrowing, i.e. higher order unification in the presence of rewriting rules.

Then type checking is generalized to elaboration by replacing all calls to conversion with calls to narrowing and by threading around the metas-env.

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

Programming platform weakness. Much of this 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, occur check).

Intricacy of algorithms. Such code is far from 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.

Code duplication. Given the complexity of 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, the two 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 (Gonthier et al., 2013; Asperti et al., 2009), to coerce data from one type to another (Luo, 1996) or to enrich data to resolve mathematical abuse of notation (Sacerdoti Coen and Tassi, 2009). 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.

2.2. *The proposed approach*

The motivation of our research is to improve over the state of the art by identifying a high-level, logic programming language suitable for the implementation of elaborators.

In particular we want the programming language to let us organize the code as follows. We want to keep a well identified software component for the kernel, not to hinder the trustworthiness of the interactive prover, but still be able to reuse the code of the kernel to build the elaborator by extending it in a modular way. Finally we want to let the user extend the elaborator in the very same way. This can be achieved thanks to the primitive and powerful notion of extensibility provided by higher order logic programming: programs are organized into clauses, and new clauses can be freely added at run-time (via implication) or at program build-time (via accumulation). In this way the rules of the kernel do not need to be re-implemented in the elaborator. On the contrary they are complemented with rules 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.

We envisage the programming language to be a logic one for two reasons. First of all the **intricacy of elaboration** can be mitigated thanks to the compactness and readability typical of declarative languages. Second, we observe that another component of each ITP, the one implementing tactics, can take real advantage from backtracking, in particular to write proof commands performing proof search (Felty and Miller, 1988).

Finally, by picking the logic language to be a higher-order one like λProlog, we posit that we can alleviate the **programming platform weakness**. Such a language supports the Higher Order Abstract Syntax approach which *identifies the object language binders and the meta-language ones*, easing the burden of taking care of binder representation, α-conversion and capture avoiding substitution. Moreover, the semi-shallow embedding approach proposed by our group in (Dunchev et al., 2016) also *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 our approach work out of the box? Not really, as we will argue in this paper.

Structure of the paper In Section 3 we implement 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. In Section 4 we discuss whether λProlog is suitable as a very high-level programming language to implement an elaborator and, if not, what should be added to it. 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 implement 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 the kernel presented in Section 3. The implementation is the first major use case for the language. The final Section 6 contains comparisons with related work and future work.

3. A modular kernel for CIC

In this section we scale the type-checker for the simply typed λ-calculus of Section 1 to a type-checker for a generic PTS that also allows cumulativity between universes and dependent products covariant in the second argument. Then we instantiate it to obtain the predicative universal fragment of Luo's ECC (Luo, 1989) and the PTS of CIC, and then we modularly extend the type-checker to the whole CIC by adding inductive types as well.

3.1. *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 (Dunchev et al., 2015)) 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.
```

The `of` predicate is made ternary:

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

It relates a term, its type and a translation (called *elaboration*) of the term that is ensured to be well typed. While in this subsection the first term is simply copied into the third one, in the next subsection the `of` predicate is improved to insert explicit type casts (coercions) when they are needed to make the first term well typed.

Here we focus on the refined rules for application and lambda abstraction.

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 implements this check. The `match_sort` and the `match_arr` predicates are used to check if the weak head normal form of the first argument is respectively a sort or a dependent product. For example, `(match_arr (arr nat x \ x) A F)` is meant to instantiate `A` with `nat` and `F` with `(x \ x)`.

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

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

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

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

The rules above have the merit of being syntax-directed (when the first argument is a ground term) and always inferring the most general type, in the sense of Luo (Luo, 1989).

We provide an implementation for `sub`, `match_sort` and `match_arr` in Section 3.3 while `succ` and `max` are described in Section 3.4.

3.2. *Coercive subtyping*

The notion of coercive subtyping (Luo et al., 2013) relates two formal systems T and $T[C]$: System T does not feature subtyping while $T[C]$ extends T with the set of subtype judgments C. The authors of (Luo et al., 2013) show that a term t well typed in $T[C]$ can be related to a term t' well typed in T, and that t' can be obtained inserting in t explicit type casts around the subterms that, in order to typecheck, required a type rule in C. In other words it gives an algorithm to simulate subtyping in T by elaborating its terms via the insertion of coercions (explicit type casts) during type checking.

```
of (app M N) BN (app RM RKN) :-
  of M TM RM, match_arr TM A1 Bx, of N A2 RN,
  coerce A2 A1 K, of (app K N) A3 RKN, sub A3 A1, BN = Bx RKN.
```

This alternative rule to type-check applications relies on the user-provided, untrusted `coerce` predicate to find a cast `K` from `A2` to `A1`. These are the clauses that the user adds to the library to implement the classic example of coercions relating the type of natural numbers, integers and rational numbers:

```
type coerce term -> term -> term -> o.
coerce nat int zpos.
coerce int rat (lam int i\ frac i (zpos (successor zero))).
coerce A C F :- coerce A B F1, coerce B C F2, F = (lam A x\app F2 (app F1 x)).
```

In the literature on coercive subtyping many flavors of coercions have been studied, e.g. between polymorphic types like lists. All kinds of coercions share the structure above, even if the construction of the composed coercion may be a bit more involved.

3.3. *Reduction and conversion rules*

To implement `sub`, `match_sort` and `match_arr` we first implement the `whd1` and `whd*` predicates that, together, implement weak head reduction. 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 -> o.

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 -> o.
match_sort T I :- whd* T (sort I).

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

type conv, conv-whd term -> term -> o.
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 -> o.
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).
```

Here and in the rest of the paper we are interested only in pure type systems that enjoy strong normalization: any reduction strategy terminates on well typed terms. Hence to ensure the termination of reduction we have to check that `sub` `match_sort` and `match_arr` are always given as input terms (actually types) synthesized by `of` and that only well typed terms (e.g. terms occurring as the third argument of `of`) are substituted inside dependent types (by the type checking rule of `app`).

3.4. *Defining the Pure Type System*

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 ECC (Luo, 1989), using the integer i to represent the universe Type_i. The `macro` directive is recognized 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.
```

3.5. *Covering the full CIC*

The type theory of Matita and Coq is an extension of ECC with global definitions, declarations, primitive inductive types, case analysis and structural recursive definitions.

We implemented the type checking rules for all these extensions 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 the relative clauses to the `whd1`, `conv`, `sub` 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, except for the verification of soundness of inductive type declarations and termination of recursive functions that we have not implemented yet. To assess the resulting kernel we linked ELPI into Matita and we made the two kernels (the original one of Matita and the one described in this paper) run side by side while type checking the arithmetic library of the system.

4. λProlog meets partial terms

In type theory terms are used (also) to represent proofs, that is derivation trees. In order to represent ongoing proofs, in other words incomplete derivation trees, implementations of type theory typically use partial terms. A partial term contains meta variables standing for missing branches in the derivation tree.

4.1. *From generative semantics to constraints*

Suppose that the λ-term `(lam T a\ P a)` is meant to encode 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, producing a new goal. The new goal, which is `(of (P x) (arr x _\ x) RP)`, for some fresh `x`, is problematic. Indeed the type checking rule for application, whose head is `of (app M N) BN (app RM RN)`, applies indefinitely to it and the its generated new goals.

The `of` predicate has 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 behavior 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 metas-env) the need for verifying if the type judgment 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 a term of type `(arr x _\ x)` has to be provided (in a context where (`of x (sort I) x`) holds), that in turn corresponds to the proof obligation $A : (\text{sort I}) \vdash A \Rightarrow A$.

The sometimes undesired generative semantics of λProlog is 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 detect problematic goals both statically and dynamically and to suspend them automatically. For example, by using the `delay` pack of SWI-Prolog (Wielemaker et al., 2012), the goal `plus(X,1,Y)` is suspended until either `X` or `Y` are instantiated. Delayed goals can be thought as *constraints* over the metavariables occurring in them.

These mechanisms, however, have never been standardized. Some of them break the clean declarative semantics of Prolog, others respect it. λProlog does not provide these kinds of facilities, or at least, does not expose them to the programmer. For example, the Teyjus system suspends unification problems outside L_λ(also known as pattern fragment) and implements some machinery to wake them up when they re-enter the fragment, but does not expose any primitive to suspend other kinds of goals.

The solution we propose is to extend the ELPI λProlog interpreter with a new builtin named `declare_constraint`.

```
type declare_constraint o -> A -> o.
```

The first argument is a goal to be suspended while the second argument is a flexible term (i.e. unification variable in head position).

We can use `declare_constraint` in conjunction with `var` as follows:

```
of X T Y :- var X, declare_constraint (of X T Y) X.
```

The clause above tells the interpreter to suspend goals of the form `(of X T Y)` whenever `X` is flexible. Instead of diverging, the running example now converges to a final state where the following goal (computation) is suspended. In the goal, `{x}` is the local signature:

```
{x} : of x (sort I) x  ?-  of (P x) (arr x _\ x) (RP x)
```

This suspended goal is to be seen as a type *constraint* on assignments to `P`: when `P` gets instantiated by a term `t`, the goal `(of (t x) (arr x _\ x) (RP x))` is resumed and `(t x)` is checked to be of type `(arr x _\ x)`. In turn such a 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` when t is ill-typed or its type is wrong, 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. For example a tactic could generate the term `(t = x\ lam x w\ Q x w)` to represent a proof step that opens a new goal (corresponding to `Q`).

The three scenarios above perfectly match the desired behavior of an elaborator. In addition to that, the set of suspended goals implicitly kept by the language interpreter represents faithfully the metas-env data structure, relieving the programmers from managing it themselves, threading it through all type rules and restoring an old copy explicitly on backtracking. Moreover, the automatic resumption of type constraints makes it impossible to obtain an ill-typed term by instantiation: the code is correct with respect to this invariant by construction.

4.1.1. *The mode directive* In order to reliably turn goals into constraints we introduce the `mode` directive, that allows to tag the arguments of a predicate with either `i` or `o`.

```
mode (of i o o). % first argument is "i", the other two "o"
```

With this directive arguments tagged with `i` (for input) are *matched* against the corresponding arguments in the goal. This is in contrast to arguments tagged with `o` (for output) are unified with the corresponding arguments in the goal.

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 (lam RT RF) :- ...
of (app M N) T (app RM RN) :- ...
of X T RX :- declare_constraint (of X T RX) X.
```

Notice that the mode declaration for `of` is not equivalent to placing a clause like `(of X T RX :- var X, !, ...)` on top, since hypothetical clauses may be placed by the runtime before this one, and they can be generative as well.

4.2. *Constraints*

Here we give a more formal description of the **declare_constraint** builtin.

The runtime state of ELPI can be thought as a triple $\mathcal{S} = (\Sigma_0, \Gamma_0, S)$ where Γ_0 is the user-provided program, Σ_0 is the signature that collects all constants appearing in Γ_0 and S is a formula of the form

$$\bigvee A_i \qquad \text{where } A_i = \bigwedge G \;\dot{\wedge}\; \bigwedge C$$

The global signature Σ_0 and program Γ_0 are never modified during execution: only S is. Therefore, to simplify the notation, in the following we will always omit to write Σ_0 and Γ_0 when showing states, and when we use a state $\mathcal{S} = (\Sigma_0, \Gamma_0, S)$ as a formula we implicitly refer to its third component S.

Each A_i is an alternative: a solution of A_i gives a solution for the initial query. Free variables occurring in A_i are implicitly existentially quantified in front of A_i or, equivalently, the sets of variables occurring in the A_is are disjoint and the existential quantifications are at the top-level, surrounding $\bigvee A_i$. An alternative is made by the conjunction of all goals to be solved. Goals in such conjunction are separated into two groups: G for active goals, and C for suspended goals (constraints). We decorate the conjunction symbol with a dot as in $\dot{\wedge}$ to visually separate the two groups of goals.

A goal is a sequent

$$\Sigma \rhd \Gamma \vdash P$$

where Σ is the local signature (a set of eigenvariables, i.e. fresh constants generated by **pi**), Γ is the hypothetical context (of clauses introduced by **=>**) and P is the predicate being proved. Note that existential variables are always quantified outside the local signature, an invariant that can be forced thanks to raising (Miller, 1992).

When the goal is suspended via **declare_constraint** an additional piece of information is attached to it, namely a flexible term K:

$$(\Sigma \rhd \Gamma \vdash P)_K$$

This extra piece of information corresponds to the second argument of the built-in predicate of which we recall the type:

```
type declare_constraint o -> A -> o.
```

The meaning of a goal as a logical formula is

$$\forall \bar{x}, \bigwedge \Gamma \Rightarrow P$$

where $\bar{x}$ binds all Σ and $\bigwedge \Gamma$ is the conjunction of all the formulas in Γ.

A program run is given by a finite or infinite sequence $\mathcal{S}_i$ of states such that $\mathcal{S}_0 = (\Sigma_0, \Gamma_0, Q \;\dot{\wedge}\; \top)$ where $Q = (\varnothing \rhd \vdash P)$ is the initial query. For each i, the sequent $\Sigma_0 : \Gamma_0 \vdash \mathcal{S}_{i+1} \Rightarrow \mathcal{S}_i$ is provable in intuitionistic logic. A final state has the form $\bigvee A_i$ such that $A_1 = \top \;\dot{\wedge}\; C$: C is the set of unresolved constraints returned to the user together with the witness for the existentially quantified variables that occur free in the initial query Q.

Normal execution proceeds looking for a uniform proof according to the rules of (Miller et al., 1991) fitted to our framework. For example, the following rule is used when the leftmost active goal of the topmost alternative is a conjunction:

$$((\Sigma \rhd \Gamma \vdash F_1 \wedge F_2) \wedge G_1 \ldots \dot{\wedge} \ldots) \vee A_1 \ldots \xmapsto[(\wedge_i)]{}$$
$$((\Sigma \rhd \Gamma \vdash F_1) \wedge (\Sigma \rhd \Gamma \vdash F_2) \wedge G_1 \ldots \dot{\wedge} \ldots) \vee A_1 \ldots$$

When the leftmost active goal of the topmost alternative is an atom different from `declare_constraint` (and cuts, that we omit from the present formalization), backchaining is employed, yielding a set of alternatives that replace the topmost alternative in the goal status:

$$((\Sigma \rhd \Gamma \vdash r\ t_1 \ldots t_n) \wedge G_1 \ldots \dot{\wedge} \ldots) \vee A_1 \ldots \qquad \xmapsto[(backchain)]{}$$
$$((\Sigma \rhd \Gamma\rho_1 \vdash P_1\rho_1) \wedge G_1\rho_1 \ldots \dot{\wedge} \ldots) \vee \ldots ((\Sigma \rhd \Gamma\rho_k \vdash P_k\rho_k) \wedge G_1\rho_k \ldots \dot{\wedge} \ldots) \vee A_1 \ldots$$

This rule can only be applied if $k > 0$ clauses part of $\Gamma_0 \cup \Gamma$ have a head that unifies with $r\ t_1 \ldots t_n$ and generates substitution ρ_i. We write $\Gamma\rho$ for the application of substitution ρ to all entries in context Γ; $P\rho$ for the application of ρ to the predicate P and and $G\rho$ for the application of ρ to a goal G (to its context and its predicate). Each clause i has a premise P_i (eventually $\top$) resulting in the new goal.

If no clause can be backchained on (i.e. $k = 0$), then backtracking takes place:

$$(\ldots \dot{\wedge} \ldots) \vee (G_1 \ldots \dot{\wedge}\, C_1 \ldots) \vee A_2 \ldots \xmapsto[(backtrack)]{}$$
$$(G_1 \ldots \dot{\wedge}\, C_1 \ldots) \vee A_2 \ldots$$

When the active goal is `declare_constraint` the state is rewritten as follows

$$((\Sigma \rhd \Gamma \vdash \textbf{declare\_constraint}\ P\ K) \wedge G_1 \ldots \dot{\wedge} \ldots) \vee A_1 \ldots \xmapsto[(suspend)]{}$$
$$(G_1 \ldots \dot{\wedge} \ldots (\Sigma \rhd \Gamma \vdash P)_K) \vee A_1 \ldots$$

The term K is said to be the key of the constraint. Whenever the key of a constraint is assigned the constraint is resumed. Resumption takes precedence over all other rules.

$$(G_0 \ldots \dot{\wedge} \ldots (\Sigma \rhd \Gamma \vdash P)_K \ldots) \vee A_1 \ldots \xmapsto[(resume)]{} ((\Sigma \rhd \Gamma \vdash P) \wedge G_0 \ldots \dot{\wedge} \ldots) \vee A_1 \ldots$$

The last two rules define how goals are moved between the collections of active goals and constraints. The standard λProlog rules only work on the first collection, preserving the second collection that stores the constraints, that are thus not changed. The collection of constraints is rewritten by user-provided rewrite rules. ELPI provides a dedicated language to write these rules inspired by Constraint Handling Rules (CHR).

4.3. *Higher Order Constraint Handling Rules*

A constraint handling rule rewrites the state of ELPI in the following way:

$$\big(G_0 \ldots \dot{\wedge} \ldots \vec{C}_i \ldots\big) \vee A_1 \ldots \xmapsto[(CHR)]{} (N \wedge G_0 \ldots \dot{\wedge} \ldots) \vee A_1 \ldots$$

Where N is a new goal and some constraints $\vec{C}_i$ have been removed. The collection of alternatives does not change, hence CHR rules can only make committed choices.

The concrete syntax in ELPI of a constraint handling rule is

```
RULE       ::= rule TO-MATCH \ TO-REMOVE | GUARD <=> TO-ADD .
TO-MATCH   ::= SEQUENT*
TO-REMOVE  ::= SEQUENT*
TO-ADD     ::= SEQUENT
GUARD      ::= TERM
SEQUENT    ::= TERM ?- TERM
```

where `TO-MATCH` and `TO-REMOVE` are lists of patterns for goals to be respectively matched or matched-and-deleted from C, the collection of constraints of the first alternative A_0. `GUARD` is a test (a λProlog goal) and `NEW` is a goal to be added in front of the conjunction of active goals. The patterns to be matched and eventually removed are sequents in an empty local signature, which is omitted in the concrete syntax.

We adopt `?-` in place of $\vdash$ in the concrete syntax to stress that the user is handling queries (and not program clauses). The term before `?-` that represent the sequent context is a list of formulae. The list is not required to be ground: it can even be, and it often is, just a meta-variable.

An example of CHR rule is the following one, partially modeling uniqueness of types.

```
rule  (H1 ?- of X nat _) (H2 ?- of X bool _) % sequents to match
      \                                      % no sequent to remove
      | true                                 % guard
    <=> ([] ?- fail).                        % new goal
```

The rule injects in G the goal `fail` if two incompatible type constraints are expected to hold on *the same* metavariable `X`. In Section 4.4 this rule is generalized to any type expression, and not just `bool` and `nat`, and an example is completely worked out. Here we focus on the operational description of rules application.

We call $\mathcal{V}$ the set of free (not bound by a `sigma`) unification variables mentioned by all terms in the rule (that is `H1, H2` and `X` in the example above); $\mathcal{M}$ and $\mathcal{D}$ the set of sequents to be matched and removed respectively; $\mathcal{T}$ the guard and $\mathcal{N}$ the new sequent. Operationally the following steps are performed towards applying a CHR rule:

1. **rename)** the signatures of all constraints are made disjoint by applying appropriate (injective) renaming of eigenvariables;
2. **reify)** the hypothetical context of every constraint is reified into an implication of a conjunction: $\Sigma \triangleright h_1, \ldots, h_n \vdash P$ becomes $\Sigma \triangleright \bigwedge h \Rightarrow P$;
3. **freeze)** Constraints in C are frozen, i.e. a bijective map $\mathcal{F}$ from existential variables to fresh distinct eigenvariables is picked, and the map is turned to a substitution applied to C. We write $\mathit{freeze}(C)$ for the application of the substitution to C. The base case of *freeze* is $\mathit{freeze}(X t_1 \ldots t_n) = (\textbf{uvar}\ \mathcal{F}(\mathrm{X})\ [\mathit{freeze}(t_1), \ldots, \mathit{freeze}(t_n)])$ where **uvar** is a global reserved symbol; all other cases just perform recursion over the term. We call *unfreeze* and denote it with freeze^{-1} the inverse operation to *freeze*, whose base case is $\mathit{freeze}^{-1}(\textbf{uvar}\ \mathrm{c}\ [t_1, \ldots, t_n]) = \mathcal{F}^{-1}(c)\ \mathit{freeze}^{-1}(t_1)\ \ldots\ \mathit{freeze}^{-1}(t_n)$
4. **select)** Non-deterministically two disjoint sets of distinct constraints $\vec{c}_i^{\,1}$ and $\vec{c}_j^{\,2}$ are selected from $\mathit{freeze}(C)$ where $c_i^h = \Sigma_i^h \triangleright H_i^h \Rightarrow P_i^h$; the ones in the first set will be

matched against $\mathcal{M}$ (the pattern for the goals to be matched), while the ones in the second against $\mathcal{D}$ (the pattern for those to be removed);

5. **guard)** A new signature $\Sigma = (\biguplus \Sigma_i^h) \uplus Img(\mathcal{F})$ is forged as the disjoint union of the involved local signatures and the new eigenvariables in the codomain of $\mathcal{F}$. Then all free variables in $\mathcal{V}$ are created under such signature (we omit here to raise them for simplicity of the discussion), i.e. we think of them as existentially quantified under Σ. Then a fresh runtime state $(\Sigma_0', \Gamma_0', S')$ is created where S' is the unary disjunction of the unary conjunction of the following sequent. The sequent tries to match the frozen constraints with the patterns in $\mathcal{M} \cup \mathcal{D}$, also reified in Σ.

$$\Sigma \rhd \quad \vdash \overrightarrow{(H_i^1 \Rightarrow P_i^1)} = \mathcal{M} \wedge \overrightarrow{(H_j^2 \Rightarrow P_j^2)} = \mathcal{D} \wedge \mathcal{T}$$

The global signature Σ_0' of the new runtime state must be a superset of Σ_0, while the program Γ_0' can be unrelated to Γ_0, only needing the rules to prove $\mathcal{T}$. For simplicity we can assume $\Gamma_0' = \Gamma_0$, which is what is currently implemented in ELPI.

6. **commit)** The new runtime state is rewritten until it reaches a normal form, leaving the old runtime state untouched in the meantime. Less operationally, assume that there exists an n s.t. $(\Sigma_0', \Gamma_0', S')$ rewrites in n steps to a final state $(\Sigma_0', \Gamma_0', (\top \curlywedge C') \vee \bigvee A_i')$. Let σ be the computed instantiation for the variables in $\mathcal{V}$. Note that these instantiations must live in the signature Σ. The program aborts if existential variables not in $\mathcal{V}$ occur in the image of σ or if C' is not $\top$. Otherwise the goal

$$\Sigma \setminus Img(\mathcal{F}) \rhd \mathit{freeze}^{-1}(\mathcal{N}\sigma)$$

is added at the front of the conjunction of active goals while the constraints that originated c_j^2 (via renaming, reification and freezing) are deleted from the delayed goals C. Since the new goal is added at the front, it will be the next processed goal, unless another constraint propagation rule can be triggered immediately.

In this version of CHR the guard plays a double role. On the one hand, when it fails, it signals that the rule should not be applied. On the other hand it is, in practice as we show in the next section, also used to synthesize the new goal from the constraints matched.

4.4. *Example: a CHR rule that models uniqueness of types*

To improve the readability of the code in this section we use two syntactic extensions of ELPI. First `(pi a b\ t)` is syntactic sugar for `(pi a\ pi b\ t)`. Second `=>` accepts as the first argument a list of terms of type `o`. In particular `([t1,t2] => t)` is operationally equivalent to `(t2 => t1 => t)`. Dually a list of terms of type `o` is considered as a conjunction, i.e. the term `[p, q]` is a valid goal and is equivalent to `(p, q)`.

In addition to that we systematically omit the third argument of the `of` predicate, since it plays no role in this example. Finally, we quantify unification variables in front of formulas, so that all variables in scope are always explicit (as the arguments of the unification variable).

The property we want to model with the CHR rule is that a term can only have one type. In type theory this means that if a term has two types `T1` and `T2` then `(sub T1 T2)` must hold. We can use this property to keep the set of constraints duplicate free: when two constraints **of** about the same variable X impose the (not yet available) term has type T1 and T2 we can remove one of the two and generate the goal (**sub** T1 T2).

A query generating such scenario is the following one (notice that X is used twice):

```
  of (app (lam nat x\ X x)
          (app (app (lam Y y\ lam nat z\ X y) _) zero)) W
```

where nat and zero are global constants and where the clause (**of** zero nat) is part of the program. The predicate **of** is called once on (X x) in a context where x is known to be of type (nat); and a second time on (X y) in a longer context where z plays no role and y has no known type. The second time (X y) is expected to be of type nat.

This is the state of the constraint store immediately after the second occurrence of X is encountered by **of**.

```
{x} : [ of x nat ] ?- of (X x) (T x)          /* suspended on X */
{y z} : [ of z nat, of y Y ] ?- of (X y) nat  /* suspended on X */
```

Constraints are already printed under disjoint signatures and with a reified context (the first two steps of CHR rules' application).

The rule we want to use to simplify the constraints' set is the following one:

```
rule  (G1 ?- of (uvar K L1) T1)                /* to match */
    \ (G2 ?- of (uvar K L2) T2)                /* to remove */
    | (ut-condition G1 G2 T1 T2 L1 L2 [] New) /* guard & build New */
  <=> (G2 ?- New)                              /* new goal */
```

where **ut-condition** G1 G2 T1 T2 L1 L2 [] New is supposed to build in New a goal asserting that T1′ and T2 are convertible where T1′ is T1 moved from context G1 to context G2.

Step 3 (freeze) synthesizes the map $\mathcal{F} = \{$(cX, X); (cY, Y); (cT, T)$\}$ where cX, cY and cT are fresh global constants. The resulting constraints are

```
{x} : [of x nat] ?- of (uvar cX [x]) (uvar cT [x])
{y z} : [of z nat, of y (uvar cY [])] ?- of (uvar cX [y]) nat
```

Selection (step 4) picks the first constraint as the one to be matched, and the second one as the one to be removed.

Step 5 runs the following query in a fresh interpreter:

```
pi x y z\ sigma G1 G2 K L1 L2 T1 T2 New\
 ([of x nat] ?- of (uvar cX [x]) (uvar cT [x])) = (G1 ?- of (uvar K L1) T1),
 ([of z nat, of y (uvar cY [])] ?- of (uvar cX [y]) nat) = (G2 ?- of (uvar K L2) T2),
 ut-condition G1 G2 T1 T2 L1 L2 [] New.
```

This goal succeeds with a substitution σ that includes the following assignments:

```
New = (of y nat, sub nat (uvar cT [y])).
G2 = [of z nat, of y (uvar cY [])]
```

Then the following goal is injected in the main interpreter (added to G, step 6):

{x y z} : freeze^{-1}(G2σ) **?-** freeze^{-1}(Newσ)

The new goal asserts that the contexts and the type of X are convertible.

Note `x` was mapped to `y`. Such piece of information is extracted from the fact that `L1` and `L2` (the arguments of the same unification variable `cX`) are respectively bound to `[x]` and `[y]`.

Hence, once *freeze*$^{-1}$ is computed, the new goal becomes

```
{ x y z } : [of z nat, of y Y] ?- (of y nat, sub nat (T y))
```

Notice that `x` is unused, hence the signature could be strengthened. Indeed what `ut-condition` does is to move terms from the constraint to be kept into the signature of the constraint to be removed. Such constraint is to be replaced with goals that are indeed equivalent to it (in the meta theory of the object language).

4.4.1. *The code of* `ut-condition` The code relies on the following invariant: one of the two constraints 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.

In an ITP like Matita this invariant is set up by the parser and kept by the elaborator as follows. The concrete syntax lets the user only write unnamed (linear) place holders that are then parsed as a fresh meta variable applied to the entire context of bound variables (i.e. it is in L_λ by construction). The elaborator never reduces terms before type checking them; in other words only elaborated terms are eventually substituted and duplicated by reduction. The first time a meta variable is encountered by the elaborator it is L_λ and hence a canonical type constraint for it is declared.

It is customary both in theory (Geuvers and Jojgov, 2002) and implementation (Asperti et al., 2012) to always keep only canonical constraints, that are collected in a set called *metavariable environment* (metas-env) and that are the only proof obligations presented to the user. The role of `ut-condition` is to keep the set of constraints duplicate free (only one type constraint for a meta variable) by turning additional constraints into invocations of `sub` (convertibility tests).

The `ut-condition` predicate has the following type:

```
type ut-condition
  list o -> list o ->               /* the contexts of K */
  term -> term ->                   /* the types of K */
  list term -> list term ->         /* the arguments (L1 and L2) */
  list (pair term term) -> o -> o.  /* accumulator and New */
```

The predicate succeeds when the first sequent is canonical.

```
ut-condition _ _ T1 T2 [] [] Todo Todo2 :-
  mk-of Todo Todo1, copy T1 T, append Todo1 [sub T T2] Todo2.
ut-condition C1 C2 T1 T2 [V1|V1S] [V2|V2S] Todo New :-
  name V1, not(mem V1S V1),        /* L_\lambda check      */
  mem C1 (of V1 TV1),              /* fetch the canonical type of V1 */
  copy V1 V2 =>                    /* substitute V2 for V1 */
    ut-condition C1 C2 T1 T2 V1S V2S [(V2,TV1)|Todo] New.
mk-of [] [].
mk-of [(V,T)|Rest] [of V T1|New] :- copy T T1, mk-of Rest New.
copy (app A B) (app A1 B1) :- copy A A1, copy B B1.
copy (lam T F) (lam T1 F1) :- copy T T1, pi x\ copy x x => copy (F x) (F1 x).
copy (arr T F) (arr T1 F1) :- copy T T1, pi x\ copy x x => copy (F x) (F1 x).
copy (sort I) (sort I).
% constants are copied to themselves
copy nat nat.
...
```

The lists `L1` and `L2`, when put side by side, represent an explicit substitution: the n-th item of `L1` (a name) is assigned the n-th term in `L2`. In order to be able to apply such substitution the **`ut-condition`** predicate loads the program with **`copy`** predicates while it traverses the two lists. The **`ut-condition`** predicate also gathers in the `Todo` list pairs of terms: the first component is the value part of the explicit substitution (from `L2`) while the second component the (canonical) type of the corresponding variable from the canonical sequent. Finally it uses **`copy`** to apply the substitution and place types coming from the canonical sequent into the other one and generate the new goal.

It is worth pointing out that `C1` is always used via the **`mem`** predicate, hence the order in which the hypothetical context is presented to the rule is not relevant.

4.5. *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 downloadable in open source form from `https://github.com/LPCIC/elpi`. The non-determinism in the operational semantics of CHR rules applications is resolved according to the so called “refined operational semantics” of CHR (Duck et al., 2004).

In addition to the new programming constructs that deal with constraints, ELPI slightly differs from the version of λProlog implemented in Teyjus (Qi et al., 2015) in a few minor aspects. First, the module system of Teyjus is not implemented. Only the `accumulate` directive is honored and has a different semantics when the same module is accumulated twice. Second, 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. ELPI has also been validated on the large code base of the Foundational Proof Certificate (Roberto Blanco, 2017) framework.

Last, ELPI is a pure interpreter written in OCaml. Embedding it into larger applications like Coq or Matita is easy and presents minimal overhead (no external program to run, no compilation chain). Finally, ELPI can be easily compiled to JavaScript and run in a browser as demonstrated by `https://voodoos.github.io/ElpIDE/`.

5. Towards an elaborator for CIC

We are developing an elaborator for CIC as a modular extension of the kernel described in Section 3. The elaborator mimics as closely as possible the behavior of the one of Matita

0.99.1 (Asperti et al., 2012), with the exception of the handling of universe constraints that follows Coq (Herbelin, 2005) and the handling of flexible-flexible narrowing cases that are delayed in our new implementation and that are instead resolved using heuristics in Matita.

In place of following the algorithm of Matita and Coq, an alternative very promising choice would have been to mimic the elaborator described in (Mazzoli and Abel, 2016) and already presented via typing rules that yield a set of higher order unification constraints to be solved later. The choice to be closer to Matita 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 optimize 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.

5.1. *Type-checking: the `of` predicate*

Using a mode declaration in conjunction with **`declare_constraint`**, we delay type-checking a metavariable, turning it into a proof obligation, i.e. a sequent of the form `G ?- of X t RX` where `G` holds type declarations for variables (`of x t x`).

```
mode (of i o o).
of K T RK :- var K, !, K = RK, declare_constraint (of K T K) K.
```

We couple this clause with a constraint handling rule very similar to the one described in Section 4.4 in order to keep the metas-env duplicate free.

5.2. *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 Russell'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, transitivity and antisymmetry of `leq`, transitivity and irreflexivity 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) <=> fail.
  rule (ltn X Y) (ltn Y X) <=> fail.
  rule (ltn X X) <=> fail.
  % reflexivity
  rule \ (leq X X).
  % antisymmetry
  rule (leq X Y) \ (leq Y X) <=> (Y = X).
  % transitivity
  rule (leq X Y) (leq Y Z) <=> (leq X Z).
  ...
  % idempotence
  rule (leq X Y) \ (leq X Y).
  ...
}
```

The code just shown is sufficient to infer universe levels for the predicative fragment of ECC. However, the ECC implemented by Matita has also a sort `prop` for impredicative propositions and a mirror copy of the predicative hierarchy to differentiate between universes of data types `type lvl` and universes of predicative propositions `cprop lvl`. Therefore the actual code implemented instantiates `@univ` with all three kind of universes and implements the `lt,succ,max` predicates according to the PTS of Matita. In particular, to compare predicative universes the code boils down to calls to the `ltn` and `leq` predicates implemented above using CHR-style propagation rules.

5.3. *Reduction: the* `whd1`*,* `whd*` *predicates*

The predicate `whd*`, defined in terms of `whd1`, computes the normal form of the input. This 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 type 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 o).
whd1 X _ :- var X, 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.

5.4. *Term comparison: the* `sub`*,* `conv` *predicates*

Conversion when at least one of the arguments is a flexible term amounts to higher order narrowing. When the unification problem falls in the L_λ fragment then the programming language can synthesize unifiers just fine.

Unfortunately it is quite common for real unification problems to fall outside the decidable fragment. In this case Matita employs an heuristic: always favor projection 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.

Thanks to the extensibility of the programming platform we are employing we can avoid hard coding heuristics and rather let the user extend the algorithm to cover very specific unification problems that arise in his formalization.

5.4.1. *Covering more than L_λ in the "bigop" library* Let's examine a concrete unification problem that is recurrent when the library of big operators of Coq (Bertot et al., 2008) is used. A sample lemma letting one re-index a sum is:

$$\forall n, \forall F : \mathbb{N} \to \mathbb{N}, \forall h : I_n \to I_n, \text{injective } h \Rightarrow \sum_{j \in I_n} F j = \sum_{j \in I_n} F(h\ j).$$

The intended way to use this lemma is to replace a sum by another one where the injective function h is used to reorder the elements of the sum. This amounts to unifying the left hand side of this equation with (a subterm of) the formula the user is trying to prove, for example $\sum_{j \in I_n} j + j$. As one expects the big sum binds j and the higher order term F is applied to such a variable. Unfortunately $F j$ does not fall in the pattern fragment: an explicit cast is inserted by Coq around j, since j ranges over the finite type I_n of natural numbers smaller than n, while F takes a bare bone natural number (the finiteness of the support explains why the injectivity of h is a sufficient condition for this theorem to hold). The cast, called `nat_of_ord`, makes the unification problem look like

```
sub (sum n (j\ add (nat_of_ord j) (nat_of_ord j))
    (sum n (j\ F (nat_of_ord j)))
```

This specific problem contains a term that is outside L_λ, namely `F (nat_of_ord j)`. Still, as proved in (Libal and Miller, 2016), this problem admits a solution (under some proviso) that is also unique.

To write the code that finds a solution one can resort to CHR rules that can manipulate the aforementioned term easily, since `F` is frozen. One can easily craft a clause that declares the previous problem as a constraint and a CHR rule that solves it.

```
sub (sum N T) (sum N F) :- declare_constraint (sub (sum N T) (sum N F)) _.
rule (G ?- sub (sum N T) (sum N (j\ uvar F [nat_of_ord j])))
```

```
  | (pi j y\ copy (nat_of_ord j) y => copy (T j) (T1 y))
 <=> (G ?- F = T1)
```

Note that the guard of the rule checks that `j` never occurs in `T` naked (no `copy j y` clause), but only inside the `nat_of_ord` context, `T1` is `T` where all occurrences of `nat_of_ord j` are replaced by `y`.

5.5. *Teaching the elaborator the contents of the library*

Another extension that proved to be crucial to large Coq development and that is intimately related to logic programming is the one to solve unification problems like the following one:

```
sub (app carrier_of_group R) integers
```

In type theory 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 their library, the standard ring of integers, and they would like to instruct the system with the heuristics that picks for `R` that group.

The code to do so is the first rule below:

```
sub (carrier_of_group X) integers :- X = integer_group.
sub (carrier_of_group X) (intersection A B) :-
  sub (carrier_of_group GA) A,
  sub (carrier_of_group 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`.

Outstanding Coq formalizations like the Odd Order Theorem (Gonthier et al., 2013) contain thousands of rules like the ones above. The elaborator of the Coq system provides an ad-hoc extension mechanism, namely Canonical Structures (Mahboubi and Tassi, 2013), to express these extensions that, in our setting, are just clauses of a specific shape.

Matita provides a slightly more general extension mechanism named Unification Hints (Asperti et al., 2009) that can be modeled by clauses as well.

5.6. *Automatic saturation*

Matita was the first prover to allow in the syntax both placeholders for an omitted term and placeholders for a possibly empty sequence of omitted terms. The concrete syntax for the latter placeholders is "...". For example, Matita accepts `eq_f2 ... plus` and elaborates it to `eq_f2 nat nat nat plus` where `eq_f2` is a constant that expects in input three types A, B, C and a function of type $A \Rightarrow B \Rightarrow C$.

The operation that fills in the dots is called *saturation* and the semantics of the dots is greedy: the dots can only be used in argument position in an application and they are turned in the minimal sequence of inferred terms that make that application type-check. The code of many tactics is highly simplified by automatic saturation: it is sufficient for the tactic writer to insert enough dots here and there and let the elaborator figure out the number of omitted arguments and their instantiation. Dots also reduce the need for implicit arguments in the prover syntax.

In our elaborator we use the syntax `hole` for a placeholder for a single term and `vect` for the dots. The code that handles them only requires a few lines:

```
type hole,vect term.
% turn a hole into a metavariable and repeat
of hole U RT :- !, of T U RT.
% try first to turn dots into zero holes
of (app M vect) TM RM :- of M TM RM.
% otherwise add one more hole and repeat
of (app M vect) U RT :-
 of M TM RM, not (var TM), match_arr TM _ _, of (app (app RM hole) vect) U RT.
```

5.7. *Coverage of the full type theory of Matita*

We modularly extend the kernel presented in Section 3 to cover all the rules of the elaborator of Matita, except for the flexible-flexible case of unification, that we currently suspend, and for the rules, already omit in the kernel, that check definitions of inductive types and termination of recursive functions. We then link the code to Matita in order to compare the behavior and performance of the two implementations.

The code can be found here: https://github.com/LPCIC/matita. Currently the majority of the elaboration problems invoked by Matita compiling its arithmetic library are solved by the λProlog code as well. The majority of the problems that do not pass are due to the different heuristics implemented, in particular for the flex-flex case.

From the point of view of code size the situation is very good: only 323 lines of ELPI are required to turn the kernel in an elaborator, whereas the implementation of the elaborator in Matita requires about 2.500 lines.

We notice however that automatic backtracking makes the debugging of the ELPI version harder when the code follows an unexpected path. In the future we need to understand how to improve traceability of the code.

6. Conclusions and related works

In this paper we validated λProlog as a good programming 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 extended λProlog with constructs to turn goals into constraints and we added a CHR like language to manipulate the set of constraints. We implemented the proposed extensions in the ELPI system and used them to extend 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 (Wielemaker et al., 2012) and SICStus Prolog (Andersson et al., 1993).

While basing the implementation of the kernel of an interactive prover on a logical framework (for instance LF) is not new, little has been done to reuse the same technology for the elaborator component. For example the MMT (Rabe, 2013) "meta" system allows to define kernel rules in LF, but resorts to the Scala language for the elaborator. Isabelle lets one describe the axioms of a logic in the Simple Theory of Types 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, a formalism that does not need the term comparison algorithm to be based on narrowing. As a consequence, no need to extend this algorithm to encompass more rewriting is perceived, and no way to extend this algorithm is given to the user. The Dedukti (Boespflug et al., 2012) language allows to describe the axioms of a logic in λP modulo equational theories. Unfortunately, at the time of writing, the system lacks an elaborator. Felty and Miller (Felty and Miller, 1990) provided encodings of LF into λProlog, Snow at al. (Snow et al., 2010) turned type inhabitation in LF into provability in λProlog. It is unclear to us if that approach could be extended to obtain elaborators. In other work, Felty and Miller implemented interactive provers in λProlog directly, representing each derivation rule of the logic with a correspoding Harrop formula in λProlog (Felty and Miller, 1988), so that proof search in λProlog implements proof search in the logic. Proof terms and tactics can then be integrated as oracles that drive the proof search.

Describing the elaborator in terms of constraints was a choice first made in Agda version 2 (Norell, 2009) and more recently in Lean (de Moura et al., 2015a). Both systems implement they own, ad-hoc, constraint solver, the former system in Haskell, while the latter in C++. The approach we followed 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. At the time of writing Lean provides a meta-programming platform (Ebner et al., 2017) limited to definition of tactics. An extension to cover the elaborator seems to be planned but not implemented yet.

A intriguing line of future work is to use the Abella (Baelde et al., 2014) theorem prover to mechanically check the correctness of the λProlog code of our kernel and elaborator. Its natural continuation would be to describe in a fully formal way the semantics of CHR rules so that one could prove, possibly in Abella, that one's extensions to the elaborator do not undermine its correctness. If both these tasks can be completed successfully, then this can call into question the standard design of interactive provers: what is the point of having a kernel component if the elaborator and all its extensions are proved to only generate terms accepted by it?

References

J. Andersson, S. Andersson, K. Boortz, M. Carlsson, H. Nilsson, T. Sjöland, and J. Widen. SICStus Prolog User"s Manual. Technical report, 1993.

A. Asperti, W. Ricciotti, C. Sacerdoti Coen, and E. 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.

A. Asperti, W. Ricciotti, C. Sacerdoti Coen, and E. Tassi. *The Matita Interactive Theorem Prover*, pages 64–69. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-22438-6.

A. Asperti, W. Ricciotti, C. Sacerdoti Coen, and E. Tassi. A Bi-Directional Refinement Algorithm for the Calculus of (Co)Inductive Constructions. *Logical Methods in Computer Science*, 8(1), 2012.

D. Baelde, K. Chaudhuri, A. Gacek, D. Miller, G. Nadathur, A. Tiu, and Y. Wang. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 7(2):1–89, 2014. ISSN 1972-5787. .

B. Barras. *Auto-validation d'un système de preuves avec familles inductives.* Thèse de doctorat, Université Paris 7, Nov. 1999.

C. Belleanne, P. Brisset, and O. Ridoux. A Pragmatic Reconstruction of Lambda-Prolog. In *Journal of Logic Programming*, 1998.

Y. Bertot, G. Gonthier, S. Ould Biha, and I. Pasca. Canonical Big Operators. In *Theorem Proving in Higher Order Logics*, volume 5170/2008 of *LNCS*, 2008. .

M. Boespflug, Q. Carbonneaux, and O. 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, 2012.

Coq development team. *The Coq proof assistant reference manual.* Inria, 2017.

L. de Moura, J. Avigad, S. Kong, and C. Roux. Elaboration in Dependent Type Theory. *CoRR*, abs/1505.04324, 2015a.

L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. *The Lean Theorem Prover (System Description)*, pages 378–388. Springer International Publishing, Cham, 2015b. ISBN 978-3-319-21401-6.

G. J. Duck, P. J. Stuckey, M. G. de la Banda, and C. Holzbaur. The refined operational semantics of constraint handling rules. In B. Demoen and V. Lifschitz, editors, *Logic Programming*, pages 90–104, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-27775-0.

C. Dunchev, F. Guidi, C. Sacerdoti Coen, and E. Tassi. ELPI: fast, Embeddable, λProlog Interpreter. In *Proceedings of LPAR*, 2015.

C. Dunchev, C. Sacerdoti Coen, and E. 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. ACM, 2016. ISBN 978-1-4503-4777-8.

G. Ebner, S. Ullrich, J. Roesch, J. Avigad, and L. de Moura. A metaprogramming framework for formal verification. *Proc. ACM Program. Lang.*, 1(ICFP):34:1–34:29, Aug. 2017. ISSN 2475-1421.

A. Felty and D. Miller. Specifying theorem provers in a higher-order logic programming language. In E. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction: Argonne, Illinois, USA, May 23–26, 1988 Proceedings*, pages 61–80, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

A. P. Felty and D. Miller. Encoding a dependent-type lambda-calculus in a logic programming language. In *Proceedings of the 10th International Conference on Automated Deduction*, pages 221–235, London, UK, 1990. Springer-Verlag. ISBN 3-540-52885-7.

H. Geuvers and G. 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.

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*, volume 7998 of *LNCS*, pages 163–179. Springer, 2013. .

H. Herbelin. Type inference with algebraic universes in the Calculus of Inductive Constructions, 2005.

T. Libal and D. Miller. Functions-as-constructors Higher-order Unification. In *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016)*, pages 1 – 17, Porto, Portugal, June 2016. Delia Kesner and Brigitte Pientka. .

Z. 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.

Z. 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.

Z. Luo, S. Soloviev, and T. Xue. Coercive subtyping: Theory and implementation. *Inf. Comput.*, 223:18–42, Feb. 2013. ISSN 0890-5401.

A. Mahboubi and E. Tassi. Canonical Structures for the working Coq user. In S. Blazy, C. Paulin, and D. Pichardie, editors, *ITP 2013, 4th Conference on Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 19–34, Rennes, France, July 2013. Springer. .

F. Mazzoli and A. Abel. Type checking through unification. *CoRR*, abs/1609.09709, 2016.

D. Miller. Unification Under a Mixed Prefix. *J. Symb. Comput.*, 14(4):321–358, 1992.

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

D. Miller and G. Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 1st edition, 2012. ISBN 052187940X, 9780521879408.

D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51, 1991.

U. Norell. Dependently Typed Programming in Agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming*, AFP'08, pages 230–266. Springer-Verlag, 2009. ISBN 3-642-04651-7, 978-3-642-04651-3.

C. Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Supérieur*. Habilitation à diriger les recherches, Université Claude Bernard Lyon I, Dec. 1996.

X. Qi, A. Gacek, S. Holte, G. Nadathur, and Z. Snow. The Teyjus System – Version 2, 2015.

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.

D. M. Roberto Blanco, Zakaria Chihani. Translating between implicit and explicit versions of proof. In *CADE-26*, 2017.

C. Sacerdoti Coen and E. Tassi. Nonuniform Coercions via Unification Hints. In *TYPES*, pages 16–29, 2009.

Z. Snow, D. Baelde, and G. Nadathur. A meta-programming approach to realizing dependently typed logic programming. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP '10, pages 187–198, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0132-9.

B. Werner. *Une Théorie des Constructions Inductives.* PhD thesis, Université Paris VII, May 1994.

J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012. ISSN 1471-0684.