Implementing HOL in an Higher Order Logic Programming Language

Cvetan Dunchev, Claudio Sacerdoti Coen, Enrico Tassi

Implementing HOL in an Higher Order Logic Programming Language

Cvetan Dunchev Claudio Sacerdoti Coen
University of Bologna
{tsvetan.dunchev, claudio.sacerdoticoen}@unibo.it

Enrico Tassi
INRIA Sophia-Antipolis
enrico.tassi@inria.fr

Abstract

We present a proof-of-concept prototype of a (constructive variant of an) HOL interactive theorem prover written in a Higher Order Logic Programming (HOLP) language, namely an extension of λProlog. The prototype is meant to support the claim, that we reinforce, that HOLP is the class of languages that provides the right abstraction level and programming primitives to obtain concise implementations of theorem provers. We identify and advocate for a programming technique, that we call semi-shallow embedding, while at the same time identifying the reasons why pure λProlog is not sufficient to support that technique, and it needs to be extended.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

Keywords λProlog, HOL, Higher Order Logic Programming, Constraints

1. Introduction

What are the programming paradigms and the programming languages better suited for the implementation of interactive theorem provers? Better suited here means providing high level primitives at the good level of abstraction, relieving the programmer from the burden of reimplementing the basic mechanisms.

Every (interactive) theorem prover has to:

1. manipulate expressions with binders up to α-conversion, and it needs to implement substitution. Binders are ubiquitous: they occur in formulae, but also in both procedural and declarative scripts (new hypotheses are named and have an associated scope), and in proof objects as well.
2. implement automatic proof search, either in the small or in the large, that requires to explore the search space via backtracking. Because the search space can be very large, the programmer needs way to control backtracking and direct the exploration.
3. manipulate incomplete data structures, i.e. data structures having parts not specified yet. Moreover, invariants have to be enforced on the data structures and lazily checked when the data structures get instantiated. Examples of incomplete data occurs in formulae (omission of types to be inferred), sequents (omission of terms to be identified later, e.g. writing X for a yet unknown witness of an existential statement), and proof objects (an incomplete proof has a proof object containing an hole).

The three features discussed above also interact in complex ways. For example, renaming bound variables in a term that contains metavariables (i.e. omitted sub-terms) must record the renaming as an explicit substitution applied to the metavariable to resume the renaming when the metavariable will be instantiated. Or, backtracking must be aware of dependencies between subgoals due to sharing of metavariables in order to split the subgoals into independent clusters of goals: once a cluster is solved, backtrack that solution is useless because search spaces are orthogonal.

In a series of papers from the 90s (Felty and Miller 1988), (Felty 1993), (Appel and Felty 1999), Amy Felty already advocated Higher Order Logic Programming (HOLP) as the programming paradigm best suited for the three tasks above. λProlog, the flagship of HOLP languages, manipulates syntax with binders, relieving the programmer from problems due to renaming and substitution; being a logic language it has backtracking for free; it allows to control backtracking via the usual Prolog's cut ("!") whose pragmatics is well known, even if it does not fit the logical semantics of the language; finally the language uses metavariables that range over both data and functions, and the complex implementation of λProlog takes care of all interactions between binders (λ-abstraction) and metavariables, addressing the problem of higher-order unification under mixed prefixes (Miller 1992).

Despite the push by Amy Felty, none of the interactive theorem provers in use is implemented in HOLP. With the exception of Mizar, implemented in Pascal for historical reasons, and the new Lean system, implemented in C++ for efficiency, all other systems (Coq, Isabelle, Agda, Matita, PVS, etc. (Wiedijk 2006)) are implemented in either Haskell, an ML variant or a Lisp flavor.

In general, functional languages provide a good compromise between performance and high-level coding, and algebraic data types can naturally encode syntax without binders. In particular, ML was born as the language for implementing the LCF theorem prover, one of the first theorem provers of the world. Nevertheless, these languages solve none of the three problems above:

1. with the exception of FreshML, that has also not been used for interactive theorem proving yet, functional languages do not abstract the details about binders and the user must encode bound variables via De Bruijn indexes (like in Coq and Matita) or it must implement α-conversion and substitution carefully (like in HOL Light). De Bruijn indexes allow the code to be very efficient, but code that handles them is very error prone.
2. the ML and Haskell families do not provide backtracking for free. A limited form of backtracking can be implemented in ML via exceptions: an exception can be raised to backtrack to a previous state, but once a function call is over it is not possible

to backtrack into it any longer. Continuations based monads to implement backtracking (plus state to handle metavariables and failure) exists (Spiwack 2010b), but the code is quite complicated and it has efficiency problems unless it is manually optimized (private communication with the authors).

3. managing metavariables, their interaction with binders and (restrictions of) higher order unification requires a lot of fragile and complex code (e.g. $\approx$ 3100 lines of OCaml code for Matita, compared to the $\approx$ 1500 lines for the kernel that implements β-reduction, conversion and the inference rules of the logic).

The situation described above is bad in two respects.

1. The code that deals with the three problems is essentially logic-independent, but it requires to be re-implemented when one wants to experiment with a new logic or implement a new system. With the exception of the monad cited above, it also seems hard to encapsulate the boilerplate code into a reusable library: a real extension of the programming language is necessary.
2. The code of the systems becomes very low-level, having to deal with all kind of intricacies due to binders and metavariables. Therefore it is hard for external programmers to contribute to the code base, for example to implement new domain specific tactics. The result is that these systems often implement in user space a second programming language, exposed to the user to write tactics, that takes care of binding, metavariables, backtracking and its control. For example, LTac (Delahaye 2000) is such a programming language for Coq, that also supports several other mini-languages to let the user customize the behavior of the system (e.g. to declare canonical structures (Mahboubi and Tassi 2013), used to provide unification hints (Asperti et al. 2009)). Not only the system becomes more complex because of the need to provide and interpret a new programming language on top, but its semantics is problematic: the behavior of the system becomes the combination of pieces of code written in multiple languages and interacting in non trivial ways. Static analysis of this kind of code is out of reach.

Adopting an HOLP language seems a promising idea. First of all the boilerplate code is pushed once and for all in the interpreter of the language, becoming decoupled from the programming logic of the theorem prover. Therefore the code of the theorem prover becomes much more clean, and very close to the inference rules it implements. Secondly, because of the decoupling, it becomes possible to experiment with alternative implementations of the HOLP language, e.g. substituting De Bruijn levels for the indexes (like in (de Bruijn 1978)) or profiling the implementation of the language on different applications. Thirdly, and most importantly, there is no more any need for ad-hoc languages in user space: the users that want can directly contribute to the code base of the theorem proving ignoring all the gory details.

Nevertheless, as we said, no system has been implemented in λProlog, despite the interest raised by the work of Felty (her paper (Felty and Miller 1988) has more than 150 citations). One reason is due to performance: logic languages are hard to optimize, and λProlog is orders of magnitudes harder than Prolog not only because of binders and higher order unification, but also because it is higher order (one can pass predicates around) and it allows a primitive (logical implication `P => Q`) to temporarily augment the code of the program (with `P`, while proving `Q`) in a lexically scoped way, making some well known Prolog static analyses and optimizations hard. Moreover, λProlog had for some time only very slow implementations, until Teyjus was born (Nadathur and Mitchell 1999) after the work of Felty. Recent work by the authors also showed that the performances of Teyjus are not so good (Dunchev et al. 2015), considering that Teyjus compiles the code to an extension of the Warren abstract machine instruction set, while the ELPI interpreter of the authors that attempts no compilation is able to run consistently faster than Teyjus (Section 6 of (Dunchev et al. 2015)).

In this paper we push seriously the idea of implementing in the ELPI variant of λProlog an interactive prover, inspired by HOL Light, but for a constructive variant of the logic. The ongoing experiment aims to come up with implementation guidelines, that differ from the ones of Felty, and to quantify the loss in term of performance w.r.t. implementations in functional languages.

We also note that the expertise we acquired implementing Matita was used already in the implementation of ELPI. Indeed, we essentially had to split out the code of Matita that deals with the three problems above, and figure out how to integrate it in an interpreter for a logic language. In the long term, we would like to scale the HOL experiment to a new implementation of Matita/Coq (or at least its logical core, that excludes the user interface).

The methodology we identified, presented in Section 4.1, requires significant extensions of λProlog, that we are implementing in ELPI and that are briefly described in the paper.

As a final remark, the interest of the work, that is still on-going, is not to obtain a new competitive implementation of HOL, that would require several men years and could be of limited interest. In particular, we only developed the system so far to the point where all the parts of a typical HOL implementation are presents to judge the feasibility and economy of the implementation.

2. λProlog in a nutshell

λProlog extends the core of Prolog in two directions.

1. it allows in the syntax λ-abstractions (written `x\p` for $\lambda x.p$), applications (written `p q` like in λ-calculus and unlike in Prolog) and metavariables both in argument and in head position (e.g. `iter [X|XS] F [Y|YS] :- F X Y, iter XS F YS.` where the metavariable `F` is to be instantiated with a binary predicate like `(x\y\y=x+x)`)
2. it generalizes Horn clauses to Higher Order Hereditary Harrop Formulae (HOHHF) clauses, whose grammar is

$$\begin{array}{l} H ::= x\ t_1\ \ldots\ t_n \mid H \wedge H \mid \forall X.H \mid \exists x.H \mid G \Rightarrow H \\ G ::= x\ t_1\ \ldots\ t_n \mid X\ t_1\ \ldots\ t_n \mid G \wedge G \mid G \vee G \\ \qquad \mid \forall x.G \mid \exists X.G \mid H \Rightarrow G \\ H, G \subseteq t ::= x \mid X \mid tt \mid x\backslash t \end{array}$$

where H ranges over HOHHF, G ranges over goal formulae, x are constants/universally quantified variables, X are existentially quantified variables and t are the higher order terms of the language. A λProlog program is a list of HOHHF. To run a λProlog program the user submits a query G that is automatically proved using the clauses in the program.

The (operational) semantics of the connectives that occurs respectively in goal formulae G / in HOHHF H is given by the introduction/elimination rules of the connectives in natural deduction for intuitionistic higher order logic. In particular:

- the goal $H_1 \wedge H_2$ is turned into the two goals H_1 and H_2, copying the program (i.e. the set of clauses/assumptions)
- the goal $H_1 \vee H_2$ is turned into the goal H_1; in case of failure of the proof search, the goal is backtracked to be H_2
- $H \Rightarrow G$ assumes H, turned into a clause, to prove G
- $\forall x.G$ introduces a fresh variable y and proves G after replacing x with y in G
- $\exists X.G$ introduces a fresh metavariable Y and proves G after replacing X with Y. Later Y can be instantiated with any term whose free variables were in scope for $\exists X.G$

- the goal t where t is atomic is proved unifying t with the (hereditary) conclusion of one of the program clauses, possibly opening new goals as well (see case $G \Rightarrow H$ below)
- assuming $H_1 \wedge H_2$ means assuming both Hs independently
- assuming $\forall X.H$ means assuming an infinite number of copies of H for all fresh metavariables Y substituted for X in H. Concretely, the copy and substitution is performed lazily.
- assuming $\exists x.H$ means generating a new fresh constant y and assuming H after replacing x with y
- assuming $G \Rightarrow H$ means assuming that H holds under the assumption that G holds as well. When the clause is used, the new goal G will be added to the set of open goals to be proved

In concrete syntax we will write `sigma X\t` for $\exists X.t$, `pi x\t` for $\forall x.t$, `a => b` or `b := a` for $a \Rightarrow b$, `a,b` for $a \wedge b$ and `a;b` for $a \vee b$. Moreover, all free uppercase/lowercase variables in a program clause are implicitly universally/existentially quantified globally/locally (like in Prolog).

`sigma` in clause positions are not in "standard" λProlog and they are accepted by our ELPI interpreter, but not from Teyjus. However, Teyjus implements a module system (Nadathur and Tong 1999) that allows to declare variables local to the list $H_1, \ldots, H_n$ of clauses of the module. The same behavior can be obtained in ELPI with (`sigma x` $H_1, \ldots, H_n$) or with the equivalent syntactic sugar { `local x.` $H_1.$ ... $H_n.$ } (where the curly braces are used to delimit the scope of the existential quantifier, i.e. the local declaration). Declaring local constants is the only and fundamental mechanism in λProlog to restrict the trusted code base.

The other major difference between the ELPI version of λProlog and the "official" one of Teyjus is that we do not enforce any type discipline, nor we try to statically enforce the restriction to HOHHF. Errors at run-time (e.g. invoking a list as a predicate) result in abortion of the program execution. On the other hand, it is for example possible to read from the terminal a query and execute it, which is prohibited by Teyjus because the correctness of the query cannot be statically enforced.

To illustrate a typical λProlog example, in Table 1 we show the code of a type-checker for closed terms of the simply typed lambda-calculus. We use the infix constant `'` to encode application, the unary constant `lam` for lambda-abstraction, and the infix constant `-->` for the function space type. The former is to be applied to a meta-level lamdba-abstraction in the spirit of higher-order abstract syntax. For example, the term $\lambda x.xx$ is encoded as `lam x \ x ' x`. Note that, in the concrete syntax, parentheses around lambda-abstractions are not necessary in λProlog when the abstraction is the last argument of an application. I.e. `lam x \ x ' x` is to be read as `lam (x \ x ' x)`.

Observe that the code of the type-checker is really minimal and in one-to-one correspondence with the two derivation rules one writes usually on paper. However, there are some differences as well. In particular, the typing judgement `term T TY` ("`T`" is a term of type "`TY`") does not mention any context Γ. Instead, in the rule for lambda-abstraction, the hypothesis that the fresh variable `x` has type `A` is directly assumed and put temporarily in the program code using logical implication `=>`. For example, the query `term (lam y y ' y) TY` will match the second clause and trigger the new query `term (x ' x) B` after having instantiated `TY` with `A --> B` where `x` is fresh (because introduced by `pi`) and of type `A`.

Finally, we will use the cut predicate of Prolog `!` with the same non-logical semantics. We say that a predicate does not have a logical semantics when it breaks the commutativity rule of conjunction or, equivalently, commutation of the semantics with instantiation.

```
1 term (M ' N) B :- term M (A --> B), term N A.
2 term (lam F) (A --> B) :- pi x\ term x A => term (F x) B.
```

Table 1. A type-checker for simply typed lambda calculus.

3. HOL in a nutshell and its ML implementation

HOL is a variant of Church's simple theory of types. Its precise syntax and semantics can be found in (Gordon and Pitts 1994) even if we will implement a constructive variant of it, and we will refer for inspiration to the HOL Light classical version and implementation. Types and terms have the following syntax:

$$T ::= \alpha \mid u\, T_1 \ldots T_n \mid T \to T$$
$$t ::= x_T \mid c_T \mid \lambda x_T.t \mid t\ t$$

where u ranges over type constructors (constants when $n = 0$) and α ranges over type variables. Contrarily to System-F, HOL has no type abstractions/applications. However, it is possible to assign to constants schematic types containing type variables $\alpha, \beta, \ldots$. E.g. the append function `@` is typed `list` $\alpha \to$ `list` $\alpha \to$ `list` α.

Variables x_T and constants c_T carry explicitly their type T and they are equal iff both the name and the types are. In particular, each occurrence of a polymorphic constant, carrying its type, explicitly carries an instantiation for every type variable occurring in its type. For example $\texttt{@}_{\texttt{list nat}\to\texttt{list nat}\to\texttt{list nat}}$ implicitly encodes the instantiation $\alpha := \texttt{nat}$. In System-F the constant occurrence would be applied to the type: `@ nat`.

HOL systems assume the existence of two predefined types, one for propositions `bool` and one for individuals `ind` (also written o and ι in the literature). Logical connectives and quantifiers are just single out constants. For example, conjunction is given the monomorphic type `bool` $\to$ `bool` $\to$ `bool` and universal quantification the polymorphic type scheme ($\alpha \to$ `bool`) $\to$ `bool`. The usual trick of higher-order syntax is used to reduce all binders to λ-abstraction: $\forall x : \texttt{nat}.P$ is represented as $\texttt{forall}_{(\texttt{nat}\to\texttt{bool})\to\texttt{bool}}(\lambda x.P)$. In the case of the HOL Light variant, the only constant that is initially assumed to exist is polymorphic equality $=_{\alpha\to\alpha\to\texttt{bool}}$, that also doubles for coimplication when $\alpha = \texttt{bool}$.

The derivation system of the logic is made of two judgements, one for typing terms and the other for proving formulae in contexts (see (Harrison 2009b) for the 10 derivation rules of the HOL Light implementation). Two examples of such rules are

$$\frac{\vdash p : \texttt{bool} \qquad \Gamma_1, p \vdash q \qquad \Gamma_2, q \vdash p}{\Gamma_1, \Gamma_2 \vdash p = q}\ \text{DEDUCT\_ANTISYM\_RULE}$$

$$\frac{\Gamma \vdash p}{\Gamma\sigma \vdash p\sigma}\ \text{INST\_RULE}$$

In the HOL Light implementation, the check for p being a boolean in the first rule is post-poned to the time the hypothesis is used (the HYP rule, here not shown). The second rule is used to apply the substitution σ that maps free variable names to terms. Its application must avoid variable capturing.

An HOL theory is a list of declarations of types, constants and axioms, the latter being just formulae assumed to hold.

The implementation of an HOL systems is rooted in the principle of conservative extensions: in order to introduce a new constant or a new type that satisfy certain axioms, the user must prove the existence of such a constant/type. We only discuss here the rule for introducing new definitions to explain a kind of complexity that involves free and bound names.

To introduce a new (polymorphic) constant c_T, the user must give a term t of type T such that all free type variables contained in t also occur in T. After checking the conditions and verifying that

no constant named c is already defined in the current theory, the latter is extended with the new constant c_T and the axiom $c_T = t$.

The condition on the free type variables of t avoids logical inconsistency and would not be necessary in System-F where every occurrence of c would be explicitly applied to every free type variable that occurs in t. In HOL, instead, only the type instantiations for the type variables that occur in the type of c can be recovered from the type of c. Therefore, if the condition is violated, c_T could be rewritten via the axiom to an instantiation of t that still contains free type variables. Example: $c_{\texttt{bool}} = !x_\alpha y_\alpha.\ x_\alpha = y_\alpha$ that from $c_{\texttt{bool}} = c_{\texttt{bool}}$ via two instantiations would imply $!x_{\texttt{bool}} y_{\texttt{bool}}.\ x_{\texttt{bool}} = y_{\texttt{bool}} \iff !x_{\texttt{unit}} y_{\texttt{unit}}.\ x_{\texttt{unit}} = y_{\texttt{unit}}$ i.e. `false` = `true`.

3.1 HOL (Light) in ML

HOL Light (Harrison 2009a) is an extremely lightweight implementation of HOL in ML that reuses the ML toplevel, following the tradition started by Milner (Gordon et al. 1979). The key idea is the possibility in ML of encoding *private data types* (Leroy et al. 2004), i.e. data types whose values can be inspected via pattern matching outside the module that defines them, but that cannot be directly constructed outside the module. Instead, from outside the module, it is only possible to invoke a function, implemented in the module, that returns an inhabitant of the type.

Private types can be used to enforce invariants: the functions of the module only builds terms that satisfy the invariants. Therefore, all inhabitants built outside the module are correct by construction, and the trusted code base for the maintenance of the invariants is lexically delimited by the module.

HOL Light implements three private types corresponding to the three judgement of the logic: 1) the type of well typed terms t such that there exists a T s.t. $\vdash t : T$; 2) the type of theorems, i.e. pairs context Γ, formula F such that $\Gamma \vdash F$; 3) the type of well-formed types. Theories could also have been implemented as a third private type (like what Coq does), but they are actually hidden inside the module that defines the type of (well typed) terms.

In particular, every inference rule of HOL corresponds to a function over theorems exported from the "kernel" module of HOL.

```
val mkComb : term -> term -> term
val DEDUCT_ANTISYM_RULE : thm -> thm -> thm
```

for example correspond respectively to the typing rule for application $t_1 t_2$ and to the inference rule shown on page 3. The function

```
val dest_thm : thm -> term list * term
```

allows to inspect a theorem to recover what has been proved. Of course, it does not have an inverse function that would correspond to claiming that $\Gamma \vdash p$ is a theorem without actually proving it.

An important feature of this approach, that dates back to LCF and puts it apart from the Curry-Howard approach used for Coq and other systems for dependently typed languages, is that *no proof object* needs to be stored. Indeed, an inhabitant of `thm` really is (in HOL Light) just a pair hypotheses/conclusion.

The functions implemented in the kernel already allow to build proofs in bottom-up style. Quite frequently, however, the user would like to mix the bottom-up and the top-down styles, and he would like to be able to use metavariables for yet unknown formulae during a top-down proof.

Again following Milner's LCF, HOL Light implements outside the kernel a mechanism for top-down proofs based on the notion of tactic. Simplifying, the tactic type is defined as

```
type tactic = sequent ->
   sequent list * instantiation * metavariables *
   justification
type justification = instantiation -> thm list -> thm
```

Ignoring the instantiation and metavariables, a tactic is a function that takes in input the sequent to be proved and return a list of new sequents to be proved and a justification. When the new sequents are proved, a list of theorems is obtained. The justification, fed with this list of theorems, returns the theorem for the initial sequent. In other words, each tactic implements a top-down reasoning step by producing the list of new subgoals plus the "inverse" bottom-up proof. When those are composed, the theorem is proved. Variables recorded as metavariables, i.e. omitted information, and instantiations for them complete the picture. We observe that:

- As we already noticed in (Sacerdoti Coen et al. 2007), LCF tactics have a few drawbacks. They only allow to reason locally on one goal, because they do not take in input the global proof state. In case of mutual dependencies between goals due to metavariables, it becomes impossible to reduce the search space using global reasoning. Moreover, it is not possible to interrupt the execution of a tactic in the middle, for example to observe the intermediate states. The tinycals introduced in (Sacerdoti Coen et al. 2007), which can now be implemented in Coq as well after (Spiwack 2010a), are a better solution but they require a first class representation of the entire proof state.
- During a top-down proof, the system temporarily records a proof object in the form of a justification, that is lazily consumed as soon as the corresponding subproof is completed (all leaves are closed). We would like to avoid this if possible.
- Every bottom-up inference rule — primitive or derived — performs some checks that are performed twice when the rule is turned into a top-down tactic. For example, the top-down tactic for the `DEDUCT_ANTISYM_RULE` rule must analyse the sequent in input to verify that it is of the form $\Gamma \vdash p = q$, yielding the two new goals $\Gamma, q \vdash p$ and $\Gamma, p \vdash q$. Later the justification `DEDUCT_ANTISYM_RULE` needs to pattern match the list of theorems in input against $\Gamma_1, q \vdash p$ and $\Gamma_2, p \vdash q$. The "duplicate" inverse checks are necessary because the kernel does not trust the outer parts of the system, but are still somehow undesired.

4. Requirements and Term Encoding

We begin the description of our implementation of (a constructive variant of) HOL in ELPI, our λProlog interpreter. The code is available at `http://lpcic.gforge.inria.fr/system.html?content=systems`. The implementation is meant as an experiment to evaluate if Higher Order Logic Programming can indeed be employed concisely and effectively for the implementation of interactive theorem provers. In this experiment, we take inspiration from HOL Light and we set the following requirements:

1. The system must have a small trusted code based, henceforth called the "kernel". Bugs outside the kernel must not allow to conclude that a non-tautological formula F is a theorem.
2. The LCF architecture does not need to record proof objects. Therefore we do not want to record proof objects as well.
3. The implementation must use genuine Higher Order Logic Programming techniques. Binders have to be handled exploiting the binding machinery of the language. Object level metavariables must be represented by the metavariables of the language.
4. Tactics must be able to inspect the global proof state.

The third requirement rules out the encoding of the HOL Light in λProlog. A mechanical encoding is possible because the language is Turing complete, but it would not benefit from the features of the language.

The second and fourth requirement rule out the technique used by Amy Felty and, more recently, by the group of Dale Miller (Miller 2014). Felty and Miller encodes the judgements of the object logic via a λProlog predicate that defined by recursion

over a proof object (Felty) or a proof certificate (Miller). For example, the proof object can decide to apply the transitivity rule for application to a goal where multiple rules apply. A proof certificate is similar to a proof object, but it can avoid recording all the needed information or encode it at a different level of detail. Therefore, a proof certificate may be non syntax directed, in the sense that multiple rules can be tried according to the same certificate, triggering backtracking if needed.

In both cases, the system can be simply turned into both an automatic or an interactive theorem prover. In the first case, it is sufficient to replace the certificate with a metavariable: the λProlog predicate will then work in a generative way, blindly enumerating all possible proof objects (and potentially diverging because of depth-first search, that can be replaced with iterative deepening to keep completeness). To obtain an interactive prover, the certificate is replaced with the request to the user to type the next rule to be tried. The user interaction can also print the sequent to be proved, but not the other open branches of the proof, that are kept at the meta-level in the and/or tree of the λProlog interpreter.

Finally, we cannot implement the kernel using a private type like in ML because Logic Programming rules out private types. To understand why, remember that every predicate in a (pure) logic programming language is invertible. Therefore it is not possible to implement a type with a predicate to reveal the data, but preventing its use to build the data as well. In particular, the predicate `dest_thm` corresponding to the function with the same name in HOL would allow to feed it any pair context-formula to obtain a theorem without any further control.

The only way to protect part of the code in λProlog is to existentially quantify a predicate to be kept secret to the rest of the code. Our kernel implements three predicates, one for recognizing well-typed types, one for the typing judgement and one for derivability, that are kept local. In order to fulfill requirement number 4, derivability will work on list of sequents (the proof state) in place of a single sequent. Finally, to avoid the repetition of code and checks in the HOL approach, we only focus for now on top-down proofs.

We detail now the implementation of the kernel.

4.1 Semi-shallow encoding of terms

Before starting, we first need to choose a syntactic representation for both terms and types that respect requirement 3, that rules out deep encodings. A deep encoding for simply typed λ-calculus with metavariables would look like this and be very similar to the data type Coq and Matita are build around:

$$t ::= var\ \texttt{NAME} \mid app\ t\ t \mid lam\ \texttt{TYPE}\ \texttt{NAME}\ t \mid meta\ \texttt{NAME}$$

Assuming `NAME`s to be either strings or De Bruijn indexes, a deep encoding forces the implementation of α-conversion, lifting, substitution, instantiation of metavariables, etc.

At the other side of the spectrum we find the shallow encoding for closed terms: we just encode object level terms with language level terms, object level metavariables with λProlog metavariables and so on. Instantiation of metavariables as well as α-conversion and β-reduction comes for free. However, it becomes impossible to implement — without extra-logical extensions of the language — both the derivation and the typing judgements. For example, the code `type (F X) B :- type X A, type F (A --> B).` does not do what it seems to do, because applied to the query `type (f a) Z`, it yields the HO unification problem `F X = f a` that admits multiple solutions like `F := x \ f a, X := 0`.

We therefore propose what we call the *semi-shallow* encoding:

$$t ::= t\ \texttt{'}\ t \mid lam\ \texttt{TYPE}\ F$$

where `'` is just an infix constructor, `F` is a λProlog function from terms to terms and object level metavariables are encoded via language level metavariables. For example, $\lambda x : o.X\ x$ is encoded as `lam o x \ X x`. Ignoring the issue about metavariables, the semi-shallow encoding was the one already used in Table 1, that essentially shows the typing judgement of our implementation of HOL Light, which includes a few other clauses for primitive constants, like `term eq (A --> A --> prop)`.

For simply typed λ-calculus (plus type variables) type checking is decidable and the typing judgement of Table 1 is syntax directed. Therefore HOL never asks the user to prove that a term has a certain type: type checking/inference is run fully automatically.

Because we can inject λProlog variables in place of types of bound variables, our type checking judgement already doubles as type inference. E.g. `term (lam A x \ x + 1) B` yields `A := nat, B := nat`.

5. Language features for semi-shallow encodings

Manipulating incomplete terms is particularly brittle in logic languages. In particular unification may instantiate meta variables "by mistake". We make two examples to cover all aspects of the problem: first parsing and pretty printing, then typing. We list here only a few clauses of these programs for the sake of conciseness.

```
pp (X <=> Y) (eq " prop ' A ' B) :- pp X A, pp Y B.
pp (X  =  Y) (eq " _ ' A ' B)    :- pp X A, pp Y B.
```

Following the spirit of Prolog the `pp` predicate relates pretty printed expressions and their low level syntactic representations. Thanks to the declarative semantics of λProlog such relation can be used in two "directions", to both print and parse. Now imagine the system has to print `(eq " TY ' X ' Y)`, that is an equation where both sides are metavariables. Unifying such term with the second argument of the first clause would, among other things, pick `prop` as the type `TY`. Printing a term would leave a visible trace in the input term. Even worse, the recursive calls over `X` and `Y` would instantiate them indefinitely: the program would enter generative mode. This problem is well known in Prolog, and various non logical predicates like `var` are provided to handle this situation. This very same problem also arises is in the implementation of automatic tactics, that have to be able to chooses the next step by inspection of the goals. Of course, inspecting the goal should leave no visible trace.

The situation is analog for typing.

```
term (lam A F) (A --> B):- pi x\ term x A => term (F x) B.
```

Imagine the `term` program is run on an incomplete input like `(lam nat n\X n) T`. The program, augmented with the clause `term n nat`, wound run on the sub goal `term (X n) B`. Such clause, as well as any other one for `term`, would apply. As a result the meta variable `X` would be instantiated by `n`. What provers implemented in ML, like Coq, do is to keep a data structure that assigns a type to each metavariable, typically called Σ. When an incomplete term is found the algorithm stops and check that the (occurrence of) the metavariable has the type declared in Σ (1). In a different part of the code, when a metavariable is instantiated, the system checks that the instantiation is compatible with type declared in Σ (2). The reading of this behavior in terms of logic programming points toward two distinct, but complementary mechanisms.

First, a goal like `term (X n) B` has to be *delayed*, a very well known mechanisms justified by the commutativity of conjunction: postponing a goal does not change the meaning of the program. Actually one can see the λProlog goals as a set of *constraints*, all to be solved at some point. Most Prolog system provides such mechanism. Teyjus has it too but only for internal use, i.e. delay hard HO unification problems falling outside the L_λ fragment. Goals are delayed on a specific metavariable and are re-scheduled

when such metavariable gets instantiated. This mechanism avoids entering generative mode and faithfully models (2).

Last, when two goals, i.e. constraints, are delayed on the same meta variable one could add to the set of goals an equation between the two types. The manipulation of the set of delayed goals is again a mechanism that accompanies many Prolog implementations in the for of *constrain handling rules*, or CHR (Frhwirth 1998). Teyjus does not provide such mechanism.

5.1 Modes and matching

To tackle this class of problems we introduce in ELPI the concept of *modes*. For example the pp predicate can be declared having the two desired modes with the following syntax:

```
mode (pp i o) as parse, (pp o i) as print.
pp (X <=> Y) (eq " prop ' A ' B) :- pp X A, pp Y B.
pp (X  =  Y) (eq " _ ' A ' B)    :- pp X A, pp Y B.
pp A A.
```

The `parse` relation is obtained from `pp` by marking its first argument as an input one. Conversely for `print`. Input arguments are *matched* against the goal, i.e. no unification variable present in the goal is instantiated as part of the unification process. T he catch all clause. `pp A A`. Such clause prints/parses each meta variable with itself, as well as prints any term with no associated notation as itself. Used in conjunction with λProlog implication, the `pp` predicate can serve as a mechanism for scoped, printed and parsed, notations, like names for bound variables. E.g.

```
?- pi w\ pp delta w =>
      print Nice (eq " nat ' w ' 0),
      parse (delta = 0) (Ugly w).
% Nice = (delta = 0)
% Ugly = x\ eq " T ' x ' 0
```

Modes alone are not sufficient to solve the problems we described for the `term` predicate, but make the declaration of a constraint quite compact:

```
mode (term i o).
term (lam A F) (A --> B) :- pi x\ term x A => term (F x) B.
term (?? as K) T :- constraint (term K T) K.
```

The `??` symbol, only available for input arguments, can be used to recognize flexible terms. The `(... as K)` syntactic sugar is reminiscent of ML's pattern matching syntax to name sub expressions, in this case the metavariable. `constraint` delays the current goal, declaring it as a new constraint.

Note that also dynamic clauses, like the `term x A` one added by the `lam` rule, are affected by the mode directive and as a consequence the higher order program `term` never enters generative mode.

5.2 Towards a CHR extension of λProlog

As a companion mechanism we intend to provide the possibility to declare constrain handling rules with the following syntax.

```
constraint term {
 rule [ term (?? as X) T1 ] [ term (?? as X) T2 ] (T1 = T2).
}
```

The first two arguments are lists of goals to be matched, the second one to be removed. The third argument is a goal to be added. The rule above hence reads: if two `term` constraints are on the same meta variable, remove the second one and add the goal `T1 = T2`.

Rules like this one are typically justified by the meta theory of the object level language. In particular this one corresponds to the uniqueness of typing.

λProlog goals live in a context (the dynamic part of the program). Making this component available to the constraint handling rules lets one implement more sophisticated propagations. For example one may check that not only the ground instances of meta variables are well typed, but also that all occurrences of the metavariable are.

```
constraint term ?- term {
 rule [ G ?- term (?? K R) _ ] [ D ?- term (?? K S) _ ] CC :-
   aux R G S D CC.

 aux [X|XS] G [Y|YS] D (TX = TY, CC) :-
  (G => term X TX), (D => term Y TY), aux XS G1 YS G2 CC.
 aux [] [] [] [] true.
 aux _ _ _ _ false.
}
```

For both performance and usability reasons one has to specify which components of a program context he is interested in. Hence the `term ?- ...` part saying that the context items we are interested in are only the ones about the `term` relation. Note that contexts like `G1` can be used in conjunction with the implication to run a goal in a custom context in the right hand side of the constraint handling rule. Finally `R` and `S` are the list of context variables visible to `K`, i.e. introduced by the `pi` binder in the `lam` rule for the `term` predicate.

It is also worth mentioning that λProlog goals should be unified taking into account injective renaming of bound variables, known as *equivariate unification*, a complex and computationally expensive algorithm. For this and other reasons, the CHR extensions to ELPI are still work in progress.

6. The kernel in λProlog

Our kernel starts implementing the well-formedness check for types and the typing/type inference judgement for terms, already discussed in Section 5.1. The last judgement implemented kernel is the derivation judgement in top-down style. It is encoded via three predicates, `prove`, `loop` and `thm`.

The predicate `loop SEQS CERTIFICATE` holds when the list of well-typed sequents `SEQS` can all be proved following instructions from the certificate `CERTIFICATE`.

The predicate `prove G CERTIFICATE` holds when the formula `G` has type `prop` and moreover `loop [(seq [] G)] CERTIFICATE` holds, where `seq [] G` represents a sequent whose context is empty and whose conclusion is the closed formula `G`.

```
prove G TACS :-
 (term G prop, !
 ; parse PG G, $print "Bad statement:" PG, fail),
 loop [ seq [] G ] TACS.
```

The code executed when type-checking fails provides a nice error message to the user. We call the type of propositions `prop` in place of `bool` because our logic is intuitionistic.

The code for `loop` is the following:

```
loop [] CERTIFICATE :- end_of_proof CERTIFICATE.
loop [ SEQ | OLD ] CERTIFICATE :-
 next_tactic [ SEQ | OLD ] CERTIFICATE ITAC,
 thm ITAC SEQ NEW,
 append' NEW OLD SEQS,
 update_certificate CERTIFICATE ITAC NEW NEW_CERTIFICATE,
 loop SEQS NEW_CERTIFICATE.
```

The predicate calls the predicates `end_of_proof`, `next_tactic`, `update_certificate` that are untrusted and defined outside of the kernel. Their intended usage is to extract from the certificate the next rule to use (a primitive inference rule or a user-defined tactic), and to update the certificate according to the new proof state. The mechanism is reminiscent of the one in (Chihani et al. 2013).

Predicate `loop` succeeds when there are no more sequents to prove, informing the certificate via `end_of_proof` (or verifying that the certificate agrees). Otherwise the predicate feeds the global proof state (the list of sequents) to `next_tactic` that, according to the certificate and the proof state, returns a rule `ITAC` to apply.

Then `loop` invokes `thm ITAC SEQ NEW` that applied the rule `ITAC` to the goal `SEQ`, reducing in top-down style a proof of it to a proof of the list `NEW` of new sequents (the hypotheses of the rule). Each primitive inference rule of HOL (Light) is implemented by `ITAC`. For example, `DEDUCT_ANTISYM_RULE` (that we simply cal `k`) is implemented by

```
thm s (seq Gamma (eq ' P ' Q))
 [ seq (P :: Gamma) Q, seq (Q :: Gamma) P ]
 :- reterm P prop.
```

As an invariant, we already know that `eq ' P ' Q` is well typed. The `reterm P prop` judgement is a variant of `term` that performs less checks knowing that `P` is already well typed. Note that, of the three premises of `DEDUCT_ANTISYM_RULE`, the typing one is automatically discharged because retyping is decidable and efficient. The same happens in HOL Light.

Once `thm` returns, the list of new sequents is inserted in the proof state and the certificate is updated via `update_certificate` before entering the next loop iteration.

All predicates involved but `prove`, i.e. `loop`, `thm`, `term`, `reterm` are declared `local` to the kernel and the code is written carefully to avoid leaking them. Indeed, if such a predicate leaked out, it would be possible for a malicious user to augment the clauses about the predicate, breaking logical correctness (e.g. executing `thm [seq [] false] [] ==> prove false X`). The problem of avoid leaking is clearly undecidable, but a simple static analysis can verify that our kernel does not leak (essentially because the predicates are only used in the kernel in head position and never in argument position and so they cannot be leaked out). The static analysis has not been implemented yet.

Compared to the HOL Light kernel, our kernel verifies whole proofs, whereas HOL Light only verifies the application of a single inference rule. The reason is that, because of the private type implementable in ML, composing inference rules outside the kernel via function application is a safe operation in ML. Missing private types, we must implement composition of rules in the kernel.

6.1 Free names in sequents

In HOL sequents can contain free variables of any type, and the `INST_RULE` allows to later instantiate them freely. In λProlog we want to work only on closed terms in order to reuse the language binding mechanism to avoid implementing α-conversion, instantiation, etc. Some inference rules, however, do introduce fresh names (typical example is the introduction rule for the universal quantifier). Because free variables are not allowed in the semi-shallow encoding, we need to find a representation of sequents where all variables are bound "outside the sequent". The one we chose puts enough `bind TYPE` binders around the sequent. For example, the primitive congruence rule for λ-abstractions is:

```
thm k (seq Gamma (eq ' (lam A S) ' (lam A T)))
 [ bind A x \ seq Gamma (eq ' (S x) ' (T x)) ].
```

Correspondingly, we introduce two new primitive rules to prove sequents that have free names:

```
thm (bind A TAC) (bind A SEQ) NEWL :-
 pi x \ term' x A => thm (TAC x) (SEQ x) (NEW x),
  put_binds' (NEW x) x A NEWL.

thm ww (bind A x \ SEQ) [ SEQ ].
```

Rule `ww` (for weakening) is simpler: if `x` does not occur in the sequent, proving `bind A x \ SEQ` is equivalent to proving `SEQ`.

The first rule says that to prove a `bind A x SEQ` using a rule `bind A TAC` which is itself parametric on a variable having the same type, it is sufficient to pick a fresh name `x`, assume that it has type `A`, and then call rule `TAC x` (the rule "instantiated" on the fresh name) on `SEQ x` (the sequent "instantiated" on the fresh name). The result will be a list of new sequents `NEW x` where `x` can occur. The predicate `put_binds'` (code of 4 lines not shown for space reasons) prefixes every sequent in the list `NEW x` with the `bind A x \` binder.

Compared to HOL Light, these two rules are new, but `INST_RULE` (and the corresponding one for type variable instantiation) are no longer necessary. Moreover, the remaining primitive rules are slightly simplified by removing α-equivalence tests where they occur in HOL Light code.

6.2 Tactics as macros

A rule in the previous paragraphs is meant to be either a primitive rule of the logic, or a user-defined tactic. In LCF tradition a tactic takes in input a local proof state and applies a bunch of other rules, either primitives or tactics.

In our code, a tactic is essentially a user-defined macro that expands to the composition of other rules. The expansion is determined by the tactic parameters but also by the (local) proof state. The invocation of the expansion is implemented in the kernel, but the expansion predicate `deftac` is untrusted and freely implemented outside.

```
thm TAC SEQ SEQS :- deftac TAC SEQ XTAC, thm XTAC SEQ SEQ.

% Example outside the kernel
deftac forall_i (seq Gamma (forall '' _ ' lam _ G)) TAC :-
 TAC = then (conv dd) (then k (bind _ x \ eq_true_intro)).
```

Note how the `forall_i` tactic for the introduction of universal quantifiers is implemented composing other tactics (`eq_true_intro`, `k` and the conversion `conv dd`) via LCF tacticals (`then, thens,` ...). Note also that, contrarily to every other interactive theorem prover out there, our scripts contains binders for the names of freshly introduced variables. For example, as we have seen in Section 6.1 the `k` rule above introduces a fresh name, and the tactic definition must start binding the name and retrieving its type (via `bind _ x`). The name can be used in the rest of the script.

The use of binders in the scripts is a clear improvement over other systems, where just strings are used to refer to the name of hypotheses, with major problems in case the names change, e.g. because of a new version of the system. Proof refactoring (Whiteside et al. 2011) should also be simplified.

A tactical `bind* TAC` is also provided to bind-and-ignore as many fresh names as possible, that are not bound in `TAC` that therefore cannot use them. It is necessary, for example, after the application of the `repeat` tactical to a tactic that generates fresh names, because it is statically unknown how many fresh names will be generated.

For the time being we do not assign names to hypotheses, like in HOL Light and unlike other theorem provers like Coq. Adding names to hypotheses via binders in the script as we did for fresh names does not introduce any new difficulty.

6.3 Tacticals

An LCF tactical is an higher order tactic. Most tacticals start applying a tactic and then continue on the resulting subgoals. Tacticals can be defined in ML outside the kernel by inspecting the result of the first tactic application and using safe function composition. We are forced again to provide a kernel-side mechanism to implement

tacticals as we did for tactics. It boils down to the new following primitive rules:

```
thm (thenll TAC1 TACN) SEQ SEQS :-
 thm TAC1 SEQ NEW,
 deftacl TACN NEW TACL,
 fold2_append' TACL NEW thm SEQS.

thm (! TAC) SEQ SEQS :- thm TAC SEQ SEQS, !.

thm id SEQ [ SEQ ].
```

The first rule apply a tactic `TAC1` and then asks the untrusted predicate `deftacl` defined in user space to expand the "tactic" (or certificate) `TACN` to a list of tactics `TACL`. The expansion can observe the list of goals generated by `TAC1` and each new tactic will be applied to the corresponding new goal by `fold2_append` that also collects the set of new, final goals.

Rule `thenll` already allows to implement in user space most LCF tacticals like `then` and `thenl`. The remaining can be implemented using the `id` rule too, that is the neutral element of composition (it does nothing).

Finally, the extra-logical `(! TAC)` tactical applies the tactic `TAC` and then suppress all backtracking via cut. It is very useful in those situations where the user wants to implement invertible tactics, i.e. tactics that may be backtracked, but that do not change provability and thus should not be backtracked.

We observe that the tactics and tacticals that can be implemented in our system go beyond the LCF ones because we inherit full-backtracking from λProlog. In particular, our tactics are naturally non-deterministic, i.e. they can have multiple solutions (that are enumerated via backtracking). The LCF version of the `then` tactical that never backtracks the first tactic corresponds to our restriction to banged tactics: `then (! TAC1) TAC2`.

6.4 Uses of the certificates

We implemented outside the kernel two kind of certificates: interactive and non interactive. A non interactive certificate is just a proof script, i.e. a list (usually a singleton) of tactics to be applied sequentially to prove the open goals. Non interactive certificates are partially inspired by the work of Chihani (Blanco et al. 2015): they also consist in a list of tactics, but at the first call a singleton list containing a metavariable is passed. When the next tactic is required, the tactic is interactively asked to the user and recorded in the certificate by instantiating the metavariable. Moreover, if the tactic is an application of the `thenll` tactical, the argument of `thenll` is a list of fresh metavariables that will be instantiated later with the tactics used in any branch.

At the end of an interactive proof, the interactive certificate has been fully instantiated recording all tactics applied, and it becomes usable as a non interactive certificate (i.e. a proof script). To improve readability of the script, we wrote a normalization function that rewrites applications of `thenll` to application of `then` and `thenl` when possible. The normalized script is pretty-printed to the user that can copy&paste it into the library.

6.5 Final considerations

As described in the previous sections, our kernel implements an higher number of basic inference rules w.r.t. the HOL Light kernel. Nevertheless, as expected, our kernel is considerably smaller: 200 lines of self-contained λProlog code vs 571 lines of OCaml code that also reuses standard library functions on lists. The greatest gain is due to saving most of the 323 lines of OCaml code that deal with instantiation, substitution, renaming and α-conversion of terms and types in HOL Light: these operations all come for free in λProlog thanks to the semi-shallow encoding.

7. User Interface Issues

We briefly sketch now the main coding issues related to the outer layer of the theorem prover, i.e. everything but the kernel.

7.1 Representation of Terms and Proof States

The internal representation of terms in an interactive prover is often very verbose and it is customary to allow to write terms using a so called "outer" syntax that allows for infix operators, mathematical notations and omission of information (e.g. types). In order to let the users extend the system easily, the outer notation should be also available when defining tactics.

HOL Light is the system with the best support of outer syntax, even when defining tactics. This is achieved by means of a pre-processor for OCaml (written in camlp5 (`http://camlp5.gforge.inria.fr`)): all strings written in backquotes in the code of HOL Light are parsed and type-checked by the pre-processor and turned into well-typed terms. The opposite is done during printing. The code of the pre-processor amounts to 2845 lines. When an user develops a library, she does not need to write any additional parsing code, but she can always briefly define a new operator as prefix/infix/etc.

In our prototype the user can directly write terms written in our semi-shallow encoding. For example, $Px \wedge Q$ should be written as `and ' (P ' x) ' Q`. To reduce the burden on the user, we provide the parse/pretty-printing predicate seen in Section 5.1. Contrarily to HOL Light, the outer syntax is not simply a string, but it is again a structured term that extends the one of the internal syntax. Infix binary operator and prefix/postfix unary operators can be directly used in the external syntax because λProlog itself allows to declare constants as such. Thus, for example, `and ' (P ' x) ' Q` can be abbreviated to `P ' x && Q` but not to `P ' x /\ Q` because `/\` is not a valid LambdaProlog constant. Moreover, infix ticks must still be used to separate between function arguments.

A second difference is that terms in the "outer" syntax can contain metavariables, allowing interpolation of (meta)variables for free like in `p X :- parse (X && X) 0, $print 0`. However, when used without care interpolation is computationally expensive. For example, the previous clause would interpolate `X` in `X && X` before calling `parse` that will traverse `X` twice. A better implementation would be `p X :- parse (Y && Y) 0, parse X Y`.

A third difference is that our syntax only admits close terms: all variables have to be quantified, either inside the term, or outside it in the code. For example, where a HOL Light user would state the polymorphic theorem `prove 'a = b => b = a'` our user must state `theorem (pi A \ ! A a \ ! b \ a = b => b = a)` where the two variables are universally quantified in the statement and the abstraction over the type A is quantified outside it to make the theorem polymorphic.

The last, and most annoying, difference is that the parsing predicate must be extended by the user every time it declares a new tactic, e.g. `parsetac (t Y) (t PY) :- parse Y PY.` where `Y` is a term argument of the tactic `t`.

All the above observations suggest that a better approach would be closer to the one of HOL Light: we could integrate in ELPI a pre-processor to detect and handle terms during the parsing and pretty-printing phase.

In the semi-shallow encoding, bound variables are represented by variables bound at the metalevel. In non-interactive scripts, the binders are visible and the user can pick any name for the variables. However, in case of an interactive proof of, say `! x \ P ' x`, after applying the introduction rule of forall the new sequent becomes `|- P ' a` where the variable `a` is now bound by a meta-level `pi a \` that was traversed during the proof search. Unfortunately and for efficiency reasons, λProlog does not remember the names of bound variables when execution goes under a `pi`, but only their

De Bruijn level (de Bruijn 1978). Therefore in the goal presented to the user the name a will be automatically generated without any control and no correlation to the initial x.

The problem can be improved at least in two ways: 1) modifying ELPI to remember names, at the price of performance; 2) making the hypothetical pre-processor discussed above insert string parameters in the syntax so that `! x \ P` would be parsed as `! "x" x \ P` and later processed via

```
do_something (! s F) :-
 pi  x \ parse s x => do_something (F x).
```

7.2 Top-level loop

HOL Light does not have an ad-hoc interactive loop: the OCaml top-level is used instead thanks to the private data types. In our implementation, instead, the toplevel loop mechanism is also part of the kernel and, as for the tactic loop, it asks a *library certificate* (`WHAT` in the code) the next command to be executed (`C`) and the continuation certificate (`CONT`):

```
check WHAT :-
 next_object WHAT C CONT,
 (C = stop, !, K = true
 ; check1 C H , check_hyps [] H, K = (H => check CONT)),
 !, K.
```

If `C` is `stop` the toplevel loop ends successfully. Otherwise the command `C` is executed (by `check1`) resulting in a new hypothesis `H` that is verified (by `check_hyps`) and then assumed in the next iteration of the loop (`H => check CONT`). The only available commands in HOL are: assuming a new axiom; declaring a new (polymorphic) definition; defining a new type as a conservative extension; proving a theorem. HOL packages can be easily implemented by the `next_object` predicate that is defined outside the kernel. For example, we implemented an `inductive_def` package that takes a piece of syntax that declares the introduction rules of a new inductively defined predicate and returns a bunch of definitions and theorems based on the Knaster-Tarski's fixpoint theorem. There are only a few kind of hypotheses returned by `check1` (that is defined in the kernel): assignment of a type to a constant (i.e. `term c ty`), declaration of a constant as a type constructor (e.g. `typ (list '' A) :- typ A.`), assigning a statement to a theorem/axiom name (i.e. `provable c statement`). The `check_hyps` predicate just verifies that constants are assigned a type/statement once.

As for tactics, we developed two kind of library certificates: the first one is non interactive and it consists in just a list of commands to be expanded by packages in the clauses of `next_object`; the second one is interactive and it just parses from standard input the next command to be executed. In the case the command starts a proof, the interactive library certificate will use an interactive certificate for the proof, effectively entering the proof mode of the theorem prover.

Finally, backtracking inside a proof is as simple as entering a `backtrack` command that just fails, invoking λProlog backtracking mechanism. Note, however, that the behavior is different than the one of HOL Light: backtracking can enumerate the next solution of a previous tactic, unless a Prolog cut (!) was used to prune the remaining solutions.

7.3 Definitional Mechanisms

We implemented definitional mechanisms similar to the one of HOL Light, but modified to have a constructive interpretation (see (Arthan 2016) for an explanation of why the ones of HOL Light entail excluded middle). The only significant change is about polymorphic constants where we preferred to employ System-F application, written in infix style using two quotes, like in `monotone '' nat ' f`. Syntactically, monotone is defined via:

```
def monotone (pi A \
 (((A --> prop) --> (A --> prop)) --> prop),
 (lam (_ A) f \ ! x \ ! y \ x <<= y ==> f ' x <<= f ' y))
```

where the type abstraction binds `A` both in the type and in the body. The new hypothesis that is added to the kernel after processing the definition is

```
pi A \ term (monotone '' A)
 (((A --> prop) --> (A --> prop)) --> prop).
```

When λProlog backchains over the hypothesis, it automatically instantiates `A` with a fresh metavariable. Therefore we get the expected behavior for free, without the need to syntactically introduce System-F type abstractions and type quantifiers.

Despite the System-F terminology we employ, note that, as expected for an HOL system, the polymorphism achieved is the Hindley-Milner one, where quantifiers do not occur inside types.

In HOL Light, the user does not even need to write the type quantifiers, that are automatically inserted to capture all metavariables that are left uninstantiated by type inference. Type inference in λProlog came for free, but there is no operator to bind all metavariables left uninstantiated. Indeed, how to implement Hindley-Milner type-inference in λProlog is an annoying problem left open for several years now.

8. Conclusions and Future Work

We presented as a concrete case study the implementation of an HOL based prover in λProlog via a semi-shallow encoding of the object logic. We identify a few limitations of the programming language and propose extensions to ease the development of an interactive prover.

A future line of research is to understand what other problems can benefits from the same extensions. Promising ones are situations where the program must manipulate partial and incomplete data structures, like during type-inference. For example, good test cases could be automatic inference of behavioral types (Giachino et al. 2015) or inference of linear types (Baillot and Hofmann 2010) via the generation of sets of inequalities then treated as constraints to be solved.

Another direction is to continue the study of constraint propagation rules in the context of an higher order language like λProlog.

Last, we could analyze how the programming language features impact on performances. The HOL library we have now is not large enough yet to compare the speed of HOL Light with our own implementation Currently, it includes only the definition of logical constants and their properties, the Knaster-Tarski fixpoint theorem, and a manual construction of natural numbers from a constructive universe. It ends with the definition of addition between natural numbers and the proof that it is commutative.

So far, the impression is that the most expensive, and recurrent, check is that terms are well typed. The HOL-light system makes an ingenious use of ML's private data types to avoid re-checking the same term over and over. Statically, the ML type system would ensure that term built out of already checked terms can be checked in a shallow way. This source of repetition is typically solved in declarative languages by tabulating, i.e. caching recent computations. In presence of binders tabulating is not an easy solution. The abstraction mechanism of λProlog via `local` constants does not seem capable to solve the problem in a satisfactory way either, since it would make the data on which the invariant holds not easy to inspect. An ambitious direction could be to study if the restriction on modes we presented in this paper could be combined with the standard type system of λProlog in order enable the programmer to express invariants similar to the ones one can express using ML's private data type.

Acknowledgments

We are greatly indebted with Dale Miller for long discussions over λProlog, its use for implementing interactive provers and the constraint programming extensions.

References

A. W. Appel and A. P. Felty. Lightweight Lemmas in lambda-Prolog. In *Logic Programming: The 1999 International Conference, Las Cruces, New Mexico, USA, November 29 - December 4, 1999*, pages 411–425. MIT Press, 1999. ISBN 0-262-54104-1.

R. Arthan. On Definitions of Constants and Types in HOL. *Journal of Automated Reasoning*, 56(3):205–219, 2016. ISSN 1573-0670. doi: 10.1007/s10817-016-9366-4.

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. doi: 10.1007/978-3-642-03359-9_8.

P. Baillot and M. Hofmann. Type Inference in Intuitionistic Linear Logic. In *Proceedings of the 12th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 26-28, 2010, Hagenberg, Austria*, pages 219–230, 2010. doi: 10.1145/1836089.1836117.

R. Blanco, Z. Chihani, and D. Miller. Translating between implicit and explicit versions of proof. Unpublished, 2015. URL `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/implicit-explicit-draft.pdf`.

Z. Chihani, D. Miller, and F. Renaud. Foundational Proof Certificates in First-Order Logic. In *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, pages 162–177, 2013. doi: 10.1007/978-3-642-38574-2_11.

N. G. de Bruijn. Lambda calculus with namefree formulas in involving symbols that represent reference transforming mappings. In *Indagationes Mathematicae*, volume 81(3), pages 348–356, 1978.

D. Delahaye. A Tactic Language for the System Coq. In *Logic for Programming and Automated Reasoning: 7th International Conference, LPAR 2000 Reunion Island, France, November 6–10, 2000 Proceedings*, pages 85–95. Springer Berlin Heidelberg, 2000. ISBN 978-3-540-44404-6. doi: 10.1007/3-540-44404-1_7.

C. Dunchev, F. Guidi, C. Sacerdoti Coen, and E. Tassi. ELPI: Fast, Embeddable, lambda-Prolog Interpreter. In *Logic for Programming, Artificial Intelligence and Reasoning: 20th International Conference, LPAR-20 2015, Suva, Fiji, 2015, Proceedings*, pages 460–468. Springer Berlin Heidelberg, 2015. ISBN 978-3-662-48899-7. doi: 10.1007/978-3-662-48899-7_32.

A. P. Felty. Implementing Tactics and Tacticals in a Higher-Order Logic Programming Language. *J. Autom. Reasoning*, 11(1):41–81, 1993. doi: 10.1007/BF00881900.

A. P. Felty and D. Miller. Specifying theorem provers in a higher-order logic programming language. In *9th International Conference on Automated Deduction, Argonne, Illinois, USA, May 23-26, 1988, Proceedings*, pages 61–80, 1988. doi: 10.1007/BFb0012823.

T. Frhwirth. Theory and practice of constraint handling rules. *The Journal of Logic Programming*, 37(13):95 – 138, 1998. ISSN 0743-1066. doi: http://dx.doi.org/10.1016/S0743-1066(98)10005-5.

E. Giachino, E. B. Johnsen, C. Laneve, and K. I. Pun. Time complexity of concurrent programs. *CoRR*, abs/1511.05104, 2015.

M. J. C. Gordon and A. M. Pitts. The HOL logic and system. In J. Bowen, editor, *Towards Verified Systems*, volume 2 of *Real-Time Safety Critical Systems*, chapter 3, pages 49–70. Elsevier Science B.V., 1994.

M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979. ISBN 3-540-09724-4. doi: 10.1007/3-540-09724-4.

J. Harrison. HOL light: An overview. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, pages 60–66, 2009a. doi: 10.1007/978-3-642-03359-9_4.

J. Harrison. HOL Light: An Overview. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, pages 60–66, Berlin, Heidelberg, 2009b. Springer Berlin Heidelberg. ISBN 978-3-642-03359-9. doi: 10.1007/978-3-642-03359-9_4.

X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system release 3.08, Documentation and users manual. In *Projet Cristal, INRIA*, 2004.

A. Mahboubi and E. Tassi. Canonical Structures for the Working Coq User. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving: 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, pages 19–34, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-39634-2. doi: 10.1007/978-3-642-39634-2_5.

D. Miller. Unification Under a Mixed Prefix. *J. Symb. Comput.*, 14(4):321–358, 1992. doi: 10.1016/0747-7171(92)90011-R.

D. Miller. Foundational Proof Certificates. In *All about Proofs, Proofs for All (APPA)*, pages 150–163, 2014.

G. Nadathur and D. J. Mitchell. System Description: Teyjus - A Compiler and Abstract Machine Based Implementation of lambda-Prolog. In *Automated Deduction - CADE-16, 16th International Conference on Automated Deduction, Trento, Italy, July 7-10, 1999, Proceedings*, pages 287–291, 1999. doi: 10.1007/3-540-48660-7_25.

G. Nadathur and G. Tong. Realizing Modularity in lambdaProlog. *Journal of Functional and Logic Programming - March 15, 1999*, 1999(Special Issue 2), 1999.

C. Sacerdoti Coen, E. Tassi, and S. Zacchiroli. Tinycals: Step by Step Tacticals. *Electr. Notes Theor. Comput. Sci.*, 174(2):125–142, 2007. doi: 10.1016/j.entcs.2006.09.026.

A. Spiwack. An abstract type for constructing tactics in Coq. In *Proof Search in Type Theory*, Edinburgh, United Kingdom, July 2010a. URL `https://hal.inria.fr/inria-00502500`.

A. Spiwack. An abstract type for constructing tactics in coq. Unpublished, 2010b. URL `http://assert-false.net/arnaud/papers/An%20abstract%20type%20for%20constructing%20tactics%20in%20Coq.pdf`.

I. Whiteside, D. Aspinall, L. Dixon, and G. Grov. Towards Formal Proof Script Refactoring. In *Intelligent Computer Mathematics - 18th Symposium, Calculemus 2011, and 10th International Conference, MKM 2011, Bertinoro, Italy, July 18-23, 2011. Proceedings*, pages 260–275, 2011. doi: 10.1007/978-3-642-22673-1_18.

F. Wiedijk. *The Seventeen Provers of the World: Foreword by Dana S. Scott (Lecture Notes in Computer Science / Lecture Notes in Artificial Intelligence)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 3540307044.