Implementing HOL in an Higher Order Logic Programming Language
Cvetan Dunchev, Claudio Sacerdoti Coen, Enrico Tassi

To cite this version:

HAL Id: hal-01394686
https://inria.hal.science/hal-01394686
Submitted on 9 Nov 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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

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 \( \alpha \)-conversion, and it needs to implement substitution. Binders are ubiquitous: they occur in formulæ, 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 formulæ (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. \( \lambda \)Prolog, 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 \( \lambda \)Prolog takes care of all interactions between binders (\( \lambda \)-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 \( \alpha \)-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 for Prolog (Spwack 2010), 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. 3100 lines of OCaml code for Matita, compared to the 1500 lines for the kernel that implements \( \beta \)-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 Tasson 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 prover ignoring all the gory details.

Nevertheless, as we said, no system has been implemented in \( \lambda \text{Prolog} \), 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 \( \lambda \text{Prolog} \) 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 \( P \Rightarrow Q \)) 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, \( \lambda \text{Prolog} \) had for some time only very slow implementations, until Tejysus was born (Radanur and Mitchell 1999) after the work of Felty. Recent work by the authors also showed that the performances of Tejyus are not so good (Dunchev et al. 2015), considering that Tejyus 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 \( \lambda \text{Prolog} \) 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 \( \lambda \text{Prolog} \), 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. \( \lambda \text{Prolog} \) in a nutshell

\( \lambda \text{Prolog} \) extends the core of Prolog in two directions.

1. it allows in the syntax \( \lambda \) abstractions (written \( \lambda x.p \) for \( \lambda x.p \)), applications (written \( p q \) like in \( \lambda \)-calculus and unlike in Prolog) and metavariables both in argument and in head position (e.g. \( \text{iter } [X|XS] F [Y|YS] :- F X Y, \text{ iter } XS F YS \)). where the metavariables is to be instantiated with a binary predicate like \( (x\backslash y \Rightarrow y+x(x)) \)

2. it generalizes Horn clauses to Higher Order Hereditary Harrop Formulas (HOHFF) clauses, whose grammar is

\[
\begin{align*}
\text{H} &::= x t_1 \ldots t_n | H \land H | \forall x.H | \exists x.H | G \Rightarrow H \\
G &::= x t_1 \ldots t_n | X t_1 \ldots t_n | G \land G | G \lor G | G \Rightarrow G \\
H, G &\subseteq t ::= x | X | t t | x t
\end{align*}
\]

where \( H \) ranges over HOHFF, \( G \) 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 \( \lambda \text{Prolog} \) program is a list of HOHFF. To run a \( \lambda \text{Prolog} \) 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 HOHFF \( H \) is given by the introduction/elimination rules of the connectives in natural deduction for intuitionistic higher order logic. In particular:

- the goal \( H_1 \land H_2 \) is turned into the two goals \( H_1 \) and \( H_2 \), copying the program (i.e. the set of clauses/assumptions)
- the goal \( H_1 \lor H_2 \) 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 $H_1 \land H_2$ means assuming both $H$s 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 $\text{sigma} \ \lambda x.t$ for $\exists X.t.\pi x.t$ for $\forall X.t.\pi x.t$ $a \Rightarrow b$ or $b := := a$ for $a \Rightarrow b$, $a,b$ for $a \land b$ and $a;b$ for $a \lor b$. 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" $\lambda$Prolog 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 $(\text{sigma } x \ H_1,\ldots,H_n)$ or with the equivalent syntactic sugar $\{\text{local } x.\ H_1,\ldots,H_n\}$ (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 $\lambda$Prolog to restrict the trusted code base.

The other major difference between the ELPI version of $\lambda$Prolog and the "official" one of Teyjus is that we do not enforce any type discipline, nor we try to statically enforce the restriction to HO-HHF. 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 $\lambda$Prolog 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 $\lambda$ to encode application, the unary constant $\text{lam}$ for lambda-abstraction, and the infix constant $\Rightarrow$ for the function space type. The former is to be applied to a meta-level lambda-abstraction in the spirit of higher-order abstract syntax. For example, the term $\text{lam } x.x$ is encoded as $\text{lam } x \ x \ x$. Note that, in the concrete syntax, parentheses around lambda-abstractions are not necessary in $\lambda$Prolog when the abstraction is the last argument of an application. I.e. $\text{lam } x \ x \ x$ is to be read as $\text{lam } (x \ x \ x)$.

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 $\text{term } T \ T Y$ ("$T$ is a term of type "$T'Y'")) does not mention any context $\Gamma$. Instead, in the rule for lambda-abstraction, the hypothesis that the fresh variable $x$ has type $\alpha$ is directly assumed and put temporarily in the program code using logical implication $\Rightarrow$. For example, the query $\text{term } (\text{lam } \ y \ y \ y \ y)$ will match the second clause and trigger the new query $\text{term } (x \ x \ x)$ after having instantiated $\text{T}$ with $\text{A} \Rightarrow \text{B}$ where $x$ is fresh (because introduced by $\pi$) and of type $\text{A}$.

Finally, we will use the cut predicate of $\lambda$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.

| $\text{term } (M \ N) \ B : = \text{term } M \ A \Rightarrow B, \text{term } N \ A.$ |
| $\text{term } (\text{lam } F) \ (A \Rightarrow B) : = \pi x.\text{term } x \ A \Rightarrow \text{term } (F \ x) \ B.$ |

### 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:

\[
T ::= a \mid u \ T_1 \ldots T_n \mid T \Rightarrow T
\]
\[
t ::= x_T \mid ct \mid \lambda x_T.t \mid t \ t
\]

where $u$ ranges over type constructors (constants when $n = 0$) and $\alpha$ ranges over type variables. Contrariwise 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 $\text{cons}$ is typed $\text{list } \alpha \rightarrow \text{list } \alpha \rightarrow \text{list } \alpha$.

Variables $x_T$ and constants $ct$ 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 $\text{cons} : \text{list } \text{nat} \rightarrow \text{list } \text{nat} \rightarrow \text{list } \text{nat}$ explicitly encodes the instantiation $\alpha := \text{nat}$. In System-F the constant occurrence would be applied to the type $\emptyset \ \text{nat}$.

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

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

\[\begin{array}{c}
\Gamma \vdash p : \text{bool} \\
\Gamma_1, p \vdash q \\
\Gamma_2, q \vdash p \\
\Gamma_1, \Gamma_2 \vdash p = q \\
\end{array}\]

\[\begin{array}{c}
\Gamma \vdash p \\
\Gamma \vdash p \sigma \\
\end{array}\]

In the HOL Light implementation, the check for $p$ being a boolean in the first rule is postponed to the time the hypothesis is used (the HYP rule, here not shown). The second rule is used to apply the substitution $\sigma$ 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 $ct_T$, 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

A lightweight implementation of an HOL system in an extension of higher order logic programming

2016/11/9
no constant named \( c \) is already defined in the current theory, the
latter is extended with the new constant \( c_F \) and the axiom \( c_F = t \).

The condition on the free type variables of \( t \) avoids logical in-
consistency and would not be necessary in System-F where every
occurrence of \( c \) would be explicitly applied to every free type vari-
able that occurs in \( t \). In HOL, instead, only the type instantiations
for the type variables that occur in the type of \( c \) can be recov-
ered from the type of \( c \). Therefore, if the condition is violated,
\( c_F \) could be rewritten via the axiom to an instantiation of \( t \) that
still contains free type variables. Example: \( c_{\text{bool}} = !x, y_1. x_0 =
y_0 \) that from \( c_{\text{bool}} = c_{\text{bool}} \) via two instantiations would imply
\( !x_0 y_1. c_{\text{bool}} = y_{\text{bool}} \iff !x_{\text{unit}} y_{\text{unit}}. x_{\text{unit}} = y_{\text{unit}} \text{ i.e. false} = \text{true} \).

3.1 HOL (Light) in ML

HOL Light (Harrison 2009a) is an extremely lightweight imple-
mentation 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 \( \Gamma \vdash t : T \); 2) the type of theorems, i.e. pairs
context \( \Gamma \), 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.

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

val dest_thm : thm -> term list * term

fun mkDeductAntisymRule thm1 thm2 = thm1 \&\& thm2

val mkDestThm : thm -> term list * term

fun mkDeductAntisymRule thm1 thm2 = thm1 \&\& thm2
```

```ml

fun mkDeductAntisymRule thm1 thm2 = thm1 \&\& thm2

fun mkDestThm : thm -> term list * term
```

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
formulas 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

```ml

fun mkDeductAntisymRule thm1 thm2 = thm1 \&\& thm2

fun mkDestThm : thm -> term list * term
```

Ignoring the instantiation and metavariables, a tactic is a func-
tion 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 rea-
soning 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 tycnals introduced in (Sacerdoti
Coen et al. 2007), which can now be implemented in Coq as
well after (Spwack 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 con-
sumed 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 — per-
forms 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 \( \Gamma, p \vdash q \) and \( \Gamma, q \vdash p \). Later the justification
DEDUCT_ANTISYM_RULE needs to pattern match the list of the-
orems in input against \( \Gamma_1, g \vdash p \) 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 construc-
tive variant of) HOL in ELPI, our λProlog interpreter. The code is
available at http://lpcic.gforge.inria.fr/system.html?content=sys-
```
over a proof object (Feltý) 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 Prolog 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 Prolog 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 Prolog 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:

\[ t ::= \text{var NAME} \mid \text{app}\ t\ t \mid \lambda\text{NAME}\ t \mid \text{meta}\ \text{NAME} \]

Assuming \text{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 \( (\text{F} X)\ B := \text{type}\ X\ A, \text{type}\ F\ (A \rightarrow B) \)

\[ \]

does not do what it seems to do, because applied to the query \( \text{type}\ \lambda a\ Z \), it yields the HO unification problem \( F \equiv x \mid a, X := 0. \)

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

\[ t ::= t \cdot t \mid \lambda\text{NAME}\ F \]

where \( \cdot \) 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, \( \lambda x: a. X \) is encoded as \( \lambda a x \backslash \ X x \). 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 \text{term}\ eq\ (\ A \rightarrow A \rightarrow \text{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. \text{term}\ (\lambda A x \backslash x + 1) B\ yields\ \ A := \text{nat}, \ B := \text{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.

\[ \]

Following the spirit of Prolog the \text{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 \( (\text{eq}\ "\ \text{prop}\ A\ B") \) :- \text{pp}\ X\ A, \text{pp}\ Y\ B.

\[ \]

This very same problem also arises in the implementation of automatic tactics, that have to be able to choose the next step by inspection of the goals. Of course, inspecting the goal should leave no visible trace.

The situation is analog for typing.

\[ \]

Imagine the \text{term} program is run on an incomplete input like \( (\lambda a\ n\ n\ X\ n)\ T \). The program, augmented with the clause \text{term}\ n\ nat, \text{wbound}\ run\ on\ the\ sub\ goal\ \text{term}\ (X\ n)\ B \).

Such clause, as well as any other one for \text{term}, would apply. As a result the meta variable \( \text{W} \) 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 \( \Sigma \). When an incomplete term is found the algorithm stops and check that the (occurrence of) the metavariable has the type declared in \( \Sigma \) (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 \( \Sigma \) (2). The reading of this behavior in terms of logic programming points toward two distinct, but complementary mechanisms.

First, a goal like \text{term}\ (X\ n)\ B\ has to be \text{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 \text{constraints}, all to be solved at some point. Most Prolog system provides such mechanism. Tejus has it too but only for internal use, i.e. delay hard HO unification problems falling outside the \text{L} \_ element.

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 metavariable 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 [Pichler 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:

```prolog
mode (pp i o) as parse, (pp o i) as print.
```

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 e catch all clause. pp A A. Such clause prints/parses each metavariable with itself, as well as prints any term with no associated notation as itself. Used in conjunction with \( \lambda \) Prolog implication, the pp predicate can serve as a mechanism for scoped, printed and parsed, notations, like names for bound variables. E.g.

```prolog
?- pl w \
  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:

```prolog
mode (term i o).
term (lam A F) (A --> B) :- pi \( x \) \( \lambda \) \( x \) A \( \rightarrow \) 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 \( \lambda \)Prolog

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

```prolog
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.

\( \lambda \)Prolog 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.

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

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 \( \lambda \)Prolog 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 \( \lambda \)Prolog

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.

```prolog
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:

```prolog
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 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 call k) is implemented by

\[
\text{thm s (seq Gamma (eq ' P ' Q))} \\
[\text{seq (P :: Gamma) Q, seq (Q :: Gamma) P}] \\
\text{:- reterm P prop.}
\]

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] [seq [] true] => 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 Prolog we do not want to work only on closed terms in order to reuse the language binding mechanism to avoid implementing \(\alpha\)-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 \(\alpha\)-abstractions is:

\[
\text{thm k (seq Gamma (eq ' (lam A S) ' (lam A T)))} \\
[\text{bind A x \ seq Gamma (eq ' (S x) ' (T x))}]
\]

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

\[
\text{thm (bind A TAC) (bind A SEQ) NEWL :=} \\
\text{pi x \ term' x A => thm (TAC x) (SEQ x) (NEW x),} \\
\text{put_binds' (NEW x) x A NEWL.}
\]

\[
\text{thm \(ww\) (bind A x \ SEQ) [ SEQ].}
\]

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

The first rule says that to prove a \(\text{bind A x SEQ}\) using a rule \(\text{bind A TAC}\) 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 \(\text{TAC x}\) (the rule “instantiated” on the fresh name) on \(\text{SEQ x}\) (the sequent “instantiated” on the fresh name). The result will be a list of new sequents \(NEW\ x\) where \(x\) can occur. The predicate \(\text{put\_binds'}\ (\text{code of 4 lines not shown for space reasons})\) prefixes every sequent in the list \(NEW\ x\) with the \(\text{bind A x \ bind}\).

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 \(\alpha\)-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.

\[
\text{thm TAC SEQ SEQS := deftac TAC SEQ XTC, thm XTC SEQ SEQ.}
\]

\% 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, 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 cases the names change, e.g. because of a new version of the system. Proof refactoring [White 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 next 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 Tactics

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:

\[
\text{tm (thenn1 TAC1 TACN) SEQ SEQS} :\!- \!
\text{tm TAC1 SEQ NEW,}
\text{deftacl TACN NEW TACL,}
\text{fold2_append \ ' TACL NEW then SEQS.}
\]

\[
\text{tm (! TAC) SEQ SEQS :\!- \! \text{tm TAC SEQ SEQS}, !.}
\]

\[
\text{tm id SEQ [ SEQ ].}
\]

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 4d 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, \(P \times Q\) should be written as ‘(P \times Q)’ or \((P \times Q)\). 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, \(\(P \times Q\)\) can be abbreviated to \(P \times Q\) but not to \(P \times Q\) because ‘/’ is not a valid LambdaProlog constant. Moreover, infix ticks must still be used to separate between function arguments.

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

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 \(\text{prove} \: \langle a = b \Rightarrow b = a \rangle\) our user must state \(\text{theorem} \: \langle \pi A \rightarrow \pi A \rangle \: \langle 1 \: \pi A \\backslash \: \pi A \rangle\) 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. \(\text{parsetac} (t \ Y) (t \ PY) :\!- \! \text{parse} Y PY\), 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 meta-level. 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 \(\langle 1 \Rightarrow P \\Rightarrow a \rangle\) where the variable \(a\) is now bound by a meta-level \(\pi A \\backslash \: \pi A\) 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
A lightweight implementation of an HOL system in an extension of higher order logic programming

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 top-level 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 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:

\[
\text{def monotone (pi A \ ((A --> prop) --> (A --> prop)) --> prop)},
\]

and 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 benefit 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


