Logic Programming Transformations for Why3 with Elpi

Matteo Manighetti

IRIF, Université Paris Cité

July 25, 2026

About Why3

Why3 is a *deductive verification platform.*

The user facing components are

- An ML-style programming language
- A specification language (first-order logic with some extensions)

```
let rec fib (n:int) : int
    requires { n >= 0 }
    ensures { fib n = result }
    variant { n }
  = if n <= 1 then n else fib (n - 1) + fib (n - 2)
```

Why3 generates Verification Conditions via Weakest Precondition. and dispatches them to external provers

It is possible to extract OCaml from WhyML. More frequently, used as an intermediate language

- C (Frama-C)
- Ada (SPARK)
- Rust (Creusot)
- EasyCrypt

About Why3

VCs are then discharged

- Either appealing directly to external provers (SMT solvers, interactive theorem provers, etc.)
- Or with some interactive preprocessing by the user

The logic of Why3 is quite rich:

- First order logic with types and polymorphism
- Inductive datatypes and predicates
- Function definitions (via ε operator)

Many of these features are not supported by SMT solvers

Why3 contains a set of **transformations** to be used in both cases:

- Preprocessing steps to dispatch VCs to SMT solvers
- Tactic-like actions for the interactive user

About Why3 transformations

In the case of dispatching to an SMT solver, **transformations** might include

- Split a VC into its components (variant decrease, array in bounds)
- Remove polymorphism
- Filter out unused or unrelated elements of the context
- Axiomatise algebraic and inductive datatypes
- Introduce universally quantified variables and hypotheses

Why3 has a plugin mechanism that can be used to add new transformations (SPARK, for example, ships a set of dedicated ones)

> New transformations are written against Why3's internal term API and task API
>
> There is no kernel: the OCaml code implementing a transformation is part of the trusted base.

A transformation is a judgment

A Why3 **task** is a context of declarations followed by a goal:

$$\Delta = \text{types, inductive defs, function defs, axioms} \qquad \Delta \vdash G$$

A **transformation** is a relation between a task and the tasks it produces:

$$\Delta \vdash G \quad \rightsquigarrow \quad [\Delta_1 \vdash G_1, \quad \ldots, \quad \Delta_n \vdash G_n]$$

> Higher-order logic programming is a native fit for this job:
>
> Backtracking, unification, variable handling via λ-trees
>
> This allows for declarative and inspectable code

λProlog: Horn clauses, plus binders and contexts

λProlog extends Horn clauses to **hereditary Harrop formulas**: clauses may contain implication and universal quantification

$$H ::= A \mid H \wedge H \mid G \Rightarrow H \mid \forall x.H$$

- `D => G` — run `G` with the extra clause `D`
- `pi x\ G` — run `G` with a **fresh** eigenvariable `x`

Terms are simply-typed λ-terms, with unification modulo $\alpha\beta\eta$ (in the pattern fragment)

Object-level binders are represented by meta-level λs (**λ-tree syntax**):

```
transform (tquant tforall V Bnd) (tquant tforall V Bnd') :-
  pi x\ ctx-vs x V => transform (Bnd x) (Bnd' x).
```

When we cross a binder, the language handles fresh-name generation, capture, substitution

Elpi

Elpi: an **Embeddable λProlog Interpreter** (Dunchev, Guidi, Sacerdoti Coen, Tassi)

- Fast interpreter written in OCaml, made to be **embedded** in host applications
- The host exposes its data and primitives as external symbols and predicates
- Extensions to λProlog: modes, determinacy (functionality) checking, syntactic constraints and CHR

Main application: **rocq-elpi** :

- Originated from the observation that several components (elaborator, typechecker…) end up growing their own ad-hoc logic programming engines
- Evolved into a full-fledged extension language allowing to write tactics, commands, extensions
- Still no complete elaborator, but several big applications

why3-elpi embeds Elpi in Why3: transformations become λProlog programs, loaded and run through Why3's plugin mechanism.

Originates from work on logic programming for proof certificate reconstruction (Foundational Proof Certificates).

A transformation is a λProlog predicate

The plugin registers each `.elpi` file as an ordinary Why3 transformation:

```
$ why3 prove file.mlw -a "elpi_intros"
```

The engine loads the API (`w3lp.elpi`) plus the user program and runs the main query

```
pred w3_run i:list tdecl, i:focused-goal, o:list focused-task.
```

- The task is **embedded** into λProlog syntax before the query
- The output tasks are **read back** into Why3 tasks, which re-declares and type-checks the terms
- Transformations can take arguments (symbols, terms): extra input arguments of `w3_run`

Everything the user writes is ordinary λProlog code over this interface

Terms: the logic of Why3 in λ-tree syntax

One `term` datatype for Why3's terms and formulas; every binder is a λ:

```
kind term type.
external symbol tapp    : lsymbol -> list term -> option ty -> term.
external symbol tbinop  : binop -> term -> term -> term.
external symbol tquant  : quant -> var -> (term -> term) -> term.
external symbol tlet    : term -> var -> (term -> term) -> term.
external symbol teps    : var -> (term -> term) -> term.
external symbol tcase   : term -> list branch -> term.
external symbol tattr   : list attribute -> term -> term.
```

Notice that there is no `tvar` term constructor!

`forall x. p x /\ q` → `tquant tforall «X» (x\ tbinop tand (tapp p [x] none) q)`

- Therefore `var` carries the printed name and type; the λ carries the **binding**
- Symbols (`lsymbol`, `var`) are **opaque** Why3 identifiers (here printed as `«X»`)

  Builtin external projections (`why3.ls-name`, `why3.var-type`, ...) allow inspection from λProlog code

Crossing a binder

Opening `tquant tforall V Bnd` = going under the λ-abstracted body

```
% x is a fresh variable standing for the Why3 symbol V
pi x\ ctx-vs x V => transform (Bnd x) (Bnd' x).
```

We move the term binder to a program binder on `x`

But `x : term` is unknown to Why3

Adding `ctx-vs x V` lets **builtins** run under the binder (`why3.var-type V`, `why3.pp-term`)

And it lets the readback turn `x` back into a Why3 variable.

Tasks and contexts

We split a task into the **global prefix** and a **goal** that can be extended with a **local prefix**:

```
symbol focused-task : list tdecl -> focused-goal -> focused-task.
symbol goal-formula : prsymbol -> term -> focused-goal.
symbol local-prop   : string -> term -> focused-goal -> focused-goal.
symbol local-symbol : lsymbol -> local-symbol-decl -> focused-goal -> focused-goal.
symbol local-type   : tysymbol -> focused-goal -> focused-goal.
```

To **introduce** a ∀ we mint a fresh lsymbol carrying the bound variable's name and type, and declare it on the spine:

```
intro-forall Pr (tquant tforall V Bnd) (local-symbol C symbol-param Inner) :-
  why3.var-name V Name, why3.var-type V Ty,
  why3.mk-ls Name [] (some Ty) C,
  intro-forall Pr (Bnd (tapp C [] none)) Inner.
intro-forall Pr Goal (goal-formula Pr Goal).
```

Tasks and contexts

Declarations-as-data is good for **rewriting** the context.

For **consulting** it, the context should be what contexts are in λProlog: clauses.

```
% load one (why3.defines Ref Decl) clause per symbol of the prefix, run K
why3.with-context Decls K
```

Lookup becomes indexed clause resolution instead of list traversal:

```
why3.premise  Pr T      % the prefix declares axiom/lemma Pr with statement T
why3.defined  Ls Body   % the prefix defines logic symbol Ls
why3.param    Ls        % Ls is declared but uninterpreted
why3.datatype Ts D      % Ts is an algebraic type with declaration D
```

- Having them as relations makes it easier to write idiomatic logic programming code
- Same mechanism at the goal level: `why3.with-hyps` loads the local hypotheses as `why3.hyp` clauses

These two angles reflect the nature of transformations as preprocessing or tactics

Example: split

The WP calculus generates big verification conditions. The first preprocessing step is a splitting:

```
split-formula (tbinop tand A B) Out :-
  split-formula A LA,
  split-formula B LB,
  std.append LA LB Out.
```

Implications are distributed:

```
split-formula (tbinop timplies P Q) Out :-
  split-formula Q LQ,
  std.map LQ (g\ g'\ g' = tbinop timplies P g) Out.
```

Example : split – Crossing a binder

```
split-formula (tquant tforall V Bnd) Out :-
  (pi x\ ctx-vs x V => split-formula (Bnd x) (Bodies x)),
  close-binder (h\ tquant tforall V h) Bodies Out.

split-formula (tlet Def V Bnd) Out :-
  (pi x\ ctx-vs x V => split-formula (Bnd x) (Bodies x)),
  close-binder (h\ tlet Def V h) Bodies Out.
```

Rewrapping the split bodies with the initial binder:

```
pred close-binder i:((term -> term) -> term), i:(term -> list term), o:list term.
close-binder Binder Bodies Out :-
  pi x\ map-close Binder x (Bodies x) Out. % instantiate the list with a fresh variable
pred map-close i:((term -> term) -> term), i:term, i:list term, o:list term.
map-close _ _ [] [].
map-close Binder X [E | Es] [Binder Body | Os] :-
  Body X = E,                              % re-abstract the body over the variable
  map-close Binder X Es Os.
```

Example: apply

```
apply-lemma Target Goal SubGoals :-
  why3.premise Target Stmt,                  % find the lemma
  open-lemma Stmt Premises Concl,            % open ∀s; split premises / concl
  Concl = Goal,                              % match: unification binds the metavars
  discharge-premises Premises SubGoals.      % premise: a hypothesis, or a subgoal
```

```
% open every leading ∀ with a fresh metavar, then peel the premise arrows
open-lemma (tquant tforall _ Bnd) Ps Concl :- open-lemma (Bnd X_) Ps Concl.
open-lemma (tbinop timplies P Rest) [P|Ps] Concl :- open-lemma Rest Ps Concl.
open-lemma Concl [] Concl.
```

Higher-order pattern unification handles instantiating the quantifiers and matching under binders

```
discharge-premises [] [].
discharge-premises [P | Ps] Out       :- why3.hyp _ P, discharge-premises Ps Out.
discharge-premises [P | Ps] [P | Out] :- discharge-premises Ps Out.
```

Example: type classes for Why3

```
type eq [@class] 'a = { equ : 'a -> 'a -> bool }

constant int_eq  [@instance] : eq int  = { equ = ... }
constant bool_eq [@instance] : eq bool = { equ = ... }
function pair_eq [@instance] (f1: eq 'a) (f2: eq 'b) : eq ('a,'b) = { equ = ... }

goal test : exists i. (equ i) (1,true) (1,true)
```

The `elpi_tc` transformation walks the prefix and compiles each instance into a clause, from its type:

```
tc-inst (eq int)      «int_eq».
tc-inst (eq bool)     «bool_eq»..
tc-inst (eq (A , B))  («pair_eq». I J) :- tc-inst (eq A) I, tc-inst (eq B) J.
```

Example: type classes for Why3

Running the query `tc-inst (eq (int, bool)) I` builds the required intance

The goal can be rewritten with the concrete instance:

```
goal test : let i = pair_eq int_eq bool_eq in
            (equ i) (1,true) (1,true)
```

The same style can be used for a `derive` mechanism that builds code/properties based on the shape of a data type

Performances

We expected some performance overhead. Luckily it is not that much.

After substituting `split_goal_right` with its λProlog implementation, on a set of 100 programs we get:

- Median slowdown: 1.43x
- Max slowdown: 1.94x

On the total runtime of transformations.

The overall runtime is dominated by the calls to external provers, so this is acceptable.

Some remarks:

- We kept implementation strictly declarative. Introducing cuts, NAF, can improve performances
- Well known double-edged swords in HOLP: taming unification, higher-order unification

For example: `split` decomposes an abstracted list into a list of abstractions. At quantifier depth n:

- Deconstruct list as `Bodies = (x\ (H x :: T x))` and recurse $\rightarrow O(n^2)$, 50x worst case slowdown
- Apply to an eigenvariable $\rightarrow$ recurse on list $\rightarrow$ HO unif on head to obtain it as an abstraction: $O(n)$

What we get

- Formulas with binders are first class data; fresh names and substitution are free
- Unification replaces hand-written matching code
- The task context comes two ways: **data** to rewrite the prefix, **clauses** to consult it
- Synthesis via proof search (typeclasses, `derive`)
- Transformations in the pure, cut-free fragment have a logical reading

Related and future work

Related work

- "Core Why3" formalization in Rocq
- Certification of Why3 transformations in Dedukti
- `rocq-elpi`: Elpi as extension language for Rocq (tactics, commands, derive)
- Foundational Proof Certificates: logic programs checking/elaborating proofs

Ongoing and future work

Extend the treatment of the logic (towards an elaborator):

- More of Why3's stock transformations as executable specifications
- Subgoals as suspended constraints: tacticals, and typing as constraints (CHR)
- Certificates: transformations that justify their output (ultimately: reconstruct from SMT trace)

Apply to other parts of Why3

- API for the WhyML language for types, typeclasses…
- Counterexample synthesis (in the style of Property Based Testing)

Thank You

Questions?

github.com/manmatteo/why3-elpi