PARSIFAL Summer 2011 Internship Report
Logically validating logic programs

Chris Martens

August 22, 2011

1 Overview

Logic programs can be used to specify systems, and logic programming languages such as Elf [Pfe91] and Lambda Prolog [MN86] provide settings intentionally designed to this end. The use of higher-order logic programming (clauses with implications as subgoals) can be used to encode hypothetical reasoning in a logic or predicates over terms with free variables in a programming language. Although Elf (LF underneath) is dependently typed and Lambda Prolog is simply-typed, they both use this same basic mechanism to achieve powerful encodings.

Where these encoding approaches diverge fundamentally is at the point of doing metatheory; that is, stating and proving theorems about the encoded system. The systems Twelf (cite), with Elf at its base, and Abella (cite), with Lambda Prolog, demonstrate these differing approaches.

On the Abella side, metatheory is done in another logic entirely called *G*. This *reasoning layer* enables the user to state inductive definitions and metatheorems that may mention *sequents* corresponding to Lambda Prolog proofs (at the *specification layer*). For instance, if `of` and `step` are Lambda Prolog predicates corresponding to the typing and stepping relations of a programming language, then one may state the preservation theorem in Abella as

```
Theorem preservation : forall E E' T,
{of E T} /\ {step E E'} -> {of E' T}.
```

The curly braces {} denote specification layer proofs. The proof is then carried out as a sequence of tactics.

The Twelf approach, on the other hand, makes use of LF's dependent types for its metatheory. A proof is *also* a logic program. Because LF is the representation language, we can state a theorem as a relation over *object-level proofs*, which are *terms* in LF. For example:

```
preservation : of E T -> step E E' -> of E' T -> type.
```

This declaration tells us that `preservation` is a predicate relating derivations of `of E T`, `step E E'`, and `of E' T`. The clauses that follow correspond to the cases of an inductive proof, with subgoals calling `preservation` recursively corresponding to induction hypotheses. In order for these clauses to constitute a proof, they must obey certain conditions as checked by the logic programming analyses `%mode`, `%worlds`, and `%total`.

What this project aims to investigate is the logical basis for said logic programming analyses. In other words, can *G* justify Twelf's justification for proofhood? The hypothesis is that if we had LF as the specification layer for Abella, then we could code our proofs at that layer (as we would in Twelf) and check the `%mode`, `%worlds`, and `%total` properties of the program at the reasoning layer.

The rest of this document goes into specific accomplishments of the summer, exploring several examples of mode analysis and presenting a logical generalization. What follows from there is more tenuous exploration of the other ideas discussed in this overview, including termination and coverage; we conclude with future directions.

2 Mode analysis in a two-level logic

Given a predicate p, a *mode declaration* for p is an assertion that designates some of its arguments as *inputs* and some as *outputs*. *Checking* such a declaration is usually described in terms of "groundness": if you feed the predicate ground inputs and it returns a substitution for the outputs, those substitutions will be ground.

In unification language, "ground" means simply that it does not contain any unification variables. So for example if $p(-,-)$ has mode $(+,-)$ (which assertion we may abbreviate as $p(+,-)$), then if we query $p(0,X)$, we expect a complete answer for X in terms of constants in the program (or no answer at all). However, a query of e.g. $p(X,Y)$ or $p(s(X),Y)$ will not necessarily return ground substitutions. Moding is part of what says we can treat a predicate like a function.

However, we would instead like an explanation of moding in logical terms that can be expressed at the Abella reasoning layer rather than in unification terms.

2.1 Example 1: Addition of natural numbers

```
plus z N N.
plus (s N) M (s P) :- plus N M P.
```

The predicate `plus` has mode $(+, +, -)$. [1] To check this, we look at each clause in turn, assume the inputs are ground and check that the outputs are ground. In the `z` case, the output is the same as the second input, so it is ground. In the `s` case, we assume `(s N)` (and therefore `N`) and `M` to be ground, so by induction, `P` is ground, so `(s P)` is ground.

For this example, we can take ground to mean "well-typed" (in the metalogic), as defined by a reasoning-level predicate on natural numbers:

```
Define $nat : nat -> prop by
$nat z ;
$nat (s N) := $nat N.
```

Then our theorem can be stated as follows:

```
Theorem plus_moding : forall N M P,
{plus N M P} -> $nat N -> $nat M -> $nat P.
```

The proof proceeds in exactly the manner sketched above and can be found in the reference files for this document.

2.2 Example 2: Edges and paths

A problem with this typing definition approach comes up in the case where we want to write programs over an open set of data. Consider a logic program that computes paths through a graph:

```
path X X.
path X Z :- edge X Y, path Y Z.
```

[1] This is not to say that is the *only* mode it has: it can also be given mode $(+, -, +)$ or $(-, +, +)$, meaning it can be used for subtraction as well as addition.

Suppose the elements related by path have the type `node`. The mode of path *should be determinable* without mentioning a specific graph (represented by inhabitants of `node` and `edge`). This means we want to leave the signature *open* to certain extensions of the signature that add inhabitants to the types `node` and `edge`.

Because we do not want to mention specific inhabitants, there is no reasoning-level predicate we can write down to give our moding theorem in terms of. As a strawman, suppose we wrote

```
Define $node : node -> prop by
$node N.
```

Now suppose we change our path program to be *ill-moded* for the mode $(+, -)$.

```
pathbad X Y.
```

The theorem

```
Theorem path_moding : forall N M,
{pathbad N M} -> $node N -> $node M.
```

will hold trivially by the (inferred) type of the input M.

Suppose instead we let `$node` be the empty definition. In this case, the same theorem would hold vacuously by casing on the `$node N`, which has no inhabitants.

We need a way to be *parametric* over extensions to the signature. We can achieve this with Abella's type declaration mechanism in the reasoning level:

```
Type $node node -> prop.
```

As this type has no induction principles attached to it, the only way to acquire such a typing is to have it by assumption, induction hypothesis or lemma.

We need one more thing in order to check the mode for path successfully, which is an assumption that `edge` has mode $(+, -)$. This is another property that can be validated or failed by the specific graph in question; by isolating it as a lemma we abstract from a *choice* of graph to a *specification* of graphs.

```
Theorem edge+- : forall X Y, {edge X Y} -> $node X -> $node Y.
skip.
```

The "*skip*" tactic provides no proof and treats the theorem as an axiom.

Now indeed we can prove our mode theorem:

```
Theorem path_moding : forall X Y,
{path X Y} -> $node X -> $node Y.
```

but not the analogous theorem for `pathbad`.

It should be noted at this point that the type declaration approach can be extended to inductive types, albeit in an unsound way, by treating the inductive equivalences as "axioms" (theorems with a skipped proof). For example, if we wanted to leave the world of natural numbers open, we could say

```
Type $nat nat -> prop.
Theorem $natz :  $nat z.
skip.
Theorem $nats : forall X,
  ($nat X -> $nat (s X)) /\ ($nat (s X) -> $nat X).
skip.
```

and still be able to prove the moding theorem for `plus`, using these axioms rather than the supported induction tactics. While the motivation for doing so may be unclear for the case of natural numbers, where the natural treatment of them is as a closed set, consider instead the types for simple lambda terms: we may soundly leave open the set of base types, but still wish to form arrow types between them inductively. The next example illustrates this setting, and, in addition, introduces the complications of contexts and higher-order programs.

2.3 Example 3: Type synthesis

Here is an encoding in lambda prolog of the lambda-calculus with type labels, along with the definition of type synthesis for it. By "synthesis" we mean that we think of the type as an output.

```
%%% sig file %%%

kind  tm, ty  type.

type  arrow   ty -> ty -> ty.
```

```
type  lam     ty -> (tm -> tm) -> tm.
type  app     tm -> tm -> tm.

type  of      tm -> ty -> o.

%%% mod file %%%

of (lam T M) (arrow T U) :- pi x\ (of x T => of (M x) U).
of (app M N) T :- of M (arrow U T), of N U.
```

What should the mode theorem for the `of` predicate look like?
As a first cut, it should be something like

```
Theorem of_moding : forall M T,
{of M T} -> $tm M -> $ty T.
```

First we must say what we mean by `$tm` and `$ty`. We want to allow for constants introduced by a hypothetical signature extension to extend the definition of terms and types, i.e. we will reason about hypothetical base types and terms (like booleans or natural numbers). Then the predicates are defined as reasoning-level types and axioms:

```
Type $ty ty -> prop.
Theorem $tyarrow : forall A B,
  ($ty (arrow A B) -> $ty A /\ $ty B)
    /\
   ($ty A /\ $ty B -> $ty (arrow A B)). skip.

Type $tm tm -> prop.
Theorem $tmapp : forall M N,
  ($tm M /\ $tm N -> $tm (app M N))
    /\
  ($tm (app M N) -> $tm M /\ $tm N). skip.
Theorem $tmlam : forall A M,
  ($ty A /\ (forall x, $tm (M x)) -> $tm (lam A M))
    /\
  ($tm (lam A M) -> $ty A /\ (forall x, $tm (M x))). skip.
```

Note the use of `forall` in the definition of `$tm`.

However, if we try to prove the theorem as stated, we cannot prove the lambda case, as we would need to invoke induction on a proof in an

extended context. We must parameterize the theorem by the context (first defining contexts).

```
Define ctx : olist -> prop by
  ctx nil ;
  nabla n, ctx (of n A::G) := ctx G.

Theorem of_moding : forall G M A,
  ctx G -> {G |- of M A} -> $tm M -> $ty A.
```

However, this gives us an additional case in the induction: the case where `of M A` is a member of `G`.

We can patch this either by requiring of the context that its types be ground (i.e. if `of E T` is in the context, then so must be `$ty T`), or by adding an additional assumption to the theorem. To avoid polluting the standard context definition, we opt for the latter:

```
Theorem ofmode : forall G M A,
  ctx G -> (forall X B, member (of X B) G -> $ty B)
  -> {G |- of M A} -> $tm M -> $ty A.
```

Note that this is an awkward requirement: the context is *only ever extended* by the `of` rules, so we *ought* to be able to perform a reasoning-level analysis of said rules to observe that they never introduce anything random into the context (only types determined by their input terms). But the notion of context is external to the specification, and the only reasoning principles on it we have are the ones we impose at the reasoning level. Andrew Cave's internship project could be applicable to this problem.

2.4 Limitations

Our treatment of moding does not soundly capture Twelf's. To see why, consider the following clause, where `a` is some element of a base type `i` and `p` relates two elements of type `i`.

```
p(X, a) :- p(Y, Z).
```

Under the interpretation above, `p` has mode $(+, -)$. It has one clause, and in that clause the output `a` is a ground atom, which gets translated at the reasoning level to a member of the type `$i`, satisfying the proof obligation.

But running the analogous code through Twelf, we get an error that says the *input* of the subgoal `p(Y, Z)` is "not necessarily ground." Indeed, operationally, this program invokes a predicate with an argument that is not determined by its own inputs: it must invent the input `Y` from thin air. But because the clauses' own output doesn't depend on that invocation, neither does our proof.

For the same reason, any predicate can be shown, by our logic, to have mode (+, ..., +): there are no proof obligations in the theorem. Indeed, the only time a predicate can fail to have all-+ mode is when some clause *uses it in a subgoal* with unground inputs. The moding theorem that Twelf proves requires that the predicate *obey the moding contracts* of other predicates. This treatment works from an operational perspective in terms of "making up inputs", but as a statement about a collection of logical propositions, it seems to make no sense.

Indeed for the sake of doing metatheory, we need not make as picky a distinction as Twelf. Such a mode error would not stop a proof from being correct, it would simply indicate an unneeded subgoal. For now, we take our definition of moding a useful notion distinct from (or, perhaps, an approximation of) Twelf's.

References

[MN86] Dale Miller and Gopalan Nadathur. Higher-order logic programming. In Ehud Shapiro, editor, *Third International Conference on Logic Programming*, volume 225 of *Lecture Notes in Computer Science*, pages 448–462. Springer Berlin / Heidelberg, 1986.

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. *Logical Frameworks*, pages 149–181, 1991.