Chapter 6

Evaluation of Elpi

In this chapter we give an evaluation of Elpi based on our experiences during this thesis. We first discuss where our thesis benefitted from Elpi and Coq-Elpi in section 6.1. Next, in section 6.2, we discuss where Elpi could be improved or where difficulties lie in using it as a commands and tactics meta programming language for our thesis. Lastly, we discuss if Elpi could be used to replace LTaC as the meta programming language for the IPM in section 6.3.

6.1 Advantages of Elpi

We will highlight the advantages of using Elpi as a meta programming language for Coq. We will discuss how logic programming is used and how Elpi interacts with Coq. Lastly, we discuss the documentation of Elpi.

Logic programming in Elpi Elpi is a logic programming language, similar to Prolog. It is not a functional programming language or an imperative language. It works best when making full use of the features of a logic programming language. This includes structuring predicates around backtracking and fully utilizing unification.

Debugging can be a pain point in programs with a lot of backtracking. It often happens that an error only surfaces later after backtracking a few times. However, Elpi includes the excellent Elpi tracer, together with the Elpi trace browser extension [TW23] for the editor Visual Studio Code. It allows one to visually inspect all paths taken by the interpreter while executing the program. This greatly helps in understanding where backtracking happened by mistake. This is very helpful when starting out with a new programming concept like logic programming.

Several other additions Elpi made to λProlog have made it easier and more concise to program in. Spilling helps to greatly reduce explicit intermediary variables. Warnings for variables that are only used once help

reduce typos. And the Elpi database allows for more modular and tactics and commands.

Interacting with Coq Coq-Elpi has worked very well in facilitating the interaction between Coq and Elpi. Quotation and anti-quotation allow for easily creating and extracting Coq terms and greatly reduces noise by embracing the Coq notations.

When you do have to work with the term constructors, the HOAS structure work well. Writing recursive function to create or interpret terms creates clean and readable code. Even though binders behave a bit unexpected as we will touch upon in the next section.

When creating Coq terms, it is essential to make sure they are properly typed. Elpi does not have a typed and an untyped term, like other Coq meta programming languages like LTaC have. But, since you have complete control over when to call the Coq type checker, you often type check a term right before using it in Coq. Thus, greatly reducing the amount of unnecessary type checking. Also encoding the type of binders using the `decl` predicate allows one to circumvent the type checker entirely when possible.

Lastly, when the type checker fails and backtracking is properly handled, the type checking error is automatically shown with the failure of the tactic. This greatly improves the experience for the user when a command or tactic does not work.

Documentation Getting started in Elpi is made easy with the excellent tutorials on writing Elpi code, and creating either a command or tactic. They explain step by step how the logic programming language works and how to interact with Elpi. They explain some mayor caveats and pitfalls and ensure that small programs are easily developed.

The documentation of the standard library of Elpi and Coq-Elpi consist of comments in the source code of the standard libraries. These comments are quite thorough and help explain most of the standard library, but do make the whole process less accessible than either a document or a website containing the documentation for the standard libraries.

6.2 Challenges with Elpi

In this section, we will discuss the challenges we encountered while working with Elpi. Despite its strengths, there are areas where Elpi posed some difficulties. We first discuss how Elpi and LTaC interact. Next, we discuss the difficulties with using binders in Elpi. Then, we show why anonymous predicates in Elpi are prone to bugs. Lastly, we discuss why debugging large programs in Elpi is hard.

Disadvantages of combining Elpi with LTaC In Elpi you have the ability to call LTaC code with the needed arguments, like terms, strings and other types. However, integrating LTaC tactics in an Elpi proof often poses significant challenges.

Since the Coq context is declared by adding rules to the Elpi context, a proof state does not simply consist of a proof variable and a type. It also consists of all the constants and there declared types. When creating proofs in Elpi we incrementally increase the Coq context and with it also the Elpi context. However, when calling an LTaC tactic on a proof variable with arguments the resulting goal has no relation to the old binders used in the proof. This makes passing values throughout the proof very difficult and often results in obscure errors surrounding binders and variables.

The result of these problems is that it is only really feasible to use LTaC tactics when they finish a branch of the proof. Only when no terms have to be passed to subsequent sections of the proof can you use LTaC code in between[1][2].

Binders in Elpi One of the main sources of trouble in the previous paragraph was binders in Elpi. While they are an essential part of the HOAS structure of Coq terms in Elpi, they can work in very unintuitive ways. Every Elpi variable is quantified over all binders it is under at declaration. A variable can thus only contain binders over which it is quantified. This leads to a myriad of errors when for example returning terms created under a binder, or when a variable gets quantified over a binder twice[3].

Binders are an essential and powerful part of Elpi, however, they are also quite unintuitive. And can hinder the features that depend on them.

Anonymous predicates Any anonymous predicates containing intermediary variables are very prone to errors. As described in section 4.7.4thesis.pdf and above, variables are bound in the uppermost predicate they are defined in. Thus, when creating an anonymous predicate where either the predicate is used multiple times, or it is used under a binder, the predicate fails and backtracks when executed. This is mitigated by adding `sigma X\` for every variable `X` at the start of the anonymous predicate. However, when making use of spilling in an anonymous predicate, you don't have access to the intermediary variable, and thus you cannot fix this.

[1] We have successfully allowed for calling the `simpl` tactic using `eiIntros`, however any more complicated LTaC tactics have to be managed carefully.

[2] Our first attempt at the commands and tactics described in this thesis was based on calling LTaC a lot more, it also called the Elpi proof generators as if they were LTaC tactics, thus creating binders for every step of the proof. This resulted in so many undebuggable errors and weird behaviors that we switched directions towards what we now call holes.

[3] This is probably a bug in Elpi, but has happened quite a few times and has let to very strange errors.

This makes anonymous predicates only useful for very small predicates. Any other predicates should be created using the normal `pred` keyword at the top level. However, especially when using CPS, you often need a small predicate which is only used once. And, an anonymous predicate would be useful[4].

Debugging large programs We already discussed how debugging small programs is a benefit of Elpi because of the tracer. However, at the moment the tracer does not function properly under larger programs. The tracer significantly increases the execution time of a program. Furthermore, the created traces are too large for the Visual Studio Code extension to ingest. You can limit traces to only a few predicates. However, this is often not enough to fully grasp the execution because of backtracking. Given that the tracer is no longer usable in debugging programs, the difficulty of debugging Elpi programs becomes apparent.

Elpi programs creating large terms need to print them often during debugging. These large terms are even longer to print as Elpi constructors and when printing using the Coq pretty printer important details can be missed. And, investigating why unclear errors messages occur becomes a lot harder without full introspection in the program trace.

Thus, either you split a program up into multiple stages during development, or you endure the slower and more laborious process of print debugging.

6.3 Elpi as the meta programming language for the IPM

In this section we will discuss the benefits and downsides of using Elpi as the meta programming language for the IPM of Iris.

Firstly, Elpi works best when the entire system is written in Elpi. Thus, when implementing the IPM in Elpi, the entire IPM needs to be ported to Elpi. A significant part of the tactics in the IPM are already implemented as part of this thesis, thus if the switch to Elpi is made, this should be doable.

Switching to Elpi could also come with several benefits. The Elpi database could allow for more modular tactics. For example, the tactic `pm_reduce` reduces a term on only pre-determined definitions. Using the Elpi database definitions could be added to `pm_reduce` whenever they are defined. Also, deeper introspection into the goal and proof term could allow for removing workarounds and creating more powerful tactics. Instead of keeping a fresh anonymous identifier counter for the Iris context, whenever a fresh identifier is needed, one could look thorough the used identifiers and create one that is

[4] In section 4.7.4thesis.pdf is an example where we still used anonymous functions and worked around the problems described here.

not yet used. Given that no type checking or elaboration needs to done during such an operation, this should not induce a significant slowdown. Thus, Elpi could allow for more powerful and modular tactics by making use of Elpi specific features.

However, using Elpi also imposes some downsides besides just the all or nothing approach. Elpi proof generators do not mimic the Coq syntax as closely as the current implementation of IPM tactics do. This makes the barrier to entry a lot higher when creating new tactics or porting existing ones to Elpi. Also, creating tactics in Elpi needs a certain base understanding of the inner working of Coq, that is not necessary with the current implementation. This results in a harder to parse code base with more verbosity.

Porting the IPM to Elpi could be a net benefit if all of the IPM is ported to Elpi. Elpi is continuously getting improved and there are possibilities for features to be added to Elpi to aid in the transition of the IPM to Elpi.