Chapter 6

Evaluation of Elpi

In this chapter, we evaluate Elpi based on our experiences during this work. We first discuss where our work benefited from Elpi and Coq-Elpi in section 6.1. Next, in section 6.2, we discuss where Elpi could be improved and where difficulties lie in using it as a meta-programming language for commands and tactics. Lastly, we discuss if Elpi can 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 works best when making full use of the features of logic programming languages. This includes structuring predicates around backtracking and fully utilizing unification.

Debugging can be a challenge with programs that require extensive backtracking. It often happens that an error only surfaces after backtracking a few times. However, Elpi includes the excellent Elpi tracer and an Elpi trace browser extension [TW23] for the editor Visual Studio Code. It enables one to visually examine all paths taken by the interpreter while executing the program. This helps in understanding where backtracking happened wrongly, and is helpful when starting with a new programming paradigm like logic programming.

Several other additions that Elpi has made to λProlog, made it easier and more concise to program. By using spilling, explicit intermediary variables are reduced. Warnings for variables that are only used once help reduce typos. Finally, the Elpi database allows for more modular 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 using term constructors, the HOAS structure works well. Writing recursive functions to create or interpret terms creates clean and readable code, even though binders can behave unexpectedly, as we will touch upon in the next section.

When creating Coq terms, it is essential to make sure they are properly typed. Elpi has no guarantees that a term is well typed, while other Coq meta-programming languages, such as Ltac do have terms that are guaranteed to be well typed. 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. This reduces unnecessary type-checking. Furthermore, encoding the types 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 improves the experience for the user when a command or tactic does not work.

Documentation Getting started in Elpi is made easier with the excellent tutorials on writing Elpi code to create either a command or tactic. They explain step by step how the logic programming language can be used. They explain some major cautions and pitfalls and ensure that small programs are easily developed.

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

6.2 Issues with Elpi

In this section, we will discuss the challenges we encountered while interacting with Elpi. Despite Elpi's strengths, there are certain areas where it encounters issues. We first discuss how Elpi and Ltac interact. Next, we discuss the difficulties with using binders in Elpi. We then show why anonymous predicates in Elpi are prone to bugs. Lastly, we discuss why debugging large programs in Elpi is difficult.

Disadvantages of combining Elpi with Ltac In Elpi you can call Ltac code with the needed arguments like terms, strings, and other types. Calling Ltac code allows one to more easily migrate from Ltac to Elpi, also it allows one to work around Coq API's not yet implemented in Elpi. However, integrating Ltac tactics into 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 their declared types. When creating proofs in Elpi we incrementally increase the Coq context and thus 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 frequently results in obscure errors surrounding binders and variables.

The result of these issues 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 were binders in Elpi. While they are an essential part of the HOAS structure of Coq terms in Elpi, they can work in 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 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 may hinder the features that depend on them.

Anonymous predicates Any anonymous predicates containing intermediary variables are susceptible 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 an anonymous predicate. However, when using spilling in an anonymous predicate, you do not have access to the intermediary variable. Therefore, it is generally not possible to use spilling in anonymous predicates.

The problems described above make anonymous predicates only useful when they are small. 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 that is only used once. Here, an anonymous predicate would be useful, as seen by the listing in section 4.7.4thesis.pdf. There, we still used anonymous predicates and worked around the issues described here.

Debugging large programs We have previously discussed the advantages of Elpi's tracer in debugging small programs. However, currently, the tracer does not function properly in 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 frequently not enough to fully grasp the execution, given the amount of backtracking. Given that the tracer is no longer usable when 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. Investigating why unclear error messages occur becomes a lot harder without full introspection in the program trace.

[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 implementing the commands and tactics described in this thesis was based on calling Ltac a lot more. This attempt also called the Elpi proof generators as if they were Ltac tactics, thus creating new binders of the entire context for every step of the proof. This resulted in many difficult to debug errors and weird behaviors. Therefore, we switched directions from these Ltac like tactics towards what we now call holes.

[3]This is a bug in Elpi that has been reported.

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

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 representative portion of the tactics in the IPM have already been implemented as part of this thesis. Therefore, the switch to Elpi should be possible.

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. Currently, these definitions need to happen before `pm_reduce` is defined. Furthermore, 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 in the Iris context, one could search through the used identifiers and choose one that has not been used. Given that no type checking or elaboration needs to be 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 drawbacks besides the all-or-nothing approach. Elpi proof generators do not mimic the Coq syntax as closely as the current implementation of IPM tactics does. This raises the barrier to entry when creating new tactics or porting existing ones to Elpi. Additionally, creating tactics in Elpi requires a certain base understanding of the Coq API's. Ultimately, this all results in a harder to parse code base with more verbosity.

Porting the IPM to Elpi could be a net benefit if the whole IPM were to be 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. Some include: Allowing Coq proofs as arguments for commands, like how Coq instances can be declared using interactive proofs. Databases which are local to a proof and reset when the proof is done. String arguments to tactics or commands, which do not have to be surrounded by quotation marks. Full access to the introduction pattern Coq API. And others not yet encountered.