Chapter 1

Introduction

Induction on inductive predicates is a fundamental aspect of reasoning about recursive structures within a logic. Separation logic [ORY01; Rey02] has proven to be a promising basis for program verification of (concurrent) imperative programs. It employs an substructural logic with additional connectives to reason about the heap of a program. Inductive predicates are an essential part of this logic, they allow one to reason about the recursive data structures present in the program.

We make use of an embedding of separation logic in a proof assistant, where all rules of separation logic are derived from the base logical constructs of the proof assistant. As a result, inductive predicates in the separation logic also follow from the logic of the proof assistant. Three major approaches have been found to define inductive predicates: structural recursion, the Banach fixpoint [Ban22], the least fixpoint as inspired by Tarski [Tar55].

- Structural recursion defines an inductive predicate by recursion on an inductive type in the proof assistant logic, e.g., defining an inductive predicate by recursion on lists from the proof assistant.
- The Banach fixpoint defines inductive predicates by guarding the recursion behind the step-indexing present in some separation logics.
- The least fixpoint takes a monotone function, the *pre fixpoint function*, describing the behavior of the inductive predicate. Then, the least fixpoint of this function corresponds to the inductive predicate. The least fixpoint also allows for proving total correctness. Thus, in this thesis, we focus on the least fixpoint approach.

Separation logic has been implemented several times in proof assistants [App06; RKV21; Chl11; BJB12]. We make use of the separation logic Iris [Jun+15; Jun+16; Kre+17; Jun+18], implemented in the proof assistant Coq as the Iris Proof Mode/MoSeL [KTB17; Kre+18]. Iris has been applied for verification of Rust [Jun+17; Dan+19; Mat+22], Go [Cha+19], Scala [Gia+20], C [Sam+21], and WebAssembly [Rao+23].

Defining inductive predicates using the least fixpoint in Iris is a very manual process. Several trivial proofs must be performed, and several intermediary objects must be defined. Furthermore, using the inductive predicates in proofs requires additional manual steps.

This thesis aims to solve this problem by adding several commands and tactics to Coq that simplify and streamline working with inductive predicates. We implement

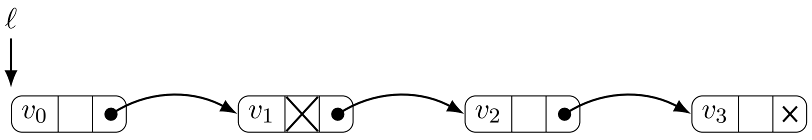

Figure 1.1: A node is shown here as three boxes next to each other, the first box contains a value. The second box is a boolean. The box is true, thus marked, if it is crossed out. The third box is a pointer, denoted by either a cross, a null pointer, or a circle with an arrow pointing to the next node.

our commands and tactics in the λProlog [MN86; Mil+91; BBR99; MN12] dialect Elpi [Dun+15; GCT19]. To use Elpi as a Coq meta-programming language, there exists the Elpi Coq connector, Coq-Elpi [Tas18].

1.1 Central example

Marked linked lists, (MLLs), developed by Harris [Har01], are non-blocking concurrent linked lists. They are the central example used in this thesis. We will use a sequential version here to give a preview of the system we developed.

MLLs, are linked lists where each node has an additional mark bit. When a node is marked, and thus the bit is set, the node is considered deleted. An example of a MLL can be found in figure 1.1. An MLL allows for deleting a node out of a list without modifying any of the other nodes, helping with concurrent usages.

In order to reason about MLLs in separation logic, we relate a heap containing an MLL to a list in the separation logic in the IPM. Using our newly developed system, this can be achieved similarly to writing any other inductive predicate.

```coq
1 eiInd
2 Inductive is_MLL : val → list val → iProp :=
3     | empty_is_MLL : is_MLL NONEV []
4     | mark_is_MLL v vs l tl :
5       l ↦ (v, #true, tl) -∗ is_MLL tl vs -∗
6       is_MLL (SOMEV #l) vs
7     | cons_is_MLL v vs tl l :
8       l ↦ (v, #false, tl) -∗ is_MLL tl vs -∗
9       is_MLL (SOMEV #l) (v :: vs).
```

The first line applies the command `eiInd` (for “Elpi Iris Inductive”) to a Coq inductive definition. Next, on line 2, we define the name of the inductive predicate `is_MLL`, together with its type `val → list val → iProp`. The predicate takes a value `val`, which will be the location of the MLL, and a list of values, representing the values in the MLL. Lastly, the type ends in `iProp`, which is the type of Iris propositions over heaps.

The first constructor `empty_is_MLL`, on line 3, relates an empty MLL, `NONEV` to an empty Coq list `[]`. The second constructor `mark_is_MLL`, on lines 4-6, relates a MLL where the first node is marked to a Coq list. `is_MLL (SOMEV #l) vs` holds if the location `l` points to a node which is marked, and the tail of the MLL, `tl`, is represented by the Coq list `vs`. This case can be found in figure 1.1 starting at node v_1.

The last constructor `cons_is_MLL`, on lines 7-9, is similar to the previous constructor, but adds the value found at location `l` to the list on line 9. This constructor holds for Figure 1.1.

The above inductive statement defines the inductive predicate `is_MLL`, together with the unfolding lemmas, constructor lemmas, and induction lemma. When we have a goal requiring induction on an `is_MLL` statement, we can simply call the `eiInduction` tactic on it. We then get goals for all the cases in the inductive predicate with the proper induction hypothesis.

This definition would not been possible using the Coq `Fixpoint`. The `Fixpoint` requires structural recursion on one of the arguments of `is_MLL`. The only candidate for structural recursion would be the second argument, since it is the only inductively defined type in the arguments. However, when a node is marked, the Coq list of values is not modified. Thus, this case is not structurally recursive and the Coq `Fixpoint` cannot be used.

1.2 Approach

Currently, to define an inductive predicate using the least fixpoint in the IPM, several steps have to be taken. First, the pre fixpoint function has to be defined. This function will model one step of the inductive predicate. Next, this predicate has to be proven monotone. Then, both the pre fixpoint function and the monotonicity proof have to be uncurryed to apply the least fixpoint lemma. Lastly, we define a curryed version of the least fixpoint applied to the pre fixpoint function. From the least fixpoint we do get the induction property and unfolding lemmas. However, they do have to be applied manually. This results in several proofs and manual intermediary definitions.

To automate this process, we take the following approach. We create the command, `eiInd`, as shown above, which is given an inductive definition in Coq, generates the pre fixpoint function, proves it monotone, and defines the fixpoint for the arity of the pre fixpoint function. Next, it proves the fixpoint properties of the defined fixpoint and generates constructor lemmas. Lastly, it generates and proves the induction lemma.

To use the inductive predicate, we create two tactics. The `eiInduction` tactic applies the induction lemma on the specified hypothesis. The `eiDestruct` tactic eliminates an inductive predicate into its possible constructors.

To accomplish these goals, we reimplement a subset of the IPM tactics in Elpi as *proof generators*. Proof generators take a hole in a proof and inhabit that hole with a proof term, any holes left in the created proof term they return as holes. These proof generators are used to generate the proof for the induction properties and are exported as IPM tactics, namely `eiIntros`, `eiSplit`, `eiEvalIn`, `eiModIntro`, `eiExFalso`, `eiClear`, `eiPure`, `eiApply` (without full specialization), `eiIntuitionistic`, and `eiExact`. The tactics themselves also allow us to evaluate how and if Elpi could be used to reimplement the full IPM.

The source code for this thesis can be found at `https://github.com/lukovdm/MasterThesisIrisElpi`.

1.3 Contributions

This thesis contains the following contributions.

Generation of Iris inductive predicates We develop a system written in Elpi that, given an inductive definition in Coq, defines the inductive predicate with associated unfolding, constructor, and induction lemmas. In addition, tactics are created that automate unfolding the inductive predicate and applying the induction lemma. *(Chapter 5thesis.pdf)*

Modular tactics in Elpi We present a way to define steps in a tactic, called *proof generators*, such that they can easily be composed. Allowing one to define simple proof generators that can be reused in many tactics. *(Section 4.7thesis.pdf)*

Generate monotonicity proof of n-arity predicates We present an algorithm which given an n-arity predicate can find a proof of monotonicity. *(Section 3.3thesis.pdf)*

Evaluation of Elpi Lastly, we evaluate Elpi with Coq-Elpi as a meta-language for Coq. We also discuss replacing Ltac with Elpi in IPM. *(Chapter 6thesis.pdf)*

1.4 Outline

We start by giving a background on Separation logic in chapter 2thesis.pdf. The chapter discusses the Iris separation logic while specifying and proving a program on MLLs. Next, in chapter 3thesis.pdf, we discuss defining representation predicates in a separation logic using least fixpoints. Thus, we show how to define a representation predicate as an inductive predicate, and then give a novel algorithm to prove it is monotone. In chapter 4thesis.pdf, we give a tutorial on Elpi by reimplementing an IPM tactic, `iIntros`. Building on the foundations of chapter 4thesis.pdf, we create the command and tactics to define inductive predicates in chapter 5thesis.pdf. In chapter 6thesis.pdf, we evaluate what was useful in Elpi and what could be improved. We also discuss how and if Elpi can be used in IPM. Lastly, we discuss related work in chapter 7thesis.pdf and show the capabilities and shortcomings of the created commands and tactics in chapter 8thesis.pdf, together with any future work.

Notation During the thesis, we will be working in two different programming languages. To always distinguish between them, the inline displays have a different color. Any `Coq displays` have a light green line next to them. Any `Elpi displays` have a light blue line next to them. Full-width listings also differentiate using green and blue lines, respectively.