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. It employs an affine 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 recursive data structures present in the program.

We make use of an embedding of separation logic in an interactive proof assistant. As a result, inductive predicates in the separation logic cannot be an axiom, and have to follow from the base logic of the proof assistant. Three mayor approaches have been found to define inductive predicates: structural recursion, the Banach fixpoint [Jun+18], the least fixpoint [App14].

- Structural recursion defines an inductive predicate by recursion on an inductive type in the base logic, e.g., defining an inductive predicate by recursion on lists defined by the proof assistant.
- The Banach fixpoint define inductive predicates by guarding the recursion behind the step-indexing present in some separation logics.
- The least fixpoint is the most general approach, it 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, whereas Banach fixpoints only allow proving partial correctness. Thus, in this thesis we focus on the least fixpoint approach.

Separation logic has been implemented several times in proof assistants [AR; 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 WebAssemble [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 a proof 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 our commands and tactics in the λProlog language Elpi [Dun+15; GCT19]. Elpi is a λProlog

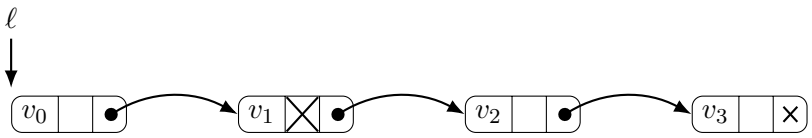

Figure 1.1: A node is shown here as three boxes next to each other, the first box contains a value. The second box is a boolean. The box is true, thus marked, if it is crossed out. The third box is a pointer, denoted by either a cross, a null pointer, or a circle with an arrow pointing to the next node.

dialect [MN86; Mil+91; BBR99; MN12]. 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), develop by Harris [Har01], are non-blocking concurrent linked lists. They are the central example used in this thesis. We will use them here to give a preview of 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
eiInd
Inductive is_MLL : val → list val → iProp :=
    | empty_is_MLL : is_MLL NONEV []
    | mark_is_MLL v vs l tl :
      l ↦ (v, #true, tl) -∗ is_MLL tl vs -∗
      is_MLL (SOMEV #l) vs
    | cons_is_MLL v vs tl l :
      l ↦ (v, #false, tl) -∗ is_MLL tl vs -∗
      is_MLL (SOMEV #l) (v :: vs).
```

The command `eiInd` (for "Elpi Iris Inductive") is prepended to the inductive statement and 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 hypotheses.

1.2 Approach

To generate inductive properties from their inductive definition using the least fixpoint we will take the following approach. We create the command, `eiInd`, as shown above. Which, given an inductive definition in Coq, generates the pre fixpoint function, proofs

in 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` tactics, which applies the induction lemma on the specified hypothesis. And, the `eiDestruct` tactic, which 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*, i.e., taking a goal in Elpi and producing a proof term which inhabits that goal. We also use these proof generators to reimplement several other IPM tactics, namely `eiIntros`, `eiSplit`, `eiEvalIn`, `eiModIntro`, `eiExFalso`, `eiClear`, `eiPure`, `eiApply` (without full specialization), `eiIntuitionistic`, and `eiExact`.

1.3 Contributions

This thesis contains the following contributions.

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

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 which can be reused in many tactics. *(Sec. 4.7)*

Generate monotonicity proof of n-ary predicates We present an algorithm which given an n-ary predicate can find a proof of monotonicity. *(Sec. 3.3)*

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. *(Ch. 6)*

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 monotone. In chapter 4thesis.pdf, we give a tutorial on Elpi by implementing an IPM tactic, `iIntros`, in Elpi. 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 usefull 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. In order 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.