Institut für Programmstrukturen
und Datenorganisation (IPD)

Lehrstuhl Prof. Dr.-Ing. Snelting

An Improved Interface for Interactive Proofs in Separation Logic

Masterarbeit von

Lars König

an der Fakultät für Informatik

```
theorem proof_example [BI PROP] (P Q R : PROP) (Φ : α → PROP) :
    P ∗ Q ∗ □ R ⊢ □ (R -∗ ∃ x, Φ x) -∗ ∃ x, Φ x ∗ P ∗ Q
:= by                                Iris Proof Mode
    iintro ⟨HP, HQ, □HR⟩ □HRΦ        --------------------------
    ispecialize HRΦ HR as HΦ         HR : R
    icases HΦ with ⟨x, HΦ⟩           HRΦ : R -∗ ∃ x, Φ x
    iexists x                        HΦ : ∃ x, Φ x
    isplit r                         -------------------------- □
    · iassumption                    HP : P
    isplit l [HP]                    HQ : Q
    · iexact HP                      -------------------------- ∗
    · iexact HQ                      ∃ x, Φ x ∗ P ∗ Q
```

Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuender Mitarbeiter:	M. Sc. Sebastian Ullrich
Abgabedatum:	30. September 2022

Abstract

Seit Software entwickelt wird, stellt sich die Frage, ob diese korrekt ist, d.h. ob sie das tut, was sie tun soll. Gegeben eine formale Spezifikation der Anforderungen, ist eine Aufgabe der Softwareverifikation also zu beweisen, ob eine Implementierung diese Spezifikation erfüllt. Diese Aufgabe kann schwierig zu lösen sein, wenn die verwendete Programmiersprache Befehle mit globalem Effekt erlaubt, sodass diese andere Befehle in unabhängigen Teilen des Programms beeinflussen können, zum Beispiel durch einen gemeinsam genutzten Heap-Speicher. *Separation-Logic* löst dieses Problem, indem es Aussagen um einen separierenden Operator erweitert, wodurch es möglich ist, Teile eines Programms als unabhängig vom Rest des Programms anzusehen. Ein Werkzeug, das Beweise zur Softwareverifikation unterstützt, sind *interaktive Theorembeweiser*. Allerdings benötigen Separation-Logic-Beweise in interaktiven Theorembeweisern, besonders mit nicht-linearem Typsystem, viel manuellen Aufwand zur Verwaltung der benötigten Datenstrukturen. Dies kann vermieden werden, indem dem Nutzer eine Schnittstelle zur Verfügung gestellt wird, die Beweise auf der typischen, höheren Abstraktionsebene ermöglicht. Diese Arbeit beschreibt eine neue Schnittstelle für Separation-Logic-Beweise in dem interaktiven Theorembeweiser *Lean 4*, basierend auf dem *Iris*-Projekt, und die Verbesserungen an dieser Schnittstelle.

For as long as software has been developed, a major concern has been to know whether it is correct in the sense that is does what it is supposed to do. Given a formal specification of the requirements, one task of software verification is to prove that an implementation fulfills the specification. This task can be difficult if the used programming language allows commands to have global effect, such that they can influence commands in unrelated parts of the program, for example through a shared heap. *Separation logic* aims to solve this problem by adding a separating operator to propositions, which makes it possible to treat parts of a program as independent of the rest. One tool that supports software verification proofs are *interactive theorem provers*. However, performing separation logic proofs in interactive theorem provers, especially with non-linear type systems, requires a lot of boilerplate code and manual bookkeeping. This can be avoided by providing the user with an interface that allows proofs on the common, higher abstraction level. This work describes a new separation logic interface for the interactive theorem prover *Lean 4* based on the *Iris* project and the improvements made to this interface.

Contents

1. Introduction

One of the main tasks of software verification is proving the correctness of software relative to a specification. Although this can already be difficult for pure functions, it becomes much more complex if functions are allowed to modify a shared global state, e.g., a shared data structure or a heap. Operations that modify a global state are often implemented using pointers or references. With the ability to modify parts of the memory unrelated to the task of the function, specifications and correctness proofs have to laboriously account for any global view of the function.

As O'Hearn et al. noticed, this is contrary to the informal understanding of the effect of functions [1]. Instead, specifications should focus on the part of the memory affected by a function. For unrelated parts, it should be possible to conclude that they are unchanged automatically. This is achieved by using *separation logic* [1] for the specifications and correctness proofs. Separation logic introduces the separating conjunction $*$ (read "sep") from the logic of bunched implications (BI) [2] to separate parts of a data structure. The proposition $P * Q$ then states "that [the propositions] P and Q hold for separate parts of a data structure" [1]. When describing the effect of a function on a data structure using pre- and postconditions, both can then be extended with the same proposition on an unrelated part of the data structure. This allows specifying the *local* effect of the function, which can automatically be extended to the *global* effect where necessary. Separation logic can furthermore be used to reason about concurrent programs [3] where two functions on separate parts of a data structure cannot interfere, even if they are executed concurrently.

One platform for software verification proofs are interactive theorem provers (ITPs), which support the user by providing a comprehensive proof interface. The available tools include logic notation, proof state displays, context management for hypotheses and functions for simple and complex steps in a proof. Performing separation logic proofs in an ITP requires extending this proof interface with new notation, extended displays, additional contexts and adapted proof steps. A proof interface for separation logic hides the functions and proofs necessary for embedding the separation logic in the base logic of the ITP, which is especially relevant if the ITP does not support substructural logics by default. The interface should also be extensible for custom separation logics, providing entry points to integrate a separation logic in the available facilities.

The aim of this work is to enable interactive separation logic proofs in the ITP *Lean 4* [4] by adding an extensible interface for separation logic. The provided implementation includes the embedding of separation logic in Lean's non-linear base logic and supports writing separation logic proofs with added notation and tactics. In addition, the management of the separation logic contexts is available together with a Lean-style display that integrates seamlessly in the available interface. As an example, the new interface is instantiated with *classical separation logic* [5] and a custom notation to show its extensibility. The syntactic and semantic description of a language model using separation logic in the pre- and postconditions of its specifications is however out-of-scope and suggested for future research on the topic.

The implementation of the new interface largely relies on the formalization of *MoSeL* [5, 6] in the different ITP *Coq*. MoSeL is an extensible separation logic framework included in the *Iris* project [7], which features its own concurrent separation logic with a complex algebra and advanced features. MoSeL succeeds the *Iris Proof Mode* [8] and can be instantiated with arbitrary affine and non-affine separation logics. Re-using parts of the implementation of MoSeL in the new separation logic interface for Lean is possible since the two ITPs have similar type systems. Their approaches for meta-programming are however different and Lean has a more flexible front end, useful, for example, for the syntax of the included tactics. The new implementation also improves the internal representation of the separation logic contexts, visible in the related functions and proofs. In addition, the interface is adapted to match the established conventions in Lean.

This work first explains the theory of separation logic, including bunched implications, in chapter 2 and gives an introduction to Lean in chapter 3. The introduction includes relevant features, such as typeclasses, macros and meta programming. The implementation of the new separation logic interface is presented in chapter 4, featuring the definition of separation logic in Lean, the provided notation and the necessary context management with the context display extension. In addition, the implemented tactics for separation logic proofs are shown together with examples of correctness proofs, available for all definitions. The interface is evaluated in chapter 5 where the instantiation for classical separation logic is shown, as well as an example of a proof using the new interface. It is then compared to MoSeL regarding its extent, the chosen proof styles and the improvements in this implementation. Lastly, a limitation of the interface is discussed together with possible solutions, preventing the instantiation of the interface for the entire Iris logic. Chapter 6 summarizes the improvements to interactive proofs in separation logic using Lean and points to possibilities for future research.

2. Separation Logic

An important feature found in many programming languages are pointers in a heap, i.e., a shared part of the memory. While this enables powerful algorithms, it also comes with difficulties for software verification. When reasoning about a small part of an algorithm that touches only a small part of the shared memory, one does not have guarantees that other parts of the memory remain unchanged. This requires manual work and a global view on the memory to ensure that local changes only have local effect. This is contrary to the informal understanding of functions as small building blocks of software and introduces unnecessary complexity.

Separation logic [1], explained in section 2.2, is an extension of Hoare logic that aims to solve this problem by providing constructs for separating the part of the memory that is required by an algorithm from the rest. This separation comes with a guarantee that the remaining part of the memory remains unchanged. This allows performing a proof on the local changes without having to worry about the global state of the memory. Separation logic builds on the logic of bunched implications, using its logical connectives to express the separation of memory parts. The logic of bunched implications is explained in section 2.1. Separation logic can be extended to reason about concurrent programs as well, including the separation of memory parts between processes, as shown in section 2.3. One modern concurrent separation logic is Iris [7]. The separation logic interface MoSeL [5] introduced with Iris is the basis of the implementation described in this work. Therefore, although the complex separation logic of Iris itself is not part of the implementation, a high-level overview is given in section 2.4 for the sake of completeness.

2.1. Bunched Implications

Linear logic [9] introduces additive and multiplicative logic connectives by restricting the use of weakening (adding a hypothesis to the context) and contraction (removing a duplicate hypothesis from the context) in *intuitionistic logic (IL)*. Weakening and contraction are only possible in the additive part of linear logic, but not in the multiplicative part. Intuitionistic logic can be defined in terms of linear logic, but it is not just the additive part of it. One reason is that linear logic does not contain the intuitionistic implication $\rightarrow$.

The logic of *bunched implications (BI)* [2] now combines IL as the additive part with the multiplicative part of linear logic (called *multiplicative intuitionistic linear logic (MILL)*). The resulting logic contains the additive connectives of IL, including the intuitionistic implication $\rightarrow$, and the multiplicative connectives of MILL, except for the multiplicative disjunction, as shown in figure 2.1. The connectives *emp*, $*$ (read "sep") and $-\!\!*$ (read "wand") are the multiplicative counterparts of $True$, $\wedge$ and $\rightarrow$. As with linear logic, weakening and contraction are only allowed for the additive connectives.

Additive	$True$, $False$, $\wedge$, $\vee$, $\rightarrow$
Multiplicative	emp, $*$, $-\!\!*$

Figure 2.1.: Additive and multiplicative connectives. The symbols used for the unit connectives $True$, $False$ and emp are defined as in MoSeL [5], while they have different symbols (1, $\perp$ and I) in the original definition of BI [2].

However, when considering logical consequences of the form $\Gamma \vdash \psi$, having two conjunctions, one that admits weakening and contraction and one that does not, leads to the question whether propositions in the context Γ can be duplicated and discarded or not. In BI this is solved by introducing different context-forming operations ";" and "," for the additive and multiplicative combination of propositions in Γ. As with the connectives, weakening and contraction are only allowed for the additive combination. The introduction rule for $\wedge$ then uses ";" as its context-forming operation, while the introduction rule for $*$ uses ",", as shown in figure 2.3. The same is true for the intuitionistic (additive) implication $\rightarrow$ and the linear (multiplicative) implication $-\!\!*$ where the former uses ";" and the latter uses "," as the context-forming operation in its introduction rule. The consequence is that the premise of $-\!\!*$ must be used exactly once in the proof of the conclusion, while the premise of $\rightarrow$ can be used an arbitrary number of times by duplicating or discarding the premise in the context.

$$\begin{aligned} \Gamma ::= {} & \phi && \text{propositional assumption} \\ & \mid \Gamma ; \Gamma && \text{additive combination} \\ & \mid \Gamma , \Gamma && \text{multiplicative combination} \end{aligned}$$

Figure 2.2.: A bunch is a tree-like structure with propositions ϕ as leafs and additive (";") or multiplicative (",") context-forming operations as inner nodes.

With different context-forming operations, the context Γ can no longer be represented as a list of propositions. Instead it is a tree-like structure (called *bunch*) with propositions ϕ as leafs and additive (";") or multiplicative (",") combination as inner nodes, as shown in figure 2.2. The propositions $True$ and emp are defined to be the units of additive and multiplicative combination, respectively. This is a simplification from the original work, where the empty contexts $\{\}_a$ and $\{\}_m$ are used instead. The definition of the deduction rules for BI is shown in figure 2.3.

Identity and Structure

$$\frac{}{\phi \vdash \phi}\ \text{Id}$$

$$\frac{\Gamma(\Delta) \vdash \phi}{\Gamma(\Delta; \Delta') \vdash \phi}\ \text{Weakening} \qquad \frac{\Gamma(\Delta; \Delta) \vdash \phi}{\Gamma(\Delta) \vdash \phi}\ \text{Contraction}$$

Additives

$$\frac{\Gamma \vdash \phi \quad \Delta \vdash \psi}{\Gamma; \Delta \vdash \phi \wedge \psi}\ \wedge\text{-Intro} \qquad \frac{\Gamma(\phi; \psi) \vdash \chi \quad \Delta \vdash \phi \wedge \psi}{\Gamma(\Delta) \vdash \chi}\ \wedge\text{-Elim}$$

$$\frac{\Gamma; \phi \vdash \psi}{\Gamma \vdash \phi \to \psi}\ \to\text{-Intro} \qquad \frac{\Gamma \vdash \phi \to \psi \quad \Delta \vdash \phi}{\Gamma; \Delta \vdash \psi}\ \to\text{-Elim}$$

$$\frac{\Gamma \vdash \phi_i}{\Gamma \vdash \phi_1 \vee \phi_2}\ (i \in \{1,2\})\ \vee\text{-Intro} \qquad \frac{\Gamma \vdash \phi \vee \psi \quad \Delta(\phi) \vdash \chi \quad \Delta(\psi) \vdash \chi}{\Delta(\Gamma) \vdash \chi}\ \vee\text{-Elim}$$

$$\frac{\Gamma \vdash \mathit{False}}{\Gamma \vdash \phi}\ \mathit{False}\text{-Elim}$$

Multiplicatives

$$\frac{\Gamma \vdash \phi \quad \Delta \vdash \psi}{\Gamma, \Delta \vdash \phi * \psi}\ *\text{-Intro} \qquad \frac{\Gamma(\phi, \psi) \vdash \chi \quad \Delta \vdash \phi * \psi}{\Gamma(\Delta) \vdash \chi}\ *\text{-Elim}$$

$$\frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \mathrel{-\!\!*} \psi}\ {-\!\!*}\text{-Intro} \qquad \frac{\Gamma \vdash \phi \mathrel{-\!\!*} \psi \quad \Delta \vdash \phi}{\Gamma, \Delta \vdash \psi}\ {-\!\!*}\text{-Elim}$$

Figure 2.3.: Deduction rules for the logic of bunched implications. The notation $\Gamma(\Delta)$ describes a bunch Γ in which a bunch Δ appears as a subtree.

2.2. Separation Logic

Similar to Hoare logic, separation logic [1] is concerned with statements $\{P\}\, C\, \{Q\}$ where P and Q are the pre- and postconditions of a code fragment C. In a language that contains pointers in a shared data structure (e.g., a heap), these conditions can contain propositions that describe the state of cells in that data structure. The central idea of separation logic is to use the separating conjunction $*$ from BI, explained in section 2.1, to combine two propositions P and R that hold for separate parts of the data structure. In consequence, P and R can be proven separately with each proof considering only the relevant part of the data structure (*parallel rule*). In general, this property is described by the *frame rule* shown in figure 2.4. It allows writing a local specification for a code fragment without the need to include the unmodified parts of the data structure - they remain unchanged by design.

$$\frac{\{P\}\, C\, \{Q\}}{\{P * R\}\, C\, \{Q * R\}}\quad \text{Modifies}(C) \cap \text{Free}(R) = \emptyset$$

Figure 2.4.: The *frame rule* of separation logic. Modifies(C) is the set of all variables assigned in C, while Free(R) is the set of all free variables in R. Since C requires a corresponding resource to modify a variable and R is added using the separating conjunction, it is required that the two sets are disjoint.

One interpretation of separation logic [1] is that the propositions in the precondition are resources that are required by the code to modify the associated cells and then used by the proof of the postcondition. The resources are *spatial* and must be used exactly once. This is visible in the definition of the frame rule in figure 2.4 where the proposition R must be in used in the pre- and postcondition. Non-spatial resources can be modeled using the weakening and contraction rules for additive connectives.

Figure 2.5 shows an example usage of the *frame rule* in a language with simple heap cell assignments. In the example, the assignment $[x] := b$ sets the value of the heap cell x to b. Using the *frame rule*, it is now possible to derive that the heap cell y remains unchanged, i.e., if the value at y was c before, it will still be c after the assignment. This is without knowing anything about the actual values of x and y. The separating conjunction $*$ alone guarantees that the two heap cells are different

$$\frac{\{x \mapsto a\}\, [x] := b\, \{x \mapsto b\}}{\{(x \mapsto a) * (y \mapsto c)\}\, [x] := b\, \{(x \mapsto b) * (y \mapsto c)\}}\ \text{Frame}$$

Figure 2.5.: Example of applying the *frame rule* to the specification of the assignment of a single heap cell.

and rules out that x and y could be aliases, something that otherwise would have to be checked manually. This example demonstrates the locality of the specification of a single heap cell assignment, which implicitly includes that the remaining heap remains unchanged.

2.3. Concurrent Separation Logic

Having a guarantee that two parts of a program don't interfere on a shared data structure simplifies proofs of sequential programs, but it is even more useful if these two program parts are executed in different processes or threads. The easiest example of applying concurrent separation logic [3] contains two program parts C and C' that are executed concurrently (written as $C \,||\, C'$). Figure 2.6 shows that if C and C' operate on disjoint parts of a data structure, it is guaranteed that they do not interfere when executed concurrently. This is expressed by combining the pre- and postconditions of the two program parts with the separating conjunction $*$. The rule of *disjoint concurrency* then states that the correctness of C and C' can be proven separately.

$$\frac{\{P\}\, C\, \{Q\} \quad \{P'\}\, C'\, \{Q'\}}{\{P * P'\}\, C \,||\, C'\, \{Q * Q'\}}$$

Figure 2.6.: The rule of disjoint concurrency for two concurrently executed program parts C and C'.

Seeing that two program parts that work on different parts of a data structure don't interfere is not surprising. But a lot of common use cases of concurrency include program parts that work on the same part of a data structure, just not at the same time. Fortunately, concurrent separation logic supports proofs of these programs as well. The core idea is based on the resource interpretation of separation logic with the addition that resources can be passed from one process to another. A concurrency primitive (e.g., a semaphore) is then seen as a resource manager that allows the process that owns the managed resource to access the associated part of a shared data structure. When a process has finished accessing the shared data structure, it returns the resource to the resource manager and another thread can receive it from there. The required specification for each instance of a resource manager is encoded in separation logic invariants. This approach enables proofs about concurrent programs that share memory between processes with guarantees that these processes never interfere.

2.4. Iris Logic

Iris [10, 7] is a higher-order concurrent separation logic based on resource algebras (an extension of partial commutative monoids) and invariants that allows specifying protocols for concurrent programs. Invariants allow gaining temporary access to a shared resource that must be given up when the invariant is reestablished after the operation. Since invariants can depend on other invariants and even themselves, Iris uses *step-indexing* to formulate rules in which a proposition P holds after one additional step of computation. This is written as $\triangleright P$ where $\triangleright$ is called the *later* modality. This requires a new kind of equivalence $\overset{n}{=}$ which comes from an *ordered family of equivalences (OFE)*. This equivalence is used to state that two propositions "are equivalent for n steps of computation", i.e., "they cannot be distinguished by a program running for no more than n steps." [7]

In addition to the shared heap memory (the *physical state*), Iris introduces a logical state, the so-called *ghost state*, whose cells have values from the user-defined resource algebra. Since this ghost state is independent of the physical state, it can be updated at any time through so-called *view shifts*. This concept is useful to keep track of the computation history of a program, but also more generally to "describe a thread's knowledge about a shared state" [10]. This allows the sharing of resources beyond an exclusive ownership of resources. To ensure that a value in the ghost state mirrors a value in the physical state, one can use invariants that require updating the value in the ghost state whenever the corresponding physical value is modified.

With its complex logic, Iris aims at "simplifying and consolidating the foundations of modern separation logics" [7], which is visible in the extensibility of the logic and its proof framework. Iris is not tied to a specific programming language, but can be instantiated with an arbitrary language instead.

3. Lean 4

Interactive theorem provers (ITPs) are integrated development environments for proving and checking formal statements. They are successfully used in mathematics and software verification. Lean [4], in the upcoming version 4, is a fully extensible ITP and functional programming language. It features dependent types, typeclasses, hygienic macros, meta programming and efficient code generation in a single language and enables coarse and fine grained automation of proof steps.

```
theorem and_replace (hr : R) :
    P ∧ Q → P ∧ R
:= by
    intro hpq
    apply And.intro ?left ?right
    · exact hpq.left
    · exact hr
```

```
· case left
R P Q : Prop
hr : R
hpq : P ∧ Q
⊢ P

· case right
R P Q : Prop
hr : R
hpq : P ∧ Q
⊢ R
```

Figure 3.1.: The left side shows an example of a proof in Lean 4. The `intro` tactic destructs an implication and makes the premise available in the local context. The `apply` tactic replaces the goal with one or more goals that imply the original goal and the tactic `exact` solves the goal by providing a proof term for it. On the right, the state after using the `apply` tactic is shown. For each goal (`left` and `right`), the state includes the available hypotheses and the goal.

Figure 3.1 shows an example of a theorem and its proof in Lean on the left side. The first two lines contain the statement to prove and its arguments, which can also be proofs. Following the keyword `by` is the proof in *tactic mode*. The proof is written by the user, but automation tactics make it possible to delegate parts of the proofs. If executed in a supported editor, the state between two tactic invocations is shown to the user, including the local context with all available hypotheses and the *goal*, as shown on the right side of figure 3.1. When starting the proof, the goal is exactly the theorem statement, but tactics may change the goal by, for example, destructing or solving parts of it. During this process, the tactics generate a proof term for the original statement, which is verified by the kernel when the proof is finished.

```
def List.get (list : List α) (idx : Fin list.length) : α :=
    ...
```

Figure 3.2.: The type of the second argument is *dependent* on the value of the first argument.

Lean is based on the *Calculus of Inductive Constructions with Universes* [11]. Building on the Curry-Howard correspondence, it enables having logical statements and proofs as common objects in a typed functional programming language. More advanced features of Lean's type system include inductive and *dependent types*. Dependent types are types that depend on elements of other types. In the example shown in figure 3.2, the *type* of the second argument `idx` depends on the *value* of the first argument `list`.

3.1. Typeclasses

Typeclasses in Lean [12] are a way of enabling *ad-hoc polymorphism* for functions. In contrast to parametric polymorphism, this allows functions to behave differently on inputs of different types. This can be useful for overloaded operators on different types, but also for requiring additional properties for certain types, for example that a relationship is reflexive.

```
class Semigroup (α : Type) where
    mul : α → α → α
    mul_assoc : ∀ a b c : α, mul (mul a b) c = mul a (mul b c)

instance : Semigroup Nat where ...

def mul_idemp_right_id [Semigroup α] (a b : α) ...
```

Figure 3.3.: Example of the use of typeclasses in Lean. `class` declares a `structure` that can be used in instance implicit arguments. `instance` defines a value of the typeclass that participates in typeclass instance search. Arguments in square brackets are instance implicit, i.e., they are not required to be explicitly provided, but synthesized by typeclass resolution.

A typeclass in Lean is an inductive datatype which can be used as the type of an *instance implicit* argument. Typeclasses are most often structures, i.e., datatypes with a number of fields. Typeclass instances are elements of these types that participate in *typeclass instance search*. When a definition or theorem requires an instance of a typeclass to be present by declaring it as an instance implicit argument, the instance is searched in the context of the definition or theorem. An example of this is shown in figure 3.3.

Typeclasses in Lean are allowed to overlap for given arguments and depend on other typeclass instances. Resolving typeclass instances therefore requires a backtracking search through the available instances. In a naïve backtracking search, however, this can cause exponential search times or non-termination, which is why Lean uses a *tabled typeclass instance search* [12] that "caches" found typeclass instances to avoid these problems in most cases. The backtracking typeclass instance search can be used to perform simple logic programming, although this is limited by the requirement that parameters have to be either *input* or *output* parameters. By default, parameters of a typeclass are input parameter, which means they have to be known to start the search for a specific typeclass instance. Output parameters have to be declared explicitly when declaring the typeclass by marking the parameter types with `outParam`. The value of the argument is then determined by the found instance during typeclass instance search. Mixed parameters, i.e., parameters that can be input or output parameters, are not supported to preserve the determinism of the typeclass instance search, following the approach for typeclasses with multiple parameters used in Haskell [13].

3.2. Macros

Extensible syntax is a typical feature of ITPs, because it reduces the cognitive overhead when processing complicated structures or expressions. This can either be used to declare new syntax that can be mixed with existing syntax, but also to establish a domain specific language (DSL) where terms are built from the new syntax exclusively. Lean features a powerful system for extending syntax [14], explained in this section.

```
infixr:25 " ⊢ " => entails

notation:25 P:26 " ⊢ " Q:25 => entails P Q

macro:25 P:term:26 " ⊢ " Q:term:25 : term => `(entails $P $Q)

syntax:25 term:26 " ⊢ " term:25 : term
macro_rules
    | `($P ⊢ $Q) => `(entails $P $Q)
```

Figure 3.4.: Examples using the most common commands for declaring new syntax in Lean. Each of the command sequences declares the same syntax with the same interpretation.

The macro system in Lean allows the user to define new syntax on different levels of abstraction, from simple notations to complex syntax transformations, such that users can choose the simplest solution that is powerful enough for their use case. Figure 3.4 shows the most common commands for declaring new syntax and its interpretation: `infix`, `notation`, `macro`, `syntax` and `macro_rules`. The `infix` command creates an infix operator with a symbol ("⊢" in the example), replacing each use of the operator with a call to the provided function ("`entails`" in the example). The user can optionally provide the precedence ("25" in the example) and associativity ("r" in the example) of the defined operator. Similar commands exist for prefix and postfix operators. The `notation` command allows a more complex expression on the right hand side where the named arguments ("`P`" and "`Q`" in the example) can be used. It is also possible to assign custom precedences to the arguments ("26" and "25" in the example). The `macro` command has two powerful features: it allows specifying the syntax category of the arguments and the result (`term` in the example) and providing the syntax to which the macro is expanded explicitly. This can be done using syntax quotations ("`` `( ) ``"), but also an arbitrary function that returns a syntax object. Macros with syntax generating functions are called *procedural macros*. The `macro` command actually consists of two steps, which can be executed manually: declaring the syntax using the command `syntax` and specifying how to expand the macro using the command `macro_rules`. Separating the two concerns allows reusing and overloading syntax, for example by matching on different types of arguments to the macro when defining its behavior.

```
notation "inc " x => add x 1

def calculate : Nat :=
    let add (a b : Nat) :=
        a + 1 + b
    inc 2
```

Figure 3.5.: An example of a macro where naïve syntax replacement would lead to *accidental name capture*, i.e., the macro would use the local function add instead of the global function in scope at the definition of the macro. In this case, the function would return 4, even though the expected behavior of `inc` is to increase its argument by one.

Macros in Lean are not implemented as naïve textual replacement, which would create problems such as *(accidental) name capture* shown in figure 3.5, but by performing *hygienic macro expansion* [14], which keeps the identifiers in an expanded macro definition inaccessible from the surrounding code. A recent advancement in Lean introduces an additional feature when defining macros called *typed macros* [15]. Using a type system for syntax categories, this prevents generating ill-formed syntax when defining macros and allows Lean to disambiguate more variable uses in syntax quotations.

3.3. Meta Programming

Lean 4 is the first version almost completely written in Lean itself, which makes it easier to provide the comprehensive reflection framework. This allows writing meta programs as normal Lean functions running in monads which provide access to internal data structures. Relevant internal data includes the environment, i.e., available definitions and theorems, and the tactic state, including the goals and their local contexts [16]. A common use case for meta programming is tactic programming, i.e., writing custom tactics that can be used when writing a proof in tactic mode.

The simplest way of writing tactics in Lean is using macros. This works by defining a macro in the syntax category `tactic`, which expands to one or more tactics. An example of this is shown in figure 3.6 where a new tactic `close` is defined that executes the tactic `simp`, passing the given identifier of a definition or theorem, followed by the tactic `done`. The newly defined tactic can then be executed like any other tactic.

```
macro "close " name:ident : tactic => `(tactic|
    simp [$name:ident] ;
    done
)
```

Figure 3.6.: Example of a macro-based tactic.

A more powerful way of writing tactics, especially when it is not possible to build them from existing tactics, is writing a meta function in the tactic monad `TacticM` with access to the relevant internal data structures. A separate tactic programming language on a higher abstraction level does not yet exist in Lean.

Figure 3.7 shows typical tasks of a monadic tactic function. The macro `elab` expands to a syntax definition and a monadic function of type `Syntax → TacticM Unit`, matching on the defined syntax to extract the arguments. For tactics with input parameters, a tactic function first needs to parse the given `Syntax` objects. This can be as simple as retrieving the identifier from a `Syntax` object, but can also include parsing a pattern from a custom syntax category. For the main part of the tactic function, two important data structures are the goals and the list of hypotheses per goal, reflecting the tactic state shown in figure 3.1. The main goal can be retrieved using the function `getMainGoal`. The returned value is a meta variable of the type `Expr`, representing a proof term expression in the kernel. This meta variable can be solved by assigning an expression to it using `assignExprMVar`. Here, the assigned expression is allowed to contain meta variables itself. The list of hypotheses available in the main goal is part of the local context, accessed using `getLCtx`. The local context can be searched by name, as shown in the example, but it is also possible to loop over the available hypotheses and, for example, inspect their types. The types and values of declarations from the local context are all of the type `Expr`, since the tactic function operates on the meta level.

```
elab "exact " name:ident : tactic => do
    -- parse syntax
    let name := name.getId
    if name.isAnonymous then
        throwUnsupportedSyntax

    -- retrieve main goal
    let goal ← getMainGoal

    -- find hypothesis in local context
    let lctx ← getLCtx
    let some decl := lctx.findFromUserName? name
        | throwError "hypothesis not found"

    -- assign expression to solve goal
    assignExprMVar goal decl.toExpr
```

Figure 3.7.: Example of a monadic tactic.

A combination of a monadic tactic function and calls to existing tactics can be achieved by using the function `evalTactic` that executes the given tactic mode syntax. An example of this technique is shown in figure 3.8. Variables of type `Syntax` can be used directly in the syntax of `evalTactic`, e.g., arguments of the surrounding tactic function. If common objects should be used in a call to `evalTactic`, they need to be converted to the syntax level first, using the function `quote`. This function performs a best-effort conversion based on the typeclass `Quote`.

```
elab "my_rfl" : tactic => do
    let hyp : Name ← findReflexivityTheorem

    evalTactic (← `(tactic|
        exact $(quote hyp)
    ))
```

Figure 3.8.: Example use of `evalTactic` to use a macro-style call to an existing tactic inside a monadic tactic function. The function `quote` converts a common object to a `Syntax` object.

4. Implementation

This chapter describes and explains the implementation of the new separation logic interface in Lean 4. It is largely based on the Coq formalization of MoSeL [5]. There are four main parts of the implementation: Section 4.1 provides the formalization of the logic, including the separation logic interface, the Lean notation for the logic and the derived laws. Since the base logic of Lean is non-linear, explicit management of the hypothesis contexts is necessary, for which an environment is presented in section 4.2. To visualize these contexts, a custom display for the tactic state is included as well. An important part of the implementation are the separation logic tactics, explained in section 4.3, which enable performing proofs on the same level of abstraction as in usual Lean logic. Section 4.4 concludes by showing the proofs done in the implementation, which includes proofs for the derived laws, the properties of propositions in form of typeclasses, the tactic theorems and the soundness of operations on the separation logic context.

4.1. Logic, Interface and Notation

The starting point for the implementation is the separation logic interface, which can be instantiated with a custom separation logic. The interface is implemented as a typeclass and split up into two parts: the interface `BIBase`, shown in figure 4.1, which includes the required connectives for a separation logic, and an extension of that interface called `BI`, which adds the axioms that each separation logic must fulfill. The interface is split up because the separation logic notation for the connectives is defined in between and used in the definition of the axioms in the second interface. A mutual definition of a typeclass and its notation is currently not possible in Lean.

Most of the connectives in `BIBase` resemble the BI connectives introduced in chapter 2. This includes the additive connectives `and`, `or` and `impl`, as well as the multiplicative connectives `emp`, `sep` and `wand`. The connectives `forall` and `exist` are the common additive variants from intuitionistic logic. The function `entails` is used to express a logical consequence of separation logic propositions (of type `PROP`) inside the base logic of Lean. The type of propositions in Lean is `Prop`, while the type `PROP` for separation logic propositions is the *carrier type* of the separation logic and can be defined by the user when instantiating the logic. Although the logic of bunched implications is used as the interface for separation logics, this implementation does not support arbitrary proofs in general BI, as described by Krebbers et al. for MoSeL [5]. Instead of a separate context type, as introduced in section 2.1 and shown in figure 2.2, a separation logic proposition is used on the left side of the

```
class BIBase (PROP : Type) where
    entails : PROP → PROP → Prop
    pure : Prop → PROP

    and : PROP → PROP → PROP
    or : PROP → PROP → PROP
    impl : PROP → PROP → PROP

    emp : PROP
    sep : PROP → PROP → PROP
    wand : PROP → PROP → PROP

    forall {α : Type} : (α → PROP) → PROP
    exist {α : Type} : (α → PROP) → PROP

    persistently : PROP → PROP
```

Figure 4.1.: Interface for the required connectives of a separation logic. This is an incomplete interface for a separation logic, since it misses the axioms that every separation logic must fulfill. The complete interface `BI` is an extension of this interface and adds the separation logic axioms. Note the difference between the type `Prop` of Lean propositions and the type `PROP` of separation logic propositions.

entailment. The different context-forming operations ";" and "," then correspond to the connectives $\land$ and $*$. This not only simplifies the definition but also allows an easier embedding of the context representation in this implementation, which is explained in section 4.2. A Lean proposition ϕ can be embedded in the separation logic as a *pure* separation logic proposition, written as $\ulcorner\phi\urcorner$. The unit connectives *True* and *False* of separation logic are defined as the pure propositions `True` and `False`. The function `persistently` is used to implement the `<pers>` *modality*, which adds additional properties to a separation logic proposition. Modalities are explained in subsection 4.1.1.

The interface `BI` extends the interface `BIBase` with the axioms required to be fulfilled by all separation logics. None of the axioms in this interface can be derived from the other axioms and all derived laws of separation logic build on these axioms. The first axiom in the interface states that the entailment relation on separation logic propositions is a *preorder*, i.e., it is reflexive and transitive. The axiom is written using the typeclass `PreOrder`, which bundles the typeclasses `Reflexive` and `Transitive`. Having one typeclass per property is useful when a definition or theorem requires a reflexive or transitive relation, but does not require it to be a preorder. Another important axiom is the introduction rule for pure propositions: $\phi \to P \vdash \ulcorner\phi\urcorner$ (note that $\to$ is the Lean implication, not the separation logic implication). It states that, for all contexts P, a pure separation logic proposition $\ulcorner\phi\urcorner$ can be proven by proving the Lean proposition ϕ without having to use the propositions from the context. This also makes an introduction rule for the separation logic proposition $True$ unnecessary, since it is defined as $\ulcorner$`True`$\urcorner$ and can therefore be proven by using the introduction rule from the base logic. The remaining axioms are the deduction rules of separation logic, building on the deduction rules for bunched implications shown in figure 2.3. Examples include the introduction rule for $\wedge$: $(P \vdash Q) \to (P \vdash R) \to P \vdash Q \wedge R$ ($\to$ is again the Lean implication) and the statements that emp is the left unit of $*$: $P \vdash emp * P$ and $emp * P \vdash P$.

4.1.1. Modalities

A modality is a unary operator for a separation logic proposition that assigns additional properties to it. Modalities are required to be monotone, i.e., if $P \vdash Q$ holds, then for all modalities m, $m\,P \vdash m\,Q$ must also hold. In addition, they must distribute over $*$. There is only one required modality, persistency, which cannot be derived from the other connectives of separation logic and is therefore part of the interface. Other modalities can be derived using the separation logic connectives and users of the interface can extend the logic with their own modalities. A separation logic proposition without modalities is spatial, i.e., it cannot be duplicated or discarded when used in a separating conjunction. The *persistency* modality `<pers>` changes this by adding two properties to the affected proposition: it allows the duplication of propositions, i.e., `<pers>` $P \vdash$ `<pers>` $P *$ `<pers>` P, and it makes the proposition absorbing, which allows discarding other propositions combined with a separating conjunction: `<pers>` $P * Q \vdash$ `<pers>` P. This property is called "absorbing", because the proposition absorbs the resources of the other proposition, i.e., the resources are not actually discarded. The idea of persistency is that the affected proposition ignores all attached resources. Since they are ignored, additional resources can be added, and duplication is possible since the proposition cannot depend on any of the ignored resources. Pure propositions, including $True$ and $False$, are always persistent. It turns out that for persistent propositions, $\wedge$ and $*$ are interchangeable. In both cases, propositions can be duplicated and discarded as long as at least one proposition in the conjunction remains, such that the held resources are never discarded.

$$\begin{aligned} \texttt{<affine>}\; P &:= emp \wedge P \\ \texttt{<absorb>}\; P &= True * P \\ \Box\, P &:= \texttt{<affine>}\; \texttt{<pers>}\; P \end{aligned}$$

Figure 4.2.: Definitions of the predefined, derived modalities.

There are three more predefined modalities which are derived from the separation logic connectives. Their definitions are shown in figure 4.2. The *absorbing* modality

`<absorb>` is a weaker version of `<pers>`, which does not make the proposition duplicable. This is visible in the definition: *True* is pure, and therefore persistent, and can be used to absorb other propositions. In contrast, P in the definition does not have any modalities and is therefore not duplicable. The *affine* modality `<affine>` on the other hand allows discarding the affected proposition itself in a separating conjunction: $P \ast$ `<affine>` $Q \vdash P$. This is achieved by placing the proposition in a non-separating conjunction with *emp*. A proposition in a non-separating conjunction can always be discarded, since the propositions in a non-separating conjunction share the held resources. However, when P is discarded, only *emp* remains, which is the unit of separating conjunction and can therefore also be discarded. Since *emp* states that no resources are known and the conjunction can be reduced to *emp*, affine propositions are not allowed to contain resources and can therefore be discarded. The *intuitionistic* modality $\Box$ is then added to create propositions with the same properties as in intuitionistic logic: they can be duplicated and discarded, but are not absorbing. For this purpose, it is important that `<affine>` is the outer modality, since otherwise the proposition would only be persistent. This is because a proposition `<pers>` `<affine>` P can contain resources, which is not allowed for affine propositions. There are also conditional variants of all four modalities, written as, for example, $\Box?p\,P$, that add the modality to the proposition iff a boolean variable `p` is true.

4.1.2. Notation

The functions in the separation logic interface are already enough to write logical statements in separation logic, but they quickly become unreadable without a notation. Compare the two equal statements in figure 4.3 for an example. When defining the notation for a system, an important question is whether to define it as a closed domain specific language (DSL) or to integrate it in the available term syntax. The first choice allows tighter control over the available constructs, but in the case of separation logic, it is in fact necessary to be able to mix common syntax constructs with separation logic notation. This is used, for instance, in the definition of conditional modalities: $\Box?p\,P :=$ `if` p `then` $\Box\,P$ `else` P. Mixing elements from separation logic syntax and Lean's syntax alone would be as easy as defining the new syntax in Lean's default syntax category `term`. However, problems arise when syntax is reused, for example in the case of $\wedge$. Although it is possible to overload syntax,

Without notation:

```
    entails (and (persistently P) Q) (sep P Q)
```

With notation:

`<pers>` $P \wedge Q \vdash P \ast Q$

Figure 4.3.: Comparison of equal statements without and with a notation for separation logic.

using the same operator for Lean propositions and separation logic propositions, which can occur together in the same statement, confuses Lean's type inference. To avoid this, the separation logic syntax is defined in the syntax category `term` with the additional notation ``` `[iprop| t] ``` to indicate that a term `t` should be interpreted as a separation logic proposition. Separation logic syntax that occurs outside of this quotation does not have an interpretation. Syntax without an interpretation in separation logic inside the `iprop` quotation is interpreted in `term` as usual. There is also the notation ``` `[term| t] ``` to force the `term` interpretation of overloaded syntax inside an `iprop` quotation. Note that the behavior of the `iprop` quotations, including the fallback to `term`, is automatically available in Coq by declaring a *notation scope* for separation logic propositions. In addition, using notation scopes allows specifying a scope for the arguments of functions and typeclasses at declaration time. Automatically inserting `iprop` quotations for certain arguments would also be possible in Lean, but is not currently part of this implementation.

The notation for the separation logic entailment is required to use the more powerful `macro` command instead of the `infix` command for infix operators, since the latter does not allow modified arguments in the expanded syntax. This is however required to pass the `iprop` quotations to the arguments, such that the arguments are also interpreted as separation logic notation. The same technique is used for most operators, as demonstrated on a few examples in figure 4.4. When a notation requires syntax that is not already defined in Lean, it is first added using the `syntax` command. The `macro_rules` command is then used to define the interpretation of the new or overloaded syntax. Notice that the left side of the syntax replacements also contains the `iprop` quotations, which limits the syntax interpretation to separation logic propositions. This is not necessary for the entailment, since a statement $P \vdash Q$ is a Lean proposition, not a separation logic proposition (see figure 4.1 for the definition). It is therefore often not necessary for the user of the separation logic interface to use the ``` `[iprop|] ``` quotation, since statements in separation logic are typically logical consequences of the form $P \vdash Q$, which can be written without additional quotations as explained. One use case where `iprop` quotations are required is when writing standalone separation logic propositions, for example to use them as arguments for a

```
macro:25 P:term:29 " ⊢ " Q:term:25 : term =>
    `(BIBase.entails `[iprop| $P] `[iprop| $Q])

syntax:35 term:36 " ∗ " term:35 : term
macro_rules
    | `(`[iprop| $P ∧ $Q]) => `(BIBase.and `[iprop| $P] `[iprop| $Q])
    | `(`[iprop| $P ∗ $Q]) => `(BIBase.sep `[iprop| $P] `[iprop| $Q])
    | `(`[iprop| True]) => `(BIBase.pure True)
```

Figure 4.4.: Examples of the definition of separation logic notation. The syntax for the operator ∧ is already defined in Lean and can be reused/overloaded.

typeclass as shown in figure 4.5. The last example in figure 4.4 is the definition of *True* in separation logic, which is defined as the pure separation logic proposition `True`. Since `True` on the right side of the syntax replacement is a Lean proposition, it is written without an `iprop` quotation.

It is also possible to define additional notation beyond the separation logic connectives, even after the logic has been instantiated. Predefined additional notation includes the notation for the predefined derived modalities and the conditional modalities, as well as bidirectional variants of the entailment and the separating and non-separating implication. The notation $\vdash P := emp \vdash P$ is introduced to simplify statements with an empty context. There are also the so-called *big operators*, which fold an operator over a list of propositions, e.g., $[\wedge]\, Ps = P_1 \wedge P_2 \wedge P_3$ if Ps contains the propositions P_1, P_2 and P_3.

```
class Persistent [BI PROP] (P : PROP) where
    persistent : P ⊢ <pers> P
class Absorbing [BI PROP] (P : PROP) where
    absorbing : <absorb> P ⊢ P

instance intuitionisticallyPersistent [BI PROP] (P : PROP) :
    Persistent `[iprop| □ P]
where
    persistent := ...

instance persistentlyAbsorbing [BI PROP] (P : PROP) :
    Absorbing `[iprop| <pers> P]
where
    absorbing := ...
```

Figure 4.5.: Examples of typeclasses that require properties of a modality for a proposition. Note that $\square\, P$ is not in relation to `<absorb>` P due to the different directions of the entailments used in the typeclasses. The instance implicit argument `[BI PROP]` requires that the type `PROP` is the carrier type of a separation logic instance. Elements of type `PROP` are then propositions in this separation logic.

4.1.3. Typeclasses

Typeclasses are widely used through the separation logic framework to require certain properties for given propositions. The most basic typeclasses assert that a proposition fulfills the properties of a modality. The typeclasses `Persistent` and `Absorbing` are shown as two examples in figure 4.5 together with an instance of each typeclass. Note that the two typeclasses use the modality on different sides of the entailment. The typeclass `Persistent` requires that the proposition P is enough to deduct the proposition `<pers>` P from it. This means, since `<pers>` P can be duplicated, P must be duplicable as well. One group of propositions for which this holds, is intuitionistic propositions, i.e., propositions with the $\Box$ modality, which is expressed by the instance `intuitionisticallyPersistent`. The proposition `persistent` to prove in `intuitionisticallyPersistent` is then $\Box\, P \vdash$ `<pers>` $\Box\, P$, which does hold. The typeclass `Absorbing` works in the opposite direction. In fact, it is easy to prove that $P \vdash$ `<absorb>` P holds for every P. The reason is that it is always possible to generate an additional $True$, which is pure and can therefore absorb other propositions, but it cannot be discarded. The typeclass `Absorbing` instead requires that P can be deducted from `<absorb>` P, which means that P must be able to hold the potential additional resources that were absorbed by `<absorb>` P. This is true for persistent propositions and visible in the typeclass instance `persistentlyAbsorbing`. These typeclasses can also be instantiated with custom modalities or constructs defined by the user of the separation logic interface to indicate that these groups of propositions fulfill the properties of the predefined modalities.

Instead of requiring properties of propositions, it is also possible to specify properties of an entire separation logic instance. For example, there are *affine* logics, in which every proposition is affine. The corresponding typeclass `BIAffine` extends the typeclass `BI` and requires that, for each proposition `P`, there must be a typeclass instance of `Affine P`. The typeclass also makes this instance available, such that it is not required to write `<affine>` P for every proposition when working with affine logics. Definitions or theorems requiring an instance of `Affine` for a proposition will then find the instance generated by `BIAffine`. There are other possible properties of separation logic instances, but none of them are currently part of this implementation.

```
abbrev Relation (α : Type) := α → α → Prop

class PreOrder (R : Relation α) extends Reflexive R, Transitive R
class Comm (R : Relation α) (f : β → β → α) where
    comm {x y : β} : R (f x y) (f y x)
```

Figure 4.6.: Examples of general-purpose typeclasses on relations. A `Relation` is defined to be a proposition on two values of the same type α.

This implementation also comes with typeclasses that are not exclusive to separation logic, but can be used on any relation. Two examples of typeclasses on relations are shown in figure 4.6. One of them is the typeclass `PreOrder`, of which an instance for the entailment relation is required in the separation logic interface represented by `BI` and `BIBase` (see figure 4.1). The typeclass `PreOrder` combines the properties of `Reflexive` and `Transitive`. The typeclass `Comm`, on the other hand, requires a new property to prove that a function f is *commutative* for all x and y in a relation R on the result type α of f. An example instance of this typeclass for separation logic is shown in figure 4.8.

```
class inductive TCOr (T U : Sort _)
    | l : [T] → TCOr T U
    | r : [U] → TCOr T U
```

Figure 4.7.: Definition of the typeclass `TCOr`, which implements a logical disjunction on typeclasses. The type of T and U is `Sort _`, which is a superset of `Prop` and `Type` and includes arbitrary typeclasses.

As mentioned in section 3.1, it is also possible to perform simple logic programming with typeclasses. This is used to require that a proposition should have at least one of two properties, which requires a logical disjunction on typeclasses. Since Lean does not include typeclasses for logic programming, they are also part of this implementation. The typeclass `TCOr` is shown in figure 4.7. `TCOr` is a typeclass based on an inductive datatype, which means there are multiple constructors with which an instance of the typeclass can be created. The two constructors each take exactly one of the two typeclasses T and U in the disjunction. This means that an instance of either typeclass is enough to instantiate `TCOr`. An example usage of this typeclass is shown in figure 4.9. There is also the typeclass `TCIte`, which requires an instance of one of two typeclasses depending on a boolean condition. The condition can, however, not be an arbitrary boolean expression, since Lean reduces only functions marked with the attribute `reducible` during typeclass instance search, which is not the case for most of the predefined boolean operators. There are workarounds for simple boolean expressions, which is enough for the requirements of this implementation.

```
instance sep_comm [BI PROP] :
    Comm (α := PROP) (· ⊣⊢ ·) (`[iprop| · ∗ ·])
where
    comm := ...
```

Figure 4.8.: Example instance of the typeclass `Comm`, shown in figure 4.6, where the relation R is the bidirectional entailment $\dashv\vdash$ and the binary function f is $*$. The proposition `comm` to prove is $x * y \dashv\vdash y * x$ for all separation logic propositions x and y. The center dot $\cdot$ in parentheses is a placeholder for an abstracted argument, i.e., `(1 + ·)` is expanded to `fun a => 1 + a`. Multiple center dots refer to different arguments and are processed in the order of their appearence in the term.

4.1.4. Derived Laws

The derived laws in this implementation are theorems that are not required to be provided by the user of the separation logic interface. Instead, they are derived from the set of separation logic axioms in the typeclass `BI`. The resulting propositions are valid for all instantiations of the separation logic interface and therefore all separation logics. The necessary proofs of the derived laws are discussed in subsection 4.4.1.

One part of the derived laws are the propositions in the instances of typeclasses, which specify properties of separation logic propositions and connectives. Examples of two instances of typeclasses describing propositions were shown in figure 4.5. An additional example of an instance of a typeclass describing a connective is presented in figure 4.8. The instance `sep_comm` states that the separation logic connective $*$ is commutative using the typeclass `Comm`. With this instance, the function `comm` can be used in proofs where the commutativity of $*$ is required. Another important instance is `entails_anti_symm`, which proves that the entailment is antisymmetrical with the bidirectional entailment as its equivalence relation. This means that if $P \vdash Q$ and $Q \vdash P$ then $P \dashv\vdash Q$.

```
theorem affinely_and_l [BI PROP] {P Q : PROP}
    : <affine> P ∧ Q ⊣⊢ <affine> (P ∧ Q)

theorem sep_elim_l [BI PROP] {P Q : PROP} [TCOr (Affine Q) (Absorbing P)]
    : P ∗ Q ⊢ P
```

Figure 4.9.: Examples of derived separation logic laws.

Most of the derived laws, however, are theorems proving additional properties of separation logic propositions and connectives. Two examples of derived laws are shown in figure 4.9. The first theorem `affinely_and_l` states that an `<affine>` modality on the left operand of $\wedge$ can be moved out to the entire conjunction and vice-versa. At first this may seem unintuitive, since discarding one operand is different from discarding the entire conjunction, but the definition of `<affine>` adds an *emp* to the conjunction, which can be used to discard each operand or the entire conjunction. Note that, like $*$, $\wedge$ is commutative and associative. The second theorem `sep_elim_l` shows that theorems can include typeclass instances to require that propositions have certain properties. In this case, the theorem requires either Q to be affine or P to be absorbing. If Q is affine, it can be discarded, and if P is absorbing, it allows discarding Q as well. This is specified using the typeclass `TCOr` introduced in figure 4.7. The theorem holds if at least one of the typeclass instances can be found. It is not necessary that, for example, P is exactly `<absorb>` P' for some P'. Figure 4.5 demonstrates this by showing an instance of `Absorbing` for `<pers>` P.

4.2. Environment and Goal Display

In section 2.1 on bunched implications, *bunches* were introduced to serve as context Γ in a statement $\Gamma \vdash \psi$ of logical consequence. A structured representation instead of a list is necessary to keep track of which parts of the context allow weakening and contraction. The introduction and elimination rules for $\rightarrow$ and $-\!\!*$, however, show that there is a duality of propositions on the left side of $\vdash$ and propositions in the premise of an implication or wand, depending on the context-forming operation. This implementation therefore follows the approach used in MoSeL and defines the entailment $P \vdash Q$ based on the definition of the separating implication $-\!\!*$. In this definition, P and Q are therefore both separation logic propositions. The context-forming operations ";" and "," in Γ then correspond to the conjunctions $\wedge$ and $*$ in P. The additive variants both admit weakening and contraction, while the multiplicative variants do not. As an example, compare the context representation $P, (Q_1; Q_2)$ using bunches with the representation as a separation logic proposition: $P * (Q_1 \wedge Q_2)$. Both the conjunctions and the context-forming operations are associative and commutative, which allows arbitrary reorderings in a multiplicative or additive part of the context. Note that it is not possible to distribute the operators over each other.

```
inductive Env (α : Type)
    | nil  : Env α
    | cons : α → Env α → Env α

structure Envs (PROP : Type) [BI PROP] where
    intuitionistic : Env PROP
    spatial        : Env PROP

def of_envs [BI PROP] : Envs PROP → PROP
    | ⟨Γp, Γs⟩ => `[iprop| □ [∧] Γp ∗ [∗] Γs]

def envs_entails [BI PROP] (Δ : Envs PROP) (Q : PROP) : Prop :=
    of_envs Δ ⊢ Q
```

Figure 4.10.: Definition of the environments, i.e., the separation logic context, including the embedding in a statement of logical consequence. The function of_envs extracts the two context parts Γ_p and Γ_s from its first argument of the type Envs PROP using pattern matching and the anonymous constructor notation $\langle a, b \rangle$ where a and b are the fields of a structure in the order of their definition.

While the form $P \vdash Q$ is convenient for the definition of separation logic, it is impractical to work with in separation logic proofs in an ITP. The separation logic interface should instead allow the user to introduce hypotheses and list the available hypotheses. This mimics the behavior of Lean, which was shown on the right side of figure 3.1. For that purpose, the context is divided into an intuitionistic part of hypotheses that can be duplicated and discarded and a spatial part of hypotheses that must be used exactly once. Each part of the context, called an *environment* in this implementation, is then represented as a list of hypotheses, encoded in the inductive datatype `Env`. The entire separation logic context is implemented as the structure `Envs`, which contains two instances of `Env`, one for the intuitionistic and one for the spatial part, as shown in figure 4.10. The Lean context is sometimes referred to as *pure context*, since it contains hypotheses that can be used in pure separation logic propositions, as explained in section 4.1.

In order to use the `Envs` structure as the context P in a statement of logical consequence $P \vdash Q$, there must be an embedding as separation logic proposition. This embedding is performed by the function `of_envs` shown in figure 4.10. For the spatial environment Γ_s, the embedding is as simple as combining the contained hypotheses with $*$, written as $[*]\,\Gamma_s$ using a big operator. The spatial conjunction ensures that the hypotheses cannot be duplicated or discarded, which is the definition of the spatial context. Note that the individual hypotheses can have modalities, which can permit duplicating or discarding them. For the intuitionistic environment Γ_p, it must be ensured that the contained hypotheses can be duplicated and discarded. For that purpose, each hypothesis is assigned with an intuitionistic modality. However, for convenience, the intuitionistic modality is written in front of the big operator in the expression $\Box\,[\wedge]\,\Gamma_p$. Note that $\Box$ distributes over $\wedge$, while for $*$ this is not true for arbitrary separation logics. The two environment representations are then combined with $*$ to a single separation logic proposition. An example of an `Envs` object and its representation as a separation logic proposition is shown in figure 4.11. The function `envs_entails` further shows how to construct a statement of logical consequence from an `Envs` object Δ and a separation logic proposition Q.

`Envs` object:

`{ intuitionistic := [Q1, Q2]`$_e$`, spatial := [P, <affine> R]`$_e$` }`

Representation as separation logic proposition:

$\Box\,(Q1 \wedge Q2) * P *$ `<affine>` R

Figure 4.11.: Example of an `Envs` object and its representation as a separation logic proposition as defined by the function `of_envs`. Note that the big operators $[\wedge]$ and $[*]$ are already expanded in this example. The notation `[ ]`$_e$ is used to indicate that the intuitionistic and spatial contexts are objects of type `Env`.

As shown in figure 3.1, the user assigns names to the introduced hypotheses and refers to them by name when they are used in a proof. It is, however, not desirable to attach the names to the hypotheses in the definition of `Envs`, since the operations on `Envs` and their proofs of soundness should not be concerned with (potentially failing) string lookups. Instead, the hypotheses are referenced using indices of the dependent type `Fin`, which ensures that the contained index value is a valid index for the target data structure. For `Envs`, this is implemented in the inductive datatype `EnvsIndex`, which contains either an index for the intuitionistic or the spatial part. The definition of `EnvsIndex` is shown in figure 4.12. For a more convenient creation of indices on the meta level, the arguments of `EnvsIndex` are the length of the intuitionistic and spatial part instead of the Env objects themselves. It is, however, possible to directly require an index for a given Envs object using the function EnvsIndex.of.

```
structure Fin (n : Nat) where
    val  : Nat
    isLt : val < n

inductive EnvsIndex (lₚ lₛ : Nat)
    | p : Fin lₚ → EnvsIndex lₚ lₛ
    | s : Fin lₛ → EnvsIndex lₚ lₛ

abbrev EnvsIndex.of [BI PROP] (Δ : Envs PROP) :=
    EnvsIndex Δ.intuitionistic.length Δ.spatial.length
```

Figure 4.12.: Definition of EnvsIndex which is used to reference hypotheses in Envs. The definition of Fin is included in Lean and only shown for completeness. Note that the datatype Nat contains only non-negative numbers.

Using EnvsIndex is a clean way of indexing a hypothesis in the context, but the user should still be able to refer to hypotheses by their assigned name. To achieve that, the names are stored on the meta level where the tactic functions and the goal display are defined. On the meta level, the hypotheses are represented as kernel expressions of type Expr. The definition of Expr allows attaching arbitrary information in a key-value store to an expression using the constructor Expr.mdata. Attached meta data is ignored by the kernel when processing expressions. When a name is assigned to a hypothesis, it is added to the Expr representing the hypothesis as meta data. When the user refers to an expression by name, the list of all hypotheses (represented as Exprs) in both parts of the context is searched for a hypothesis with this name attached as meta data. When the name is not assigned to any hypothesis, the lookup fails in the implementation of the tactic and no EnvsIndex object will be created. The string lookup in the context is therefore irrelevant for the implementation of operations on Envs. As a consequence, the assigned names also do not appear in the proof terms generated for separation logic proofs. Note that this is different from the Coq implementation of MoSeL, where the names are stored directly in Envs. A comparison of both approaches is shown in section 5.3.

As an example, an application of envs_entails is shown in figure 4.13 as a kernel expression. The separation logic context in envs_entails contains a single proposition in the intuitionistic context with the name HQ attached as meta data. The applications of the function envs_entails and the constructors of Envs and Env are represented as combinations of the Expr constructors app and const. The additional arguments for the separation logic proposition type PROP and the required typeclass instances are omitted for readability. The free variables `_uniq.65 and

`` `_uniq.64 `` in the `Expr` constructor `fvar` represent the separation logic goal and the hypothesis `HQ`, respectively.

$$\begin{aligned}
\Phi &:= \texttt{app (app (const `envs\_entails) } \Delta\texttt{) (fvar `\_uniq.65)} \\
\Delta &:= \texttt{app (app (const `Envs.mk) } \Gamma_p\texttt{) } \Gamma_s \\
\Gamma_p &:= \texttt{app (app (const `Env.cons) } P\texttt{) (const `Env.nil)} \\
P &:= \texttt{mdata [(`name, DataValue.ofName `HQ)] (fvar `\_uniq.64)}
\end{aligned}$$

Figure 4.13.: Example of the (simplified) representation of a Lean goal Φ containing a separation logic context Δ in an application of `envs_entails` as a kernel expression of type `Expr`. Placeholders for expression terms are marked in green.

The tactic implementations do not directly modify `Envs`, but use the provided operations shown in figure 4.14. Each of the operations has a soundness proof which is used in the correctness proof of the tactic implementations. The `Env` datatype has the same structure as a common list, which includes that elements are added in the front of the list using the constructor `Env.cons`. This is reasonable since the environment indices implemented in `EnvsIndex` count the hypotheses front-to-back. It is however not the preferred way of adding hypotheses to a context (spatial or intuitionistic) when performing a proof.

```
def append [BI PROP] : (p : Bool) → (P : PROP) → (Δ : Envs PROP) →
    Envs PROP

def delete [BI PROP] : (rp : Bool) → (Δ : Envs PROP) →
    (i : EnvsIndex.of Δ) → Envs PROP

def lookup [BI PROP] : (Δ : Envs PROP) → (i : EnvsIndex.of Δ) →
    Bool × PROP

def replace [BI PROP] : (Δ : Envs PROP) → (rp : Bool) →
    (i : EnvsIndex.of Δ) → (p : Bool) → (P : PROP) → Envs PROP

def split [BI PROP] : (Δ : Envs PROP) → (mask : List Bool) →
    (mask.length = Δ.spatial.length) → Envs PROP × Envs PROP
```

Figure 4.14.: Signatures of important operations on `Envs`. The shown functions are implemented using additional functions on `Env`. In Lean, tuple types are written as `A × B` where `A` and `B` are types.

When the user introduces a hypothesis to a context, it should be shown on the bottom of the list of hypotheses in the respective context, as shown in figure 3.1.

This is done using the function `append`, which adds a hypothesis P to the end of a context in Δ. The context is determined by the parameter p where `true` indicates that the hypothesis is added to the intuitionistic context and `false` that it is added to the spatial context. The parameter is defined as `Bool` in order to use it in conditional modalities like $\square ?p\, P$. A hypothesis can be removed from a context with the function `delete` where the `EnvsIndex` i references the hypothesis in a specific context. The correct context in Δ can be determined by the constructor used to create the `EnvsIndex` instances as shown in figure 4.12. The additional parameter `rp` (short for *remove intuitionistic*) is considered when deleting a hypothesis from the intuitionistic context and ignored when i references a spatial hypothesis. Calling the function with `rp` = `true` is, for example, useful when the referenced hypothesis is destructed and no longer needed. The value `false` is specified when a hypothesis is used in a proof, since intuitionistic hypotheses can be used more than once. A function that does not modify the environments is `lookup`. This function is used to retrieve the hypothesis referenced by i from Δ. The resulting tuple contains a boolean value that indicates whether the hypothesis comes from the intuitionistic (`true`) or spatial (`false`) context and the retrieved proposition. The function always succeeds since elements of `EnvsIndex.of` Δ must contain a valid index for Δ. The function `replace` combines the functions `delete` and `append`. Note that the inserted proposition is added to the end of the respective context, which is the expected behavior when considering the goal display. Using `replace` instead of `delete` and `append` has the additional advantage of being able to use the soundness proof for the entire operation.

The function `split` on `Envs` is the most complex one and also shown in figure 4.14. It is used to split the spatial context Δ`.spatial` into two parts, such that each hypothesis is in exactly one of the two parts. The hypotheses are distributed using the list `mask` which is required to be of the same length as Δ`.spatial`. The boolean values in `mask` define whether the hypothesis at the same position in Δ`.spatial` is placed in the left part (`true`) or the right part (`false`) of the result. This function is used when performing a proof of two propositions combined with the separating conjunction $*$. Since every proposition in the spatial context can only be used once, the user must decide whether to use it for the proof of the proposition on the left-hand or right-hand side of $*$. Since the propositions in the intuitionistic context can be used multiple times, they are available for the proofs on both sides of $*$. The entire intuitionistic context of Δ is therefore used in both parts of the result. The approach of using a boolean mask to split the hypotheses is different from the one in the Coq formalization of MoSeL. There, a list of string identifiers is given and a boolean value that indicates whether the referenced hypotheses should be placed in the left or right part of the result. The remaining hypotheses are placed in the opposite part. The two approaches are compared in section 5.3.

There are cases in which the spatial context is required to contain only affine propositions, e.g., when a proof is concluded and the remaining hypotheses must be discarded. The intuitionistic context does not need to be checked since intuitionistic hypotheses are always affine. Instead of iterating over the hypotheses in the spatial

```
class AffineEnv [BI PROP] (Γ : Env PROP) where
    affineEnv : ∀ P, P ∈ Γ → Affine P

instance affineEnvNil [BI PROP] :
    AffineEnv (PROP := PROP) .nil
instance affineEnvConcat [BI PROP] (P : PROP) (Γ : Env PROP) :
    [Affine P] → [AffineEnv Γ] → AffineEnv (.cons P Γ)
instance affineEnvBi (Γ : Env PROP) :
    [BIAffine PROP] → AffineEnv Γ
```

Figure 4.15.: Definition and instances of the typeclass `AffineEnv`, which is used to determine whether a given environment Γ contains only affine hypotheses. The argument `PROP` is specified explicitly in the instance `affineEnvNil` because it cannot be inferred automatically for the typeclass `AffineEnv`.

context, this check is implemented using the typeclass `AffineEnv`, shown in figure 4.15. This allows extensions to the separation logic interface to introduce additional modalities or structures with new instances of the typeclass `Affine`. The typeclass `AffineEnv` requires that for each proposition P, it must hold that if P is contained in the environment Γ, there must be an instance of the typeclass `Affine` for P. The more convenient notation $\forall P \in \Gamma$ is not yet defined in Lean. The instances of `AffineEnv` in figure 4.15 show three cases in which an environment contains only affine hypotheses. The first two instances implement an inductive approach. The instance `affineEnvNil` states that the empty environment contains only affine hypotheses, while the instance `affineEnvConcat` performs the induction step by stating that an affine environment Γ remains affine when an affine hypothesis P is added. The last instance `affineEnvBi` is a shortcut for when the separation logic contains only affine propositions, indicated by an instance of the typeclass `BIAffine`.

The last part of the implementation of the separation logic context is the goal display. It is supposed to list the introduced hypotheses in the same way as the local context in Lean is shown (see figure 3.1). The display should contain both the intuitionistic and the spatial context in addition to the already available Lean context, as well as the current goal Q on the right side of the Lean goal $P \vdash Q$. An example of the resulting display is shown in figure 4.16. Notice that the goal display produces only the part after the ⊢ in line 5 where the Lean goal would usually be displayed. This is because the goal display is implemented as a so-called *delaborator* for the function `envs_entails`, which contains the `Envs` object Δ, as well as the separation logic proposition Q in the Lean proposition `of_envs` $\Delta \vdash Q$. This proposition is the (Lean) goal during a separation logic proof. A delaborator maps a meta level expression of the type `Expr`, representing an application of the associated function, to a `Syntax` object that can be shown to the user. The implementation of the goal display first destructs the expressions for ⊢, `Envs` and `Env` and then retrieves the

```
 1  case Ind_1                               case Ind_2
 2  PROP : Type                              PROP : Type
 3  inst : BI PROP                           inst : BI PROP
 4  P Q1 Q2 R : PROP                         P Q1 Q2 R : PROP
 5  ⊢ Iris Proof Mode                        ⊢ Iris Proof Mode
 6  -----------------------------            -----------------------------
 7  HQ1 : Q1                                 HQ : Q1 ∧ Q2
 8  HQ2 : Q2                                 ----------------------------- □
 9  ----------------------------- □          HP1 : □ P
10  HP : P                                   HP2 : □ P
11  HR : <affine> R                          HR : R
12  ----------------------------- ∗          ----------------------------- ∗
13  P -∗ R                                   R
```

Figure 4.16.: Examples of the separation logic goal display showing three contexts per goal (`Ind_1` and `Ind_2` in this example): The Lean context in lines 2-4, the intuitionistic context above the line with the intuitionistic modality □ and the spatial context above the line with the separating conjunction $*$. Notice that hypotheses with an intuitionistic modality, like $\Box P$ in the context of the right goal, can also appear in the spatial context. The goals $P \mathrel{-\!\!*} R$ and R are shown below the last line.

hypotheses of both contexts as expressions. The resulting expressions contain the names of the hypotheses as meta data. An example of an environment as an `Expr` is shown in figure 4.13. The destruction of `Expr` objects is explained in the context of tactics in subsection 4.3.3. The produced syntax is then assembled, including the hypotheses of both contexts with their names, the goal and additional styling elements (lines and symbols) shown in figure 4.16. The syntax for the hypotheses is created by the delaborator for separation logic propositions.

4.3. Tactics

With the ability to instantiate custom separation logics, write statements using the established notation, and visualize the context of proofs steps, the only thing missing from the separation logic interface are the tactics that allow writing proofs on the high abstraction level common in ITPs. Separation logic proofs would also be possible without these tactics by applying the provided axioms and derived laws, but they would be much more complicated and less readable. Instead, using the provided tactics, the separation logic proofs read like other Lean proofs regarding both the abstraction level and the known proof concepts like introduction and destruction of hypotheses. Although the implementation of the tactics is based on MoSeL, their signatures, including names and syntax patterns, follow the Lean conventions. A list of all provided tactics can be found in figure A.1.

4.3.1. Separation Logic Proof Mode

When writing a separation logic theorem, the statement to prove is a logical consequence of the form $P \vdash Q$ where P and Q are separation logic propositions. The entire statement is a Lean proposition as visible from the definition of $\vdash$ in figure 4.1. Since the separation logic tactics and the goal display require a structured context to introduce and access hypotheses, the user must first enter the *separation logic proof mode*. The structured context is then represented as an `Envs` object. The concept of separation logic environments was explained in section 4.2. The two tactics used to enter and leave the separation logic proof mode are `istart` and `istop`. The "`i`" prefix is used to distinguish the separation logic tactics from the Lean tactics. The tactic `istart` enters the proof mode by turning the goal from $P \vdash Q$ into an application of `envs_entails`, which includes an instance of `Envs`. Since the proposition P can be arbitrarily complex, it is not trivial to decide where it should be split into hypotheses in the created context. The `Envs` object is therefore created with two empty `Env` objects, i.e., both the intuitionistic and the spatial context are empty. This is done in two steps: First the typeclass `AsEmpValid` is used to turn the proposition $P \vdash Q$ into $\mathit{emp} \vdash (P \mathrel{-\!\!*} Q)$, justified by the elimination rule for $\mathrel{-\!\!*}$ shown in figure 2.3. Then, the goal is turned into `envs_entails` Δ_0 $(P \mathrel{-\!\!*} Q)$ where Δ_0 is the empty `Envs` object. The user can now reintroduce the proposition P to the context and destruct it if necessary. The tactic `istart` is also called implicitly by other tactics where applicable, such that the user can start a proof with, for example, the tactic `iintro` without having to enter the proof mode manually beforehand. The tactic `istop` works in the opposite direction and is used to remove the application of `envs_entails` from the Lean proposition in the goal. This is done by reducing the definition of `envs_entails` and `of_envs` shown in figure 4.10. The hypotheses in the intuitionistic and spatial context are combined using $\wedge$ and $*$, respectively, and the proposition representing the intuitionistic context is prepended with an $\square$. The resulting proposition is then again of the form $P \vdash Q$ where P is an unstructured separation logic proposition containing all hypotheses from the `Envs` object Δ.

4.3.2. Tactics and Theorems

As explained in section 3.3, custom tactics in Lean are usually implemented using the `elab` macro which generates the tactic syntax and registers the given monadic function to elaborate usages of the tactic. The implementation of the tactic functions follows the example of MoSeL and adds theorems used by the monadic functions to modify the focused Lean goal, i.e., the separation logic context and goal. The additional theorems allow implementing the tactic behavior on the theorem/object level instead of dealing with kernel expressions on the meta level. Another advantage is that the tactic theorems come with correctness proofs for the implemented actions, whereas arbitrary meta operations may create misformed proof terms which would then be rejected by the verification in the kernel. Figure 4.17 shows the implementation of the simplest tactic in this implementation: `iex_falso`.

```
elab "iex_falso" : tactic => do
    evalTactic (← `(tactic|
        refine tac_ex_falso _ ?_
    ))

theorem tac_ex_falso [BI PROP] {Δ : Envs PROP} (Q : PROP) :
    envs_entails Δ `[iprop| False] → envs_entails Δ Q
```

Figure 4.17.: Implementation of the tactic `iex_falso` showing the two parts of tactic implementations: a monadic function and a number of accompanying theorems. The tactic `iex_falso` replaces the separation logic goal Q with $False$, keeping the context Δ unchanged.

In this example, the monadic function does nothing more than use the Lean tactic `refine` to replace the goal with a proposition that implies the goal. The required implication is the tactic theorem that is passed to `refine`. Since the monadic function is executed on the meta level, the usage of `refine` must be wrapped in `evalTactic`, as shown in figure 3.8. The tactic theorem `tac_ex_falso` states that for all environments Δ and separation logic propositions Q, if the goal is of the form `envs_entails` Δ Q (separation logic proof mode is activated), it can be replaced with `envs_entails` Δ $False$, since the latter implies the former.

4.3.3. Meta Operations

Usually, the monadic tactic functions consist of more than just using `refine` to transform the goal. One example of a task on the meta level is parsing additional parts of the tactic syntax. In the example of the tactic `isplit`, shown in figure 4.18, these additional parts include the variables `side` and `hyps`. Both variables are `TSyntax` objects, i.e., typed syntax objects with their syntax kind as type. The syntax kind of `side` is determined by the custom category `splitSide`, which is also shown in figure 4.18, while `hyps` is an array of elements of the syntax kind `ident`, which is predefined in Lean. In order to determine which side the argument `side` specifies (internally encoded as a boolean value), it is matched against the two possible values of `splitSide`: `l` and `r`. The syntax used in the match is written using syntax quotations ("`` `( ) ``"). The optional syntax kind produced by the syntax quotations is prepended using a vertical bar as separator. Since the syntax category `splitSide` can be extended after its declaration, it is necessary to include a last case in the match, throwing an exception for all other values. The argument `hyps` is processed by extracting the string identifier from each syntax element in the array.

Another important task on the meta level is extracting information from the Lean goals, which are available to meta functions as kernel expressions of type `Expr`. Implementations of the separation logic tactics often need to destruct the separation logic proof mode goal `envs_entails` Δ Q. This destruction is performed by the function

```
declare_syntax_cat splitSide
syntax "l" : splitSide
syntax "r" : splitSide

elab "isplit" side:splitSide "[" hyps:ident,* "]" : tactic => do
    let splitRight ← match side with
        | `(splitSide| l) => pure false
        | `(splitSide| r) => pure true
        | _  => throwUnsupportedSyntax
    let names := hyps.getElems.map (·.getId)
    ...
```

Figure 4.18.: Example of parsing additional syntax elements in a monadic tactic function. The tactic `isplit` is used to destruct a separating conjunction $*$ and distribute the hypotheses from the spatial context to the spatial contexts of the created goals. Its syntax contains the `side` on which the listed hypotheses are placed and the identifiers of these hypotheses in square brackets separated by commas.

`extractEnvsEntailsFromGoal` shown in figure 4.19. The function first retrieves all unsolved goals using `getUnsolvedGoals` and extracts the first goal from the returned list. It is common for Lean tactics to use the first goal as the main goal to work on. If there are no goals, an exception is thrown. The variable `goal` then contains a meta variable representing the goal. A meta variable has a type, which is the goal proposition shown in the goal display, and can have a value assigned to it, which would be a proof of the proposition. The function `extractEnvsEntailsFromGoal` works on the type level and receives the type of the meta variable `goal` from the function `getMVarType`. This type is given as a kernel expression of type `Expr`. Since Lean does not support expression quotations by default [1], the expression is destructed manually using `extractEnvsEntails?`. After reducing any pending function calls in the given expression, this function checks that the expression is an application (`Expr.app`) of the function `envs_entails` with two arguments Δ and Q, both also expressions. If so, it further checks that the first argument is an application of the constructor of `Envs` with the two arguments Γ_p and Γ_s. If all checks succeed, the three values Γ_p, Γ_s and Q are returned. The function `extractEnvsEntailsFromGoal` continues by returning the same values. The implementation of the function `findHypothesis` uses `extractEnvsEntailsFromGoal` to find a hypothesis in the intuitionistic or spatial context by name. Since it is a meta function, both the contexts and the contained hypotheses are expressions of type `Expr`. After extracting the intuitionistic and spatial contexts from the goal, `findHypothesis` continues by destructing repeated applications (`Expr.app`) of `Env.cons`, checking if the meta data (`Expr.mdata`) asso-

[1]There is the package `quote4` [17] for expression quotations in Lean 4, but it is currently not used in this implementation.

ciated with the hypotheses in the contexts contains the specified name. If successful, the index of the found hypothesis is returned.

```
def extractEnvsEntailsFromGoal : TacticM (Expr × Expr × Expr) := do
    let goal :: _ ← getUnsolvedGoals
        | throwNoGoalsToBeSolved
    let expr ← getMVarType goal

    let some (Γp, Γs, Q) ← extractEnvsEntails? expr
        | throwError "not in proof mode"
    return (Γp, Γs, Q)
```

Figure 4.19.: Implementation of the function `extractEnvsEntailsFromGoal`. The vertical bar in a line after a `let` contains the value to return if the match in the `let` statement is unsuccessful.

Some tactics need to inspect hypotheses from the Lean context, e.g., the tactic `iassumption_lean` shown in figure 4.20. This is easier to implement than using hypotheses from the separation logic context, since Lean provides the necessary functions as part of its meta API. The Lean hypotheses are contained in the so-called *local context* of each goal. In order to access the local context, `iassumption_lean` first calls `getUnsolvedGoals` to retrieve the main goal. It then uses `getMCtx` to receive the *meta variable context* where information on all meta variables is stored. Finding the *meta variable declaration* of `goal` using `findDecl?`, the declaration `decl` can be used to access the local context (`decl.lctx`). The local context contains an array of *local declarations*, which are exactly the hypotheses in the Lean context of `goal`. These local declarations have, beside other properties, a name (`userName`) and a type (`type`). The tactic `iassumption_lean` uses the type of each hypothesis to validate that it is of the form $\vdash Q$ with an empty context. Note that the type is again given as an expression of type `Expr`. If the hypothesis is of the required form, the goal is closed using the theorem `tac_assumption_lean` where the name

```
elab "iassumption_lean" : tactic => do
    let goal :: _ ← getUnsolvedGoals
        | throwNoGoalsToBeSolved
    let some decl := (← getMCtx).findDecl? goal
        | throwError "ill-formed proof environment"

    for h in decl.lctx do
        let (name, type) := (h.userName, ← instantiateMVars h.type)
        ...
```

Figure 4.20.: Implementation of the tactic `iassumption_lean`. The tactic closes a separation logic goal with a Lean proposition of the type $\vdash Q$.

of the hypothesis is used in the construction of the syntax object for the tactic call as shown in figure 3.8.

4.3.4. Typeclasses

As shown in figure 4.17, tactics in the separation logic interface are implemented as monadic functions together with accompanying theorems. These theorems are used to modify the Lean goal and come with correctness proofs. With the use of typeclasses, the theorems also become extensible for the user of the separation logic interface. Figure 4.21 shows one of the theorems used in the tactic `iintro` as an example of this. The theorem `tac_wand_intro_intuitionistic` is used to introduce (a modified version P' of) the premise P of a goal $P \mathrel{-\!\!*} Q$ (or similar) to the intuitionistic context. For that purpose, it states (in the last line) that a Lean goal of the form `envs_entails` $\Delta\, R$, i.e., a separation logic goal R with a context Δ, can be transformed into the proposition `envs_entails` $\Delta'\, Q$ where $\Delta' = \Delta$`.append` `true` P' is the new context containing the introduced proposition. The modified context is created from the previous context Δ using the function `append`, which adds the proposition P' to one of the two separation logic contexts. The intuitionistic context is specified by providing `true` for the first argument.

The instance implicit arguments are used to destruct and validate properties of the propositions using typeclasses as explained in section 3.1. The typeclass used first is `FromWand`, which extract the premise P and conclusion Q from a wand R and is also shown in figure 4.21. The proposition R from the original goal is given as a normal argument, while the other propositions P and Q are specified to be output parameters (`outParam`), which are determined by the typeclass instance as explained in section 3.1. To keep the check extensible for the user of the separation logic interface, R does not have be constructed using $\mathrel{-\!\!*}$, but it must be derivable from $P \mathrel{-\!\!*} Q$. This allows treating the separation logic goal as $P \mathrel{-\!\!*} Q$, since the original wand R is derivable from it. The predefined instance of `FromWand` handles the default case and specifies $R = P \mathrel{-\!\!*} Q$. The proposition $P \mathrel{-\!\!*} Q$ is then destructed to introduce the premise: P is added to the intuitionistic context as P' and Q replaces the separation logic goal.

The remaining part of `tac_wand_intro_intuitionistic`, shown in figure 4.21, is concerned with validating that the premise P can be treated as intuitionistic, which is required in order to add it to the intuitionistic context. To be able to treat a proposition as intuitionistic, it must be affine and persistent. This is fulfilled by a proposition $\square\, P$, but also for propositions with certain combinations of the `<affine>` and `<pers>` modalities. Note that not all propositions with combinations of the two modalities have both properties, e.g., `<pers>` `<affine>` P is not affine, as explained in subsection 4.1.1. Having P as the premise of the wand, `tac_wand_intro_intuitionistic` therefore first checks that P is affine, using the typeclass `Affine`. P can however also be treated as affine if Q is absorbing, which is checked using the typeclass `Absorbing`. The two required typeclass instances are therefore combined using `TCOr`, which implements a logical disjunction on typeclasses

```
class FromWand [BI PROP] (R : PROP) (P Q : outParam PROP)
where
    from_wand : (P -∗ Q) ⊢ R
class IntoPersistent (p : Bool) [BI PROP] (P : PROP) (P′ : outParam PROP)
where
    into_persistent : <pers>?p P ⊢ <pers> P′

instance intoPersistentPersistently (p : Bool) [BI PROP] (P P′ : PROP) :
    [IntoPersistent true P P′] → IntoPersistent p `[iprop| <pers> P] P′
instance intoPersistentAffinely (p : Bool) [BI PROP] (P P′ : PROP) :
    [IntoPersistent p P P′] → IntoPersistent p `[iprop| <affine> P] P′
instance intoPersistentHere [BI PROP] (P : PROP) :
    IntoPersistent true P P

theorem tac_wand_intro_intuitionistic [BI PROP] {Δ : Envs PROP}
    {P P′ Q R : PROP} : [FromWand R P Q] →
    [IntoPersistent false P P′] → [TCOr (Affine P) (Absorbing Q)] →
    envs_entails (Δ.append true P′) Q → envs_entails Δ R
```

Figure 4.21.: The tactic theorem `tac_wand_intro_intuitionistic` (bottom) together with two of the typeclasses used to validate and destruct the separation logic hypotheses (top). In addition, instances of the typeclass `IntoPersistent` are shown (middle).

as shown in figure 4.7. The remaining check that P is persistent is performed by the typeclass `IntoPersistent`, which is also responsible for removing `<affine>`, `<pers>` and $\Box$ modalities from P. Since P will be added to the intuitionistic context, it is unnecessary to keep those modalities. The definition of `IntoPersistent` and the instances mentioned in this paragraph are shown in figure 4.21. Note that a proposition `<affine>` `<pers>` P is persistent, even though `<affine>` is the outer modality. The used typeclass `IntoPersistent` resembles the typeclass `Persistent` introduced in subsection 4.1.3, but in addition to checking that P is persistent, it generates a new proposition P' without the listed modalities. The boolean parameter p is used to indicate whether it is already established that P is persistent and set to `false` when the typeclass instance search is started in `tac_wand_intro_intuitionistic`. Using p is necessary to continue removing other modalities after the first `<pers>` modality is found. The instance `intoPersistentPersistently` therefore specifies that if the goal is of the form `<pers>`?p `<pers>` P for any boolean value p (right side of the implication), the typeclass instance search can continue with P (without the `<pers>` modality) and p set to `true` (left side of the implication), since it is already established that P is persistent. The result P' of the continued search is used as the result of `intoPersistentPersistently`. One instance that removes a modality without changing p is `intoPersistentAffinely`. Here the only change is that the typeclass instance search is continued with P, having stripped the `<affine>` modality.

The result P' of the continued search is again used as the result. If there are no more modalities to remove, i.e., the argument P of `IntoPersistent` does not match any of the arguments in `intoPersistentPersistently` and `intoPersistentAffinely`, the typeclass `intoPersistentHere` is used to end the typeclass instance search by stating that P is persistent if p was set to `true` anywhere during the typeclass instance search. There are also other instances of `IntoPersistent` that provide support for extensions with new instances of `Persistent` for custom modalities or constructs. Note that the typeclass instance search in this use case requires priorities for the typeclass instances, e.g., the typeclass instance `intoPersistentHere` has a lower priority than `intoPersistentPersistently` and `intoPersistentAffinely`. After the typeclass instance search is finished, `tac_wand_intro_intuitionistic` introduces the result P' to the intuitionistic context.

4.3.5. Composite Tactics

One of the most powerful tactics is `icases`, which destructs a hypothesis in a context using a pattern. Besides others, there are patterns to destruct $\wedge$, $*$ and $\vee$, as well as to move hypotheses between the intuitionistic and the spatial context. A destructed hypothesis is replaced with the resulting hypotheses in its context. Figure 4.22 shows an example usage of the tactic. The first step in the example is the destruction of the existential quantifier $\exists$ in HP, which is done using the anonymous constructor notation ⟨`x, ...`⟩ where `x` is the bound variable. Existentially bound variables are introduced to the pure context. Next, the three-argument separating conjunction $P1 * \cdots *$ `<affine>` `<pers>` $(P3 \mathrel{-\!\!*} P4)$ is destructed using again the anonymous constructor notation: ⟨`HP1, ..., □HP3`⟩. The intuitionistic modality $\square$ indicates that the hypothesis `HP3` should be placed in the intuitionistic context. This is possible since the separation logic proposition is both affine and persistent, which is enough to treat it as intuitionistic as explained in subsection 4.1.1. Note that in the Coq formalization of MoSeL, there are two conjunction patterns, one with two arguments and one with multiple arguments. This is also discussed in section 5.2. The last step is destructing the disjunction $\square\, P2 \vee P2$ inside the conjunction. The notation uses a vertical bar "|" to separate the arguments of the disjunction: `□HP2 | HP2`. Using the same name for both arguments is allowed, since they are placed in different goals (`Ind_1` and `Ind_2`). It is also allowed to move only one of the hypotheses between contexts, here by using $\square$ on the first argument of the disjunction. The tactic `icases` can be used as a standalone tactic, as shown in this example, but it is also automatically executed when introducing hypotheses. The tactic `iintro` therefore allows using `icases` patterns instead of names for the introduced hypotheses.

The implementation of `icases` traverses the parsed pattern syntax recursively and executes various actions for the different pattern constructs. Non-recursive actions like renaming a hypothesis or moving it to the intuitionistic context are directly executed by calling the corresponding tactics `irename` or `iintuitionistic`. For recursive actions like the destruction of a conjunction, additional functions are used to destruct the proposition and return the resulting propositions together with the

Separation logic proposition HP:

$\exists\, x, P1\, x * (\Box\, P2 \vee P2) *$ `<affine>` `<pers>` $(P3 \mathrel{-\!\!*} P4)$

Usage of `icases`:

```
icases HP with ⟨x, ⟨HP1, □HP2 | HP2, □HP3⟩⟩
```

Resulting separation logic contexts:

```
case Ind_1                       case Ind_2
x : Nat                          x : Nat
⊢ Iris Proof Mode                ⊢ Iris Proof Mode
-----------------------------    -----------------------------
HP2 : P2                         HP3 : P3 -∗ P4
HP3 : P3 -∗ P4                   ----------------------------- □
----------------------------- □  HP1 : P1 x
HP1 : P1 x                       HP2 : P2
----------------------------- ∗  ----------------------------- ∗
```

Figure 4.22.: Example usage of the tactic `icases` on a separation logic proposition HP. Before the call to `icases`, the proposition in the example is assumed to be in the spatial context. The two goal displays show the contexts of the two goals created by executing `icases` on HP with the resulting hypotheses. Additional elements in the Lean context, as well as the separation logic goal, are omitted in this example.

remaining parts of the pattern. To support destruction patterns with more than two arguments, a stack-based approach is used to destruct the binary right-associative separation logic connectives $\wedge$, $*$ and $\vee$. The recursive calls to destruct the arguments of destructed propositions are performed on the top level of the implementation to separate the goal manipulations from the destruction of the propositions. This is relevant for the destruction of disjunctions, which generates one goal per argument. In addition, arguments that are processed later in surrounding patterns have to be applied to all of the generated goals. The destruction of a proposition using a specific pattern usually considers various cases, such as different ways of destructing a non-separating conjunction $\wedge$ depending on the pattern arguments. Each destruction variant uses a theorem to perform the necessary changes to the separation logic goal and contexts, similar to the example shown in figure 4.17. The theorems themselves use typeclasses to destruct the propositions as demonstrated in figure 4.21. This allows users to extend the implementation of `icases` for their custom connectives.

4.4. Proofs

The implementation of the separation logic interface contains propositions in many different places and often Lean expects a proof of these propositions. This guarantees that the separation logic and the transformations performed in the interface are sound relative to their definitions, as well as it allows users to validate their separation logic proofs within Lean's kernel. Without these proofs, a mapping from the separation logic proof to a proof in Lean's base logic would not be possible. Proofs are required in different places in this implementation: Building on the axioms in the separation logic interface `BI` (see figure 4.1), additional laws of separation logic propositions are derived, as shown in subsection 4.1.4. Proofs are required to validate that the stated laws are correct for arbitrary separation logics. Another important category contains the propositions made in typeclasses, which are used to state that separation logic propositions or relations fulfill certain properties. The concept of using typeclasses for this was explained in section 3.1. Proving the related propositions guarantees that the separation logic propositions for which a typeclass is instantiated have the stated properties. The most complex propositions in the separation logic interface are those describing the modifications of tactics on a separation logic goal and its context. These propositions are used by the tactic implementations to execute the modifications, as described in subsection 4.3.2. Adding proofs for them justifies that the modifications can be used to create correct and validatable proofs. The last class of propositions are those made about the separation logic environments introduced in section 4.2. Proving these ensures that the embedding in separation logic statements and the defined operations are sound. This section explains proof examples from each class of propositions and some of the basic techniques used in the proofs.

4.4.1. Derived Laws

The first class of propositions builds on the separation logic axioms specified in the interface `BI` for a separation logic. From these axioms, many additional laws can be derived, which are part of this implementation and used in proofs of other parts of the separation logic interface. An example of a derived law considering the behavior of the `<pers>` modality in the context of non-separating conjunctions $\land$ is shown together with its proofs in figure 4.23. The proposition made in the theorem `persistently_and` states that the terms `<pers>` $(P \land Q)$ and `<pers>` $P \land$ `<pers>` Q are equivalent, expressed by the bidirectional entailment $\dashv\vdash$ between them. In the proof of the theorem, the first step is to split this bidirectional entailment into two entailments, since the two directions require different proofs. This is done by applying the field `anti_symm` of the typeclass `AntiSymm`, which states that the relation $\vdash$ is antisymmetrical with the relation $\dashv\vdash$ as its equivalence relation. Applying this proposition generates two goals `left` and `right` for the two directions. The goal `right` for the reverse direction `<pers>` $P \land$ `<pers>` $Q \vdash$ `<pers>` $(P \land Q)$ is directly provable with the separation logic axiom `persistently_and_2`. The opposite direction does not require a separate axiom, but is derivable from other laws. Since the right side of the entailment `<pers>` $(P \land Q) \vdash$ `<pers>` $P \land$ `<pers>` Q is a $\land$, it is destructed by applying the introduction rule `and_intro` to prove the two arguments of $\land$ separately. The left and right side of $\vdash$ in both remaining goals are wrapped in a `<pers>` modality, which allows using the `persistently_mono` rule stating that $(P \vdash Q) \to$ (`<pers>` $P \vdash$ `<pers>` Q). The last step is proving $P \land Q \vdash P$

and $P \land Q \vdash Q$, which is done using the two elimination rules of $\land$, `and_elim_l` and `and_elim_r`, respectively.

```
theorem persistently_and [BI PROP] {P Q : PROP} :
    <pers> (P ∧ Q) ⊣⊢ <pers> P ∧ <pers> Q
:= by
    apply anti_symm
    case left =>
        apply and_intro
        <;> apply persistently_mono
        · exact and_elim_l
        · exact and_elim_r
    case right =>
        exact persistently_and_2
```

Figure 4.23.: Proof of the derived law `persistently_and` stating that `<pers>` can be distributed over $\land$. The notation `<;>` *tac* applies the tactic *tac* to every goal created by the preceding tactic, while the notation · *tacs* applies the tactics *tacs* to the first unsolved goal and closes it.

It is often impractical to completely destruct complex propositions using introduction rules. Instead, it is much more convenient to use so-called *rewriting*. In normal Lean proofs, this is as simple as replacing instances of a proposition a with a proposition b anywhere in a proposition given a proof that $a = b$. This is more complicated in separation logic where statements have the form $P \vdash Q$ or $P \dashv\vdash Q$. The latter is easier to handle here, since it states that P and Q are equivalent, i.e., P can be replaced with Q anywhere in a separation logic statement. Implementing this in Lean is however not straightforward, since Lean only supports rewriting with equalities instead of equivalences. The easiest solution is to define the bidirectional entailment $\dashv\vdash$ as equality and use the rewriting facilities provided by Lean. The user of the separation logic interface is then required to ensure that the equality on separation logic propositions resembles the equivalence relation $\dashv\vdash$. This can be achieved by using setoids that provide an equality for equivalence classes on an embedded type, which would be the actual type of the separation logic propositions. The setoid type would then be used as the carrier type in the `BI` interface. While this allows using Lean's rewriting tactics, it comes with major limitations for the separation logic interface, further discussed in section 5.4.

Rewriting with a separation logic statement of the form $P \vdash Q$ is however entirely unsupported by Lean. This is different from Coq, which is why the following solution differs from the Coq formalization of MoSeL, although it is based on the same concepts. Rewriting with statements $P \vdash Q$ has a different effect compared to rewriting with equivalences or equalities. First, it is unidirectional, i.e., in a given position, either P can be replaced with Q or Q can be replaced with P, but never both. Second, the direction in which the rewriting occurs is dependent on the position of

```
@[rwMonoRule]
theorem absorbingly_mono [BI PROP] {P Q : PROP} :
    (P ⊢ Q) → <absorb> P ⊢ <absorb> Q

@[rwMonoRule]
theorem and_mono [BI PROP] {P P′ Q Q′ : PROP} :
    (P ⊢ Q) → (P′ ⊢ Q′) → P ∧ P′ ⊢ Q ∧ Q′
```

Figure 4.24.: Examples of two rules used for rewriting in separation logic statements. The notation `@[`*attr*`]` associates the attribute *attr* with the following definition or theorem.

the replacement in a statement. The rules stating how rewriting is allowed for certain separation logic connectives are one of the key points of rewriting with entailments. The base rule describes rewriting around the entailment $\vdash$ itself: Rewriting with $P \vdash Q$ on the left side of $\vdash$ allows replacing P with Q (forward direction) while rewriting on the right side of $\vdash$ allows replacing Q with P (reverse direction). This is a direct consequence of the transitivity of $\vdash$ which states that if $P \vdash Q$ holds then it is sufficient to prove $Q \vdash R$ to receive a proof of $P \vdash R$. In the example of rewriting on the left side of $\vdash$, this means that given $P \vdash Q$, the goal $P \vdash R$ can be replaced with $Q \vdash R$, which is the same as replacing P with Q on the left side of $\vdash$. Examples of further rewriting rules for separation logic connectives are shown in figure 4.24. The theorem `absorbingly_mono` states that when rewriting with $P \vdash Q$, it is also possible to rewrite with `<absorb>` $P \vdash$ `<absorb>` Q. Combined with the rewriting rule for $\vdash$, this allows replacing P with Q when rewriting with $P \vdash Q$ on the left side of `<absorb>` $P \vdash R$. This is done by first using `absorbingly_mono` to derive `<absorb>` $P \vdash$ `<absorb>` Q from $P \vdash Q$ and then using the transitivity of $\vdash$ to replace `<absorb>` P with `<absorb>` Q and receive `<absorb>` $Q \vdash R$. The theorem `and_mono` states the same for a binary separation logic connective, i.e., given $P \vdash Q$ it is possible to replace P with Q on the left or the right side of $\wedge$. For constructing proofs where one side of a binary connective should remain unchanged, the reflexivity of $\vdash$ can be used, i.e., $P \vdash P$ for all separation logic propositions P. In the example of `and_mono`, this allows using the rewriting rule as $(P \vdash Q) \to P \wedge Q' \vdash Q \wedge Q'$ as well. Both the reflexivity and transitivity of $\vdash$ are part of the `PreOrder` typeclass of which an instance for $\vdash$ is required in the interface `BI` for separation logics. There are more complex rewriting rules for other separation logic connectives, e.g., the rewriting direction in a $-\!\!*$ changes for the premise, but stays the same for the conclusion.

Applying the rewriting rules by hand allows performing the rewrite, but it is still impractical and clutters proofs with the destruction of separation logic connectives. Using the meta programming capabilities of Lean, this implementation therefore includes the tactic `rw′` which automates applying the rewriting rules and enables Lean-style rewriting with entailments, i.e., reflexive and transitive relations with rewriting rules. The tactic first applies the transitivity rule to enable rewriting on the specified side of the entailment, e.g., ⊢. It then recursively tries to apply the available rewriting rules to find a position in which the given rewrite term can be used. The tactic assumes that there is at most one rewriting rule per connective and does therefore not perform backtracking. All generated goals that do not match the rewrite term are closed using the reflexivity of the entailment. Transitivity and reflexivity of the entailment relation are required using the typeclasses `Transitive` and `Reflexive`, while the rewriting rules can be added using the attribute `rwMonoRule`, as shown in figure 4.24.

```
theorem absorbingly_wand [BI PROP] {P Q : PROP} :
    <absorb> (P -∗ Q) ⊢ <absorb> P -∗ <absorb> Q
:= by
   apply wand_intro_l        -- <absorb> P ∗ <absorb> (P -∗ Q) ⊢ <absorb> Q
   rw [← absorbingly_sep]    -- <absorb> (P ∗ (P -∗ Q)) ⊢ <absorb> Q
   rw' [wand_elim_r]         -- <absorb> Q ⊢ <absorb> Q
```

Figure 4.25.: Example of a proof using the tactic `rw'`. The right side shows the goal after the tactic in the same line on the left side has been executed.

An example of a proof, in which the new rewriting tactic `rw'` for entailments is used, is shown in figure 4.25. After the theorem `wand_intro_l` is applied to move `<absorb>` P from the right side of the entailment to the left, the Lean tactic `rw` is used with `absorbingly_sep` to move the `<absorb>` modality out of the separating conjunction $*$. The theorem `absorbingly_sep` is a bidirectional entailment, i.e., an equality, which is why `rw` can perform the rewrite. The theorem `wand_elim_r` on the other hand is a unidirectional entailment ⊢ requiring `rw'`. The syntax is the same as for the Lean tactic, but in this example it allows rewriting with the proposition $P * (P \mathrel{-\!\!*} Q) \vdash Q$ inside of the `<absorb>` modality. The remaining goal `<absorb>` $Q \vdash$ `<absorb>` Q is automatically solved by `rw'` using the reflexivity of ⊢.

4.4.2. Typeclass Instances

Proofs are also part of many typeclass instances used to specify properties of separation logic propositions and relations. The proofs are typically short and use derived laws in order to make the results reusable for other proofs. Two proofs that are a part of typeclass instances are shown as an example in figure 4.26.

The first proof in figure 4.26 is part of the instance `andAffineL`. The field `affine` of the typeclass `Affine` states that if P is affine, then $P \wedge Q$ is affine for all separation logic propositions P and Q. The proposition to prove is $P \wedge Q \vdash emp$ and the tactic `rw'` is used to rewrite with two entailments. The first rewrite term is again the field `affine`, this time as part of the instance `Affine P`, which replaces P with emp. The typeclass instance `Affine P` is given as an instance implicit argument of `andAffineL` and guarantees that the known proposition P is affine. The remaining statement to prove is $emp \wedge Q \vdash emp$, which can directly be solved using the left elimination rule of $\wedge$.

```
instance andAffineL [BI PROP] (P Q : PROP) :
    [Affine P] →
    Affine `[iprop| P ∧ Q]
where
    affine := by
        rw' [affine, and_elim_l]

instance intoPersistentIntuitionistically [BI PROP]
    (p : Bool) (P Q : PROP)
    [instP : IntoPersistent true P Q] :
    IntoPersistent p `[iprop| □ P] Q
where
    into_persistent := by
        rw' [← instP.into_persistent]
        cases p
        case false =>
            exact intuitionistically_into_persistently_1
        case true =>
            apply persistently_if_mono
            exact intuitionistically_elim
```

Figure 4.26.: Examples of instances that state properties of separation logic propositions. Both instances contain the proof that the properties hold for the specified propositions.

The proof of the second instance `intoPersistentlyIntuitionistically` in figure 4.26 requires a case distinction to prove the proposition `into_persistent`. The statement to show is, if it is already established that P is persistent and removing its modalities results in Q, then adding an $\square$ in front of P results in a valid instance of `IntoPersistent` without requiring again that P is persistent (hence p is not required to be `true` in the resulting instance). For a definition of the typeclass `IntoPersistent` see figure 4.21. The proof again starts by rewriting with the proposition given by the available typeclass instance, which is `<pers>`?`true` $P \vdash$ `<pers>` Q. The notation $\leftarrow$ is used to rewrite with the entailment in the reverse direction, in this case on the right side of the entailment in the goal. After that, the statement left to prove is `<pers>`?$p \, \square \, P \vdash$ `<pers>`?`true` P, which either requires that $\square \, P$ is persistent ($p =$ `false`) or that $\square$ can be dropped from $\square \, P$ ($p =$ `true`). The two cases of p are distinguished using the tactic `cases`. The first case `false` is directly solved using the derived law `intuitionistically_into_persistently_1`. The second case has a conditional `<pers>` modality with the same value of p on both sides. The theorem `persistently_if_mono` can therefore be applied, which states that $P \vdash Q$ is sufficient to prove `<pers>`?$p \, P \vdash$ `<pers>`?$p \, Q$ for all p. The remaining statement is then $\square \, P \vdash P$, which is proven using `intuitionistically_elim`.

4.4.3. Tactic Theorems

The theorems used by the separation logic tactics to modify the separation logic goals and their contexts are the most complex ones since they combine requirements of properties of propositions and changes to the separation logic environments and goals. These theorems commonly use the typeclasses explained in subsection 4.1.3 and subsection 4.3.4, as well as the environment operations explained in section 4.2. Their proofs typically rely on the separation logic axioms, the derived laws and the soundness proofs of the environment operations.

Figure 4.27 shows an example of a theorem used in the tactic `iintro` together with its proof. The theorem `tac_wand_intro_intuitionistic` was already shown in figure 4.21 with the definition of the two typeclasses `FromWand` and `IntoPersistent`, however without the proof. The described modification moves the premise P of a wand R to the intuitionistic context as P' with the modalities related to the $\Box$ modality stripped. The necessary proof ensures that if the user provides a proof for Q with the modified context containing P', it implies the original proposition with the separation logic goal R and P' not in the context. After reducing the definition of `envs_entails` (see figure 4.10) using `simp only`, the proof of `of_envs` (`Envs.append` `true` $P'\,\Delta) \vdash Q$ is therefore introduced as the hypothesis `h_entails` and the remaining goal after all introductions in line 7 is `of_envs` $\Delta \vdash R$. The typeclass instance `FromWand` $R\;P\;Q$ guarantees that R can be replaced with the wand $P \mathrel{-\!\!*} Q$ on the right side of $\vdash$, which is done by rewriting with `from_wand` in line 8. The second rewrite term in this line is the soundness proof of the environment operation `append`, which states that a proposition P' can be appended to an environment Δ in the conclusion of a wand with P' as its premise, i.e., `of_envs` $\Delta \vdash \Box?p\,P' \mathrel{-\!\!*}$ `of_envs` (`Envs.append` $p\,P'\,\Delta$). If P' should be added to the intuitionistic context ($p =$ `true`), it is required to have the intuitionistic modality $\Box$. Since the right side of $\vdash$ in the goal is $P \mathrel{-\!\!*} Q$ after using `from_wand`, the introduction rule `wand_intro_l` can be used to move P from the premise of the wand to the left side of $\vdash$. The goal after applying the introduction rule in line 9 is then $P * (\Box?$`true` $P' \mathrel{-\!\!*}$ `of_envs` (`Envs.append` `true` $P'\,\Delta)) \vdash Q$. This almost allows using a wand elimination rule, except that P does not yet have the intuitionistic modality and may carry additional modalities that are already removed from P'. The intuitionistic modality can be added in two different ways, depending on whether the instance `inst_affine_absorbing` of `TCOr` contains an instance of the typeclasses `Affine` or `Absorbing`. A case distinction on `inst_affine_absorbing` is performed in line 10 to prove the two cases `l` and `r` separately.

In the case `l`, with an instance of `Affine` available, the theorem `affine_affinely` can be used in line 13 to replace P with `<affine>` P. Rewriting with the derived law `persistently_if_intro_false` to add a `<pers>`?`false` in front of P is only required to use `into_persistent` in line 15 which replaces `<pers>`?`false` P with `<pers>` P' as guaranteed by the instance of `IntoPersistent`. The proposition `<affine><pers>` P' is then equivalent to the premise $\Box?$`true` P' of the wand and the elimination rule `wand_elim_r` can be used in line 16 to receive `of_envs` (`Envs.append` `true` $P'\,\Delta) \vdash Q$, which is exactly the new goal available as `h_entails`.

```
theorem tac_wand_intro_intuitionistic [BI PROP] {Δ : Envs PROP}
    {P P′ Q : PROP} (R : PROP) : [FromWand R P Q] →
    [IntoPersistent false P P′] → [TCOr (Affine P) (Absorbing Q)] →
    envs_entails (Δ.append true P′) Q → envs_entails Δ R
:= by
    simp only [envs_entails]
    intro _ _ inst_affine_absorbing h_entails
    rw′ [← from_wand, envs_append_sound true P′]
    apply wand_intro_l ?_
    cases inst_affine_absorbing
    case l =>
        rw′ [
            ← affine_affinely P,
            persistently_if_intro_false P,
            into_persistent,
            wand_elim_r,
            h_entails]
    case r =>
        rw′ [
            persistently_if_intro_false P,
            into_persistent,
            ← absorbingly_intuitionistically_into_persistently,
            absorbingly_sep_l,
            wand_elim_r,
            h_entails,
            absorbing]
```

Figure 4.27.: Proof of the tactic theorem `tac_wand_intro_intuitionistic` used in the tactic `iintro`. Note that the tactic `rw′` redirects equality rewrites to `rw`.

The proof of the case `r` is similar, although instead of an instance of `Affine` P, an instance of `Absorbing` Q is available, which requires a slightly different approach. First, `into_persistent` is used again in line 21 to replace P with `<pers>`P'. Then, differing from the proof of case `l`, `absorbingly_intuitionistically_into_persistently` is used in line 22 to replace the `<pers>` modality with `<absorb>` $\Box$. This law does not require an instance of `Absorbing` yet. After that, the `<absorb>` modality of `<absorb>` $\Box\, P'$ is extended to the entire left side of $\vdash$ using `absorbingly_sep_l` in line 23. The remaining goal is then `<absorb>` $(\Box\, P' * (\Box ?$`true` $P' \mathrel{-\!\!*} \ldots)) \vdash Q$. Using the same steps as in case `l` allows resolving everything under the `<absorb>` modality to Q, leaving `<absorb>` $Q \vdash Q$ as the goal after line 25. This is solved exactly by the field `absorbing` of the typeclass `Absorbing`.

4.4.4. Environments

The correctness proofs of the modifications to the separation logic proof mode environments, performed in the tactics, rely on the soundness proofs of the environment operations. The tactic theorems use the operations defined on the combined environment type `Envs`, which are implemented using operations on the `Env` objects representing the intuitionistic and spatial context. The definition of the environment types and their embedding in separation logic propositions was presented in figure 4.10. Since this definition is different from the one in the Coq formalization of MoSeL, its soundness proofs are also unrelated. The operations on `Env` come with their own soundness proofs, which are used as parts of the soundness proofs of operations on `Envs`. Since the `Env` objects do not have their own embeddings in separation logic propositions, their soundness proofs specify the behavior of the operations in the context of big operators.

As an example, the soundness proof of the operation `append` on the environments type `Envs` is shown in figure 4.28. This proof was used in the proof of the tactic theorem `tac_wand_intro_intuitionistic` in figure 4.27. The theorem states that it is possible to add a proposition Q to the environment Δ using `append` if Q is given as the premise of a $\mathrel{-\!\!*}$. If Q is added to the intuitionistic context ($p =$ `true`), it must have the intuitionistic modality $\Box$. The proof of the theorem starts by moving the premise $\Box ?p\, Q$ to the left side of $\vdash$ using `wand_intro_l`. A case distinction is then necessary to treat the two cases of adding Q to the intuitionistic or spatial context. In both cases, targeted using the notation `<;>`, the conditional modality is resolved and the definition of `of_envs` is reduced. The resulting goals contain the embedding of `Envs` as a separation logic proposition using big operators, as shown in figure 4.10. The goal in the first case is then $Q*(\Box[\wedge]\Gamma_p)*([*]\Gamma_s) \vdash (\Box[\wedge]\Gamma_p)*([*]($`Env.append`$\Gamma_s Q))$ where Γ_p is the intuitionistic and Γ_s the spatial context. The other case is similar, but has a $\Box Q$ instead of Q on the left side and Q is appended to Γ_p instead of Γ_s on the right side. The key theorem applied in the case $p =$ `false` is `env_big_op_sep_append` which states that a proposition Q that is appended to an environment in $[*]$ can instead be combined using $*$ on the right side of $[*]$. The definition of `env_big_op_sep_append` together with its proof is shown in figure 4.29. After moving Q outside of $[*]$, all parts

```
theorem envs_append_sound [BI PROP] {Δ : Envs PROP}
    (p : Bool) (Q : PROP) :
    of_envs Δ ⊢ □?p Q -∗ of_envs (Δ.append p Q)
:= by
    apply wand_intro_l ?_
    cases p
    <;> simp only [bi_intuitionistically_if, of_envs]
    case false =>
        rw' [
            env_big_op_sep_append,
            (assoc : _ ∗ (_ ∗ Q) ⊣⊢ _),
            (comm : _ ∗ Q ⊣⊢ _)]
    case true =>
        rw' [
            env_big_op_and_append,
            (comm : _ ∗ □ Q ⊣⊢ _),
            ← (assoc : _ ⊣⊢ (□ Q ∗ _) ∗ _)]
```

Figure 4.28.: Soundness proof of the operation `append` on the environments type `Envs`. The patterns behind the commutativity and associativity rules `comm` and `assoc` instruct Lean on where to use them in the proposition. The notation `<;>` *tac* applies the tactic *tac* to every goal created by the preceding tactic.

(Q and the embedding of the two contexts) are equal on both sides and combined using separating conjunctions $*$. The only thing left is to use associativity and commutativity to bring the propositions in the correct structure. The proof of the case $p =$ `true` is similar, but uses `env_big_op_and_append` instead, which moves a proposition Q outside of $\square\,[\wedge]$ as $\square\, Q$ and combines them with $*$. It is then again enough to use associativity and commutativity rules.

The theorem `env_big_op_sep_append` shown in figure 4.29 is an example of a soundness proof of an operation on `Env`. It is used in the soundness proof `envs_append_sound` shown in figure 4.28 and states that a proposition Q can be moved out of a call to `append` inside the big operator $[*]$ and instead combined with a single $*$. The proposition of the theorem is an equivalence, which means the reverse direction is possible as well. For this proof, it is not enough to unfold the definition of the big operator $[*]$. The constructor `cons` of the inductive datatype `Env` prepends the added proposition, as visible in its definition in figure 4.10, which is why $[*]$ performs a right fold of $*$ over an environment Γ. This means that a proposition on the left side of Γ can be separated from $[*]$. The operation `append` on the other hand adds a proposition to the right side of Γ such that the user finds introduced propositions on the end of the hypothesis list. It is therefore required to perform an induction over Γ to proof the soundness of `append`. After reducing the definition of `Env.append` in both goals, the goal in the base case is

```
theorem env_big_op_sep_append [BI PROP] {Γ : Env PROP} {Q : PROP} :
    [∗] (Γ.append Q) ⊣⊢ [∗] Γ ∗ Q
:= by
    induction Γ
    <;> simp only [Env.append]
    case nil =>
        simp only [big_op]
        rw′ [(left_id : emp ∗ _ ⊣⊢ _)]
    case cons Q′ Γ′ h_ind =>
        rw′ [
            !big_op_sep_cons,
            h_ind,
            ← (assoc : _ ⊣⊢ (Q′ ∗ _) ∗ _)]
```

Figure 4.29.: Soundness proof of the operation append on the environment type `Env`. The notation "`!`" in front of a rewrite term in `rw′` performs the rewrite as often as possible.

$[*]$ (`cons` Q `nil`) $\dashv\vdash$ $[*]$ `nil` $* Q$ where `cons` and `nil` are the two constructors of `Env`. Reducing the definition of $[*]$ implemented as `big_op` on the left side of $\dashv\vdash$ yields Q while on the right side *emp* remains as the result of $[*]$ `nil`. The statement left to prove is $Q \dashv\vdash emp * Q$ which is true since *emp* is the left unit of $*$. The induction step is then allowed to use the induction hypothesis $[*]$ (Γ'`.append`Q) $\dashv\vdash$ $[*]\,\Gamma' * Q$ to prove that $[*]$ (`cons` P (Γ'`.append` Q)) $\dashv\vdash$ $[*]$ (`cons` $P\,\Gamma'$) $* Q$. Applying `big_op_sep_cons` on both sides of $\dashv\vdash$ moves P out of the big operator and allows rewriting with the induction hypothesis on the left side of $\dashv\vdash$. In addition to reducing the definition of `big_op`, the theorem `big_op_sep_cons` handles the case of an empty remaining context Γ'. The remaining goal is then $P * ([*]\,\Gamma' * Q) \dashv\vdash (P * [*]\,\Gamma') * Q$ which is solved using the associativity of $[*]$. This concludes that a proposition Q that is appended on the right side of Γ can be moved out of $[*]$ to the right.

Another important theorem is `env_big_op_sep_delete_get`, which states that it is possible to move out any hypothesis from a context Γ inside a $[*]$ using the functions `delete` and `get` on `Env`. Both functions have to be used with the same index i in Γ. The proof of the theorem, shown in figure 4.30, is again an induction, this time over the environment Γ and the index i together. This is made possible by providing the custom eliminator `env_idx_rec`, which handles contradicting cases (valid index with empty environment) and transforms the induction hypothesis. The goal in the base case is $[*]$ (`cons` $P'\,\Gamma'$) $\dashv\vdash$ $[*]$ ((`cons` $P'\,\Gamma'$)`.delete` i_0) $*$ ((`cons` $P'\,\Gamma'$)`.get` i_0) where i_0 is the index with value 0 in Γ. Reducing the definition of `Env.delete` removes P' from `cons` $P'\,\Gamma'$, while doing the same with `Env.get` selects P'. The remaining goal is then $[*]$ (`cons` $P'\,\Gamma'$) $\dashv\vdash$ $[*]\,\Gamma' * P'$ which is solved by moving P' out of `cons` using `big_op_sep_cons` again and reordering the terms with `comm`. The goal in the induction step is the same as in the base case, except instead of i_0 the

```
theorem env_big_op_sep_delete_get [BI PROP]
    {Γ : Env PROP} (i : Fin Γ.length) :
    [∗] Γ ⊣⊢ [∗] (Γ.delete i) ∗ (Γ.get i)
:= by
    induction Γ, i using env_idx_rec
    case zero P′ Γ′ _ =>
        rw′ [
            Env.delete,
            Env.get,
            big_op_sep_cons,
            (comm : P′ ∗ _ ⊣⊢ _)]
    case succ P′ Γ′ i′ _ _ h_ind =>
        rw′ [
            env_delete_cons,
            env_get_cons,
            !big_op_sep_cons,
            ← (assoc : _ ⊣⊢ (P′ ∗ _) ∗ _),
            ← h_ind]
```

Figure 4.30.: Proof that is is possible to extract a single hypothesis from the context Γ inside a $[*]$ using `delete` and `get`. The required index i is given as a `Fin` object with the length of Γ as the upper bound.

index is i'_1 with the value i'`.val` $+ 1$ for some index i'. The induction hypothesis `h_ind` states that $[*]\,\Gamma' \dashv\vdash [*]\,(\Gamma'$`.delete` $i') * \Gamma'$`.get` i' for all i'. Since the value of the index i'_1 is not 0, both `Env.delete` and `Env.get` skip P' in `cons` $P'\,\Gamma'$ when reduced. The theorems `env_delete_cons` and `env_get_cons` in addition handle the necessary transformation of the `Fin` index. The goal after reducing both functions is then $[*]$ (`cons` $P'\,\Gamma'$) $\dashv\vdash [*]$ (`cons` P' $(\Gamma'$`.delete` $i')) * (\Gamma'$`.get` $i')$ with the index i'_1 eliminated. Moving P' out of `cons` and applying the associativity rule `assoc` for $*$ leaves the goal $P' * [*]\,\Gamma' \dashv\vdash P' * [*]\,(\Gamma'$`.delete` $i') * (\Gamma'$`.get` $i')$ which is solved by rewriting with the induction hypothesis `h_ind`. This proves that any hypothesis in a context Γ can be moved out of $[*]\,\Gamma$ and appended using a separating conjunction $*$, independent of its position in Γ.

5. Evaluation

This chapter evaluates the usage of this implementation of a separation logic interface in the Lean theorem prover and compares it to the Coq implementation of MoSeL [5], which was the basis for this implementation. The evaluation includes instantiating the separation logic interface for *classical separation logic* in section 5.1, including proofs of the axioms and a custom notation, as well as an example of a setoid instance for separation logic propositions. It further shows an example of a proof in a separation logic in section 5.2 using the provided tactics and compares it to an equivalent proof in the Coq implementation, as well as to a proof of a similar statement in the pure logic of Lean. One difference between the two implementations is the definition of the environment, which is compared in section 5.3. Lastly, this chapter discusses a limitation of this separation logic interface regarding the rewriting of equivalences using Lean's `Setoid` class in section 5.4. This is relevant for an instantiation of this interface with the Iris separation logic. There are additional, smaller differences between the two implementations, originating in the availability of different features in the two ITPs, which are not included in the evaluation.

5.1. Logic Instance

The starting point for using the provided separation logic interface is to instantiate the typeclasses `BIBase` and `BI` presented in section 4.1. The typeclass `BIBase` contains the separation logic connectives and was shown in figure 4.1. The typeclass `BI` extends `BIBase` with the necessary axioms for a separation logic. An instantiation for a custom separation logic must provide instances of both typeclasses to use the full separation logic interface including the tactics.

As an example, the typeclass `BIBase` is instantiated for *classical separation logic* [5, 1] in figure 5.1. The resources in classical separation logic are cells in a heap, which is expressed using the function `State`. This function describes a heap by mapping indices of the type `Nat` to values of the type `Option Val`. These values will be available if the resource for the cell is owned. The value type `Val` can be chosen individually for each classical separation logic statement. The type of classical separation logic propositions is called `HeapProp` and is defined to be a proposition `Prop` on a part of the heap, represented by an object of `State Val`, which consists of the owned resources. The type `Val` is again a parameter for the definition of `HeapProp`. The instance shown in figure 5.1 then defines the required separation logic connectives in terms of `HeapProp`. The provided connectives are implemented as propositions on a given state σ. For example, the separation logic proposition

```
abbrev State (Val : Type) := Nat → Option Val
abbrev HeapProp (Val : Type) := State Val → Prop

instance (Val : Type) : BIBase (HeapProp Val) where
    entails P Q := ∀ σ, P σ → Q σ
    emp := fun σ => σ = ∅
    pure φ := fun _ => φ
    and P Q := fun σ => P σ ∧ Q σ
    or P Q := fun σ => P σ ∨ Q σ
    impl P Q := fun σ => P σ → Q σ
    «forall» Ψ := fun σ => ∀ a, Ψ a σ
    exist Ψ := fun σ => ∃ a, Ψ a σ
    sep P Q := fun σ =>
        ∃ σ₁ σ₂ , σ = σ₁ ∪ σ₂ ∧ σ₁ || σ₂ ∧ P σ₁ ∧ Q σ₂
    wand P Q := fun σ => ∀ σ', σ || σ' → P σ' → Q (σ ∪ σ')
    persistently P := fun _ => P ∅
```

Figure 5.1.: Part of the instantiation of the separation logic interface for classical separation logic. The notation $\sigma_x \cup \sigma_y$ represents the union of the two states σ_x and σ_y, while the notation $\sigma_x \mathbin{||} \sigma_y$ is a proposition of disjointness for σ_x and σ_y.

emp asserts that the state is empty, i.e., that no resources are owned. In contrast, the functions `pure` and `persistently` ignore the given state, which makes them persistent as described in subsection 4.1.1. The difference between the additive non-separating conjunction `and` and the multiplicative separating conjunction `sep` is visible in the respective definitions. In the definition of `and`, both P and Q are allowed to use the entire known state σ, while for `sep`, the state must be split in two disjoint states σ_1 and σ_2, which are made available to P and Q, respectively. The disjoint union on states is expressed by stating that σ is the union of σ_1 and σ_2 using $\cup$ and requiring σ_1 and σ_2 to be disjoint with the operator $||$. The entailment on classical separation logic propositions is a Lean proposition and states that the consequence Q must hold on all states on which the premise or context P holds.

As mentioned in subsection 4.1.2, the user can define custom notations for the instantiated logic that can be used together with the predefined notation. This is done in the same way as in the implementation of the interface by providing additional syntax (if necessary) using Lean's `syntax` command. The interpretation of the syntax as separation logic proposition is then defined using the command `macro_rules`. Terms that describe separation logic propositions have to be wrapped in an `` `[iprop| ] `` quotation on both sides of the syntax transformation. An example of a custom notation for classical separation logic is shown in figure 5.2. Here, the operator $\mapsto$ is defined to allow propositions on single heap cells. The term $l \mapsto v$ states that the cell with the index l contains the value v, as visible in the definition of `maps_to`. Note that the terms l and v in the `macro_rules` command do not have

```
def maps_to (l : Nat) (v : Val) : HeapProp Val :=
    fun state => state l = some v

syntax:52 term:53 " ↦ " term:53 : term
macro_rules
| `(`[iprop| $l ↦ $v]) => `(maps_to $l $v)
```

Figure 5.2.: Example of a custom notation for the instantiated separation logic.

`` `[iprop| ] `` quotations on the right side of the syntax transformation, since they are not separation logic propositions, but of the types `Nat` and `Val`, respectively.

As explained, the user of the separation logic interface must provide an instance of the typeclass `BI` with the proofs of the separation logic axioms. While the instance of the typeclass for classical separation logic is too long to show it completely, the three examples of separation logic axioms already mentioned in section 4.1 are shown in figure 5.3 with their proofs.

The first axiom `pure_intro` in figure 5.3 states that $\phi \to P \vdash \ulcorner\phi\urcorner$ where $\to$ is the implication on Lean propositions. The proof starts by reducing the definition of the separation logic connectives `entails` and `pure` given in the instantiation of `BIBase`. Since the definition of `entails` specifies that the implication between the propositions must hold for all states σ and `pure` ignores the state, the resulting goal after reducing is $\phi \to \forall\, \sigma, P\,\sigma \to \phi$. Introducing ϕ as `h` and ignoring σ and $P\,\sigma$, the goal is solved by providing the proof `h` of ϕ. Although this seems trivial in the logic of Lean, the proof shows that it is possible to ignore the context of an entailment for a pure goal in classical separation logic.

As a second example, the proof of the $\wedge$-introduction rule `and_intro` is shown in figure 5.3. The proposition to prove is $(P \vdash Q) \to (P \vdash R) \to P \vdash Q \wedge R$, which is again reduced to the underlying definition using `simp only`. The resulting statement is first destructed by introducing the implication premises `h_PQ` and `h_PR`. The first hypothesis `h_PQ` has the type $\forall\, \sigma, P\,\sigma \to Q\,\sigma$ and the type of `h_PR` is the same with R instead of Q. The remaining statement is also universally quantified over σ, which originates in the definition of $\vdash$ in classical separation logic, as shown in figure 5.1. Introducing σ leaves $P\,\sigma \to Q\,\sigma \wedge R\,\sigma$ from which $P\,\sigma$ is introduced as `h_P`. The resulting conjunction is destructed using the tactic `constructor` and the two sides of $\wedge$ are proven separately. The proof of the left side, provided with `exact`, is an instantiation of `h_PQ` for the state σ with `h_P` as the proof of $P\,\sigma$. The proof of the right side is the same with `h_PR` instead of `h_PQ`.

The last proof shown in figure 5.3 ensures that *emp* is the left unit of $*$ on the left side of $\vdash$. The corresponding axiom is called `emp_sep_2` and requires that $emp * P \vdash P$. Reducing all relevant definitions leaves the existential from the definition of `sep`, shown in figure 5.1 on the left side of the implication caused by the definition of $\vdash$ in classical separation logic. The existential is destructed using the anonymous constructor notation $\langle\,\rangle$, which introduces the two states σ_1 and σ_2,

```
instance : BI (HeapProp Val) where
    equiv_entails :=
        ...
    pure_intro := by
        simp only [BIBase.entails, BIBase.pure]
        intro _ _ h _ _
        exact h
    and_intro := by
        simp only [BIBase.entails, BIBase.and]
        intro _ _ _ h_PQ h_PR σ h_P
        constructor
        · exact h_PQ σ h_P
        · exact h_PR σ h_P
    emp_sep_2 := by
        simp only [BIBase.entails, BIBase.sep, BIBase.emp]
        intro _ _ ⟨σ₁, σ₂, h_union, h_disjoint, h_empty, h_P⟩
        rw [h_empty] at h_union
        rw [h_union]
        rw [← empty_union]
        exact h_P
    ...
```

Figure 5.3.: Selection of proofs of the separation logic axioms in the instantiation of the typeclass `BI` for classical separating logic with propositions of type `HeapProp`. The notation · *tacs* applies the tactics *tacs* to the first unsolved goal and closes it. Additional underscores after the tactic `intro` are often used to ignore implicit variables.

as well as the proofs `h_union` and `h_disjoint`, stating together that the disjoint union of σ_1 and σ_2 is equal to the original state σ. This is part of the behavior of $*$ where each resource must either be used in the left argument or the right argument. The proofs of the two arguments of $*$ are introduced as `h_emp` and `h_P` where `h_emp` states that σ_1 must be the empty state $\emptyset$, following the definition of `emp` in classical separation logic. Rewriting with `h_emp` turns the union proof `h_union` into $\sigma = \emptyset \cup \sigma_2$ which implies $\sigma = \sigma_2$, proven by the theorem `empty_union` for arbitrary states. This fact is used to change the remaining goal from $P\,\sigma$ to $P\,\sigma_2$, which is exactly the proof of the second argument of $*$, the hypothesis `h_P`.

One important proof in the instance of the typeclass `BI` is `equiv_entails`, which connects the entailment with the equality on separation logic propositions. It would be favorable to use an equivalence relation instead, but as explained in subsection 4.4.1, in order to use Lean's rewriting tactic, equality is required. If it is not possible to use the equality on separation logic propositions, one solution is to substitute the separation logic type with a quotient type instead, replacing the equality on separation logic propositions with an equality on their equivalence classes. The

```
instance heapPropSetoid (Val : Type) : Setoid (HeapProp Val) where
    r P Q := ∀ σ, P σ ↔ Q σ
    iseqv := { refl := ..., symm := ..., trans := ... }

instance (Val : Type) : BIBase (Quotient (heapPropSetoid Val))
```

Figure 5.4.: Setoid instance for classical separation logic propositions and the usage of the setoid type in the declaration of the `BIBase` instance.

associated equivalence relation is specified as part of the `Setoid` instance for the separation logic proposition type. The limitations of this approach are discussed in section 5.4. Figure 5.4 shows the instantiation of the `Setoid` typeclass for classical separation logic propositions. In addition to the equivalence relation `r`, the typeclass `Setoid` requires a proof `iseqv` that the specified relation is in fact an equivalence relation, i.e., that it is reflexive, symmetrical and transitive. For classical separation logic, the natural equivalence relation is the bidirectional implication on propositions, given the definition of the entailment shown in figure 5.1. It is easy to prove that this relation is an equivalence relation by using the according theorems of the bidirectional implication. The interfaces `BIBase` and `BI` are then instantiated with the quotient of the separation logic proposition type, i.e., for equivalence classes of separation logic propositions. This requires proving that the separation logic connectives return the same result for all members of an equivalence class. As an example, the connective `and` requires a proof that $P \approx P' \to Q \approx Q' \to P\,\sigma \wedge Q\,\sigma = P'\,\sigma \wedge Q'\,\sigma$ for all states σ based on the definition of `and` shown in figure 5.1. The proof is again simple for classical separation logic, since the equivalence relation $\approx$ is the bidirectional implication $\leftrightarrow$.

For classical separation logic, however, it is not necessary to use a setoid type in the instantiation of the typeclasses `BI` and `BIBase`. The reason for that is that Lean already allows proving an equality with a bidirectional implication, a fact that is called *propositional extensionality*. This is used in the form of the theorem `propext` in the proof of `equiv_entails` to replace the required equality with a bidirectional implication. The remaining proof is then to show that a bidirectional implication is equivalent to implications in both directions.

5.2. Proof Example and Comparison

The main goal of the separation logic interface is to provide users with the infrastructure to write proofs in separation logic. This requires the definition of the logic, the management of the separation logic context, the notation for separation logic propositions and tactics to assist in writing the proofs. All of these elements were presented in chapter 4. Figure 5.5 now shows an example of a separation logic proof using this implementation and compares it to a proof of the same proposition in the Coq formalization of MoSeL. At the end of the section, a proof of a similar statement in the pure logic of Lean is shown to demonstrate that the interface reaches the same level of abstraction for separation logic proofs. The proposition in the two separation logic theorems is written using the same notation in Lean and Coq, differing only slightly in the argument for the `BI` instance. Note that the shown proof does not require a specific instance of the separation logic interface, but works for all separation logics.

```
theorem proof_example [BI PROP]
    (P Q R : PROP)
    (Φ : α → PROP) :
    P ∗ Q ∗ □ R ⊢
    □ (R -∗ ∃ x, Φ x) -∗
    ∃ x, Φ x ∗ P ∗ Q
:= by
    iintro ⟨HP, HQ, □HR⟩ □HRΦ
    ispecialize HRΦ HR as HΦ
    icases HΦ with ⟨x, HΦ⟩
    iexists x
    isplit r
    · iassumption
    isplit l [HP]
    · iexact HP
    · iexact HQ
```

```
Lemma proof_example {A} {PROP : bi}
    (P Q R : PROP)
    (Φ : A → PROP) :
    P ∗ Q ∗ □ R ⊢
    □ (R -∗ ∃ x, Φ x) -∗
    ∃ x, Φ x ∗ P ∗ Q.
Proof.
    iIntros "[HP [HQ #HR]] #HRΦ".
    iDestruct ("HRΦ" with "HR")
        as (x) "HΦ".
    iExists x.
    iFrame.
    iAssumption.
Qed.
```

Figure 5.5.: Example of a separation logic proof in Lean using this implementation (left) and in Coq using MoSeL (right).

The proof in figure 5.5 then starts by introducing the hypotheses HP, HQ, HR and $H R\Phi$. The last two hypotheses are introduced to the intuitionistic context, specified in Lean using the prefix "□" and in Coq using "#". There is however a greater difference in the notation of the introduction patterns. In Lean, the introduction pattern is written as a Lean term, while in Coq, it is written as a string literal. This is because the Coq implementation is required to parse the pattern string itself. In contrast, the Lean implementation uses the Lean parser and matches directly on the defined syntax. This allows extending the syntax of conjunction and disjunction patterns to multiple arguments easily. The conjunction and disjunction patterns in Coq allow only two arguments by default, but there is an additional notation using the operator "&" for conjunctions with more than two arguments.

The next step in the proof is to process the hypothesis $H R\Phi$. The necessary actions include providing the required proof of R and destructing the existential quantifier. Resolving a wand with a proof of the premise is done using the tactic `ispecialize`, which is implicitly called in the Coq version through the pattern in the first argument of `iDestruct` (the Coq version of `icases` in Lean). The tactic `ispecialize` introduces (besides other features) the conclusion of a wand to the context. If this hypothesis and the hypothesis with the proof of the premise are in the spatial context, they are discarded. Otherwise they are kept in addition to the new hypothesis. After executing `ispecialize` in the example, the hypothesis $H\Phi : \exists x, \Phi x$ is part of the intuitionistic context. The temporary hypothesis generated by the first pattern in `iDestruct` is unnamed. The hypotheses $HR\Phi$ and HR remain in the intuitionistic context, but are no longer needed in the proof. The existential quantifier is then destructed using `icases` in Lean and `iDestruct` in Coq. The naming of the tactics follows the respective conventions in the two ITPs. In Lean, the anonymous constructor pattern $\langle\, \rangle$ is used again, while in Coq, parentheses are used to introduce variables from existential quantifiers. As a result, the hypothesis $H\Phi : \Phi\, x$ is placed in the intuitionistic context in both cases, replacing the previous hypothesis $H\Phi$ in Lean.

After introducing and destructing the hypotheses, the existential quantifier in the goal is resolved by providing the witness x. This is done in both versions of the proof shown in figure 5.5 using the tactic `iexists`. The Coq version of the proof can now use the tactic `iFrame` to apply the frame rule (see figure 2.4). The tactic uses the hypotheses HP and HQ to solve the propositions P and Q in the goal, leaving $\Phi\, x$ to be solved by `iAssumption`. Since the tactic `iframe` is not yet available in this implementation, the goal must be destructed manually using the tactic `isplit`. The tactic generates two goals for the two arguments of a separating conjunction and requires specifying how the hypotheses in the spatial context are split. For that purpose, the first argument indicates for which side the hypotheses should be listed (`l` for left and `r` for right). The remaining hypotheses will automatically be included in the spatial context of the other side. If no hypotheses are specified, then all hypotheses are moved to the respective side. The intuitionistic context is available to all generated goals. The first call to `isplit` therefore leaves no hypotheses in the spatial context of the left argument $\Phi\, x$ for which the tactic `iassumption` finds the hypothesis $H\Phi$ as a proof. The remaining conjunction $P * Q$ is again destructed using `isplit`, this time moving the hypothesis HP to the left side and (implictly) HQ to the right. Both hypotheses are then used to close the respective goals using the tactic `iexact`. Destructing the goal manually would work the same way in Coq.

The proof example in figure 5.5 already shows differences between the two implementations, e.g., the availability of certain tactics. The Coq implementation of MoSeL lists 20 core tactics, 15 of which are currently supported in the Lean implementation. The missing tactics can often be replaced with other tactics and include, for example, `iApply`, `iRevert` and `iAssert`. There are additional, more specific tactics, such as for modalities, induction and rewriting/simplification, that are also not part of the Lean implementation yet. Figure A.1 shows a complete list of the currently supported tactics, including smaller tactics that are not mentioned in the list of tactics in the Coq implementation. Another difference between the two implementations is the use of patterns in the arguments of tactics. The Coq implementation includes powerful patterns for selecting, introducing and specializing hypotheses, while the Lean implementation often requires multiple tactic calls to achieve the same behavior. This is visible in the example shown in figure 5.5 where the behavior of the tactic call `iDestruct` in the Coq implementation is achieved in the Lean implementation using a combination of `ispecialize` and `icases`. The specialization of $HR\Phi$ is performed in the Coq implementation using the specialization pattern in the first argument of `iDestruct`. Introduction patterns are also more powerful in Coq and allow, for example, immediate rewriting with an introduced hypothesis, although the syntax of the patterns is more general in Lean. The shorter style in the Coq proofs using patterns is not necessarily an advantage and is rather the result of the different proof styles in Lean and Coq, independent of the separation logic frameworks.

```
theorem proof_example [BI PROP]
    (P Q R : PROP)
    (Φ : α → PROP) :
    P ∗ Q ∗ □ R ⊢
    □ (R -∗ ∃ x, Φ x) -∗
    ∃ x, Φ x ∗ P ∗ Q
:= by
    iintro ⟨HP, HQ, □HR⟩ □HRΦ
    ispecialize HRΦ HR as HΦ
    icases HΦ with ⟨x, HΦ⟩
    iexists x
    isplit r
    · iassumption
    isplit l [HP]
    · iexact HP
    · iexact HQ
```

```
theorem proof_example
    (P Q R : Prop)
    (Φ : α → Prop) :
    P ∧ Q ∧ R →
    (R → ∃ x, Φ x) →
    ∃ x, Φ x ∧ P ∧ Q
:= by
    intro ⟨HP, HQ, HR⟩ HΦ
    specialize HΦ HR
    cases HΦ with | intro x HΦ =>
    apply Exists.intro x
    constructor
    · assumption
    constructor
    · exact HP
    · exact HQ
```

Figure 5.6.: Comparison of the separation logic proof from figure 5.5 (left) to a proof of a similar (yet weaker) statement in the pure logic of Lean (right).

Figure 5.6 compares the proof in figure 5.5 to a proof in the pure logic of Lean. The statement in the pure proof is obtained by converting all separating conjunctions to non-separating conjunctions and removing the modalities. In addition, the base logic of Lean is intuitionistic, which means that its propositions can be used multiple times. In the shown proofs, most tactic usages can be directly translated, including parts of the syntax, e.g., the pattern in the calls to `iintro` and `intro` or the keyword `with` in the syntax of `icases` and `cases`. The tactic `ispecialize` supports renaming the specialized hypothesis using the keyword `as`, which is not possible in the standard Lean tactic. On the other hand, the tactic `cases` is more powerful than `icases` and supports the destruction of arbitrary inductive types by generating individual goals for the constructors. Similarly, the more general tactic `apply`, which is not available in this implementation, can be used to destruct an existential quantifier. Although the proofs are not equal, the comparison visualizes that this implementation of a separation logic interface supports proofs on the same level of abstraction as used for common proofs in Lean.

```
inductive Env (α : Type)
    | nil  : Env α
    | cons : α → Env α → Env α

def get : (Γ : Env α) → Fin (Γ.length) → α
    | .cons a _ , ⟨0    , _⟩ => a
    | .cons _ as, ⟨i + 1, h⟩ => as.get ⟨i, Nat.lt_of_succ_lt_succ h⟩

Inductive env (A : Type) : Type :=
    | Enil  : env A
    | Esnoc : env A → ident → A → env A.

Fixpoint env_lookup {A} (i : ident) (Γ : env A) : option A :=
    match Γ with
    | Enil => None
    | Esnoc Γ j x => if ident_beq i j
        then Some x
        else env_lookup i Γ
    end.
```

Figure 5.7.: Definition of the inductive type `Env` representing one of the separation logic contexts in the Lean (top) and Coq (bottom) implementation. In addition, the operation for retrieving a hypothesis from a context is shown for both implementations.

5.3. Environment Definition

A major point in which this implementation differs from the Coq formalization of MoSeL is the definition of the environment, which models one of the separation logic contexts (intuitionistic or spatial). The key difference is that this implementation uses indices of a dependent finite datatype with the length of the environment as its upper bound. The definition of the inductive datatype `Env` is then of the same structure as a common list. On the other hand, the environment in the Coq version is a map with string identifiers as keys. Both definitions are shown together with the operations for retrieving a hypothesis from the environment in figure 5.7. The complete implementation of the environment datatypes was described in section 4.2. The signatures of the operations `get` and `envs_lookup` in figure 5.7 already show one difference between the two implementations: In Lean, the operation returns a hypothesis of type α, since the finite index is always valid. In Coq, on the other hand, the lookup may fail if the identifier is not a key in the map, which is why the function returns an element of the type `option A`. The implementation of the Coq operation `env_lookup` shows this lookup where the identifier i is compared to every identifier j in the map. In contrast, in the function `get` in the Lean implementation, the list is traversed until the correct position is found ($i = 0$). The case `.nil` does not need to be considered, since the index i is always a valid index in Γ. There is however additional work necessary to adapt the proof of the finite index when decreasing its value, performed by the predefined theorem `Nat.lt_of_succ_lt_succ`. Similar transformations are also required for other operations, the most complex one being from an index j of the type `EnvsIndex.of` Δ to the index j' of the type `EnvsIndex.of` (Δ`.delete` rp i) referring to the same hypothesis in the modified environments. Since the user of the separation logic interface specifies hypotheses by name in both the Lean and the Coq implementation, a lookup with string identifiers is also necessary in the Lean version. It is however performed on the meta level in the tactic implementations as described in subsection 4.3.3.

```
theorem tac_wand_intro [BI PROP]
    {Δ : Envs PROP} {P Q : PROP} (R : PROP) :
    [FromWand R P Q] →
    envs_entails (Δ.append false P) Q →
    envs_entails Δ R
```

```
Lemma tac_wand_intro {PROP : bi}
    (Δ : envs PROP) (i : ident) (P Q R : PROP) :
    FromWand R P Q →
    match envs_app false (Esnoc Enil i P) Δ with
    | None    => False
    | Some Δ' => envs_entails Δ' Q
    end →
    envs_entails Δ R.
```

Figure 5.8.: Example of the usage of the environment operation append/`envs_app` to add a hypothesis to a separation logic context in the tactic theorem `tac_wand_intro` in the Lean (top) and Coq (bottom) implementation.

The fact that environment operations can fail is also visible in tactic theorems using these definitions. The theorem `tac_wand_intro` is shown in figure 5.8 as an example of using the environment operation `append` (`envs_app` in Coq) to add a hypothesis P to the context Δ. Note that the shown operation modifies an entire `Envs` object, choosing the appropriate context based on the boolean parameter of `append`. In this case, the hypothesis is added to the spatial context, since the parameter is `false`. In the Lean implementation, the operation `append` directly returns the new `Envs` object and is always successful. In contrast, in the Coq implementation, the result of the operation `envs_app` is either the modified `Envs` object, in which case it is inserted into `envs_entails`, or `None` if the operation failed. As the result of an unsuccessful operation, the proposition `False` becomes a premise in the theorem and it cannot be used to transform the environment. The operation `envs_append` fails if the identifier i of the added hypothesis P is already present in the context Δ. On the tactic level this can also happen in the Lean implementation, but the necessary check for this is performed before the tactic theorems are used.

```
Inductive env_wf {A} : env A → Prop :=
    | Enil_wf : env_wf Enil
    | Esnoc_wf Γ i x :
        Γ !! i = None → env_wf Γ → env_wf (Esnoc Γ i x).

Record envs_wf' {PROP : bi} (Γp Γs : env PROP) := {
    env_intuitionistic_valid : env_wf Γp;
    env_spatial_valid : env_wf Γs;
    envs_disjoint i : Γp !! i = None ∨ Γs !! i = None
}.

Definition of_envs' {PROP : bi} (Γp Γs : env PROP) : PROP :=
    (⌜envs_wf' Γp Γs⌝ ∧ □ [∧] Γp ∗ [∗] Γs)%I.
```

Figure 5.9.: Definitions of well-formedness, i.e., that all identifiers are unique, for the environment types `env` and `envs` in the Coq implementation of the separation logic interface. The operator `!!` is a shorthand for the function `envs_lookup` and the postfix operator `%I` is used to indicate that the preceding term should be interpreted as separation logic proposition, similar to the `iprop` quotation in this implementation.

One additional requirement for naming the hypotheses in the separation logic context is that the names must be unique. While this is handled entirely on the meta level in Lean, since the environment operations are only concerned with indices, the Coq implementation requires a proof of the uniqueness. The necessary definitions are shown in figure 5.9. For a single `env` object, this proof is encoded in the inductive type `env_wf`, which characterizes the well-formedness of an `env` object. It states that an `env` object is well-formed if it is either empty (`Enil_wf`) or the identifier i of the next hypothesis x does not appear in the previous context Γ and Γ is already well-formed (`Esnoc_wf`). This well-formedness property is required for both the intuitionistic and spatial context. For that purpose, the record `envs_wf` combines two well-formedness proofs and adds the requirement that the two `env` objects must be disjoint, i.e., that no identifier is used in both contexts. An object of `envs_wf` states the well-formedness of an entire `envs` object consisting of the intuitionistic context *Γp* and the spatial context *Γs*. The embedding of the environment, for the Lean implementation explained in section 4.2, must then also carry this well-formedness proof. It is therefore added as a pure proposition to the embedding of the intuitionistic and spatial context in `of_envs'`. The well-formedness proofs must be reestablished in the soundness proofs of the operations where necessary.

A related task involving the names of hypothesis is the generation of fresh names that are not specified by the user, which can be useful for introducing anonymous hypotheses. Since hypothesis names are only available on the meta level in the Lean implementation, new names can be generated using the function `mkFreshUserName` from Lean's meta API. The environment representation and its operations are completely detached from this process. In the Coq implementation, on the other hand, a different approach is used to generate new hypothesis names on the object level. For that purpose, the field `env_counter` is added to the datatype `envs` besides the intuitionistic and the spatial context. The counter is increased by the tactic `iFresh` when a new hypothesis name is generated. Additional effort is necessary to ensure that the value of the counter does not have an influence on the equality of environment objects and that no additional steps are generated in the proof term when the value of the counter is increased.

```
def split : (Γ : Env α) → (mask : List Bool) →
    (mask.length = Γ.length) → Env α × Env α
    | .nil, .nil, _ => (.nil, .nil)
    | .cons a as, b :: bs, h =>
      let (ls, rs) := split as bs (length_cons_list_cons h)
      if b then (.cons a ls, rs) else (ls, .cons a rs)
```

```
Fixpoint envs_split_go {PROP} (js : list ident) (Δ1 Δ2 : envs PROP) :
   option (envs PROP * envs PROP) :=
    match js with
    | [] => Some (Δ1, Δ2)
    | j :: js =>
      '(p,P,Δ1') ← envs_lookup_delete true j Δ1;
      if p : bool
          then envs_split_go js Δ1 Δ2
          else envs_split_go js Δ1' (envs_snoc Δ2 false j P)
    end.

Definition envs_split {PROP} (d : direction) (js : list ident)
    (Δ : envs PROP) : option (envs PROP * envs PROP) :=
    '(Δ1,Δ2) ← envs_split_go js Δ (envs_clear_spatial Δ);
    if d is Right then Some (Δ1,Δ2) else Some (Δ2,Δ1).
```

Figure 5.10.: Definition of the environment operations for splitting the spatial context in Lean (top) and Coq (bottom). Note that the operation is defined for a single environment (Env) in Lean, while it is defined for the combined separation logic environments (envs) in Coq.

Another operation where this implementation uses a different approach, is the splitting of the spatial context in the tactic `isplit`. The idea of the approach was already described in section 4.2. Figure 5.10 now shows the implementation of the environment operations in the Lean and the Coq implementation. The function `split` in the Lean implementation uses a boolean mask `mask` to distribute the hypotheses in the environment Γ between two environments. The mask is required to have the same length as Γ, which is guaranteed by the proof `h`. The implementation then consists of a match on the environment and the mask, appending the next hypothesis to the left environment after the recursive call if the value in the mask is `true` and to the right environment if the value is `false`. The cases in which the environment is empty and the mask is not or vice versa do not have to be taken into account due to the proof that they have the same length. The proof that the remaining parts `as` and `bs` of the environment and the mask have the same length is performed by `length_cons_list_cons`. The Coq implementation, on the other hand, uses string identifiers in the implementation of the function `envs_lookup` shown in figure 5.7. The function `envs_split` moves the hypotheses referenced by the identifiers `js` to the left or right environment, depending on the argument `d`, and keeps the remaining hypotheses in the other environment. For that purpose, the helper function `envs_split_go` is called with the environment Δ, containing all hypotheses, and the environment `envs_clear_spatial` Δ with an empty spatial context. The intuitionistic context is available in both environments as explained in section 4.2. The function `envs_split_go` then moves the hypotheses `js` from the first environment $\mathit{\Delta}$`1` to the environment $\mathit{\Delta}$`2`. This is done by iterating over the list `js` and removing and retrieving each hypothesis `P` from $\mathit{\Delta}$`1` using the identifier `j`. The resulting environment is called $\mathit{\Delta}$`1`$'$. One important case distinction is to check whether the removed hypothesis was part of the intuitionistic context, in which case the environment $\mathit{\Delta}$`1`$'$ cannot be used, since the split considers only the spatial context. This problem does not occur in Lean, since the boolean mask for the spatial context cannot target the intuitionistic context. If the hypothesis was part of the spatial context, it is added to the environment $\mathit{\Delta}$`2`. The function `envs_split_go` always performs the move of the hypotheses from left to right ($\mathit{\Delta}$`1` to $\mathit{\Delta}$`2`), so the function `envs_split` is required to swap the environments if the direction `d` is `Left`. The user of the tactic `isplit` in the Lean implementation specifies the hypotheses in the same way as in Coq, but the hypothesis names are already resolved on the tactic level where the mask for the spatial context is created.

5.4. Limitations of Setoid Rewriting

The use of so-called *rewriting* in the necessary proofs of this separation logic interface was discussed in subsection 4.4.1. It was also mentioned that Lean's rewriting tactic `rw` only supports rewriting with equalities and that this is a major limitation for a separation logic interface. In a separation logic interface, two relations are relevant for rewriting: equivalences of the form $P \dashv\vdash Q$ (or similar) and entailments of the form $P \vdash Q$. The approach in this interface is to use Lean's rewriting tactic for the former relation and the custom rewriting tactic `rw'` for the latter, since rewriting with implications or entailments is unsupported. In order to use Lean's `rw` tactic for the equivalence relation $\dashv\vdash$, it is necessary to define it as an equality. This often requires using a setoid type for the separation logic propositions, as explained in section 5.1.

There are however equivalence relations for which Lean's `Setoid` class cannot be instantiated. One example is the relation $\overset{n}{=}$ from an *ordered family of equivalences (OFE)* in the Iris separation logic described in section 2.4. The relation is a step-indexed equivalence, which cannot be expressed in the non-parameterized class `Setoid`. Rewriting with this equivalence is in addition only allowed under certain, so-called *non-expansive*, functions, which preserve an equivalence when applied to its arguments. In contrast to Coq, Lean's rewriting tactics do not offer support for this conditional rewriting, which is one reason for the current inability to instantiate this separation logic interface with the Iris logic.

One possible solution could be to generalize the custom rewriting tactic `rw'`, which is part of this implementation, to (parameterized) equivalences. This would require adding rewrite rules for the bidirectional entailment $\dashv\vdash$ and supported separation logic connectives, as described for the entailment $\vdash$ in subsection 4.4.1. For the step-indexed equivalence $\overset{n}{=}$, additional rewrite rules and proofs of the non-expansiveness of the connectives would also be required.

6. Conclusion and Future Work

This work provides an interface for separation logic proofs in the interactive theorem prover Lean 4. After explaining the theoretical background of separation logic [1] in chapter 2, including the logic of bunched implications [2] and the concurrent separation logic Iris [7], a short introduction to the interactive theorem prover Lean [4] was given in chapter 3. The mentioned topics include typeclasses, used in the interface to require properties of propositions and relations, macros and meta programming, which are both relevant for the provided notation and tactics. Chapter 4 continued with describing the main parts of the implementation: the typeclasses for instantiating the interface with a custom separation logic, the notation for separation logic propositions, the context management for separation logic proofs and the context display, as well as the tactics used in separation logic proofs. In addition, an overview of the proofs of the theorems in the implementation was given.

The evaluation in chapter 5 showed that the provided separation logic interface can be instantiated with custom separation logics and an example instance was given for *classical separation logic* [1]. A proof of a separation logic statement using the provided tactics was presented and compared to a proof using the separation logic interface MoSeL [5] in Coq. The proof example showed that the provided interface is capable of supporting separation logic proofs in Lean and highlighted advantages, different styles and features that are not yet implemented compared to MoSeL. A proof of a similar statement in the pure logic of Lean indicates that the proofs in the provided separation logic interface are able to reach the abstraction level of common proofs in Lean. Discussing the definition of internal data structures made clear that this implementation contains improvements over the implementation of MoSeL.

One improvement of this implementation is based on the decision to keep the identifiers of hypotheses entirely on the meta level, which leads to a simpler definition of the separation logic context. This is visible in the definition of the respective datatypes, the functions and theorems that use them, and the soundness proofs of the operations. It does however require complex index transformations and the tactic implementations have to manually process kernel expressions. In contrast, using string identifiers on the object level in the implementation of MoSeL requires a more complex environment definition and the operations and additional proofs on the object level cause larger proof terms for the validation in the kernel.

The presented separation logic interface is already capable of performing interactive proofs in separation logic, but more work is necessary to improve the interface and allow instantiating it with more complex logics, such as the concurrent separation logic Iris. The necessary changes include the definition of equivalence, a more general logic interface and additional separation logic modalities. The implementation should also be extended by adding more tactics to simplify proofs containing, for example, modalities or induction. There are furthermore possibilities to improve the usability, as well as the error handling of the already available tactics. Additional examples of instances with different separation logics, as well as semantic languages models using the provided separation logic interface would help to further evaluate the implementation.

Bibliography

[1] P. O'Hearn, J. Reynolds, and H. Yang, "Local Reasoning about Programs that Alter Data Structures", en, in *Computer Science Logic*, G. Goos, J. Hartmanis, J. van Leeuwen, and L. Fribourg, Eds., vol. 2142, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 1–19, ISBN: 978-3-540-42554-0 978-3-540-44802-0. DOI: `10.1007/3-540-44802-0_1`.

[2] P. W. O'Hearn and D. J. Pym, "The Logic of Bunched Implications", en, *Bulletin of Symbolic Logic*, vol. 5, no. 2, pp. 215–244, Jun. 1999, ISSN: 1079-8986, 1943-5894. DOI: `10.2307/421090`.

[3] P. W. O'Hearn, "Resources, concurrency, and local reasoning", en, *Theoretical Computer Science*, vol. 375, no. 1-3, pp. 271–307, May 2007, ISSN: 03043975. DOI: `10.1016/j.tcs.2006.12.035`.

[4] L. de Moura and S. Ullrich, "The Lean 4 Theorem Prover and Programming Language", in *Automated Deduction – CADE 28*, A. Platzer and G. Sutcliffe, Eds., Cham: Springer International Publishing, 2021, pp. 625–635, ISBN: 978-3-030-79876-5.

[5] R. Krebbers *et al.*, "MoSeL: A general, extensible modal framework for interactive proofs in separation logic", en, *Proceedings of the ACM on Programming Languages*, vol. 2, no. ICFP, pp. 1–30, Jul. 2018, ISSN: 2475-1421. DOI: `10.1145/3236772`.

[6] R. Jung, R. Krebbers, *et al.*, *The Coq development for Iris*, en. [Online]. Available: `https://gitlab.mpi-sws.org/iris/iris/-/tree/96883dbdd500f9688fb5f241361969a098dfec63` (visited on 2022-09-19).

[7] R. Jung, R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer, "Iris from the ground up: A modular foundation for higher-order concurrent separation logic", en, *Journal of Functional Programming*, vol. 28, e20, 2018, ISSN: 0956-7968, 1469-7653. DOI: `10.1017/S0956796818000151`.

[8] R. Krebbers, A. Timany, and L. Birkedal, "Interactive proofs in higher-order concurrent separation logic", en, in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, Paris France: ACM, Jan. 2017, pp. 205–217, ISBN: 978-1-4503-4660-3. DOI: `10.1145/3009837.3009855`.

[9] J.-Y. Girard, "Linear Logic", en, *Theoretical Computer Science*, vol. 50, no. 1, pp. 1–101, 1987.

[10] R. Jung *et al.*, "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning", en, in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Mumbai India: ACM, Jan. 2015, pp. 637–650, ISBN: 978-1-4503-3300-9. DOI: `10.1145/2676726.2676980`.

[11] L. de Moura, J. Avigad, S. Kong, and C. Roux, *Elaboration in Dependent Type Theory*, en, arXiv:1505.04324 [cs], Dec. 2015. [Online]. Available: `http://arxiv.org/abs/1505.04324`.

[12] D. Selsam, S. Ullrich, and L. de Moura, *Tabled Typeclass Resolution*, en, arXiv:2001.04301 [cs], Jan. 2020. [Online]. Available: `http://arxiv.org/abs/2001.04301`.

[13] M. P. Jones, "Type Classes with Functional Dependencies", en, in *Programming Languages and Systems*, G. Goos, J. Hartmanis, J. van Leeuwen, and G. Smolka, Eds., vol. 1782, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 230–244, ISBN: 978-3-540-67262-3 978-3-540-46425-9. DOI: `10.1007/3-540-46425-5_15`.

[14] S. Ullrich and L. de Moura, "Beyond Notations: Hygienic Macro Expansion for Theorem Proving Languages", en, *Logical Methods in Computer Science*, vol. Volume 18, Issue 2, p. 7421, Apr. 2022, ISSN: 1860-5974. DOI: `10.46298/lmcs-18(2:1)2022`.

[15] S. Ullrich, *Typed Macros*, en, Jun. 2022. [Online]. Available: `https://github.com/leanprover/lean4/pull/1251` (visited on 2022-08-21).

[16] A. Paulino *et al.*, *A Lean 4 Metaprogramming Book*, en. [Online]. Available: `https://github.com/arthurpaulino/lean4-metaprogramming-book` (visited on 2022-08-22).

[17] G. Ebner, *Expression Quotations for Lean 4*, en. [Online]. Available: `https://github.com/gebner/quote4` (visited on 2022-09-02).

A. Appendix

`istart`
  Start the separation logic proof mode.

`istop`
  Stop the separation logic proof mode.

`irename` *nameFrom* `to` *nameTo*
  Rename the hypothesis *nameFrom* to *nameTo*.

`iclear` *hyp*
  Discard the hypothesis *hyp*.

`ispecialize` *hyp args* `as` *name*
  Specialize the wand or universal quantifier *hyp* with the hypotheses and variables *args* and assign the name *name* to the resulting hypothesis.

`iexists` x
  Resolve an existential quantifier in the goal by introducing the variable x.

`iexact` *hyp*
  Solve the goal with the hypothesis *hyp*.

`iassumption_lean`
  Solve the goal with a hypothesis of the type $\vdash Q$ from the Lean context.

`iassumption`
  Solve the goal with a hypothesis from any context (intuitionistic, spatial or pure).

Figure A.1.: List of all tactics currently supported in this implementation of a separation logic interface for Lean 4. The "`i`" prefix is used to distinguish the separation logic tactics from the Lean tactics.

`iex_falso`

Change the goal to $False$.

`ipure` *hyp*

Move the hypothesis *hyp* to the pure context.

`iintuitionistic` *hyp*

Move the hypothesis *hyp* to the intuitionistic context.

`ispatial` *hyp*

Move the hypothesis *hyp* to the spatial context.

`iemp_intro`

Solve the goal if it is *emp* and discard all hypotheses.

`ipure_intro`

Turn a goal of the form $\ulcorner\phi\urcorner$ into a Lean goal ϕ.

`isplit`

Split a conjunction $\wedge$ into two goals, using the entire spatial context in both of them.

`isplit` *(*`l`/`r`*)* `[`*hyps*`]`

Split a separating conjunction $*$ into two goals using the hypotheses *hyps* as the spatial context for the left (`l`) or right (`r`) side. The remaining hypotheses in the spatial context are used for the opposite site.

`ileft`
`iright`

Choose to prove the left or right side of a disjunction in the goal.

`icases` *hyp* `with` *pat*

Destruct the hypothesis *hyp* using the pattern *pat*.

`iintro` *pats*

Introduce up to multiple hypotheses and destruct them using the patterns *pats*.

Figure A.1.: Continued list of all tactics currently supported in this implementation of a separation logic interface for Lean 4.