DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Towards Automated Proofs in Higher-Order Concurrent Separation Logic

Florian Sextl

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Towards Automated Proofs in Higher-Order Concurrent Separation Logic

Automatisierung von Beweisen in nebenläufiger Separationslogik höherer Ordnung

Author:	Florian Sextl
Supervisor:	Prof. Tobias Nipkow, Ph.D.
Advisors:	Mohammad Abdulaziz, Ph.D.
	M.Sc. Simon Roßkopf
	M.Sc. Fabian Huch
Submission Date:	May 16, 2022

I confirm that this master's thesis is my own work and I have documented all sources and material used.

München, Florian Sextl

Acknowledgments

First and foremost, I want to thank my advisors Mohammad Abdulaziz, Simon Roßkopf and Fabian Huch who supported me immensely throughout this project. I especially want to thank them for the many hours they spent discussing all details of my work with me in our regular meetings and their great feedback and suggestions about and for my work. I am also extremely grateful for all the support and advice they provided me with regarding my further academic career.

I would also like to thank Ralf Jung, who acted as an external advisor for this work and who provided me with great insights into the intricate details of Iris and the Coq system. Although, we did not have many opportunities to discuss these topics, his input helped me immensely with understanding all necessary ideas and notions.

Other than that, I also want to thank Tobias Nipkow and all the other members of the Chair for Logic and Verification at TU Munich that I had the pleasure to work with in the last years. Their guidance and the knowledge I was able to build up over this time enabled me to successfully finish this project in a way that would not have been possible otherwise.

Last but not least, I also want to thank all other people who helped me answer many of the questions that came up throughout my work. I am especially gratefully to both the members of the Isabelle and the Iris community for building open and welcoming spaces for constructive and fruitful discussions as well as asking questions and receiving answers from experienced researchers.

Abstract

The Iris framework implemented in Coq allows defining higher-order concurrent program logics based on separation logic. This work investigates how well that framework can be ported to Isabelle/HOL and whether the proof automation for such a port can be as efficient as or better than in Coq. To this end, a partial port of Iris has been developed that contains all necessary facilities to work with selected real-world examples. In this process, we analyzed how Coq features used by Iris can be translated to Isabelle and found that the whole framework can in general be recreated there, although with certain caveats. A full port is necessarily more verbose with regard to composable proofs and requires either an axiomatic extension to HOL or an mapping from syntax to semantics outside of the logic depending on how the Iris logic is embedded. In another step, we developed specialized proof automation methods for our Iris port and compared them with existing Coq machinery for the same purpose. To this end, we developed translation techniques for common mechanization patterns from Coq to Isabelle. We conclude that at least in the context of the Iris logic neither Isabelle nor Coq provide significantly stronger proof automation compared to the other system.

Das im Beweisassistenten Coq implementierte Rahmenwerk Iris ermöglicht es nebenläufige Programmlogiken höherer Ordnung basierend auf Separationslogik zu definieren. Diese Arbeit untersucht wie gut sich dieses Rahmenwerk nach Isabelle/HOL übertragen lässt und ob die Beweisautomatisierung für solch eine Übersetzung effizienter oder besser sein kann als für das Original. Zu diesem Zweck wurde eine partieller Übersetzung von Iris entwickelt, welche mit alle notwendigen Funktionen ausgestattet ist um ausgewählte realistische Beispiele handhaben zu können. Als Teil dieses Prozesses analysierten wir Ansätze um Merkmale von Coq die Iris nutzt nach Isabelle zu übertragen und kamen zu dem Ergebnis, dass eine vollständige Übersetzung möglich wäre, obgleich diese gewissen Einschränkungen unterliegen würde. Eine vollständige Übersetzung würde eine ausführlichere Handhabung von modularen Beweisen benötigen. Des Weiteren würde abhängig von der gewählten Einbettung der Iris Logik entweder eine axiomatische Erweiterung des HOL Systems oder eine explizite Übersetzung von Syntax zu Bedeutung außerhalb der Logik notwendig. In einem weiteren Schritt entwickelten wir spezialisierte Beweisautomatisierungsmethoden für unsere Übersetzung von Iris und verglichen diese mit der vergleichbaren Maschinerie in Coq. Zu diesem Zweck entwickelten wir Übersetzungsmethodiken für übliche Beweismuster. Wir kommen letztlich zu dem Schluss, dass im Kontext des Iris Rahmenwerks weder das Isabelle noch das Coq System eine klar stärkere Beweisautomatisierung ermöglichen.

Contents

1 Introduction

“One of the reasons I prefer higher-order logic to dependent type theories — apart from simple semantics, equality that works and no need to put everything in the kernel — is that dependent types seem to make automation much more difficult.”

— *Lawrence C. Paulson*, Machine Logic [39]

Our modern world relies heavily on digital technology. Many of the involved systems are based on low level concurrent principles that rely on direct memory management, e.g. locks. Implementing these primitives correctly and efficiently is not trivial. Yet, erroneous implementations, especially of memory accesses, are a common reason for software vulnerabilities. For instance, the latest report about zero-day exploits by the Google Project Zero team [14] relates around 70% of disclosed faults in 2021 to memory corruption vulnerabilities, which mostly stem from few, well known patterns. These patterns can often be found easily by formal methods such as program verification. As a result, formal verification can improve the correctness of implementations of the aforementioned primitives and guarantee the absence of certain classes of bugs and vulnerabilities.

Reasoning based on separation logic is especially useful for low level, concurrent memory accesses. Variations of this logic allow their users to express fine-grained propositions about resources such as memory locations and their ownership even in the context of concurrent computation. In addition, separation logic facilitates program verification in a modular and well automatable fashion. For these reasons, many specialized separation logics have been developed for a broad range of verification tasks over the last twenty years. Unfortunately, many of the developed logical systems are not compatible with each other and make combining developments impossible. To unify this diverse research field and its “next 700 separation logics” [36], the Iris framework strives to provide a shared platform for all sorts of specialized program logics based on higher-order, concurrent separation logic [20, 21, 25, 45].

The Iris framework is formalized in the widely used proof assistant Coq [18]. Although this allows a great number of users to utilize the framework, it is not available for users of other proof systems. These users can not make use of the framework, because the underlying logic systems of the proof tools they are using are often not compatible with the Coq logic. Among these tools, the interactive theorem prover Isabelle with its object logic Isabelle/HOL is one of the most widely used ones. Even though the Isabelle community has developed several useful separation logic libraries, none of these are as generic as the Iris framework and can, therefore, be used to formalize the same kind of complex program specifications in a correspondingly specialized program logic. To be more precise, there exists no framework for higher-order, concurrent separation logics that would allow different developments to be compatible like Iris does. Therefore, a port of Iris to Isabelle/HOL would enrich the Isabelle ecosystem with a new way to define specialized program logics for composable developments. Moreover, such a port could help in attaining a better understanding of how to make Isabelle/HOL and Coq better interoperable. As a result, this thesis strives to answer the question whether the Iris framework can be ported to Isabelle and how Coq-specific logical primitives can be translated.

However, even a generic logical framework such as Iris, which allows expressing many useful program specifications, can only improve its users' experience in formalizing proofs if it comes with strong and adjustable proof automation. The initial quote by Larry Paulson, who originally developed the Isabelle system, expresses the widespread assumption that the limitations of the HOL logic allow for better proof automation in comparison to other logic systems. Therefore, this work examines proofs in the Iris logic and contrasts comparable proof principles from both Coq and Isabelle to find evidence for or against this common conjecture.

The main contributions of this thesis are a partial port of Iris to Isabelle and the thus obtained results to the aforementioned research questions. The port contains all necessary components to verify selected small, but realistic examples and is available as open source software [42]. It consists of about 10k lines of Isabelle code, including experimental alternative approaches to porting specific implementation details. However, the port also contains axiomatizations of central concepts for which we did not find an adequate formalization. In addition to the port, we developed idiomatic translations for several proof automation mechanisms and convenience features of Iris and surveyed how feasible and idiomatic they would make a full port to Isabelle.

2 Related Work

Although there have been a number of works about parallel developments in more than one proof system, there are not as many about porting developments between them. Chen et al. formalized and proved Tarjan's strongly connected components algorithm in Why3, Coq, and Isabelle [4]. They kept their formalizations as similar as possible and chose to not use idiomatic features of the different systems to make them better comparable. In contrast, we decided to aim for an idiomatic translation of the Iris framework to show whether a full port would be possible and could be integrated in further developments as well as the Coq formalization.

In contrast, Yushkovskiy and Nawaz et al. published a survey each on proof systems in general [33, 52]. Whereas the former focuses on comparing only Coq and Isabelle on a high level, the latter contrasts a greater number of different systems with each other. In contrast to our work, both surveys only investigate high-level differences in an abstract manner and do not explore how to translate specific developments between the systems. Such a high-level overview might help deciding which system to use in a green-field development but does not suffice to port an existing formalization between the systems.

It is also necessary to note that a full Iris port to Isabelle would not be the only separation logic framework available. The Archive of Formal Proofs contains three different separation logic developments by Klein et al. [23], Hou et al. [17] and Lammich and Meis [27]. The development by Klein et al. provides a formalization of the algebra of abstract separation logic, for which the library by Hou et al. adds a specialized automatic solver. Propositions in this logic are defined as assertions on values of an arbitrary resource type. Similarly, the development by Lammich and Meis defines propositions in the logic to be HOL predicates on a fixed heap type. This library is an extension to the Imperative HOL framework and the related Refinement framework (cf. [26]). It also comes with automated proof methods for the logic's entailments. In contrast to the step-indexed Iris logic with embedded meta assertions, the simpler logics for these frameworks allow for a strong, yet simple proof automation. However, all of these developments are aimed at simple verification of primarily imperative programs and, thus, differ significantly from the more generic Iris framework. As a result, the developments all serve different purposes.

We also need to consider work related to Iris. This project is the first to attempt a port of the Iris framework, but there is currently another project going on by Lars König to port the Iris Proof Mode [25] and MoSeL [24] to Lean. The project has only started as of the time of this writing and investigates mostly orthogonal questions to our work. For this reason, their work is quite likely to find other difficulties to overcome and, thus, further the overall understanding of how one can port complex developments such as Iris between widely used theorem provers.

On another note, our work on automating proofs in the Iris logic is mostly related to the Diaframe tool by Mulder et al. [32]. This tool allows full automation of a commonly used subset of Iris formulae and provides at least partial automation for other proof goals. Section 6.4 contains a more detailed description of how their automation approach works and how our work relates to it. For this reason, our work can also be compared to other automatic program verification tools based on separation logic such as Caper [13] in the same way as Diaframe. The Diaframe paper already provides a detailed survey of how these tools relate and differ from its approach.

3 Background

The following paragraphs introduce the most relevant basics that are required to understand our work on the central research questions. To this end, both the Isabelle and the Coq proof assistants are introduced with a focus on how they differ on a high level. In a next step, we explain the fundamentals of separation logics needed for understanding Iris, which is then introduced in chapter 4. Throughout this thesis, implementations and names of functions and definitions in Coq or Isabelle are denoted with `typewriter` font, whereas *italic* font is used for logical formulae.

3.1 Isabelle/HOL

The interactive theorem prover Isabelle [37, 38] is a LCF-style [16] logical framework. This means that it allows users to define object logics on top of a meta logic and reduces proofs to an abstract datatype, which is constructed by a small, trusted logic kernel. The meta logic is called Isabelle/Pure and consists of an intuitionistic fragment of higher-order logic, which supports three types of dependence (cf. [50]): terms depending on terms (i.e. functions), proofs depending on terms (i.e. universal quantification), and proofs depending on proofs (i.e. implication). Proofs in Isabelle/Pure are instances of an abstract datatype, which can only[1] be constructed from axioms and inference rules. These inference rules are based on higher-order unification and higher-order resolution. This approach to proof representation is called the LCF architecture after the LCF system by Gordon [16] and allows the system to "forget" the concrete proof steps after the proof object has been constructed. For this reason, the exact proof script provided by the user is irrelevant to the proof as long as Isabelle's kernel can use it to actually construct the correct proof object. As a result, proofs in Isabelle depend only on the trusted logic kernel. Additionally, the proof terms can be explicitly recorded to be checked by external verifiers (cf. [49, §2.5]).

Isabelle/HOL In contrast to Isabelle/Pure, Isabelle/HOL [34] is based on a simply typed version (cf. [6]) of classical set theory called *Higher Order Logic* (HOL, cf. [15] for the theorem proving environment of the same name). It is the most widely used object logic for Isabelle, partially due to it being implemented more closely to Pure than other object logics. To be more precise, HOL types are represented as Pure types,

[1]Isabelle also allows the user to explicitly opt-in to omitted proofs denoted by the `sorry` keyword. Though, this feature is only meant for prototyping.

which makes the translation of terms and proofs from HOL to Pure straightforward and aids much in the development of low-level automated proof mechanisms. However, this implementation detail also fixes the flexibility of the HOL type system to its Pure counterpart. As an example, HOL can not have higher kinded type variables as Pure does not support these either. Yet, Isabelle/HOL supports the classification of types by their sorts, which are lists of *type classes* a type is an instance of (cf. [49]). These (axiomatic) type classes are similar to the ones found in Haskell but support a wider range of objects for ad-hoc polymorphism. They do not only support fixing terms and functions for a type, but also axioms about these. Axiomatic type classes in Isabelle/HOL are defined as a special subtype of *locales*, which are comparable to fixed proof contexts containing types, terms and assumptions.

The HOL type system is based on three fundamentals: the `bool` type for booleans and propositions, the infinite type `ind` of individuals (e.g. used as the basis for natural numbers) and the function type constructor ($\Rightarrow$). All other types are defined by either combining existing types and type constructors or naming a subset of values of an existing type as a new type. The latter option is commonly called *semantic subtyping* and can, for example, be used to guarantee a type invariant by construction. More complex type definitions, such as algebraic data types, can be implemented in terms of product and sum types or by defining an initial algebra/final coalgebra. The Isabelle datatype package by Traytel et al. (cf. [46]) handles such datatype definitions automatically and ensures that the defined types adhere to the notion of so called Bounded Natural Functors. This approach also allows for (co-)recursive datatypes, thus enabling HOL to support both types with infinitely many unique inhabitants and infinitely sized values. However, HOL functions over recursive types must terminate to be sound. As a result, all function definitions are either not recursive or must follow a decreasing induction schema. Although these schemas can often be induced by the system, the user must provide a proof that the function will terminate in all cases for complex recursion patterns.

In-line Isabelle/ML Isabelle allows its users to write system level code in-line in its own domain-specific language Isabelle/ML. This language extends Isabelle's implementation language Standard ML (subsequently ML) with the Isabelle/Pure specific APIs and enables direct interaction with the system. This kind of interaction can be used to write logic level code generation, custom proof procedures, and handling of system data structures, inter alia. The ability to write such functionalities in-line and evaluate them alongside object logic definitions requires no recompilation of the system and enables a straightforward development cycle. For this reason, many Isabelle developments facilitate in-line ML code for small mechanization tasks. Likewise, this work makes heavy use of in-line ML code to reduce boilerplate code and overcome limits to the definition of custom proof automation methods.

3.2 Coq

In contrast to Isabelle, the Coq system is based on only one fixed logic — the *Calculus of Inductive Constructions* (CIC). This logic is based on the Calculus of Constructions (CoC) by Coquand and Huet [7], which is the type theory at the top of the lambda cube by Barendregt [1]. The CoC allows dependence over all three dimensions of the cube and, therefore, supports polymorphic terms (terms depending on types), type constructors (types depending on types), and dependent types (types depending on terms). In addition, the CIC extends this system with the notion of inductive definitions of terms and types. These allow the user to work with recursive definitions provided that these have a (least) fixed-point and terminate. Other than inductive definitions, Coq users often utilize dependent records to bundle types, terms and properties into a single unit. These records are also the basis for Coq's type classes and *canonical structures*; these are two similar principles used for abstracting formalization details and structuring proof reasoning. Both of these use cases are described in more detail in later sections.

Unlike Isabelle, Coq does not separate propositions and terms but utilizes the Curry-Howard isomorphism to interpret types in the CIC as propositions and terms as proofs. Based on this fundamental notion, validating a proof becomes the same as checking whether its term is of the claimed type. Therefore, Coq's kernel employs type checking, term evaluation, which is necessary to type check dependent types, and higher-order unification to verify given proofs. Although the existence of a term with the corresponding type is already enough to prove a proposition in the Curry-Howard correspondence and the actual term can therefore be ignored after type checking, proofs in Coq are in general relevant and are most of the time kept available[2]. These proofs have to be constructed as terms and can sometimes be used as such afterwards again. Their size and complexity can also have a great impact on the proof's runtime and, although this does not contradict proof irrelevance, might effect the user's acceptance of these specific proofs. On the other hand, the Coq system follows the de Bruijn criterion, i.e. a Coq proof term contains all information necessary to validate it and can thus be easily exported to be checked by other tools, as well (cf. [5, section 1.2.3]).

Although Coq also supports program extraction from its internal programming language Gallina to its implementation language OCaml, the system does not support dynamic extensions in the way Isabelle does with in-line ML code. Every new functionality written in OCaml can only be added to Coq as a plugin. This process requires recompilation of the involved components. For this reason, Coq has been extended with several libraries for meta programming languages, which allow the user to define new functionalities in the user space and without the need to recompile. One such library for tactical reasoning, Ltac, is described in more detail in section 6.2.

[2]See also the difference between `Prop` and `SProp` as described in [18].

3.3 Embedding of Logics

Similarly to how both the CIC and Isabelle/Pure are implemented in terms of Standard ML and OCaml respectively, other logics can be embedded in their programming languages as well. For Isabelle, this is also the way that object logics such as Isabelle/HOL are formalized on top of Pure. Although the general idea is equivalent for all of these levels of embedded logics, there exist relevant differences, which also play a large role for the formalization of a logic framework such as Iris. In general, most logic systems are implemented in one of two embedding methods: *shallow* or *deep* embedding (cf. [51]). These categories describe how the logic interacts with the meta logic system it is implemented in and can also be used in other contexts such as programming languages and their semantics.

Shallow Embedding In general, a shallow embedding reuses existing structures like types and terms and can therefore exploit all facilities of the meta system. In the context of programming languages, domain specific languages are a common example for shallow embeddings, as they often consist of specialized constructions of their host language. In the context of logics, most systems nowadays contain a lambda calculus for constructing formulae, which makes it possible to share structures between an object logic and its meta system. For example, the Isabelle/HOL object logic is embedded shallowly into the Isabelle/Pure meta logic by reusing its type system. Therefore, all well-formed HOL terms are also well-formed Pure terms and can be handled in the same way in the context of term manipulation and proof automation.

Whereas the Isabelle/HOL system only reuses the Pure type system and adds basic terms via axiomatization, the Iris formalization in Coq is a shallow embedding based on a single Gallina type and no term axiomatizations. This allows Iris propositions to be defined directly with semantics encoded in Coq's proposition type `Prop`. For this reason, it is not only trivial to add new logical operators to Iris but also possible to directly relate semantic properties with them. In addition, a shallow embedding makes it easy to embed meta terms in object level formulae and, therefore, lift definitions and reasoning to whichever level is most convenient.

Deep Embedding In contrast to a shallow embedded logic, a deep embedded logic is in most cases strictly distinct from the meta system. This means that terms and formulae of the embedded logic are instances of a meta datatype, e.g. based on the constructors of an algebraic datatype. These terms can then be typed with an equally embedded type system or even left untyped. Moreover, the semantics of the embedded logical operators and primitives need to be defined separately from their syntax by translating them into meta level definitions. Although this translation can often be used to relate syntax and semantics directly, it is in general possible to perform arbitrary transformations to the syntax and break the semantic meaning by doing so. This kind of transformations allow for code optimizations and proof mechanisms based on rewriting, but require an external soundness proof to show that these operations do not break the intended semantical meaning. This process can be aided by intermediate stages of the transformation. As an example, it is possible to extend the syntactic formulae with meta terms to represent certain semantic properties. Yet, due to the representation of embedded terms as instances of an abstract datatype, these meta terms can only have one of a fixed number of different types. Especially a type system without generalized algebraic datatypes such as HOL can not support other ways of embedding meta level terms directly into the object level.

In general, both embedding methods can be used to formalize many logical systems or program language semantics without any significant difference. They mostly differ in how mechanizations for them are implemented. Pretty printing for a better readability and proof automation are two common examples of such mechanisms.

3.4 Separation Logic

Reasoning about programs with shared mutable data structures can be quite involved. To solve this problem, Reynolds introduced the notion of separation logic in his seminal work [41] and based it on the logic of bunched implications by O'Hearn and Pym [35]. Since its first introduction, many different flavors of separation logic have been developed for a great number of different purposes. Yet, they all share the same basic idea: abstracting resources as part of the program state and making reasoning about these local. As an example, abstract memory representation is a simple and often utilized resource; mostly as a heap map from abstract location identifiers to memory cell contents. For simplicity, the following section focuses on an abstract program state representation containing only a heap h. Let $h \vdash P$ denote that the proposition P holds for the abstract program state encoded by h. Moreover, let $\oplus$ denote the disjoint composition of two heaps, i.e. the union of the maps if their domains are disjoint.

Most separation logics contain both the classic logical operators such as conjunction or implication and the resource-specific operators *separating conjunction* (denoted by $*$) and *magic wand* ($-\!\!*$). The separating conjunction $P * Q$ denotes that the two propositions P and Q hold for distinct parts of the heap. This means that the compound proposition holds for a program state in which the heap h can be divided into two distinct subheaps $h1$ and $h2$ (i.e. $h = h1 \oplus h2$) such that both $h1 \vdash P$ and $h2 \vdash Q$ hold. Thus, the separating conjunction operator functions as a sort of conjunction for predicates about the heap. Similarly, the magic wand operator can be seen as an implication or extension on heap predicates. This means that $P -\!\!* Q$ holds for a state h if given any state h' such that $h' \vdash P$, the combined state $h \oplus h'$ satisfies Q. As an example, assume that $h \vdash P * (P -\!\!* Q)$ holds. This is equivalent to the fact that there exist two distinct heaps $h1, h2$ such that $h = h1 \oplus h2$, $h1 \vdash P$ and $h2 \vdash P -\!\!* Q$ all hold. Due to the semantics of the magic wand operator, one can then deduce that $h \vdash Q$ also holds.

Although the described semantics only hold for the very simple case of a heap map, separation logic can easily be defined to work with arbitrary other resources in a similar way. Moreover, the separating conjunction and magic wand operators enable modular and well structured reasoning about these resources. For this reason, separation logic has become a standard tool for reasoning about programs with direct memory manipulation, shared mutable data structures, and pointers. This holds for both interactive theorem provers and fully automated tools. The latter are in general based on either completely decidable separation logics (cf. [2]) or on other variations that allow for good proof automation.

Classical and Affine Separation Logics Separation logics can be divided into two categories: *classical* and *affine* separation logics. These categories differ in both the semantics of propositions and whether they have associated weakening rules. The classical separation logic requires propositions to only hold for the exactly necessary resources. In the context of our heap example, a proposition that does not refer to the heap can only hold for a given empty heap. On the contrary, propositions also hold for larger resources in an affine separating logic as long as they contain the minimal necessary fraction. In the heap example, a proposition that only expects the cell at a location l to hold a value v holds also for a heap that additionally contains other values at other locations at well. In the context of affine separation logic, it is possible to define weakening rules, with which unneeded assumptions can be dropped. The intuition behind these rules is that from assertions that expect more resources one can always deduce an assertion that requires only a part of these.

4 Introduction to Iris

The Iris framework allows its users to define higher-order concurrent program logics with a separation logic at their core. For this, they have to provide certain predicates and definitions about the programming language they want to use. Based on these, the actual program logic is instantiated on top of Iris' base logic. Iris also comes with lots of mechanisms to make reasoning in its program logics more comfortable and straightforward. In addition, the framework contains an instantiation for the simple, ML-like language *HeapLang*, in which most examples in the Iris formalization in Coq and this thesis are implemented.

The following sections introduce the Iris basics necessary to understand the rest of the thesis. It should be noted that the complete formal definitions for all introduced concepts can be found in the Iris documentation [45], where they are defined abstractly. For this reason, that document can be seen as the theoretical background for both the Coq formalization and our port.

4.1 Resources in Iris

Many low level programs and data structures manipulate specific resources such as memory cells or locks. For this reason, Iris allows the user to define their own resource types and reason about these inside the Iris logic. To be useable with Iris, these resource types need to be *resource algebras*. A simplified definition of the corresponding type class in Isabelle can be found in listing 1. Resource algebras are structures similar to partial commutative monoids and include a validity predicate, a (partial) core operator, and a total composition operator. These operators need to adhere to some specific rules such as associativity but can in general be reduced to the following intuitions. The validity predicate decides which elements of the resource can be used without breaking the soundness of the other operations. The core operator computes a (unique) identity element with regard to the composition for an input resource element if possible. Whereas the core is also allowed to be partial, composition needs to be defined for all resource elements. To satisfy this property, some resource types need to be extended with a dedicated invalid element for all possible compositions that should not result in sound behavior.

Although it suffices for most resource types to be resource algebras, reasoning about how a program manipulates resources can require differentiation of different program steps. For this reason, certain resource types need to be *Ordered Families of Equivalences*

```
class ra =
    fixes valid :: 'a ⇒ bool
        and core :: 'a ⇒ 'a option
        and comp :: 'a ⇒ 'a ⇒ 'a (infix ·)
    assumes ra_assoc: (a · b) · c = a · (b · c)
        and ra_comm: a · b = b · a
        and ra_core_id: core a = Some a' ⟹ a' · a = a
        and ra_valid_op: valid (a · b) ⟹ valid a
        ...
```

Listing 1: Resource algebra type class.

```
class ofe =
    fixes n_equiv :: nat ⇒ 'a ⇒ 'a ⇒ bool (infix ≗ⁿ)
    assumes ofe_refl: x ≗ⁿ x
        and ofe_sym: x ≗ⁿ y ↔ y ≗ⁿ x
        and ofe_trans: x ≗ⁿ y ⟹ y ≗ⁿ z ⟹ x ≗ⁿ z
        and ofe_mono: m ≤ n ⟹ x ≗ⁿ y ⟹ x ≗ᵐ y
```

Listing 2: OFE type class.

(OFE). This means that these resource types must be associated with a family of equivalence relations indexed by natural numbers. This family is easily expressed as a single step-indexed equivalence relation that is downwards monotone with regard to the index. A simplified formalization of this property can be found in listing 2. The basic idea of OFEs is to encode equivalence for computation steps up to an index. In particular, more computation steps can only show that two values are distinct and never that they were actually equivalent. This concept is similar to how pattern matching algorithms work. These algorithms are allowed to return a negative result whenever they found a difference in parts of the two input terms but must not succeed without having checked the full terms.

Apart from OFEs, Iris also has the concept of *Complete OFE*s (COFEs), which are equipped with a notion of limits with regard to step-indexed equivalences. Although the exact definition is not relevant for this work, the notion of COFEs is quite central to Iris in that they enable the definition of fixed-points for some functions on a COFE based on Banach's Fixed-Point theorem. This property is especially relevant for the uniform predicates introduced in section 4.2.

```
class camera = ofe +
    fixes valid :: 'a ⇒ nat ⇒ bool
    ...
    assumes camera_extend: valid a n ⟹ a ≝n (b1 · b2) ⟹
        ∃c1 c2. a = c1 · c2 ∧ c1 ≝n b1 ∧ c2 ≝n b2
    ...
```

Listing 3: Camera type class.

Whereas OFEs lift the notion of equivalence to a step-indexed semantics, resource algebras are generalized in a similar way to so called *cameras*[1]. For this reason, the camera structure extends the OFE structure and requires a step-index validity predicate, as well as other step-index related properties. A simplified version is shown in listing 3. Due to the need for step-indexing for some resource types, cameras and OFEs are more generally used classifications instead of resource algebras. The notion of resources that do not rely on step-indexing can easily be achieved by defining the OFE and camera instances such that they ignore the step index. This kind of instantiation is denoted as discrete OFEs or discrete cameras respectively. Other than discrete cameras, Iris also features the notion of cameras with a total core or with a common unit element with regard to composition. These variants allow for simpler reasoning about some camera properties and can sometimes be obtained by the combination of several cameras. For example, the `option` type constructor found in both HOL and Gallina lifts a given camera to a unital camera by adding the `None` constructor as the unit and lifting all operators through the `Some` constructor.

4.2 Uniform Predicates

Iris enables users to implicitly reason about resources through its propositions of type `iProp`. These propositions form the separation logic that underlies the Iris program logics and enable the user to abstractly reason about program resources. However, these propositions are internally defined as predicates over resources and the step-index. These predicates are required to be *uniform*, i.e. they need to be monotone with regard to composition and step-index and ignore invalid elements. The monotonicity requirement denotes that if a predicate holds for a fixed resource object a and an index n, then it also needs to hold for all smaller indices $m \leq n$ and extensions b to the resource object. For this notion, an extension b to a resource object a is a resource that is equivalent to the composition $a \cdot c$ of a and some other resource c.

[1] For the origin of this name, we refer the interested reader to the Iris Appendix [45] and the fundamental works about Iris [20, 21].

$$\lceil P \rceil := \lambda\_\ \_.\ P \qquad P \wedge Q := \lambda a\ n.\ P\ a\ n\ \wedge\ Q\ a\ n \qquad \forall x.\ P\ x := \lambda a\ n.\ \forall x.\ P\ x\ a\ n$$

$$\Box\, P := \lambda a\ n.\ P\ \texttt{core}(a)\ n \qquad \triangleright P := \lambda a\ n.\ n = 0 \vee P\ a\ (n-1)$$

$$P * Q := \lambda a\ n.\ \exists b_1\ b_2.\ a \overset{n}{=} b_1 \cdot b_2 \wedge P\ b_1\ n\ \wedge Q\ b_2\ n$$

$$P \mathrel{-\!\!*} Q := \lambda a\ n.\ \forall m\ b.\ m \leq n \longrightarrow \texttt{valid}\ (a \cdot b)\ m \longrightarrow P\ b\ m \longrightarrow Q\ (a \cdot b)\ m$$

Figure 4.1: Selected basic Iris operators defined as uniform predicates.

Based on this definition, uniform predicates form an OFE and COFE but not a camera. The aforementioned ability to define fixed-points for COFEs makes it possible to define recursive uniform predicates and is relevant for several fundamental usages of these. Although uniform predicates are already quite versatile, they are by definition only defined over a single camera resource. The actual proposition type of Iris `iProp` supports the use of arbitrary many different resource types. This property requires a few technicalities to provide a simple yet expressive user interface and gets examined in more detail in sections 5.3 and 5.4.

4.3 Iris Base Logic

Iris comes with a number of basic propositions that form the Iris base logic and are sufficient to define most more complex propositions. Some of these propositions are listed in fig. 4.1 with their definition in terms of uniform predicates. Of these, the *pure* predicate is the most basic one, as it embeds meta level propositions into the Iris logic. This proposition can be seen as a uniform predicate that does not depend on the resource and step arguments (cf. fig. 4.1) and is denoted by the syntax $\lceil P \rceil$ for a Coq/HOL proposition P. Other than the pure proposition, the Iris base logic also contains the classic logical operators conjunction ($\wedge$) and disjunction ($\vee$). The semantics of these operators can be seen as simple point-wise lifting of the corresponding meta-level operators to the level of uniform predicates. Similarly, the Iris base logic contains the universal ($\forall$) and existential quantifiers ($\exists$) that are allowed to range over arbitrary meta-level types. Their semantics is also point-wise lifted from the meta-level. It is also noteworthy that the quantifiers are allowed to range over Iris propositions and higher-order predicates, too.

In contrast to these lifted propositions, the separating conjunction and magic wand operators are defined in terms of camera functions as depicted in fig. 4.1. Iris encodes an affine separation logic and assigns the operator's semantics accordingly. The semantics of the separating conjunction operator can be seen as a direct translation of the semantics introduced in section 3.4, but with the camera composition instead of the heap addition. Likewise, the magic wand operator has the semantics of a resource extension by composition. Similarly to the heap version, the Iris magic wand only reasons about extensions that are valid with regard to the camera validity operator.

Apart from these standard logical operators, the Iris base logic also contains more specialized operators. Of these, the persistence ($\square$) and later ($\triangleright$) modalities are in general the most relevant operators. The general notion of a persistent assertion denotes that it holds independently from other assertions. This property implies that the proposition only takes the duplicable parts of resources into consideration. These duplicable parts remain present even under camera composition, which results in the assertion holding regardless of other facts in the context. Accordingly, the persistence operator lifts the embedded proposition to a persistent level and makes it ignore non-duplicable resource parts. It does so by reasoning about the core of a given camera object as depicted in fig. 4.1. This definition results in the correct notion, as the core is an identity element with regard to camera composition for both itself and the original object and is therefore duplicable, e.g. $\texttt{core}(x) = \texttt{core}(x) \cdot \texttt{core}(x)$. As a result of the reduction to duplicable resources, conjunction and separating conjunction become equivalent below the persistence modality.

By contrast, the later modality denotes that the wrapped proposition holds at the next step-index, i.e. at the next step of computation. This notion is encoded by reducing the step-index input of the according uniform predicate as depicted in fig. 4.1. The ability to delay propositions is central to the Iris logic, as it guarantees the logic's soundness against a paradox which occurs otherwise (cf. [20, 21, 45]).

Other than the operators and definitions introduced above, the *entailment* proposition is the most central concept to the Iris logic. The entailment $P \vdash Q$ of two Iris propositions P and Q can be intuitively seen as an implication on valid resources. To be more precise, it encodes that for all valid combinations of a resource object and a step-index for which P holds, Q must also hold. This also means that, if P holds for all valid resource and step combinations, than Q must also hold for all of these. Interestingly, in this case the proposition $P \mathrel{-\!\!*} Q$ also holds for all valid cases. In general, most proof goals and reasoning rules of the Iris logic are based on entailment and can be combined by derived rules such as entailment transitivity, also often called the cut rule.

4.4 Advanced Propositions

The Iris framework does not only contain the base logic but also many other language-agnostic propositions that can be used to formalize more complex program specifications. These propositions are mostly related to three of the core reasoning principles in Iris: *ghost state*, *impredicative invariants*, and *weakest preconditions*. Of these three, ghost state is the most fundamental, as it describes parts of an abstract program state without a direct representation in an execution of this program. An example of this could be an abstract state machine to model how a certain data structure can be used or access permissions that are only enforced at the language type level. For simplicity, every kind of resource can be seen as a piece of ghost state, even if it has an execution time correspondent such as the actual memory. To make direct reasoning about these kinds of resources possible, a proposition that can `own` a piece of ghost state is required. To this end, Iris introduces the `own` predicate, which takes a camera resource and a name and ensures that this resource is part of any satisfying global state. As such, `own` $\gamma\ x$ encodes the ownership of a given resource object x that is included in a bigger ghost state piece with name γ. This notion becomes apparent with the following rule `own` $\gamma\ x *$ `own` $\gamma\ y \dashv\vdash$ `own` $\gamma\ (x \cdot y)$, where $\dashv\vdash$ denotes entailment in both directions. In this context, a single `own` $\gamma\ x$ proposition is called a ghost location in the Iris literature.

In addition to ghost states, Iris comes with higher-order invariants to encode propositions that hold at all times and even in a concurrent setting. This notion can be understood best in the context of a concurrent program. There, an invariant describes a property that needs to hold after each program step if it held before. Similarly, Iris invariants can also be understood as containers owning resources that all threads can access at the same time. In this context, a thread can gain temporary full access to the invariant and even modify the underlying resources for the duration of an atomic step, as long as they are restored afterwards. Based on this understanding, predicates about the memory representation of locks or protocols for accessing critical sections (cf. [20, section I.3.4]) are common use cases for invariants. The fact that the encapsulated propositions in invariants can themselves be any Iris proposition such as `own` or even another invariant makes these to one of the most important features of Iris and allows the user to define arbitrarily nested propositions. The implications for the implementation of this feature are investigated in greater detail in section 5.4 and are, therefore, deferred to that section.

Based on higher-order ghost state, i.e. the combination of ghost locations and invariants, users can define many useful purely logical specifications for data structures and programs. Yet, the predicates and propositions defined thus far allow no direct assertion about program steps. For this purpose, Iris defines a weakest precondition predicate `WP`. It encodes the notion of a minimal required precondition `WP e {v. Q}`, such that after evaluating the expression `e` the postcondition Q holds for the resulting value `v`. The predicate requires the framework to be instantiated with a concrete programming language first but is otherwise defined only in terms of the Iris base logic. For the language-dependent part, the weakest precondition depends on the type of program expressions, the language's reducibility predicate and computation step relation, as well as an associated predicate about concrete program states. For simplicity, the concrete formalization of the weakest precondition predicate is omitted as it is not relevant for this work. Yet, it is relevant to know that the weakest precondition of an expression needs to be defined recursively. If the expression is already a value, the weakest precondition is just the same as the postcondition predicate applied to the value. If, however, the expression can still be evaluated or reduced further, e.g. for a call-by-value function application, then the weakest precondition needs to also contain the notion of all further steps. Due to this, the definition of the weakest precondition predicate is quite involved and relies on the aforementioned fixed-point construction for COFEs to be defined in a sound way.

The weakest precondition predicate is often found at the top level of program specifications and proofs and is therefore the main tool in using Iris for program verification. As such, an entailment from some hypotheses on the left-hand side to a single weakest precondition predicate on the right-hand side is a common proof goal. This kind of goal is also underlying Iris' hoare logic expressions. Nonetheless, a typical proof state for such specifications involves a simple weakest precondition on the right side of an entailment. This position is also relevant for reasoning about the goal, as some Iris rules can only be applied if the entailment's ride-hand side is of a specific form. For example, after doing one programs step, i.e. after reducing the expression of the weakest precondition predicate once, a single later modality can be lifted from the left-hand side of the entailment.

5 About porting Iris to Isabelle

Porting a program from one programming language to another is often a rather straightforward process. Most programming languages used today are turing complete and can therefore be reduced to a few core principles. Porting these is then often just a one-to-one translation of corresponding principles. Furthermore, most mainstream programming language also feature some sort of foreign function interface to enable direct inter-language communication. With this, it is easily possible to obtain a single program from a source code base with parts written in C, Rust and Haskell, for example.

Although this kind of interoperability is desirable for interactive theorem provers as well, it is not as easily achievable as with programming languages. Whereas programming languages have the common ground of machine instructions or the operating system's APIs, proof assistants need to be based on suitable logic systems to have a common ground for any direct interoperability[1]. For the same reason, porting developments from one logic to the other can be rather tricky if they rely on inherent features of the source system that are not or not easily available in the destination system. This problem also came up in our development of a partial port of the Iris framework to Isabelle/HOL and is discussed in the following sections in more detail.

5.1 About porting from Coq to Isabelle

Often, porting inductive definitions such as recursive functions and algebraic datatypes between Gallina and Isabelle/HOL can be trivial, as both languages have similar syntax and ways to define such logical objects. In addition, many propositions that can be expressed in any common logical system are of a similar form in HOL and Coq. This observation seems to be trivial as mathematical expressions should not depend on the system they're expressed in but can be surprising given the fact that both the weak type theory of Isabelle/Pure and HOL are quite different from the calculus of inductive constructions that Coq is based on.

Despite the syntactic similarities, there are also several commonly found differences that make porting definitions and lemmata more intricate. For one thing, Iris utilizes the finite `gset` and `gmap` types from the std++ library [12], as some uses cases in Iris

[1]Interoperability between proof systems is still a central topic in current research projects such as the EuroProofNet COST action `https://europroofnet.github.io/groups/`. As our work focuses only on porting Iris and not making the Coq formalization interoperable with our Isabelle implementation, we choose to discuss this topic no further.

require the involved sets and maps to be finite. Similarly, Isabelle/HOL has the finite set type `fset` and the finite map type `fmap` in its standard library, which can be used for these use cases. However, most automation and standard tools in Isabelle/HOL are designed to work with unrestricted sets and maps, i.e. these sets and maps can potentially be infinitely sized. The according `set` and `map` types are related to their finite counterparts but are necessarily distinct. To be more precise, the finite types are semantic subtypes of the unrestricted ones. Therefore, the handling of finite sets and maps differs significantly between Coq with std++ and Isabelle/HOL. Especially proofs about instances of the finite types require more boilerplate code, as reasoning steps for these types often come with additional prerequisites. Additionally, some proofs may require the user to relate the finite instances back to the unrestricted type to allow for the necessary proof steps. In these cases, the lifting/transfer package of Isabelle/HOL can be used to make reasoning more straightforward. This package introduces sophisticated mechanisms to lift definitions and proof states for types that are defined from subsets of other types, such as semantic subtypes.

The lifting/transfer package is also utilized to make handling of uniform predicates easier, as these can be defined as a semantic subtype of functions. In this case, the lifting/transfer package is used similarly to the sealing mechanism utilized in the Coq formalization of Iris. This mechanism is introduced in std++ and provides the user with fine-grained control other when the sealed definitions are unfolded by the system. This functionality allows the user to choose whether they want to reason on the semantic level of uniform predicates, i.e. how they handle camera objects, or only with already proven properties of propositions.

On another note, the embedding of meta level propositions into the Iris logic by the pure predicate can potentially also become problematic for translating usages of Iris to Isabelle/HOL. HOL has not the same facilities as the CIC and, even though this chapter presents several general translation techniques, not every valid Coq definition can be ported. Especially more intricate definitions that rely heavily on dependent types or the generic usage of type classes like monads can not be expressed easily in Isabelle/HOL and might even be impossible to translate correctly. Yet, this may be a negligible problem, as the pure operator is often only used for simple propositions that fall into the shared logical subset of HOL and CIC.

5.2 Algebraic Hierarchies

Many formal developments in computer science and especially mathematics classify logical objects in algebraic hierarchies. This classification makes it possible to abstract over different entities and reduces the amount of boilerplate code necessary to express the same properties for each new object. Iris' OFE, COFE, and cameras are a rather small example of such an algebraic hierarchy and their formalization motivates this section.

```
instantiation prod :: (camera, camera) camera begin
    definition valid_prod :: 'a×'b ⇒ nat ⇒ bool where
        valid_prod (x, y) n = valid x n ∧ valid y n
    definition core_prod :: 'a×'b ⇒ ('a×'b) option where
        core_prod (x, y) = case (core x, core y) of
            (Some x', Some y') ⇒ Some (x', y')
            | _ ⇒ None
    definition comp_prod :: 'a×'b ⇒ 'a×'b ⇒ 'a×'b where
        comp_prod (x, y) (a, b) = (x·a, y·b)
end
```

Listing 4: Camera type class instance for the product type `prod`.

Coq offers two idiomatic ways of constructing such hierarchies: type classes and canonical structures. Although both concepts are essentially just syntactic sugar and specialized machinery for dependent records, they have quite different strengths and weaknesses. The algebraic hierarchy of Iris is implemented by combining both concepts to overcome the respective shortcomings. Whereas the central categories such as OFE and cameras are formalized as canonical structures with the mixin pattern (cf. [29]), smaller and less central concepts such as the camera functions are mostly formalized as type classes. The utilization of canonical structures for the central concepts avoids the potentially exponential blowup in the size of proof terms that an equivalent unbundled type class formalization could lead to (cf. [19, 44]). Similarly, the usage of type classes for the other concepts reduces boilerplate code required for the instance search and makes debugging problems with instance resolution easier due to a custom debugging facility for this mechanism.

In contrast to Coq, Isabelle has no canonical structures and its type classes are based on locales, i.e. instead of records. Therefore, they are not record instances and thus values of the programming language, but special logical contexts that are bound to a single type. Despite this difference, both type classes and locales are the idiomatic way of constructing algebraic hierarchies in Isabelle/HOL, as they enable users to abstract over multiple types, terms and propositions. Therefore, formalizing the central concepts of Iris is straightforward in Isabelle/HOL as seen in listings 2 and 3. Instantiating the type classes of the algebraic hierarchy is similarly straightforward. For example, the product type $(\times)$ can be instantiated as a camera by lifting the camera operations of the two argument types as depicted in listing 4.

Although most instances can be translated equally directly, there still are a few pitfalls that make this translation a bit more verbose. As an example, Isabelle/HOL does only support defining type class instances or locale interpretations explicitly, but not via a parameterized constructor or something equivalent. In contrast, Iris contains several instance constructors, such as `discreteO` and `iso_cmra_mixin`, that can be used to equip types with a discrete OFE definition or define a camera instance for a new type by an isomorphism from an already defined camera. It would be possible to provide such facilities in Isabelle by generating the according type class instances via custom ML code. The necessary ML code is more verbose than the equivalent construction in Coq and obfuscates the concrete usages of possible proof goals that would, for example, be required to construct the camera definition via isomorphism. These trade-offs make a direct translation to Isabelle only feasible for very straightforward to use constructions and it is often just simpler to directly construct the class instance in the normal way. Other than that, Isabelle's type classes are also more limited than the ones in Coq in that they only support a single instance per type. Yet, this fact is actually an advantage as it makes type class instance resolution a trivial operation and prevents any form of exponential blowup due to the direct corelation of types and class instances.

5.3 Dynamic Composable Resources

The uniform predicates introduced above are generic over the user-provided resource camera. This works well for a single contained development but becomes cumbersome if one wants to combine two separate developments. If the related facts are defined over different cameras, they would need to be redefined to be composable. The Iris formalization in Coq overcomes this problem by generalizing all facts and definitions over a compound resource type. This compound type is then equipped with a type predicate to denote that it contains a specific camera. In this way, it is possible to make theorems generic in the exact resource type and still require certain cameras to be available.

The compound resource type is formalized as a product type of finite maps from ghost variable names to the actual camera objects (cf. [45, section 7.5]). These maps support an arbitrary number of instances of each camera type and are thus also necessary to achieve actual composability. Whereas these maps can be defined in terms of standard map types such as `gmap`, the product type requires a rather non-standard way to look up single camera types in it, i.e., as the domain type of a map. A list of camera maps with a lookup mechanism based on dependent types is the solution to this requirement utilized in the Coq formalization. The concrete implementation is not relevant for our work; nonetheless, it should be obvious that the dependent type based mechanisms can not be trivially translated to Isabelle/HOL. It should also be noted that the `own` predicate is implemented in terms of the camera map lookup mechanism, which makes this mechanism necessary to every feature-complete port of Iris.

For this reason, our port employs a workaround based on on-demand code generation and locales to emulate the dynamically composable resource type in Isabelle. To this end, our compound resource type is just the simple product type of partial maps ($\rightharpoonup$) from ghost names to camera instances. For constructing and accessing single ghost

```
1     type_synonym 'a cmra_map = ghost_name ⇀ 'a
2     type_synonym resource = cmra1 cmra_map × cmra2 cmra_map × ...
3         × cmraN cmra_map
4
5     definition get_cmra2 :: ghost_name ⇒ resource ⇒ cmra2 option where
6         get_cmra2 name res = (fst (snd res)) name
7     definition constr_cmra2 :: ghost_name ⇒ cmra2 ⇒ resource where
8         constr_cmra2 name x = (∅, [name ↦ x], ∅, ..., ∅)
9
10    locale inRes =
11        fixes get_cmra :: ghost_name ⇒ 'res ⇒ 'camera option
12        and put_cmra :: ghost_name ⇒ 'camera ⇒ 'res
13        assumes ...
14
15    interpretation cmra2In: inRes get_cmra2 constr_cmra2
```

Listing 5: The compound resource type and `inRes` locale.

values, specific getter and constructor methods of the form shown in listing 5, lines 5–8, are utilized. Whereas the getters just traverse the tuple and then look up the given name in the correct map, the constructors build a new resource object by introducing a single-valued map for the given name and camera object and setting the other tuple entries to the empty map. By doing so, OFE equivalence and validity of the resource object are the same as for the inserted camera object, which allows us to define the `own` predicate for any camera in a sound way based on the corresponding constructor function. It is trivial to see that the getter and constructor functions follow the same simple pattern and are only trivial boilerplate code. For this reason, we decided to generate them given a specific compound resource type. The code generation facility is implemented in Isabelle/ML and constructs the functions stepwise by simply iterating over the tuple type.

The getters and especially the constructors are then abstracted via the locale `inRes` defined in listing 5, lines 10–13. Apart from the functions, this locale also fixes a few relevant properties about them. Similarly to the functions themselves, the locale interpretations can also be generated via ML code. For this, it is possible to exploit the regular structure of our generated functions and provide an automated solver that uses a small number of common rules. Based on the locale, it is now also possible to define the generic `own` predicate to work with any resource type. Equipped with these definitions, users can then define lemmata that rely on the presence of a certain camera by assuming the respective `inRes` interpretation. This requires more boilerplate code than the Coq equivalent but is otherwise equally expressive.

```
Context `{!inRes Σ (excl unit)}.
    Definition locked (γ : ghost_name) : iProp Σ := own γ (Excl ()).
    ...
```

Listing 6: The formalization of the `locked` predicate in Coq.

```
context
    fixes get_lock :: ghost_name ⇒ 'res ⇒ unit excl option
      and constr_lock :: ghost_name ⇒ unit excl ⇒ 'res
    assumes lock_inG: inRes get_lock constr_lock
begin
    definition locked :: ghost_name ⇒ 'res upred where
        locked γ = own constr_lock γ (Excl ())
    ...
```

Listing 7: The formalization of the `locked` predicate in Isabelle/HOL.

The exact difference becomes apparent in the following example, which focuses on the modular definition of a simple lock primitive. Such a primitive requires a `locked` predicate that encodes that the current thread has exclusive access to the locked resources. A simple formalization of the `locked` assertion involves the `excl` camera wrapper around the unit type. This wrapper type has a single valid constructor `Excl`, which can not be composed with any other element and, thus, guarantees exclusive access. Based on this intuition, it is straightforward to define the `locked` predicate in both Coq and Isabelle/HOL as seen in listings 6 and 7. Whereas the Coq version requires only a single `inRes` assumption in the context, the Isabelle version is quite verbose as it requires the user to define both correctly typed getter and constructor functions per camera in addition to the `inRes` assumptions. Moreover, the two usages differ in that the Isabelle `inRes` assumption is actually a plain theorem that is obtained from instantiating the locale. In addition, the `own` predicate also requires the constructor explicitly. Similarly, one needs to use the named assumptions such as `lock_inG` for reasoning about the defined constants in this context. The reason for this behavior is that all theorems that are defined in the context of `inRes` require an instance fact to be applicable, i.e. a theorem with content P is then of the form `inRes ?get ?constr` $\Longrightarrow P$.

5.4 Impredicative Invariants

The Iris base logic is expressive enough to give specifications for many non-concurrent data structures and programs. Though, for concurrent data structures one mostly wants to have some semi-consistent state predicates that always hold for all threads and that might only be violated for the runtime of atomic operations. To satisfy this need, Iris provides the impredicative invariants introduced above. These invariants can be described completely in terms of combinators of the Iris base logic and are based on the so called *world satisfaction* proposition. This proposition essentially acts as a data base for all registered invariants. It keeps them in a map, indexed by the invariant's name and also stores whether the invariant is currently open, i.e. whether its contents can be used freely for the current duration of an atomic operation. Based on this world satisfaction proposition, one can then define *fancy updates*. This operator carries the notation of an update to the world satisfaction state, i.e. it encodes a change of the closed and opened invariant sets. This state change is encoded in so called *masks*, which are the sets of names of invariants available at that point, i.e. invariants that have been allocated before and are not open at this point. To be more precise, masks denote a minimal required set of available invariants. Based on this, the fancy update assertion ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} P$ holds iff a logical update of the state can be performed, such that at the end P holds and the mask $\mathcal{E}_1$ is changed to $\mathcal{E}_2$. The actual invariant proposition $\boxed{P}^{\mathcal{N}}$ for embedded proposition P and name $\mathcal{N}$ is then defined in terms of fancy updates (cf. [45, section 9.4]), although we omit the concrete definition for simplicity in this work.

In Iris the impredicativity of invariants is their most important property, i.e. they can contain arbitrary other Iris propositions such as ghost state ownership, other invariants or weakest preconditions. This fact allows the Iris logic to be as expressive as it is and enables the usage of complex program specifications.

Due to the fact that the world satisfaction proposition stores invariants, and thus Iris propositions, the type of Iris propositions needs to be defined recursively. Above, we introduced the simplified assumption that uniform predicates instantiated with some resource camera would be enough, but due to the propositions being part of this resource type, one actually gets a recursive domain equation for the Iris proposition type `iProp` of this form:

$$iProp := UPred(Res(iProp))$$

Here, `UPred` stands for the type constructor of uniform predicates applied to the free type constructor `Res` for a resource type that can rely on `iProp`. A solution to this equation is guaranteed by the America-Rutten theorem (cf. [3, 45]) and gets computed explicitly in the Iris Coq formalization. The main point of the America-Rutten theorem is the existence of a fixed-point solution for recursive domain equations with a specific kind of bifunctors from the category of COFEs to the category of cameras. This equation

```
type_synonym 'res::camera upred = 'res ⇒ nat ⇒ bool
definition upred_sep :: 'res upred ⇒ 'res upred ⇒ 'res upred where
    upred_sep P Q a n = ∃b1 b2. n_equiv n a (b1 · b2) ∧ P b1 n ∧ Q b2 n
```

Listing 8: A shallow embedding of the separating conjunction.

is then slightly different than our first approximation:

$$iProp := UPred(Res(iPropFP, iPropFP)), \text{ with } iPropFP \cong UPred(Res(iPropFP, iPropFP))$$

This construction can not be directly translated to Isabelle/HOL. The reason for this shortcoming is HOL's lack of higher kinded type constructors and thus a common classification of (bi-)functors. Yet, even without the general notion of a functor, it is possible to prove specific camera types to be the correct functors for the America-Rutten theorem and define the corresponding map functions in HOL. Similar to the class of functors, it is also not possible to express domain equations in terms of types. However, this limitation is not relevant for the case of *iPropFP*, as the equation there only requires a fixed-point up to type isomorphism, which can be expressed as an Isabelle/HOL locale. Lastly, Isabelle/HOL does not support reasoning about types in the way Coq does, which makes proving the America-Rutten theorem in HOL at least very difficult.

All of these facts make it seem that Iris' `iProp` type can not be formalized (easily) in Isabelle/HOL. Yet, we found that the extent of this limitation depends on whether one employs a shallow or deep embedding to formalize Iris in Isabelle. As described above, the Iris Coq formalization is based on a shallow embedding of the Iris base logic. This decision was made to obtain easier integration of Coq syntax and types into the logic [21]. Similarly, our Isabelle port of Iris is also based on a shallow embedding with the uniform predicates being step-indexed HOL predicates over the given camera. A simplified version of this encoding can be found in listing 8. It is quite important to note that the type for uniform predicates is generic in the camera argument type, which disqualifies the type to be a bounded natural functor. As introduced above, a type needs to be classified as a functor of this sort to be usable with Isabelle/HOL's datatype package. This package is the standard way of constructing non-trivial types and also one of the few ways to define types based on recursive domain equations like the `iProp` type. As a result, the aforementioned constructions seem to be the only practical way to prove the existence of the `iProp` fixed-point type in the case of a shallow embedding. For this reason, we decided to axiomatize the fixed-point type for our Iris port. For this, a new type is declared and used for the definition of `iProp`. The isomorphism between these two types is then introduced as an axiomatized locale instance.

In contrast, a deep embedding of the Iris logic could be defined without the need for an explicit fixed-point computation or axiomatization, yet induces other limitations.

The recursive `iProp` type can actually be encoded via the Isabelle/HOL datatype package in the case of a deep embedding and does not rely on the America-Rutten theorem any more. Yet, a deep embedding of the Iris logic also comes with its own problems, which are partially what motivated the shallow Coq formalization in the first place. Iris relies heavily on meta logic propositions to enrich its logic without the need to re-introduce all necessary logical objects. For this reason, the Iris logic contains embedded meta propositions and quantifiers over variables that are also allowed to be used in these meta propositions. Yet, the combination of these quantifiers with arbitrarily typed variables makes deeply embedding the Iris logic into Isabelle/HOL tricky. First of all, the algebraic datatype for the uniform predicates must contain all basic operators of the logic. Therefore, quantifiers must also be constructors of this datatype. For this, one could define constructors to take a function from the variable type to a proposition, which is a common formalization method for quantifiers. In this case, the quantifier constructors would need to take a function from an arbitrary type to uniform predicates, e.g. `Exist ('b ⇒ 'a upred)` where `'a` is the underlying camera type. If a formula requires more than one type of variables, it can not be encoded in this way due to the problems described in section 3.3. As an example, the proposition `Exist (λb::bool. ⌈b⌉)` can not be embedded as the `P` in `Exist (λx::(ghost_name ⇀ bool). P ∗ own γ x)`.

To overcome this problem, one can introduce an abstract variable format and require an explicit mapping from this format to actual terms for the semantics. The format for the variables can, e.g., be strings or natural numbers and can be combined with a De Bruijn index notation for bound variables. Based on this idea, it is possible to formalize quantifiers over arbitrary typed variables and use them throughout the formula. However, this approach also makes it necessary to explicitly set up a mapping from variables to a user defined semantic representation, e.g. a specific meta level term or function. Such a mapping needs to take the abstract variable representation as input and return the corresponding HOL term. However, these HOL terms can be of various types, which makes it impossible to encode such a mapping in HOL itself. In addition, the mapping would need to keep track of all introduced variables and other user defined mappings in a unified data structure, which, again, can not be expressed in HOL due to the combination of arbitrary types. Instead, it is necessary to implement both the mapping and the transformation from deep embedded formulae to their semantics in Isabelle/ML. Moreover, the embedded meta propositions used by the pure operator are allowed to contain variables that are bound by a quantifier. For this reason, the pure operator can not take HOL terms but must take terms encoded in a complete deep embedded version of the HOL language.

6 Automation

Interactive theorem provers have been in use for aiding humans with doing mathematical proofs for more than half a century now. They can both be used to verify user given proofs, support the construction of proofs, and even provide whole proof scripts on their own. This kind of proof automation reduces the amount of domain specific and proof engineering related knowledge needed to make larger developments feasible. As a result, proof assistants such as Isabelle and Coq have been successfully used for the formalization and verification of whole operating system kernels [22], optimized compilers [28] and many other developments. Despite these achievements and many successful fully automatic reasoning tools, even theorems that humans would deem trivial can sometimes not be proved automatically (e.g. due to the undecidability of certain logical systems). For handling these, proof assistants nowadays come with a large number of tools to build specialized automation.

In the following sections, four different levels of proof automation and abstraction are discussed. They are organized by the level of abstraction and usability they provide and evaluated with regard to how they are used in both the Iris formalization in Coq and our port. For this, we employ and reference a spin lock implementation in HeapLang throughout the chapter. This implementation can also be found in both the Iris and the Diaframe formalization and is also mentioned in related works [25, 32]. For brevity, the focus is put only on the `release` method shown in listing 9 together with its specification. In addition, the relevant predicates are defined in listing 10. The corresponding Isabelle/HOL code is in general equivalent and, therefore, omitted. The core idea of this example is a simple lock that guards some resources R by connecting the ownership of the ghost variable γ with a physical heap cell l containing a boolean flag. The boolean flag denotes whether the lock is held by a thread and is checked and set with atomic operations utilized in further functions not shown here. For this section, just the `release` method is enough, as it already requires the handling of invariants and ghost state in the combination with weakest preconditions, but involves no further complexity such as atomic memory access. Apart from the introduced predicates, the `locked` assertion introduced in section 5.3 is also reused.

The `release` function is defined as a HeapLang lambda term and takes a location variable `l` as an input, to which it stores ($\longleftarrow$) the value `false`. For the predicates, points-to facts `l`$\mapsto$`x`, which denote that the HeapLang heap contains a cell at location `l` with contents `x`, and the invariant implementation `inv` are introduced as auxiliary notions. In this context, the function call `inv N P` corresponds to $\boxed{P}^{\mathcal{N}}$.

```
Definition release : val := λ: "l", "l" ⟵ #false.
Lemma release_spec γ lk R :
    is_lock γ lk R ∗ locked γ ∗ R ⊢ WP (release lk) {v. ⌈v = #()⌉ ∗ True}
```

Listing 9: Release method of spin lock.

```
Definition lock_inv (γ : gname) (l : loc) (R : iProp Σ) : iProp Σ :=
    ∃ b : bool, l ↦ #b ∗ (⌈b = true⌉ ∨ ⌈b = false⌉ ∗ locked γ ∗ R).
Definition is_lock (γ : gname) (lk : val) (R : iProp Σ) : iProp Σ :=
    ∃ l : loc, ⌈lk = #l⌉ ∧ inv N (lock_inv γ l R).
```

Listing 10: The predicates used for the spin lock specifications.

6.1 Low level proof principles

Proofs in both Coq and Isabelle are essentially data structures in their respective implementation languages OCaml and Standard ML. Therefore, they can be constructed at the implementation level by meta functions over proof states. These functions are called tactics in both systems and, apart from their types, are just normal functions in the respective languages. This also means that they can perform arbitrary computations apart from changing the proof state. Such computations are commonly used for debug printing or storing and retrieving of data from a global storage. However, the actual proof state transformations follow strict rules and can't break the logic's soundness. The systems come with several tools to manipulate the proof state in standard ways. For Coq, proof state manipulation typically includes the construction of a proof program term. In contrast, Isabelle tactics mostly perform resolution with a rule to rewrite or substitute the goal term. Despite this difference in effect on the underlying data, many standard tactics in both systems can be used in similar ways.

In Isabelle, the `resolve_tac` tactic is one of the most fundamental proof principles, as it performs higher-order resolution of the goal with a set of given theorems. This allows the user to reason in a backwards way by refining the goal to a possibly easier to solve formula. Therefore, `resolve_tac` can be used in a comparable way to Coq's `apply` tactic.

The usefulness of these tactics becomes apparent with the following example:

$$\frac{\texttt{locked}\ \gamma\ *\ R \vdash \texttt{locked}\ \gamma\ *\ R}{\texttt{locked}\ \gamma\ *\ R\ *\ l \mapsto \texttt{\#false} \vdash \texttt{locked}\ \gamma\ *\ l \mapsto \texttt{\#false}\ *\ R}$$

This is a single step in the proof of `release_spec` and denotes that for proving the lower formula it suffices to prove the upper one with the points-to fact removed on both sides. The justification for this step are the rules Frame and Move2R depicted in eq. (6.1). The step can be performed via two applications of `resolve_tac`; first to switch the order of the right-hand side with Move2R, then to remove the points-to fact with Frame.

$$\frac{P \vdash Q}{P * R \vdash Q * R}\ \textsc{Frame} \qquad \frac{\Delta \vdash P * R * Q}{\Delta \vdash P * Q * R}\ \textsc{Move2R} \qquad \frac{}{P \vdash P}\ \textsc{Refl} \tag{6.1}$$

Even though this example only shows a very basic reasoning step, most sophisticated (automated) proof utilities are implemented based on such fundamental steps. For this, the tactics are combined via self-explanatory tacticals such as `THEN` or `REPEAT` in Isabelle. Coq provides similar functionalities, albeit with different names. In case a user needs to write custom tactics in OCaml, they need to develop a Coq plugin containing those, which can only be used after recompiling the system with the new plugin included. In contrast, Isabelle's ML tactics can also be defined at theory evaluation time and therefore be developed in parallel to the formalization. Isabelle/ML's integrated Standard ML parser and interpreter is the reason for this advantage. Despite the apparent disadvantage in comparison to Isabelle, Coq also has several powerful, yet easy-to-use facilities to define sophisticated proof automation methods, which are described in the next sections.

6.2 Higher level proof principles

Although low level tactics provide fine-grained control over the handling of proof goals, they are often too verbose and cumbersome to use. Especially the need to recompile Coq plugins when changing OCaml tactics slows down development. In addition, many tactics do not require the fine-grained control a low level tactic provides but can be expressed in higher level patterns such as proof term matching and control flow tacticals. For this reason, both Isabelle and Coq have been complemented with higher level tactic languages that require little to no low level code for defining useful proof procedures.

There are several higher level meta-programming languages for Coq tactics, yet the Iris formalization and specifically the Iris Proof Mode [25] make only[1] use of the Ltac tactic language [10, 18]. It contains constructions to match a Coq term against user provided patterns, apply tactics to proof states, and control the sequence of these tactics. In addition, Ltac provides facilities to also parse arguments such as hypothesis names in the same way as common builtin Coq tactics do. More importantly, the Ltac language contains primitives to operate on tactic values as well as Coq syntax items.

[1]The formalization also contains a few lines of Ltac2 code used to handle string encoding. Yet, we ignore these as Ltac2 is the dedicated, but still experimental, successor of Ltac.

The latter functionality allows Ltac users to define both pattern matching against and partial evaluation of Coq terms. Ltac also supports anonymous tactics as well as local definitions and can, thus, be used to define complex tactics with a straightforward structure. Moreover, Ltac comes with a number of small helper functions to check given terms for certain properties and guide the tactical control flow based on these. In summary, Ltac provides the Coq user with all necessary tools to build high level, flexible, and expressive tactics that can be used to guide automatic proof search or improve the interactive proof experience.

In contrast to Coq, Isabelle's low level tactics, i.e. ML level tactics, are easier to use and are, thus, more often utilized for highly specialized automated solvers. Despite this fact, Isabelle comes with the Isar framework [48, 50] that allows for high level reasoning in both Isabelle/Pure and object logics.

The Isar framework allows the user to write proofs in the Isar proof language in one of two styles[2]: either in the so called procedural style indicated by the `apply` keyword or in the structured style. The latter style supports explicitly named auxiliary lemmas and intermediate proof steps as well as high level reasoning principles such as locally fixing a variable or defining a term. It also features structured case distinction and induction proofs and deductive reasoning threads. Due to the explicit intermediate states and sequence of proof steps, this style offers a great similarity to classical mathematical proofs on paper and is often preferred over procedural style proofs for its clarity.

Procedural Isar proofs consist of proof scripts similarly to Coq, where single steps consist mostly of the apply keyword followed by an Isar method call. Proofs in this style are often used for prototyping automated methods and are generally discouraged for all but very short proofs by the Isabelle Community Conventions. Yet, procedural proofs are more commonly utilized in the context of program verification proofs. Apart from procedural poofs, the underlying Isar methods are also the basic building blocks for any reasoning in Isar. They abstract tactical reasoning from the raw goals to the Isar proof context and are, therefore, roughly equivalent to Ltac tactics. Isar methods can either be defined in terms of ML code or within the Eisbach method language [30]. The Eisbach language was directly inspired by the Ltac language and, as a result, features similar functionalities. Of these, the ability to pattern match proof goals, lemmata and arbitrary terms as well as define control flow of methods similarly to the control flow at the tactic level are the most relevant ones. In addition, Eisbach provides the user with simple tools to define method arguments that are either other methods, typed terms, (possibly named) theorems, or temporary additions to theorem databases.

Despite their similarities, Eisbach is by design strictly less powerful than Ltac. For example, Eisbach's pattern matching method is rather limited in comparison to the Ltac version. On top of that, Eisbach has no local definitions or term transformation procedures other than destructuring. These shortcomings are justified by Isabelle's

[2]It is also possible to mix the two styles. However, mixing both styles is discouraged by the Isabelle Community Conventions in all circumstances.

```
method framing =
  match conclusion in "_ ∗ P ⊢ _" for P ⇒ ‹moveR P, rule Frame, framing?›
  | _ ⇒ ‹rule Refl›
```

Listing 11: The `framing` method to automatically solve goals with framing.

inclusion mechanism for ML code, which allows the user to define the missing parts as Isar level commands that can be independent from Eisbach. For this reason, the language serves the purpose of providing a minimal framework and examples for how to construct one's own meta language based on it rather than being an equivalent meta-programming language to Ltac. Nonetheless, Eisbach methods facilitate strong automation even without ML extensions.

As an example, the aforementioned subgoal that occurs in the proof of `release_spec` can be proven completely by a simple, yet generic Eisbach method. To be more precise, this goal requires only a simple handling of frames to be proven:

$$\texttt{locked}\,\gamma * R * l \mapsto \texttt{\#false} \vdash \texttt{locked}\,\gamma * l \mapsto \texttt{\#false} * R$$

The method in listing 11 is able to prove this goal fully automatically. For this, it relies on another method `moveR`, which moves a given term to the head position in the right-hand side of the entailment if possible. The implementation of such a method can be done in several different ways but is not relevant for this example. The `framing` method matches the proof goal conclusion against a pattern with an entailment and a single fixed variable. This variable gets instantiated in the case of a match and can then be used as input to other methods. If the proof goal still contains a separating conjunction in its left side, its head will be searched for in the right-hand side and moved to the respective head position. After this step, the Frame rule can be applied with the `rule` method, which in this context is equivalent to the `resolve_tac` tactic. In a next step, the method `framing` is called again recursively to remove more frames if possible, i.e. the ? operator denotes allowed failure. In the other case, the method tries to solve the goal by reflexivity. In general, the Eisbach method is more generic than the tactic based approach above, as it can automatically find the right moving rules.

6.3 Basic proof automation methods

Both the Isabelle and Coq systems come with a number of included automated solvers that support standard formalization methods. Higher-order logic programming is one of these methods supported by both systems (indirectly). This kind of logic programming can be used to both classify and compute logical entities in both HOL and the CIC. To simulate logic programming in an interactive theorem prover, one can, for example, define clauses as predicates and theorems and use proof search mechanisms that can do unification, rule application and backtracking to evaluate the logic program in a procedural way. Although this approach works for both Coq and Isabelle, the idiomatic usage differs significantly in these systems.

It is most idiomatic to implement logic programming in Coq by the use of type classes[3]. This approach is heavily utilized by several large Coq developments including the Iris Proof Mode [25] and was first introduced by Spitters and van der Weegen [44]. Type class based logic programming is based on the idea of encoding predicates over values, types, and facts into a single type class and using the class instances as clauses. Coq's type class instance search algorithm can then compute the correct instance for given parameters. The `Frame` class is a simple example used to compute the remainder of a framing operation, i.e. a symmetric removal of equivalent terms on both sides of an entailment. This class is defined as

```
Class Frame P Q R := frame : R ∗ Q ⊢ P
```

and will, given a hypothesis `Q` and a conclusion `P`, compute the remaining conclusion `R` after a backwards-framing step, e.g. by eq. (6.2).

$$\textsc{FrameLP}\quad \frac{\texttt{Frame}\ P\ Q\ R \qquad S \vdash R}{S * Q \vdash P} \tag{6.2}$$

A simple instance can be defined like

```
Instance frame_sep_refl P Q : Frame (P ∗ Q) Q P .
```

As described before, type classes are not a native part of Coq's type system but rather a special way of using dependent records [43]. Similarly, the type class instance search method is based on Coq's `eauto` tactic and a custom fact database that stores instances. This tactic performs bounded proof search based on the aforementioned principles with the stored instances as applicable rules and will fail if no fitting instance was found. The underlying search algorithm can not only be adjusted with several parameters to fit different use cases but also exchanged for a user-provided custom instance search or computation method (cf. [18, chapter *Typeclasses*] or [40, chapter *UseAuto*]). In addition to these possible optimizations, type class based logic programming has another great advantage over more naive approaches: Type class constraints are resolved independently from rule application and unification. As a result, rules are not applied if the type class constraints can't be solved, which makes earlier backtracking possible and aids automation.

In contrast to Coq, type classes in Isabelle/HOL can not be used to encode logic programming. Instead, users can utilize (inductive) predicates, introduction rules

[3]It is also possible to either use plain (inductive) predicates, theorems, and a custom decision procedure or canonical structures to achieve logic programming in Coq. However, since the Iris formalization only makes use of type classes to this end, we ignore these alternatives for the scope of this work.

and Isabelle's Classical Reasoner package to achieve an equivalent form of logic programming. Although this is fundamentally the same approach underlying the type classes in Coq, the Isabelle variant requires more boilerplate code to set up custom logic programming databases (e.g. with named theorems) and add the corresponding attributes to the introduction rules. With this encoding, the aforementioned `Frame` predicate is equivalent to

```
definition frame where frame P Q R ≡ R ∗ Q ⊢ P
named_theorems frame_rule
lemma frame_refl [frame_rule]: frame (P ∗ Q) Q P
```

The Classical Reasoner package is a generic Isabelle/Pure library that contains several different (semi-)automated proof methods for solving goals in classical (first order) logic [50, chapter 9.4]. All of these are based on the fundamental idea of simulating sequent calculus style reasoning with natural deduction rules. These rules come in three different flavors: introduction, destruction, and elimination rules. Introduction rules are primarily meant for backward-chaining of facts, i.e. exchanging the current goal with the premise of the rule if the goal can be unified with the rule's conclusion. On the contrary, destruction and elimination rules are meant to deduce new facts from assumptions and are thus used for forward reasoning. Thereby, the difference between destruction and elimination rule is solely the syntactic form they are defined in. In addition to the differentiation of the rule type, each rule can be *safe* or *unsafe*. An unsafe rule can potentially lead to an unprovable goal and should therefore not be applied eagerly, whereas safe rules can not lead to such goals and are allowed to be applied eagerly. The important point here is that the classification needs to be done by the user, which means that they need to figure out whether a rule is safe to use.

Apart from the classical rules, some solvers also support simplification rules and enable combining several rewriting and deduction steps into a single method call. Due to this ability and the greater number in different search strategies, the Classical Reasoner package methods are in general stronger than equivalent Coq tactics. However, for the use case of logic programming both systems are essentially on par as this kind of reasoning requires no simplification and will often operate on only a small amount of possible rules.

Both the Coq and Isabelle versions of the frame predicate can be used together with the framing rule FrameLP in eq. (6.2) to finish the already known intermediate goal of the proof of `release_spec` from listing 9:

$$\texttt{locked}\,\gamma \ast R \ast l \mapsto \texttt{\#false} \vdash \texttt{locked}\,\gamma \ast l \mapsto \texttt{\#false} \ast R$$

If one wants apply the framing rule to this goal in a backwards fashion, they need to find a suitable $?R$ such that $\texttt{frame}\ (\texttt{locked}\,\gamma \ast l \mapsto \texttt{\#false} \ast R)\ (l \mapsto \texttt{\#false})\ ?R$ is a valid instance. It is easy to see that $?R$ needs to be $(\texttt{locked}\,\gamma \ast R)$. Both the type class based approach in Coq and the classical reasoning approach in Isabelle can find this solution without problems. The new goal is then

$$\texttt{locked}\ \gamma\ *\ R \vdash \texttt{locked}\ \gamma\ *\ R$$

which holds trivially due to the reflexivity of entailments in separation logic and can alternatively also be proved via framing. The framing approach via logic programming is especially better suited for automatic proofs, as it does not require the same moving of terms as the methods used above.

6.4 Sophisticated proof automation for Iris proofs

All aforementioned reasoning principles are quite helpful for interactive and mostly procedural proofs but in general not strong or ergonomic enough to automatically solve interesting goals, especially in the Iris base logic. Although it is not possible to have a full proof automation for such a complex and strong logic, there are still many single steps that can be automated.

6.4.1 Iris Proof Mode

The Iris Proof Mode (IPM) [25] combines many such steps into practical tactics to support users in interactive, high level proofs. To this end, the user only needs to combine the IPM tactics according to their own high level proof idea. IPM tactics are mostly implemented as Ltac tactics and contain reasoning steps based on several custom logic programming type classes like `Frame`. They are also meant to mimic existing Coq tactics in how they work and which arguments they take. For example, the `iApply` tactic corresponds to Coq's `apply` tactic but tries to apply rules that involve Iris entailments by using Iris level hypotheses. These hypotheses are organized in an own proof context that separates pure Coq premises, as well as persistent and spatial Iris hypotheses and allows referring to object logic hypotheses by a given name similarly to meta level premises in the normal proof mode. To this end, the IPM ensures that the correct rules for reasoning about Iris entailments are used and a normal form for the hypotheses is maintained.

It is insightful to explore how the `iDestruct` IPM tactic works on the hypothesis Hlock: `is_lock` γ *lk* R, which arises in the proof of the specification in listing 9. In this context, the exact tactic invocation used in the Iris formalization is `iDestruct "Hlock" as (l ->) "#Hinv"`. It first unfolds the definition of `is_lock` as described in listing 10. It then continues to introduce the existentially quantified variable into the environment as a fresh variable with name `l`. In a next step, the IPM tactic splits the conjunction, moves the pure equality into the context and uses it to rewrite all occurrences of `lk` to `l`. In the last step, the tactic moves the remaining invariant into the persistent hypothesis context and returns the new proof state back to the user.

6.4.2 Diaframe

In contrast to the IPM, the Diaframe library for Iris [32] aims not at improving interactive proofs but proving high level Iris goals fully automatically. Diaframe achieves this high level of automation by performing a proof search that is guided by the goal's logical connectives and utilizes a bi-abduction based hint system. Hints of this form can be used to compute both a frame and antiframe, i.e. remaining terms on the hypothesis and goal side, cf. $P * \mathit{antiframe} \vdash Q * \mathit{frame}$. Moreover, they can be seen as goal transformation rules that can be applied in certain situations and can also be provided by the user. Abstractly speaking, the proof search procedure consists of alternating symbolic execution of program steps under `WP` predicates and two additional intertwined reasoning steps. These steps consist of rewriting the goal until it reaches a normal form and searching for an applicable hint afterwards. These hints are then used to do another reasoning step, leading to another recursive application of the proof search procedure. The proof search steps are implemented as Ltac tactics, with type classes to guide the rewriting and hint search. By applying hints of the aforementioned bi-abduction form, a fixed hypothesis (P) and a fixed goal (Q) are consumed and in exchange the computed frame and antiframe terms are inserted into the goal.

Due to this twofold focus on one hypothesis and goal each, the hint search procedure is subdivided into two phases. In the first phase Diaframe tries to find hints primarily by matching against the hypothesis, whereas in the second phase it focuses more on the goal term. These phases are called left phase and right phase respectively [31]. The two phase heuristic makes it possible to treat certain atomic formulae such as invariants as nested connectives, too. More generally, the hint search iterates over all Iris level hypotheses and tries to find an applicable hint. It does so by first checking all hints in its database (i.e. all instances of the according type class) and, if there is no applicable hint, uses recursive hint rules to conduct a syntax-directed search with backtracking to find a fitting compound hint instead. This step contains the two phases described above. If there is no applicable hint, Diaframe backtracks to the next hypothesis and repeats the hint search for this term. If it can't find an applicable hint for any hypothesis, Diaframe tries to apply a last resort rule, i.e. a rule that might not rely on any hypothesis and should not be applied if any other rule could be applied instead. The introduction of fancy updates eq. (6.3) is an example of such a last resort rule.

$$P \vdash {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}} P \tag{6.3}$$

This rule can both be used to introduce a new fancy update on top of a hypothesis or drop the topmost fancy update on the goal. The latter can lead to the goal becoming unprovable, as many hypothesis transformations depend on the presence of certain modalities or wrappers on the goal side. For this reason, the rule is only used as a last resort to drop a fancy update on the goal. For similar reasons, Diaframe does only support a limited, but very common subset of Iris goals. For any other goals, it tries to solve as much as possible and returns the remaining proof obligations back to the user for them to continue in an interactive way. Despite these limitations, Diaframe employs a strong enough proof automation to automatically prove the spin lock example without any additional hints or the need to switch to interactive mode.

6.4.3 Automation in Isabelle

Both the IPM tactics and the Diaframe proof search procedure are implemented in terms of Ltac tactics and type class logic programming. For this reason, it is rather straightforward to port either of these to Isabelle/HOL based on the translations developed above. Despite the possibility of a direct port, we decided to use a more idiomatic way for providing similar functionalities in Isabelle. For example, we decided to not port the IPM proof context with its named hypotheses but keep the hypotheses in a single term. Whereas a name indexed list of premises mirrors the standard Coq proof mode and is, therefore, more idiomatic in Coq, Isabelle proofs do not contain such a list in general. Instead, the Iris rules are normalized to either work on the whole hypothesis term or the head term of a separating conjunction of several hypotheses. To apply a rule to a specific hypothesis, the system then needs to move the correct term to the head position by utilizing the fact that the separating conjunction is commutative and associative. It is even more versatile to move terms by splitting/framing rules based on the aforementioned logic programming.

Although the structured proof style of Isar would make the proof structure easier to understand, it does not work well for more involved Iris proofs. The reason for this is that many Iris rules rely on the structure of the goal term and can only be used to a limited degree in a more structured way. As a result, we find it more idiomatic to keep the procedural proof style of the IPM for interactive proofs and focus on using Isabelle's basic proof automation principles to make them as straightforward as possible.

It is noteworthy that the IPM tactics and our corresponding methods differ in the format for arguments. Whereas the Ltac tactics take pattern strings containing hypothesis names, the Isabelle counterparts take pattern terms instead. These pattern terms can contain schematic variables, i.e. free variables that Isabelle's higher-order unification procedure can instantiate, and are mostly used to match hypothesis terms. Patterns with more than one atom are encoded as separating conjunctions of all involved atoms. Due to Eisbach's matching facility, the single atom patterns can be accessed via term destructuring. Pattern terms are especially useful for moving regardless of the exact moving method. To this end, the pattern $\triangleright(?l \mapsto ?v)$ can be used to move the atom $\triangleright l \mapsto \#x$ to the head of the hypothesis term for further rule applications. Similarly, patterns can also be used to guide splitting and framing. Both splitting and framing with more than one atom to extract can be a difficult problem to solve via logic programming as rules must assume a certain order of atoms to not get stuck or loop forever. The goal `Frame` $(P * Q * S)$ $(S * P)$ $?R$ would be an example where this becomes problematic. Here, one requires a rule that relates the S in both the first and the second argument in a reasonable way. If the method instead matched the head of the first argument, i.e. S, against the pattern $S * P$, it can find that it can move this atom to the second argument. Normally, framing and splitting goals can be thought of as computing a third argument to a logic programming predicate. In this case, they rather divide the atoms in the first argument — the input term — into the two other arguments based on the given pattern. This can be done easily with Eisbach's term matching and is, therefore, an important tool for building sophisticated (semi-)automation methods in Isabelle.

On another note, the aspect whether the patterns include wrappers such as modalities, is also relevant. This mostly makes a difference for splitting goals but not for framing goals, as framing makes it possible to move terms out of certain wrappers. In the context of splitting goals, we classify pattern terms according to this property into two classes: *inner* and *outer* patterns. Whereas inner patterns contain only the required atoms, outer patterns also contain all wrappers that should be on top of the atom. The former can potentially match too many atoms, whereas the latter requires explicit handling of all wrappers via pattern matching on both the pattern term and the input term. As an example, consider the input term $P * \triangleright(Q * P)$ where one would like to extract the term $\triangleright P$. In this case, simply P would be the correct inner pattern and matches both occurrences of this atom. On the other hand, $\triangleright P$ would be the outer pattern leading to exactly the term one wanted to extract. Despite this potential problem, we found that in most cases the inner pattern works reasonably well.

In general, our approach allows us to develop several useful semi-automated proof methods on par with corresponding IPM tactics. Most of these methods are written exclusively in Eisbach; only a few ML methods are necessary to overcome certain Eisbach limitations. An exemplary proof of the `release` specification with these methods is depicted in listing 12. This proof is only slightly simplified, but can otherwise be found in our development [42, SpinLock.thy]. The general idea of the proof is to use the `wp_store` rule in line 14, which describes the semantics of a store in HeapLang. This rule requires a points-to fact about the location to which a new value gets stored as a hypothesis. The proof starts with unfolding relevant terms and beta-expanding the lambda abstraction in the definition of `release`. For this, the `iDestruct` method is used to both execute the beta-expansion and destruct the parameter `lk` into the underlying location value. In the next step, the invariant inside `is_lock` is accessed to extract the required points-to fact in line 7. Here, the `iMod` method applies the `inv_acc` rule to access the invariant and removes modalities from the extracted hypotheses. Due to the definition of `inv`, the extracted proposition is embedded in a later modality. This modality then needs to be handled explicitly to be able to access the points-to fact inside `lock_inv`. For this reason, `move_sepL` moves the correct term to the head position, where the points-to fact is extracted and the later modality is stripped with `later_elim`. Finally, the `wp_store` rule is applied

```
1   lemma release_spec:
2       is_lock γ lk R ∗ locked γ ∗ R ⊢ WP (release lk) { λv. ⌈v = ()⌉ ∗ True }
3   (*Unfold release definition*)
4   apply (simp add: release_def is_lock_def ...)
5   apply (iDestruct rule: wp_pure[OF pure_exec_beta])
6   (*Open invariant*)
7   apply (iMod rule: inv_acc[OF subset_UNIV])
8   (*Apply wp_store*)
9   apply (move_sepL ▷?P)
10  apply iExistsL
11  apply (iApply rule: upred_entail_eqL[OF upred_later_sep])
12  apply (move_sepL ▷(?l ↦?v))
13  apply later_elim
14  apply (iWP rule: wp_store)
15  (*Cleanup*)
16  apply (entails_substR rule: upred_laterI)
17  apply (entails_substR rule: wp_value)
18  apply (iApply_wand_as_rule (∃x. ?P x) (?l ↦?v ∗ own constr_lock ?n ?x ∗ R))
19  apply (iExistsR False)
20  apply (entails_substR rule: upred_laterI)
21  by frame_single+
```

Listing 12: Proof of `release_spec` with IPM-inspired Eisbach methods.

with the `iWP` method, which handles additional steps necessary when reasoning about program steps with a weakest precondition. In the last steps of the proof, the remaining goal terms are cleaned up by stripping them of modalities, closing the invariant, and framing them. As an intermediate step, the invariant is closed implicitly in line 18 by exploiting the corresponding magic wand hypothesis obtained from opening it earlier. The `iApply_wand_as_rule` method utilized for this step is a specialized version of `iApply` and searches for a magic wand that has the first argument (i.e. $\exists x.\ ?P\ x$) as its left-hand side. It then opens a new subgoal in which the user must prove that this left-hand side proposition follows from the hypotheses that match the second argument to the method; the points-to fact, the ghost variable of the lock, and the locked proposition in this case. This new subgoal can then be proven in parallel to the remainder of the original goal by first instantiating an existential quantifier and then dropping a last later modality before finishing the proof with framing.

Similarly to the described Eisbach methods for interactive proofs, we also built an experimental, fully automated proof method based on the general ideas of Diaframe's proof search procedure. For this, the procedure iterates over the hypotheses by destructuring the separating conjunctions similarly to Diaframe's hint search. For each hypothesis, it then calls the `auto` method from the Classical Reasoner package with a specific selection of simplification and introduction rules. These rules cover all basic rewriting steps and also most standard Diaframe hints and can be extended via named theorems. Unfortunately, this approach is not strong enough to do all relevant proof steps, because some of these require a high proof search depth and the correct order of rule applications. In addition, the binary classification into safe and unsafe does not suffice to provide the proof search mechanism with enough information about when to use which rule. Some steps require a certain sequence of rules that can not be encoded simply by these standard attributes. In this case, simple, yet specialized Eisbach methods are required. For example, framing and moving as well as the application of last resort rules require this approach. Handling these cases can lead to unprovable goals and, therefore, introduce unnecessary overhead to an exhaustive proof search via the Classical Reasoner package. Nonetheless, the solver is able to prove the specification of `release` fully automated while following the same proof ideas as described in listing 12. The corresponding lemma can be found in our development [42, SpinLock.thy] as well.

In conclusion, our experiment shows that it is possible to have sophisticated full automation for Iris proofs in Isabelle. Some steps performed by explicit Ltacs in Diaframe can even be performed by Isabelle's inherent proof automation mechanisms. Yet, our experiment did not include a thorough handling of rule application to nested arguments. Although Diaframe's recursive hint search could be adopted to solve this problem, we did not implement this or any other solution for our experimental solver. However, we came up with an alternative solution approach that was not implemented due to time constraints. This approach would adapt reasoning rules on the fly to the current goal by leveraging a similar recursive distinction as Diaframe but via ML code.

7 Discussion and Conclusion

In this thesis, we present our partial port of the Iris framework to Isabelle/HOL. We also investigate several intricate details of our port in chapter 5 and show how to translate the underlying Coq features into HOL. To summarize, Isabelle's type classes and locales in combination with custom on-demand code generation from Isabelle/ML suffice to port most of the Iris functionalities. As our port contains all relevant parts to fully formalize and prove a few selected examples, there are no generally necessary parts left that can not be ported to Isabelle. Despite this, one limitation remains with regard to pure propositions. The encapsulated meta assertions can in general contain arbitrarily complex expressions that might also make use of Coq functionalities that can in general not be translated to Isabelle/HOL. In practice, this limitation is quite unlikely to ever be relevant as most interesting program logic semantics can already be expressed within the Iris base logic and its standard extensions such as invariants. Even though this caveat is unlikely to ever be of relevance, the described problems in correctly translating the construction of the `iProp` type limit the actual usefulness of a full port of Iris.

7.1 Discussion of a full Port

An optimal full port of the Iris framework would consist of a definitional extension to HOL, i.e. an extension that can be defined completely inside the HOL logic and type system, with only very little overhead and a convenient user experience. Our approach to a shallow embedding of the Iris logic can not be used for an definitional but only for an axiomatic extension based on our results. This fact limits the usability of our port to developments that can afford to use this extended version of HOL. On the other hand, more generic libraries that must not assume such an extension can not make full use of this Iris port. Even if the axiomatic extension is acceptable for some developments, it is not yet known whether the axiomatization is sound. A proof of this property would need to be developed outside of HOL and was not realized in the scope of this work. Yet, the proof of the America-Rutten theorem and the associated fixed-point construction in Coq are a strong indication that the axiomatization is justified. Moreover, it is quite common to define types in Isabelle/HOL axiomaticly and integrate them by providing some kind of isomorphism to other types or sets. In general, axiomatic extensions to HOL are not uncommon, although definitional extensions are usually preferred if possible. As a result, our port can provide users with the same functionalities as the Coq formalization, but requires them to accept our axiomatization based on the mentioned justifications.

Similarly, a deep embedded port faces caveats with regard to overhead and convenience, even though it solves the problem of the `iProp` construction. Due to the aforementioned reasons, any deep embedding of the Iris base logic can not be translated to its corresponding semantics without the use of ML-based code generation. Although the generated semantics can be used to prove the same kinds of formulae as with the shallow embedded approach, it is not possible to reason about the translation process from syntax to semantics. This also implies that one can not back-relate semantic properties with the syntactic logical operators and, therefore, lack the ability to fully utilize semantic based rewriting steps used in the context of proof automation. In addition to the overhead of registering variable mappings and deep embedding HOL terms for the pure predicate, this inability makes the deep embedding approach less suited for a full port of Iris.

We conclude that a full port of the Iris framework is technically possible, yet from a proof engineering perspective not (yet) favorable. Further work may improve on this situation, but our results show a theoretical bound on how useful a full port really would be. Whereas the shallow embedding we employed for our partial port relies on the axiomatization but allows for the same level of convenience as the Coq version, the explored deep embeddings do work without the need for an axiomatization yet suffers from limitations in accessibility.

7.2 Discussion of Automation

In addition to the question whether a full port of Iris is possible and feasible, this work also explores how well Iris proofs can be automated in Isabelle/HOL in comparison to Coq. We especially tried to find sufficient evidence to the widely believed superiority of proof automation in Isabelle. Yet, our results show that the Coq formalization of Iris already allows for very strong proof automation and any port to Isabelle can not improve on it significantly. At the same time, we also developed methods to relate the implementation of abstract reasoning principles in the Iris logic in both systems. In this context, it became apparent that both the Ltac and the Eisbach languages are highly useful for constructing sophisticated, idiomatic proof methods. To this end, the Ltac language provides the high-level capabilities necessary to reason about dependently typed proof goals without the overhead of writing a Coq plugin. Similarly, the Eisbach language provides abstraction over common proof guidance patterns and can easily be extended by more idiomatic, fully automated proof methods written in Isabelle/ML. In conclusion, both languages can be used to build equivalent proof methods, at least in the context of the Iris logic.

Whereas the similarities between Ltac and Eisbach hold for all kinds of developments, the usage of logic programming and related ideas to guide Iris proofs is more likely to be only applicable in the constrained context of an embedded logic such as Iris. For this kind of context, we can find no decisive indication that either of Coq or Isabelle enable significantly better proof automation mechanisms. We trace this result back to the fact that Iris does mostly rely on Coq specific features such as dependent types for the sake of convenience only. None of the basic logical connectives or algebraic structures relies ony any such features, resulting in our straightforward port and a similarly straightforward translation of proof principles.

However, we have found one aspect of proof automation in which Isabelle has a slight advantage over Coq. To be more specific, Isabelle comes with great mechanisms to support the definition and handling of semantic subtypes and the Sledgehammer tool for automatic proof search with the help of external tools. Both topics are not in general Isabelle-specific but have Coq counterparts, which are just less idiomatic and thus also less sophisticated. For example, the definition of uniform predicates as shallow embedded semantic subtypes of HOL predicates can utilize the full power of the lifting/transfer package, which makes defining new predicates or proving general reasoning rules quite straightforward. The similar Coq procedure of sealing and unsealing the uniform predicate definitions is simply less well integrated with general proof automation and makes breaking the abstraction less idiomatic than in Isabelle. Many fundamental proof rules for the Iris base logic can be directly derived from transfering the proof goal to the semantic level and applying one of Isabelle's standard proof automation methods. In addition, more involved rules can often be solved by either combining already defined rules or doing an transfer step and utilizing related rules to reason at the semantic level.

In these cases, Isabelle's Sledgehammer tool can often find all necessary theorems completely automatic and, thus, reduce the amount of work that the user has to do. Although the CoqHammer tool can be used in similar ways, it is deemed less idiomatic than its Isabelle equivalent and is not as well integrated into the normal workflow of a Coq developer. In general, both tools apply similar techniques for goal and proof translation as well as the premise selection. Yet, it is not clear, whether they can be compared in a neutral setting as they operate on quite different logics and existing literature has not applied them to the same problems. Recent work by Desharnais et al. [11] about Sledgehammer shows that it can solve around 70% of randomly chosen goals from the Archive of Formal Proofs, whereas related work by Czajka et al. [8, 9] presents that CoqHammer can solve around 40% of the goals in Coq's standard library. These results are obtained in contexts that are quite different from the Iris logic. Therefore, it is possible that both tools perform comparably well in the Iris context due to it mostly lying in a shared logical fragment. In practice, however, automatic proof search works better in Isabelle due to its better integration into the normal proof workflow. It should also be noted that both hammers can be fine tuned to fit a specific logical fragment and might perform quite well for Iris' logic and program logics instantiated in the framework if done so.

To conclude, our work has not found any significant difference in the proof automation capabilities for the Iris logic apart from a few technicalities. On the contrary, we were able to confirm that both systems allow for quite sophisticated proof automation techniques, which can be used to ease the use with and the development of the Iris framework and its port.

7.3 Future Work

In this thesis, we conclude that a full port of Iris to Isabelle is possible, albeit not optimal. Yet, we see several possible optimizations that could result in a truly useful full port of Iris. First of all, it needs to be investigated further whether a shallow or a deep embedding of the logic works better. For this, it can be expected that the America-Rutten theorem is provable in Isabelle/HOLCF, which is a definitional embedding of domain theory in HOL and supports reasoning about functors, inter alia. Such a proof would already increase the confidence in our axiomatic extension to HOL. Similarly, proving our axiomatization to be sound would justify the chosen approach. Moreover, it should be investigated whether the actual fixed-point construction for `iProp` can be done in Isabelle/HOLCF and whether it could be used in the context of the otherwise normal HOL types, as well.

For a deep embedding, the Nominal2 package by Urban et al. [47] might also be worth investigating, as it can improve the general handling of bound variables, e.g. in quantifiers. This might not solve the problem of the semantics of these binders but would be a stronger argument for the deep embedding. In general, our exploration of how well a deep embedding could be used for full Iris proofs was not exhaustive and introduces the need for further investigation.

We also expect that further work on automation in the port can be fruitful. In this context, we suggest to further investigate how the strongly context-dependent and verbose reasoning in the Iris logic could be integrated with structured Isar proofs with fully automated intermediate steps. An integration with the existing Refinement framework is another possible research direction from which our port might profit. In addition with the already strong automation of this library, this could result in a strong and very flexible framework for developing fully verified algorithms from a purely functional high-level specification. The Refinement framework could then transform the high-level specification into a lower level, equivalent imperative program, for which the Iris logic would provide the user with the ability to reason in more detail than with the current model.

Bibliography

[1] H. Barendregt. "Introduction to generalized type systems." In: *Journal of Functional Programming* 1.2 (1991), pp. 125–154. DOI: 10.1017/S0956796800020025.

[2] J. Berdine, C. Calcagno, and P. W. O'Hearn. "A Decidable Fragment of Separation Logic." In: *Foundations of Software Technology and Theoretical Computer Science*. Ed. by K. Lodaya and M. Mahajan. Vol. 3328. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 97–109. DOI: 10.1007/978-3-540-30538-5_9.

[3] L. Birkedal, K. Støvring, and J. Thamsborg. "The category-theoretic solution of recursive metric-space equations." In: *Theoretical Computer Science* 411.47 (2010), pp. 4102–4122. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2010.07.010.

[4] R. Chen, C. Cohen, J.-J. Lévy, S. Merz, and L. Théry. "Formal Proofs of Tarjan's Strongly Connected Components Algorithm in Why3, Coq and Isabelle." In: *10th International Conference on Interactive Theorem Proving (ITP 2019)*. Ed. by J. Harrison, J. O'Leary, and A. Tolmach. Vol. 141. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 13:1–13:19. ISBN: 978-3-95977-122-1. DOI: 10.4230/LIPIcs.ITP.2019.13.

[5] A. Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013. ISBN: 9780262317870. DOI: 10.7551/mitpress/9153.001.0001.

[6] A. Church. "A formulation of the simple theory of types." In: *A Formulation of the Simple Theory of Types* 5.2 (1940), pp. 56–68. ISSN: 0022-4812. DOI: 10.2307/2266170.

[7] T. Coquand and G. Huet. *The calculus of constructions*. Tech. rep. RR-0530. INRIA, 1986.

[8] Ł. Czajka. "Practical Proof Search for Coq by Type Inhabitation." In: *Automated Reasoning*. Ed. by N. Peltier and V. Sofronie-Stokkermans. Cham: Springer International Publishing, 2020, pp. 28–57. ISBN: 978-3-030-51054-1.

[9] Ł. Czajka and C. Kaliszyk. "Hammer for Coq: Automation for Dependent Type Theory." In: *Journal of automated reasoning* 61.1 (2018), pp. 423–453. DOI: 10.1007/s10817-018-9458-4.

[10] D. Delahaye. “A Tactic Language for the System Coq.” In: *Logic for Programming and Automated Reasoning*. Ed. by M. Parigot and A. Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 85–95. ISBN: 978-3-540-44404-6.

[11] M. Desharnais, P. Vukmirović, J. Blanchette, and M. Wenzel. “Seventeen Provers under the Hammer.” Accepted at *13th Conference on Interactive Theorem Proving (ITP 2022)*. 2022.

[12] std++ developers and contributors. *Coq-std++*. URL: `https://gitlab.mpi-sws.org/iris/stdpp/-/tree/master`.

[13] T. Dinsdale-Young, P. da Rocha Pinto, K. J. Andersen, and L. Birkedal. “Caper: Automatic Verification for Fine-grained Concurrency.” In: *Proceedings of the 26th European Symposium on Programming (ESOP'17)*. 2017, pp. 420–447. DOI: `10.1007/978-3-662-54434-1_16`.

[14] Google Project Zero. *The More You Know, The More You Know You Don't Know – A Year in Review of 0-days Used In-the-Wild in 2021*. URL: `https://googleprojectzero.blogspot.com/2022/04/the-more-you-know-more-you-know-you.html`.

[15] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. USA: Cambridge University Press, 1993. ISBN: 0521441897.

[16] M. J. C. Gordon. *Edinburgh LCF*. Lecture Notes in Computer Science. Berlin: Springer, 1979. ISBN: 3540097244.

[17] Z. Hou, D. Sanan, A. Tiu, R. Gore, and R. Clouston. “Separata: Isabelle tactics for Separation Algebra.” In: *Archive of Formal Proofs* (2016). `https://isa-afp.org/entries/Separata.html`, Formal proof development. ISSN: 2150-914x.

[18] Inria, CNRS, and contributors. *Coq Reference Manual*. URL: `https://coq.inria.fr/distrib/current/refman/`.

[19] R. Jung. *Exponential blowup when using unbundled typeclasses to model algebraic hierarchies*. 2019. URL: `https://www.ralfj.de/blog/2019/05/15/typeclasses-exponential-blowup.html`.

[20] R. Jung. “Understanding and evolving the Rust programming language.” PhD thesis. Universität des Saarlandes, 2020. DOI: `10.22028/D291-31946`.

[21] 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.” In: *Journal of Functional Programming* 28 (2018), e20. DOI: `10.1017/S0956796818000151`.

[22] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. "SeL4: Formal Verification of an OS Kernel." In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 207–220. ISBN: 9781605587523. DOI: `10.1145/1629575.1629596`.

[23] G. Klein, R. Kolanski, and A. Boyton. "Separation Algebra." In: *Archive of Formal Proofs* (2012). `https://isa-afp.org/entries/Separation_Algebra.html`, Formal proof development. ISSN: 2150-914x.

[24] R. Krebbers, J.-H. Jourdan, R. Jung, J. Tassarotti, J.-O. Kaiser, A. Timany, A. Charguéraud, and D. Dreyer. "MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic." In: *Proc. ACM Program. Lang.* 2.ICFP (2018). DOI: `10.1145/3236772`.

[25] R. Krebbers, A. Timany, and L. Birkedal. "Interactive Proofs in Higher-Order Concurrent Separation Logic." In: *SIGPLAN Not.* 52.1 (2017), pp. 205–217. ISSN: 0362-1340. DOI: `10.1145/3093333.3009855`.

[26] P. Lammich. "Refinement to Imperative HOL." In: *Journal of Automated Reasoning* 62.4 (2019), pp. 481–503. ISSN: 1573-0670. DOI: `10.1007/s10817-017-9437-1`.

[27] P. Lammich and R. Meis. "A Separation Logic Framework for Imperative HOL." In: *Archive of Formal Proofs* (2012). `https://isa-afp.org/entries/Separation_Logic_Imperative_HOL.html`, Formal proof development. ISSN: 2150-914x.

[28] X. Leroy. "A formally verified compiler back-end." In: *Journal of Automated Reasoning* 43.4 (2009), pp. 363–446.

[29] A. Mahboubi and E. Tassi. "Canonical Structures for the Working Coq User." In: *Interactive Theorem Proving*. Ed. by D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, S. Blazy, C. Paulin-Mohring, and D. Pichardie. Vol. 7998. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 19–34. ISBN: 978-3-642-39633-5. DOI: `10.1007/978-3-642-39634-2{\textunderscore}5`.

[30] D. Matichuk. "Automation for Proof Engineering: Machine-Checked Proofs At Scale." PhD thesis. Sydney, Australia: UNSW, 2018.

[31] I. Mulder, R. Krebbers, and H. Geuvers. *Appendix of 'Diaframe: Automated Verification of Fine-Grained Concurrent Programs in Iris'*. 2022. DOI: `10.5281/zenodo.6330596`.

[32] I. Mulder, R. Krebbers, and H. Geuvers. "Diaframe: Automated Verification of Fine-Grained Concurrent Programs in Iris." In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI '22. New York, NY, USA: Association for Computing Machinery, 2022. DOI: `10.1145/3519939.3523432`.

[33] M. S. Nawaz, M. Malik, Y. Li, M. Sun, and M. I. U. Lali. *A Survey on Theorem Provers in Formal Methods*. 2019. DOI: `10.48550/ARXIV.1912.03028`.

[34] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. `https://isabelle.in.tum.de/doc/tutorial.pdf`, Updated Version used as Isabelle/HOL Tutorial. Berlin: Springer, 2002.

[35] P. W. O'Hearn and D. J. Pym. "The Logic of Bunched Implications." In: *Bulletin of Symbolic Logic* 5.2 (1999), pp. 215–244. DOI: `10.2307/421090`.

[36] M. Parkinson. "The Next 700 Separation Logics." In: *Verified Software: Theories, Tools, Experiments*. Ed. by G. T. Leavens, P. O'Hearn, and S. K. Rajamani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 169–182.

[37] L. C. Paulson. *Isabelle: A Generic Theorem Prover*. Lecture Notes in Computer Science. Berlin: Springer, 1994. ISBN: 3540582444.

[38] L. C. Paulson. "Natural deduction as higher-order resolution." In: *The Journal of Logic Programming* 3.3 (1986), pp. 237–258. ISSN: 07431066. DOI: `10.1016/0743-1066(86)90015-4`.

[39] L. C. Paulson. *Sledgehammer: some history, some tips*. URL: `https://lawrencecpaulson.github.io/2022/04/13/Sledgehammer.html`.

[40] B. C. Pierce, A. Azevedo de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, and B. Yorgey. *Programming Language Foundations*. URL: `https://softwarefoundations.cis.upenn.edu/plf-current/index.html`.

[41] J. C. Reynolds. "Separation logic: a logic for shared mutable data structures." In: *17th annual IEEE symposium on logic in computer science*. IEEE Comput. Soc, 22-25 July 2002, pp. 55–74. ISBN: 0-7695-1483-9. DOI: `10.1109/LICS.2002.1029817`.

[42] F. Sextl. *Partial Iris port to Isabelle/HOL*. URL: `https://github.com/firefighterduck/isariris`.

[43] M. Sozeau and O. Nicolas. "First-Class Type Classes." In: *Lecture Notes in Computer Science* (2008). DOI: `10.1007/978-3-540-71067-7\_23`.

[44] B. Spitters and E. van der Weegen. "Type classes for mathematics in type theory." In: *Mathematical Structures in Computer Science* 21.4 (2011), pp. 795–825. DOI: `10.1017/S0960129511000119`.

[45] The Iris authors. *The Iris 3.4 Documentation*. URL: `https://plv.mpi-sws.org/iris/appendix-3.4.pdf`.

[46] D. Traytel, A. Popescu, and J. Blanchette. "Foundational, Compositional (Co)datatypes for Higher-Order Logic: Category Theory Applied to Theorem Proving." In: 2012, pp. 596–605. ISBN: 978-1-4673-2263-8. DOI: `10.1109/LICS.2012.75`.

[47] C. Urban, S. Berghofer, and C. Kaliszyk. "Nominal 2." In: *Archive of Formal Proofs* (2013). `https://isa-afp.org/entries/Nominal2.html`, Formal proof development. ISSN: 2150-914x.

[48] M. Wenzel. "Isabelle/Isar — a versatile environment for human-readable formal proof documents." Dissertation. München: Technische Universität München, 2002.

[49] M. Wenzel. *The Isabelle/Isar Implementation*. URL: `https://isabelle.in.tum.de/dist/Isabelle2021-1/doc/implementation.pdf`.

[50] M. Wenzel. *The Isabelle/Isar Reference Manual*. URL: `https://isabelle.in.tum.de/doc/isar-ref.pdf`.

[51] M. Wildmoser and T. Nipkow. "Certifying Machine Code Safety: Shallow Versus Deep Embedding." In: *Theorem Proving in Higher Order Logics*. Ed. by K. Slind, A. Bunker, and G. Gopalakrishnan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 305–320.

[52] A. Yushkovskiy. "Comparison of Two Theorem Provers: Isabelle/HOL and Coq." In: *Proceedings of the Seminar in Computer Science (CS-E4000)* (2018). DOI: `10.48550/ARXIV.1808.09701`.