An Extensible Theorem Proving Frontend

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Sebastian Andreas Ullrich

Tag der mündlichen Prüfung:	26.05.2023
1. Referent:	Prof. Dr.-Ing. Gregor Snelting
2. Referent:	Prof. Dr. rer. nat. Jasmin Blanchette

Contents

Abstract vii

Zusammenfassung ix

Acknowledgements xi

1 Introduction 1
- 1.1 Interactive Theorem Proving . . . 1
- 1.2 Structure . . . 2
- 1.3 Lean . . . 4
  - 1.3.1 Programming in Lean . . . 4
  - 1.3.2 Proving in Lean . . . 6
  - 1.3.3 The Essence of Lean . . . 8
  - 1.3.4 A Short History of Lean . . . 10

2 A Retrospective of Extensibility in Lean 3 13
- 2.1 Lean 3 as a Programming Language . . . 14
- 2.2 Lean 3 as a Metaprogramming Language . . . 17
- 2.3 Advanced Tactic Programming . . . 22
- 2.4 Interactive Proving . . . 27
- 2.5 Related Work . . . 31

3 An Overview of Lean 4 33
- 3.1 Architecture . . . 34
- 3.2 The Kernel and Type Theory . . . 35
  - 3.2.1 Internalizations . . . 35
  - 3.2.2 η-Conversion for Structure Types . . . 38
  - 3.2.3 Mutual Inductive Types . . . 40

3.3 The (Future of the) Module System 45
3.4 The Parser 49
3.5 The Elaborator 52
3.6 The Code Generator 54
3.7 The User Interface 57

4 A Macro System for Theorem Provers 61
4.1 Lean 4 Macro System by Example 63
4.2 Hygiene Algorithm 67
4.2.1 Expansion Algorithm 68
4.2.2 Examples 70
4.3 Implementation 72
4.3.1 Extended Quasiquotations 76
4.4 Typed Syntax 79
4.5 Integrating Macros into Elaboration 82
4.6 Tactic Hygiene 84
4.7 Best-Effort Eager Name Analysis in Macros 86
4.8 Related Work 88

5 An Imperative Extension of *do* Notation 93
5.1 Local Mutation 95
5.2 Early Return 105
5.3 Iteration 108
5.4 Implementation 114
5.4.1 Reference Implementation 115
5.4.2 Full Implementation 117
5.5 Reasoning 119
5.6 Formalization 119
5.7 Evaluation 127
5.8 Related Work 130

6 An Efficient Reference Counting Scheme for Functional Programming 133
6.1 IR Syntax 135
6.2 IR by Example 137
6.3 Semantics of the Reference-Counting IR 140
6.4 A Compiler from λ_{pure} to λ_{RC} 144
6.4.1 Inserting Destructive Update Operations 144

6.4.2 Inferring Borrowing Signatures 147
6.4.3 Inserting Reference Counting Operations 149
6.4.4 Preserving Tail Calls 153
6.5 Optimizing Functional Data Structures for `reset`/`reuse` . . 153
6.6 Runtime Considerations 155
6.7 Experimental Evaluation 157
6.8 Related Work 163

7 Conclusion 167
7.1 Future Work 168

A Macro Implementation of *do* Notation 171
A.1 Basic do Notation 171
A.2 Mutable Variables 172
A.3 Early Return 176
A.4 Iteration 177

B Formal Correctness of *do* Translation 183
B.1 Contexts 183
B.2 Intrinsically Typed Representation of do Statements 185
B.3 Dynamic Evaluation Function 187
B.4 Translation Functions 188
B.5 Equivalence Proof 193
B.6 Partial Evaluation 199

C Formal Reference Counting Semantics & Proof of Correctness 203
C.1 Pure Semantics 203
C.2 Well-Formedness 205
C.3 reset/reuse 206
C.4 Borrow Inference 207
C.5 A Type System for RC-Correct Programs 208
C.6 Proof of Semantics Preservation 212
C.7 Proof of Compilation Well-Typedness 218

Bibliography 225

Index 241

List of Publications 243

Abstract

Interactive theorem provers (ITPs) are tools that can formally verify as well as help with writing computerized proofs, such as ones about mathematics or the correctness of programs. In recent years, extensive formalization projects have been completed using ITPs. The Lean theorem prover in particular has not only been used to formalize established theorems in mathematics but is now being utilized for formalizing novel mathematical topics as well. The goal of the Lean project is nothing less than to revolutionize how mathematicians work by making formalized proofs a realistic alternative to pen-and-paper proofs, removing the need for laborious step-wise refereeing and ensuring that all necessary proof steps are recorded accurately instead of requiring interpretation and implicit background knowledge from the reader. Making this goal a reality will require further breakthroughs in formalization efficiency and usability in order to close the convenience gap between pen and machine.

As one step towards this goal, this thesis discusses the design and implementation of a theorem prover frontend fully extensible by users as part of the latest version of Lean, Lean 4. The frontend is responsible for a particular aspect of formalization usability: accepting the user's input in a syntactical form that should optimize for multiple, partially contradictory goals: compactness, readability by humans, and unambiguous interpretability by the prover system. As the set of mathematical notations in use is extensive, growing every year, and often specific to a field, author, or even single paper, the frontend must allow users to extend it ad hoc with new, expressive notations with flexible interpretation rules. The same desire for flexible input syntax and interpretation rules can also be found at the level of individual proof steps ("tactics") and higher levels of organizing formalizations and programs.

The core part of this extensibility is an expressive macro system I have developed for Lean that covers both simple syntax transformations ("sugars") and complex, type-aware *elaboration*. The macro system is based on a novel *hygiene* algorithm inspired by that of the Lisp-family language Racket but custom-built for ITPs whose task is to ensure that lexical scoping works as intuitively expected even for complex macros. I have taken care in making the macro system approachable in general by providing multiple abstraction levels of increasing expressivity but based on the same fundamental principles. As one example use case of the macro system, I present an imperative extension of *do* notation known from Haskell and an implementation thereof as macros. The extended syntax has been integrated into Lean 4 and fundamentally changed the way both Lean developers as well as users write monadic and even pure Lean programs. I describe the notation's formal semantics and prove in Lean that this semantics coincides with the meaning of the program after macro expansion.

While the macro system is a central part of the extensible frontend, it is tightly connected to and dependent on other components. I give a description of the whole frontend and the greater Lean system, focusing on parts I have made significant contributions to. As motivation for this work, I review earlier attempts at extensibility of the frontend in Lean 3 and discuss lessons learned from them. Finally, I describe an efficient reference counting scheme for functional programming that has been instrumental in making the extensibility work in practice by allowing us to rewrite Lean in Lean itself. By transparently reusing allocations, the scheme, just like the extended do notation, combines benefits of imperative and pure functional programming into a new paradigm I call "pure imperative programming".

Zusammenfassung

Interaktive Theorembeweiser sind Softwarewerkzeuge zum computergestützten Beweisen, d.h. sie können entsprechend kodierte Beweise von logischen Aussagen sowohl verifizieren als auch beim Erstellen dieser unterstützen. In den letzten Jahren wurden weitreichende Formalisierungsprojekte über Mathematik sowie Programmverifikation mit solchen Theorembeweisern bewältigt. Der Theorembeweiser Lean insbesondere wurde nicht nur erfolgreich zum Verifizieren lange bekannter mathematischer Theoreme verwendet, sondern auch zur Unterstützung von aktueller mathematischer Forschung. Das Ziel des Lean-Projekts ist nichts weniger als die Arbeitsweise von Mathematikern grundlegend zu verändern, indem mit dem Computer formalisierte Beweise eine praktible Alternative zu solchen mit Stift und Papier werden sollen. Aufwändige manuelle Gutachten zur Korrektheit von Beweisen wären damit hinfällig und gleichzeitig wäre garantiert, dass alle nötigen Beweisschritte exakt erfasst sind, statt der Interpretation und dem Hintergrundwissen des Lesers überlassen zu sein. Um dieses Ziel zu erreichen, sind jedoch noch weitere Fortschritte hinsichtlich Effizienz und Nutzbarkeit von Theorembeweisern nötig.

Als Schritt in Richtung dieses Ziels beschreibt diese Dissertation eine neue, vollständig erweiterbare Theorembeweiser-Benutzerschnittstelle („frontend“) im Rahmen von Lean 4, der nächsten Version von Lean. Aufgabe dieser Benutzerschnittstelle ist die textuelle Beschreibung und Entgegennahme der Beweiseingabe in einer Syntax, die mehrere teils widersprüchliche Ziele optimieren sollte: Kompaktheit, Lesbarkeit für menschliche Benutzer und Eindeutigkeit in der Interpretation durch den Theorembeweiser. Da in der geschriebenen Mathematik eine umfangreiche Menge an verschiedenen Notationen existiert, die von Jahr zu Jahr weiter wächst und sich gleichzeitig zwischen verschiedenen Feldern, Autoren

oder sogar einzelnen Arbeiten unterscheiden kann, muss solch eine Schnittstelle es Benutzern erlauben, sie jederzeit mit neuen, ausdrucksfähigen Notationen zu erweitern und ihnen mit flexiblen Regeln Bedeutung zuzuschreiben. Dieser Wunsch nach Flexibilität der Eingabesprache lässt sich weiterhin auch auf der Ebene der einzelnen Beweisschritte („Taktiken“) sowie höheren Ebenen der Beweis- und Programmorganisation wiederfinden.

Den Kernteil dieser gewünschten Erweiterbarkeit habe ich mit einem ausdrucksstarken Makrosystem für Lean realisiert, mit dem sich sowohl einfach Syntaxtransformationen („syntaktischer Zucker“) also auch komplexe, typgesteuerte Übersetzung in die Kernsprache des Beweisers ausdrücken lassen. Das Makrosystem basiert auf einem neuartigen Algorithmus für *Makrohygiene*, basierend auf dem der Lisp-Sprache Racket und von mir an die spezifischen Anforderungen von Theorembeweisern angepasst, dessen Aufgabe es ist zu gewährleisten, dass lexikalische Geltungsbereiche von Bezeichnern selbst für komplexe Makros wie intuitiv erwartet funktionieren. Besonders habe ich beim Entwurf des Makrosystems darauf geachtet, das System einfach zugänglich zu gestalten, indem mehrere Abstraktionsebenen bereitgestellt werden, die sich in ihrer Ausdrucksstärke unterscheiden, aber auf den gleichen fundamentalen Prinzipien wie der erwähnten Makrohygiene beruhen. Als ein Anwendungsbeispiel des Makrosystems beschreibe ich eine Erweiterung der aus Haskell bekannten „do“-Notation um weitere imperative Sprachfeatures. Die erweiterte Syntax ist in Lean 4 eingeflossen und hat grundsätzlich die Art und Weise verändert, wie sowohl Entwickler als auch Benutzer monadischen, aber auch puren Code schreiben.

Das Makrosystem stellt das „Herz“ des erweiterbaren Frontends dar, ist gleichzeitig aber auch eng mit anderen Softwarekomponenten innerhalb der Benutzerschnittstelle verknüpft oder von ihnen abhängig. Ich stelle das gesamte Frontend und das umgebende Lean-System vor mit Fokus auf Teilen, an denen ich maßgeblich mitgewirkt habe. Schließlich beschreibe ich noch ein effizientes Referenzzählungsschema für funktionale Programmierung, welches eine Neuimplementierung von Lean in Lean selbst und damit das erweiterbare Frontend erst ermöglicht hat. Spezifische Optimierungen darin zur Wiederverwendung von Allokationen vereinen, ähnlich wie die erweiterte do-Notation, die Vorteile von imperativer und pur funktionaler Programmierung in einem neuen Paradigma, das ich „pure imperative Programmierung“ nenne.

Acknowledgements

I wish to thank my advisor, Prof. Gregor Snelting, for his guidance and support. His trust in me and willingness to support development of another theorem prover wholly different from the one already established at the group cannot be understated as contributions to the existence of this thesis. I also thank Prof. Jasmin Blanchette for agreeing to review this thesis and for his many helpful nitpicks.

I cannot thank enough my partner in crime and the creator of Lean, Leonardo de Moura. I don't think either of us quite realized the extent of the journey we embarked on together some four years ago. Being afforded the opportunity to take part in shaping a language enjoyed by many people truly is a once-in-a-lifetime opportunity. The amount I benefited from Leo's immense knowledge and experience along the way cannot be repaid. Likewise, I must thank Prof. Jeremy Avigad not only for graciously inviting me to CMU and advising my master's thesis there, but then to facilitate a stay at Microsoft Research on top during which I met Leo in person for the first time, kickstarting our collaboration in earnest.

I also thank my former and current colleagues in Karlsruhe for creating a productive yet fun and relaxed working atmosphere with entirely too many Sebastians: Johannes Bechberger, Simon Bischof, Sebastian Buchwald, Andreas Fried, Sebastian Graf, Martin Hecker, Denis Lohner, Manuel Mohr, Martin Mohr, Jakob von Raumer, Brigitte Sehan-Hill, Max Wagner, and Andreas Zwinkau. Andreas, Jakob, Max, and Sebastian Graf in particular were exposed to many of my more or less sane ideas and provided valuable feedback. Denis together with Joachim Breitner sparked the fascination with theorem provers in me, even if it was not quite the right one. Johannes graciously extended his Temci project for my needs. Martin Mohr provided valuable advice and templates for actually finishing

the PhD. Sebastian Graf, Jakob, and Max as well as Mario Carneiro, Leo, and Yannick Forster proofread parts of this thesis and provided important feedback.

Even more people gave helpful advice and feedback on papers this thesis is based on: Thomas Ball, Christiano Braga, David Thrane Christiansen, Gabriel Ebner, Matthew Flatt, Johannes Hölzl, Alexis King, Daan Leijen, Gregory Malecha, Simon Peyton Jones, Tahina Ramananandro, Daniel Selsam, and Nikhil Swamy, thank you!

During my PhD, I had the pleasure to supervise and work with bright students that advanced the Lean ecosystem and showed us that becoming fluent in Lean is possible in a short amount of time: Niklas Bülow, Markus Himmel, Marc Huisinga, Lars König, and Joscha Mennicken, thank you for your contributions and interest in Lean.

Finally, I wish to thank my parents for their love and support, as well as my high school maths teacher Bernhard Gärttner for his central role in sending me on this path. By teaching me love of both mathematics and programming at the same time and encouraging, supporting, and challenging me over many years, he could not have better prepared me for the topic of theorem proving even if neither of us knew it existed at that point.

No large language models were involved in the writing of this thesis.

There is much work to do
– Conor McBride, *Epigram: Practical Programming with Dependent Types* [McBride, 2005]

1 Introduction

1.1 Interactive Theorem Proving

The idea of using a computer to not only *compute* mathematical terms but also *prove* mathematical and other theorems is by now long established, beginning in the 1970s with Robin Milner et al.'s *LCF* system described in the adequately titled *LCF: a way of doing proofs with a machine* [Milner, 1979] and Nicolaas Govert de Bruijn et al.'s *Automath* system [de Bruijn, 1970] developed around the same time. While the two systems follow different fundamental approaches that both can be found in derived systems to this day, what they have in common is *interactivity*. Back then as now it was infeasible to have theorems of high complexity proved automatically by a computer due to the gigantic search space, and so human and machine must join forces to develop such proofs together. This back-and-forth conversation consists of the human stating the proposition to be proved in some way the computer can understand and then proposing how to break it apart in successive proof steps (some of which may be found by the machine automatically when the step size is small enough) to which the computer will respond with the remaining *proof goals* if it believes the steps are sound or an appropriate error if it does not.

Thus, despite recent advances in having at least competition-level mathematical problems solved automatically by a machine [Polu et al., 2022, Lample et al., 2022][1], this style of *interactive theorem proving* still appears to be the most promising approach to a formalized and computer-verified representation of our collective mathematical knowledge as well as other

[1] Which is still work done in the context of, and verified by, an interactive theorem prover.

topics full of high-complexity proofs such as program verification. One particular system that has recently come into prominence from tackling this challenge is the Lean programming language and theorem prover, which is the focus of this thesis. Through a combination of desirable features as well as some fortuitous circumstances, Lean has managed to capture an expansive community of mathematicians contributing to a unified mathematics library, *mathlib* [The mathlib Community, 2020]. At the time of writing, mathlib has, after just five years of development, reached the milestone of more than one million lines of code (a term that, as we will soon learn, is readily applicable to proofs as well) and 100,000 theorems contributed by 280 authors [The mathlib Community, 2022b] that neither Lean developers nor users would have believed possible just a few years before, and does not show any signs of slowing down. In just the last year (2022), we have witnessed the completion of the Liquid Tensor Experiment [The mathlib Community, 2022a] addressing a formalization challenge posed by Fields medalist Peter Scholze, the completion of the sphere eversion project [van Doorn et al., 2023], and the completion of a formalization [Bloom and Mehta, 2022] of a new result in number theory [Bloom, 2021] preceding the completion of the informal refereeing.

The topic of this thesis is my work on supporting this development into the future as part of the latest version of Lean, Lean 4. In particular, my PhD has been focused on making it easier for users to express themselves and encode their concepts in Lean and similar systems in a natural way. I have done so by introducing an unparalleled degree of extensibility into the theorem prover's *frontend*, the part that implements the interaction with users. The *macro system* that resulted from this work is now a central part of Lean and likewise the central chapters of this thesis discuss implementation and use of this system.

1.2 Structure

The main contents of this thesis are divided into five chapters that partially build on top of each other (Fig. 1.1). After the introduction, each chapter lists the main contributions as well as acknowledges parts and prior published work it is based on that were done in cooperation with other authors. Chapter 2 reviews initial extensibility efforts in Lean 3 and discusses its shortcomings. While it motivates and sets up the challenges addressed by

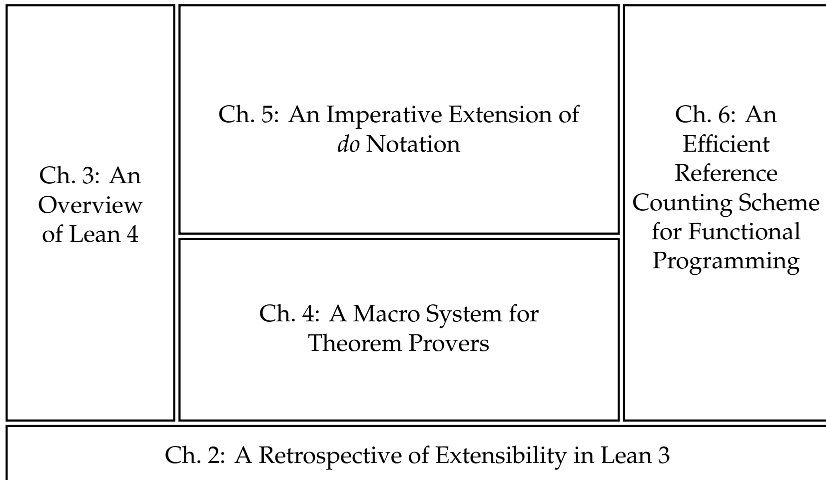


Figure 1.1: Visualization of content chapters of this thesis, with chapters placed on top of chapters they build on. Box sizes are roughly proportional to chapter lengths.

the following chapters, it is not required reading for them. Chapter 3 gives an overview of Lean 4 and summarizes significant changes not addressed by later chapters. Chapter 4 describes the novel macro system I designed for Lean 4, the core ingredient of the extensible frontend. As one significant application of the macro system, I discuss an extension of functional *do* notation with additional imperative elements in Chapter 5, implemented purely as macros in Appendix A, and formally prove correctness of the translation in respect to a natural semantics (Appendix B). Chapter 6 follows the same idea of combining imperative and functional properties, but on the runtime instead of the language level: by transparently introducing opportunities for reusing allocations, I show how we can optimize pure functional code and utilize imperative data structures such as arrays in it. I prove soundness of the compilation steps in Appendix C. Chapter 7 finally concludes the thesis and points out possible future research directions.

In the rest of the chapter, I will give a short introduction to Lean.

1.3 Lean

1.3.1 Programming in Lean

Lean is, as mentioned, both a programming language and a theorem prover, both aspects of which we will take a detailed look at in this thesis in separation as well as in combination in the form of *metaprogramming*. For readers familiar with functional programming, the basics of the purely functional language Lean should be relatively easy to grasp: the syntax is reminiscent of the ML family of languages, with some additional influences, including limited whitespace sensitivity, from Haskell.

```
-- a function definition using lambda notation and type inference
def f := fun x => x + 1
-- the same function, using explicit parameters and types
def f (x : Nat) : Nat := x + 1

-- a named product type with named constructor and fields
structure Point where mk ::
  x : Nat
  y : Nat

-- pattern matching on the structure's constructor
def getX (p : Point) := match p with
  | Point.mk x _ => x
  -- `match` is terminated by deindentation

-- an equivalent function, using a top-level pattern matching
     shorthand
def getX (p : Point)
  | Point.mk x _ => x

-- an equivalent function, using convenient "projection notation"
def getX (p : Point) := p.x
```

Lean makes heavy use of *typeclasses* [Wadler and Blott, 1989] to automatically infer certain properties about types and other terms.

```
-- a parametric typeclass
class ToString (α : Type) where
  toString : α → String
-- make `toString` available outside of the `ToString` namespace
export ToString (toString)

instance : ToString Point where
  -- implicitly infer `ToString` instance of `Nat`, not shown here
  toString p := toString p.x ++ "," ++ toString p.y

#eval toString (Point.mk 1 2)  -- 1,2
```

Lean programs can be compiled or executed right in the editor using the `#eval` command shown above. When used as a programming language, Lean has eager evaluation semantics.

An important advanced ingredient of Lean is *inductive types*, which are a generalization of algebraic data types. For example, the type `Nat` of natural numbers used above is defined as the standard inductive Peano representation.

```
inductive Nat where
  | zero : Nat
  | succ : Nat → Nat
```

Note though that Lean uses a more optimized representation internally and at run time, as discussed for the special case of the kernel in Section 3.2.

Inductive types in fact are, together with function types and universes (see below), the only kind of type available in Lean's type theory. A structure fundamentally is simply an inductive type with one constructor, and a typeclass is a special kind of structure.

What makes inductive types more general than standard algebraic data types is their ability to form *type families* using *dependent types*, that is, the result types of constructors do not have to be uniform and they may depend on *values*, not just other types. The prime example is the type of lists of a given length.

```
inductive Vector (α : Type) : Nat → Type where
  | nil : Vector α 0
  | cons : α → (n : Nat) → Vector α n → Vector α (n + 1)
```

This declaration defines a dependent type former `Vector : Type → Nat → Type` that maps a given element type and a natural number, which is not a

type, to our new type. The type family `Vector` α[2] then can be constructed in two ways: the constructor `nil` creates an empty list of length zero, and the constructor `cons`, given an element of type `α` and a list of length n, creates a list of length $n + 1$. The second constructor thus is a dependent function, using the notation `(a : α) → β a` for the *dependent function type* with domain `α` and range `β a` that may vary for each `a : α`.

A more thorough introduction to programming in Lean can be found in [Christiansen, 2022]. For the implementation parts comprising the majority of this thesis, the above basics on Lean programming without details on inductive types, dependent types, or universes should be sufficient for comprehending the text. In fact, there are exactly two, closely related uses of a dependent type in the discussed implementation (can you spot them?). The main and original motivation for all these advanced features is, of course, Lean's other aspect, theorem proving.

1.3.2 Proving in Lean

Getting more formal as is appropriate for this second aspect, Lean's type theory is based on a variant of the type theory of Coq [The Coq Team, 2017], the Calculus of Inductive Constructions [Paulin-Mohring, 2015], which adds the aforementioned inductive types to the previously developed Calculus of Constructions [Coquand and Huet, 1988]. Other dependently typed proof assistants and programming languages such as Agda [Norell, 2009] and Idris [Brady, 2013], which all have influenced the design of Lean, have similar foundations. The Calculus of Constructions is the most expressive corner of the λ-cube [Barendregt, 1991], i.e. a λ-calculus where terms and types may freely depend on other terms and types — indeed, there is no more syntactic distinction between terms and types. Following in Automath's tradition, the main interest of such strong type systems to theorem provers is the renowned *Curry-Howard correspondence* [Howard, 1980]: proofs are programs, propositions are types!

More specifically, it turns out that the simply typed λ-calculus corresponds to intuitionistic propositional logic: function types correspond to

[2] Unlike the list length, α is constant in all uses of `Vector` in its constructors, i.e. `Vector` is *parametric* over it. We thus call α a *(uniform) parameter* and use Lean's parameter syntax to the left of the colon for it, while varying parameters like `n` are called *indices* and must be specified to the right of the colon.

implications, variables to hypotheses, lambda abstraction to implication introduction, function application to modus ponens.

```
-- trivial implication, also the identity function
-- an `example` is a temporary declaration without a name
-- `p` is automatically quantified as a type variable
example : p → p := fun h => h

-- the "principle of simplification" [Whitehead and Russell, 1910],
-- also the K combinator
example : p → q → p := fun hp hq => hp

-- the "principle of the syllogism" [Whitehead and Russell, 1910],
-- also function composition
example (hpq : p → q) (hqr : q → r) : p → r :=
  fun hp => hqr (hpq hp)
```

If we want more than propositional logic, however, the question is: what is the type theoretic equivalent of universal quantification? The answer is the dependent function type, with the Calculus of Constructions corresponding to higher order intuitionistic logic.

```
-- introduction of universal quantification is lambda abstraction
example (p : α → Prop) : ∀ x : α, (p x → p x) := fun x px => px

-- elimination of universal quantification is function application
example (p : α → Prop) (h : ∀ x : α, p x) (y : α) : p y := h y
```

Here `∀ x : α, e` is simply another way to write the dependent function type `(x : α) → e`. We usually reserve it for specifying types in `Prop`, the universe of propositions. *Universes* are mostly a technical means to avoid Girard's paradox [Girard, 1972]: we want every term, including types, to have a type, but the type of (simple) types `Type` may not be its own type without inviting inconsistency. Thus Lean features an infinite hierarchy of type universes `Type : Type 1 : Type 2 : ... : Type u : ...` where `u` is a *universe variable*. `Prop` is an important outlier: it sits at the base of the hierarchy, `Prop : Type`, but has two distinct properties: it is *impredicative*, which in this context means that function types into a proposition are themselves propositions,

```
example (p : α → Prop) : Prop := ∀ x, p x
```

and is *irrelevant*, meaning that any two proofs of the same proposition are the same and interchangeable (*definitionally equal*).

```
-- proof by reflexivity, `rfl : x = x`
example (p : Prop) (h1 h2 : p) : h1 = h2 := rfl
```

For more details on Lean's type theory, see [Carneiro, 2019] for a description of the theory of Lean 3, which we will extend to that of Lean 4 in Section 3.2.

While having a single language for terms, types, proofs, and propositions greatly reduces the number of concepts and syntax one has to keep in mind, the term language is not in fact the primary language in which Lean proofs are written. Instead, proofs are usually (such as in Appendix B) structured as a sequence of proof steps called *tactics*, which may implement arbitrarily powerful automation, preceded by the keyword `by` usable in any term context. Thus while input *terms* roughly reflect the structure of the produced internal lambda expression, tactics abstract even further from that, more following the mostly linear flow of an informal proof text than a highly nested expression tree.

```
example (hpq : p → q) (hqr : q → r) : p → r := by
  intro hp    -- remaining goal: ..., hp : p ⊢ r
  apply hqr   -- remaining goal: ... ⊢ q
  apply hpq   -- remaining goal: ... ⊢ p
  exact hp    -- goals accomplished

-- the same theorem, using [Limperg and From, 2023]'s proof search
    tactic
example (hpq : p → q) (hqr : q → r) : p → r := by aesop
```

1.3.3 The Essence of Lean

The basic language and type theory summarized above for the most part are not significant departures from other type theory-based systems that precede Lean, such as Coq and Agda. This begs the question: why go through the effort of inventing a new system in the first place?

The main motivation that led Leonardo de Moura[3] to start the Lean project at Microsoft Research in 2013 was twofold:

[3] whom I will take the liberty to also abbreviate as "Leo" in the subsequent many mentions

- Creating a platform for developing *white-box automation*. As the main architect of the Z3 SMT solver [de Moura and Bjørner, 2008], Leo was acutely aware of the strengths but also limitations of standard SMT implementations, in particular their non-configurability and one-size-fits-all design [de Moura and Passmore, 2013], which might thus be described as *black-box* automation. A white-box approach then would mean to expose the components making up an SMT solver to users in such a way that they can recombine and reconfigure them according to their specific needs. An interactive theorem prover's tactic language would be a natural vehicle for such a system, allowing for rapid, incremental development of tailor-made automation, and would help close the gap between these two theorem proving approaches.
- Quite in contrast to the above point of extensive automation, condensing standard dependent type theory down to a minimal theory, which ultimately gave rise to the name *Lean*. While the sophistication of tactics should be essentially limitless, implementing a verifier for their output should be as simple, and thus usually as error-free, as possible. At the same time, the base language should be sufficiently expressive for the formalization of advanced mathematics and for program verification, for which dependent type theory seemed like the most promising foundation. Thus the *lean*-ness of Lean should be understood in comparison to other implementations of dependent type theory, which does come with a certain degree of fundamental complexity compared to simpler logic systems. Compared to Coq, whose type theory Lean's is inspired by and still closest to, this in particular relates to replacing the primitive concepts of termination checking, fixpoint operators, and pattern matching with primitive *recursor* functions [Dybjer, 1994], as well as forgoing the type system-embedded module system.

While the second goal of a small type theory and kernel has been achieved with the very first version of Lean, leading to a proliferation of independent type checkers for Lean 3, and has seen only minor corrections since (Section 3.2), work on the first goal at the time of writing is still mostly focused on the fundamental *creating the platform* part — as it turns out, developing an interactive theorem prover is no small feat even before considering advanced automation. Having said that, after initial experiments with SMT primitives in Lean 3 [Ebner et al., 2017] from our side, we are happy to see

that with Lean 4, the platform has progressed enough to support other peoples' work on novel white-box automation [Limperg and From, 2023], even if not based on SMT.

What shapes the perception of Lean in the eyes of end-users at the current time, however, is perhaps relatively independent of the above achieved and outstanding goals. The current direction of the Lean project and ecosystem can be attributed instead primarily to the collaboration with the group of Jeremy Avigad that led to the development of mathlib, as detailed in the next subsection, and with it a clear focus on classical, non-constructive mathematics. This focus is partly a technical decision expressed in Lean's theory as in the unconditional assumption of definitional proof irrelevance mentioned above, but to a greater degree is a social convention by the mathlib authors who readily embraced classical axioms like the excluded middle and the axiom of choice, owing to their backgrounds in informal classical mathematics. The perception of Lean as a system suitable for classical mathematicians to learn and to formalize their own topics in ultimately is what led to the unprecedented influx from this crowd into the community and the resulting development of mathlib and of the impressive works built on top of it. At the same time, the Lean system itself remain sufficiently agnostic to allow for other applications such as program verification as well.

1.3.4 A Short History of Lean

As a majority of the work described in this thesis is built on lessons we have learned from previous versions of Lean, I close this chapter with a short description of Lean's development history over its various major versions before taking a closer look at Lean 3's extensibility features and their limitations in the next chapter. Because of the explorative nature of the Lean project, each version came with significant changes in the system and an effectively rewritten standard library such that no backward compatibility could be provided.

Lean 0.1 (2014) was the first published prototype of Lean, initiated by Leonardo de Moura (Microsoft Research) and later joined by Soonho Kong (then Carnegie Mellon University). While Lean 0.1 introduced the ML-like syntax that in many parts has persisted throughout all Lean versions as well as a simple `simp` tactic, it notably did not yet

feature support for inductive types, with types being introduced via ad hoc axioms instead. Lean 0.1 also featured prototypical support for syntax and tactic extensibility via Lua scripts, but this part was never further developed or put to use before being dropped in Lean 3 in favor of extensions written in Lean itself.

Lean 2 (2015) [de Moura et al., 2015b], initially called Lean 0.2, was the first official release of Lean. It was also the first release I personally have used and started contributing to. Apart from adding crucial missing features such as proper support for inductive types and extending the set of built-in tactics, Lean 2 added support for Homotopy Type Theory (HoTT) [The Univalent Foundations Program, 2013] as a major feature, with univalence as a noncomputable axiom and proof irrelevance disabled. Lean 2 came with a nontrivial standard and math library developed in collaboration with Jeremy Avigad's group, with a separate version for HoTT.

Lean 3 (2017) replaced the unused support for extensibility via Lua with a radically different approach: making Lean a programming language that can be used to define tactics and other metaprograms in Lean itself [Ebner et al., 2017], as described in the next chapter. Another major change from Lean 2 was the removal of support for HoTT, which was done because of concerns over how to implement efficient tactics in the absence of proof irrelevance and the code duplication resulting from this as well as uncertainty at that time over whether "book HoTT" or the recent computational version of Cubical Type Theory [Cohen et al., 2015] was preferable. The *mathlib* library was eventually spun off as an independent, community-managed project so as to focus the Lean 3 standard library to a core set of libraries essential to both formalization of mathematics and formal verification of programs.

Lean 4 (2023) [de Moura and Ullrich, 2021] finally is the most recent release of Lean at the time of writing and the version most of this thesis is dedicated to. If making Lean a metaprogramming language in Lean 3 was a radical idea at the time, Lean 4 dwarfed this effort by making Lean a *general-purpose* programming language sufficiently expressive and efficient for reimplementing most parts of Lean in Lean itself. We will see in the following chapters how this significant

> rewrite and the system and language changes introduced by it are a direct continuation of the extensibility work in Lean 3 and a desire to lift its fundamental limitations. Leo and I started development of Lean 4 in 2018, with most of the fundamental design we were busy implementing the following four years established during an internship of mine at Microsoft Research later that year. While Lean 4 is not directly compatible with earlier versions, a mixed automated/manual port of all of Lean 3 mathlib has just been finished at the time of writing, with resources being updated to designate Lean 4 as the official and recommended version of Lean.

Given the frequency of major Lean releases with almost no backward compatibility, inevitably the question arises: when will we switch to Lean 5 and start from scratch again? While the need for a subsequent major version is impossible to project so early into the life cycle of a new release, we can at least take the escalating time between releases as a hint towards stability, especially considering that the implementation of Lean 4 took roughly as long as that of all three versions before it combined. I think I can speak for all core developers that at this point, we do not even want to think about any effort that comes close to the Lean 4 rewrite. In many ways, the foundations of Lean 4, particularly the extensible frontend, are as generic as we can make them and there are no plans to revolutionize them yet again.

I felt that pressure of time that is perhaps the
surest indication we have left childhood behind
– Gene Wolfe, *The Book of the New Sun*

2 A Retrospective of Extensibility in Lean 3

A central characteristic of dependent type theory is its computational nature: type checking needs to reduce definitions to successfully decide equivalence of terms, which gives rise to a general way to reduce terms to a normal form or, in other words, to compute their values. Given that an effective dependently typed language must provide expressive ways to specify such definitions, it is not too surprising that this same proving language would be adapted not only for programming, but *metaprogramming*, that is, for developing programs that help in constructing other terms, proofs, and programs. Metaprogramming has a long history in interactive theorem proving, with the programming language ML literally designed as the *Meta Language* of the LCF proof assistant for writing automation [Gordon et al., 1978]. A more recent development is the idea of using the same language as both the metalanguage and its target language [van der Walt and Swierstra, 2012, Brady, 2013, Ziliani et al., 2015, Christiansen and Brady, 2016], which Lean adopted starting with Lean 3. In other words, Lean 3+ enables users to write proof automation ("tactics") in the same language used for specifying the propositions to be proved: the Lean language.

Contributions. In this chapter, I present and review the extensible tactic system of Lean 3 and related parts. While not the first metaprogramming system for theorem proving of its kind, it has become a well-adopted, main feature of Lean 3 and continues in extended form in Lean 4. I discuss many restrictions that were lifted in Lean 4, and why they could not be solved in Lean 3. Just like these restrictions laid the foundation for the development

```
meta def collatz : nat → nat :=
λ n,
if n % 2 = 0 then
  collatz (n / 2)
else
  collatz (2 * n + 1)
```

```
0:  scnstr #0        10: cfun nat.add
1:  scnstr #2        11: ginvoke
2:  push 0                collatz
3:  cfun nat.mod     12: goto 17
4:  cfun             13: scnstr #2
     nat.dec_eq      14: push 0
5:  cases2 13        15: cfun nat.div
6:  scnstr #1        16: ginvoke
7:  push 0                collatz
8:  scnstr #2        17: ret
9:  cfun nat.mul
```

Figure 2.1: A simple, possibly non-terminating Lean 3 program implementing the Collatz sequence, and its bytecode in text representation.

of Lean 4, this chapter will lay the foundation for the later chapters of this thesis that discuss the solutions we have found for lifting these restrictions.

Acknowledgements. Lean 3's tactic system described in this chapter is joint work with Gabriel Ebner, Jared Roesch, Jeremy Avigad, and Leonardo de Moura [Ebner et al., 2017]. My main contributions are Section 2.4 on interactive use and parsing of tactics, and the parts on user-defined attributes and expression reflection in Section 2.3. The parts marked with *Discussion* are novel work written for this thesis in hindsight of our experience with Lean 3 after publication and the development of Lean 4.

2.1 Lean 3 as a Programming Language

Lean 3 became a programming language by means of a relatively simple translation to an untyped, stack-based bytecode format that can be run in a built-in interpreter (Fig. 2.1). In order to function as a metaprogramming language, such code can make use of a selective set of functions that have been exported from the underlying C++ code base. These functions are marked with a new `meta` modifier on the Lean side that restricts their use to functions that in turn are marked `meta`, effectively functioning as a new lexical namespace; in particular, they cannot be used in theorems, for which the `meta` modifier is not accepted. The reason for this restriction was to

make sure we could not affect Lean 3's soundness by accidentally exporting a C++ function with a Lean type that was provably uninhabited, perhaps because of a minor oversight in the translation of the C++ type to Lean. Since the function does not have a definition on the Lean side, the type checker would not be able to catch this case like it usually could, and we could prove arbitrary theorems from it if it were not for the `meta` namespace. Essentially each exported function acts as a new axiom, and the `meta` modifier immediately restricts uses of this axiom to metaprogramming applications. Because metaprograms are not part of Lean's Trusted Code Base — after execution, their output is checked by the kernel just as if we had written down the output manually — soundness is therefore unaffected by the introduction of such functions.

While the syntax of a `meta` function is identical to that of a regular one, the soundness-ensuring namespace restriction explained above in turn allow us to lift a different restriction of regular functions for `meta` functions, making them more convenient for programming: they can make use of arbitrary recursion, i.e. do not have to prove that they terminate and may in fact not do so as seen in Fig. 2.1. The Lean kernel does not accept any kind of recursion in regular definitions; recursion is instead transformed by the frontend into applications of *recursors* that encode the recursion principle of a type, for which the recursion must be provably structural or, more generally, well-founded.[1] `meta` functions on the other hand are passed to the Lean 3 kernel with a specific flag that enforces the namespace restriction but accepts recursive uses as is without further checks.

Discussion. The programming capabilities of Lean 3 are clearly trimmed to the use case of metaprogramming: the bytecode interpreter is a simple extension that provides acceptable performance for stringing together invocations of the much faster C++ primitives. However, while the system is *expressive* enough to implement more ambitious tactics such as the superposition prover described in Section 5.2 of [Ebner et al., 2017], it turned out not to be *performant* enough to enable use of the tactic in practice. A companion native backend was under development at the time as mentioned in [Ebner et al., 2017], but did not appear until Lean 4 with a

[1] Well-founded recursion is in fact a special case of structural recursion over a well-foundedness inductive predicate.

from-scratch implementation. For the simplicity of implementation, and because the eager, immutable representation of runtime values guaranteed the absence of cycles, Leo used reference counting for garbage-collecting Lean objects, based on the same intrusive smart pointers also used by existing persistent data structures in the C++ implementation such as kernel terms. Reference counting operations are implied by the interpreter's and primitives' use of these smart pointers to Lean objects, with the only special optimization being a destructive update of arrays with reference count 1. The Lean 3 runtime has basic support for multi-threading, but the implementation is relatively simplistic: in order to avoid the overhead of atomic reference counting, object graphs are simply copied wholesale when contained in the initial closure of a newly started thread. We kept reference counting in Lean 4 because of its comparable simplicity and ease of interfacing with other languages, but greatly improved its performance and associated optimizations (Chapter 6): reference counting instructions are explicit in a dedicated intermediate representation and thus open to compile-time optimization, destructive updates of unique values have been extended to arbitrary Lean types, and Lean objects can be referenced from multiple threads without penalizing performance of single-threaded code.

The `meta` modifier too was clearly developed for the sake of metaprogramming. While its properties could be convenient for regular programming as well, the choice of keyword name as well as its transitive nature made this slightly awkward. Lean 4, as a general-purpose programming language, improved in this regard by removing `meta` and splitting its responsibilities into two separate concerns:

- The safe definition of external functions is delegated to the `opaque` command, which, in contrast to that of Lean 3, postulates the existence of a function given that its type is known to be inhabited. This can be done by providing an instance of the `Inhabited` typeclass or an explicit term of the specified type, which do not affect the runtime behavior of the function but merely guarantee the soundness of its existence.

- The use of unbounded recursion is covered by the `partial` modifier, which does not enforce a namespacing restriction like `meta` did. Instead, it uses `opaque` for its external interface: the user must show that the function type is inhabited and, like with external functions,

the function's implementation is completely opaque to Lean's logic. This is sufficient to ensure soundness: in a non-terminating function such as

```
partial def f (x : Nat) : Nat := 1 + f x
```

the mere existence of a constant `f : Nat → Nat` is not an issue since we know we could substitute any use of it with e.g. the identity function. However, we must not allow external definitions or theorems to extract the contradictory equation `f x = 1 + f x`.

2.2 Lean 3 as a Metaprogramming Language

In contrast to dedicated tactic languages such as Ltac [Delahaye, 2000] that are based on high-level primitives such as locating hypotheses by unification with a pattern term, Lean 3 metaprogramming is based on a comparatively low-level monadic API exposing the same data structures the base system is built on. This part is relatively unchanged in Lean 4 apart from the data structures being reimplemented in Lean and thus being even more accessible to metaprograms, so I will list the most important ones here:

- The *global context* or *environment* holds the top-level declarations defined so far as well as further metadata. The environment is implemented in what is commonly called the *LCF approach*[2]: it is an abstract data type to metaprogramming code that can only be extended with new declarations by an exported kernel function that rejects any input that is not well-typed in the environment before the extension, guaranteeing that the environment is consistent at all times even in the presence of arbitrary metaprograms.[3] The LCF

[2] Note, however, that like other dependently-typed provers, Lean does not use the LCF approach on the proof level: proofs are lambda terms according to the Curry-Howard correspondence, not abstract data types that can only be constructed by kernel inference rules.

[3] As long as the executed metaprograms are memory-safe, that is. In the extreme case, write access to the `/proc/self/mem` file is sufficient for breaking memory safety on Linux. Numerous external proof checkers for Lean 3 can be used to guard against such soundness

design of the environment directly shapes the design of Lean and similar proof assistants up to their surface syntax and even interaction on the user level. As the environment can only be extended one step at a time by new declarations that may only reference existing declarations, a Lean file is an ordered list of declarations where a declaration may only reference declarations lexically before it, and the file is processed in this order both in batch mode and interactively in the user's editor. The kernel and type theory allow a restricted set of mutually recursive declarations, but these are handled as an atomic declaration block both in the surface syntax and during processing. One distinctive advantage of this processing model is that *incremental* processing in the editor is relatively simple to implement: after a textual change, it is sufficient to restore the state of the environment at the point of the last declaration strictly before the change and continue processing from there on. As long as the environment is implemented as a persistent data structure and all involved operations are pure, it is guaranteed that this incremental approach is equivalent to reprocessing the file from the beginning.

The ability to access and extend the environment is what made Lean 3 a metaprogramming language. The remaining contexts described in the following are of great help when building metaprograms, especially tactics, but are not strictly necessary for synthesizing new declarations.

- The *local context* is the collection of currently free variables. As (well-typed) Lean terms directly occurring in top-level declarations are always closed terms, the only way to observe free variables in metaprograms is by introducing them yourself or "entering" an existing variable binding.

  More specifically, Lean uses a *locally nameless* term representation [McBride and McKinna, 2004]. Bound variables are represented as de Bruijn indices. While it is technically possible for a metaprogram to observe and manipulate the de Bruijn indices of bound variables directly, e.g. during structural recursion over a term, the preferred approach is to immediately convert such *un*bound vari-

issues resulting from implementation details instead of the core type theory.

ables (de Bruijn indices not contained within a corresponding lambda abstraction) into *free* variables that are references (via names) into the local context. For each binder type, there are monadic library functions that do this by choosing a fresh name for the free variable, then substituting it for the unbound index throughout the term. Thus brittle indexing arithmetics are hidden from the metaprogramming developer and replaced with simple unique named references. Free variables are furthermore distinguished from bound variables by storing additional data in the local context: a user-facing name and a type (the binding's domain). The former ensures we can pretty-print open terms correctly given the local context, while the latter does the same for type inference.

- The *metavariable context* similarly is the set of current *metavariables*, also called unification variables, existential variables, or simply *holes*. I will note that the name unification variable can be confusing in the context of Lean since they can be solved not only by unification but also other processes such as typeclass inference or tactic execution (see next item), whereas the latter two names accurately suggest that all metavariables are eventually substituted by concrete terms if elaboration succeeds. The kernel type checker does not use a metavariable context and will reject remaining metavariables just like free and unbound variables. A metavariable term itself is again a simple reference by unique name into the corresponding context. In addition to a user-facing name and a type, an entry in the metavariable context also contains a reference to the local context in which the metavariable was created, as assigning a term containing a free variable to a metavariable whose original context did not include it would lead to an unbound binding and should be rejected immediately.
- Finally, in the special case of tactic metaprogramming, we have an additional *tactic context* that keeps track of the list of goals. A goal is a metavariable that stands for the eventual proof of the statement expressed by the metavariable's type. The initial tactic context when executing a tactic block is a single fresh metavariable of the type expected at the point of the tactic block. Tactics may generate additional goals by assigning a term containing multiple new metavariables to the current goal(s). It is ultimately up to the

tactic's implementation whether it registers additional metavariables in the tactic context as new goals or not (in which case they are usually filled by unification while proving other goals), as there are sensible use cases for both behaviors.

With these ingredients, we can implement higher-level functions such as the `assumption` tactic from [Ebner et al., 2017] that solves a goal if there is a hypothesis that matches the goal's conclusion:

```
meta def find : expr → list expr
    → tactic expr
| e []        := failed
| e (h :: hs) :=
  do t ← infer_type h,
     (unify e t >> return h) <|>
     find e hs
```

```
meta def assumption : tactic unit
    :=
do { ctx ← local_context,
     t   ← target,
     h   ← find t ctx,
     exact h }
<|> fail "assumption tactic
    failed"
```

The `tactic` monad (introduced as a `meta` constant) gives us access to all described contexts, as well as error handling. We start by retrieving the local context as a list of free variables as well as the *target* type, that is, the conclusion of the first goal in the tactic context. A simple recursive function `find` then locates the first hypothesis in the local context whose type is unifiable with the target type, or else fails. If it found a hypothesis `h`, we close the goal via the `exact` tactic function that assigns the given term to the goal's metavariable and removes it from the list of goals. Throughout, we use standard monadic notation such as >> for sequencing, `<|>` of the `alternative` typeclass for backtracking on errors, and a Haskell-like `do` notation, which we will look at in much more detail in the context of Lean 4 in Chapter 5.

The last missing ingredient is a way to bridge between the standard object level and the meta-level in order to actually use tactics in practice. The `by` keyword does so by accepting a tactic block, i.e. a list of tactic invocations, in term position. It elaborates to the term assigned to the initial goal by after running the tactic block. The tactic block is translated to a monadic sequence of the involved tactics, compiled to bytecode, and finally interpreted as described in the previous section.

```
lemma simple (p q : Prop) (h₁ : p) (h₂ : q) : q :=
by assumption
```

Discussion. Lean 3's low-level tactic metaprogramming framework set the foundation for a healthy ecosystem of user-defined tactics: at the time of writing, the mathlib documentation [Mathlib, 2022] lists 117 tactics contributed by various authors compared to 89 tactics implemented in Lean itself. We were positively surprised to see that authors of these included not only seasoned functional programmers but also some mathematicians with no prior exposure to monadic programming. While one might imagine a higher-level tactic language in the style of Ltac or others built on top of this foundation that would not require knowledge of monads for writing simple automation, this need did not seem to manifest itself in the Lean community so far, though it might also have proved difficult with Lean 3's limited syntactic extension capabilities compared to Lean 4. For comparison, a succinct implementation of `assumption` in Ltac might look like

```
match goal with
| H : ?A |- ?A => apply H
end.
```

I have already mentioned in the previous section that the performance of the Lean 3 interpreter prevented the implementation of more ambitious automation as Lean 3 user tactics. Ironically, in one case Lean 3 turned out to do more compilation than was advisable: the translation of a `by` block to a term of type `tactic unit`, which then went on to be compiled and interpreted, was simple to implement, but introduced unnecessary compilation overhead for a monadic program that is usually run exactly once.[4] We fixed this in Lean 4 by essentially introducing a dedicated tactic block interpreter via the Lean 4 macro system that directly consumes the surface syntax without further translation steps, as we shall see later in Section 4.6.

While the section title references "Metaprogramming", we focused on the special case of tactic metaprogramming so far, as that was the clear focus in Lean 3. In fact, other entry points for metaprograms I added to Lean 3, user-defined notations and commands, heavily suffered from limited access to the parser and elaborator compared to their built-in C++

[4] with the exception of tactic blocks inside of combinators such as `repeat` that may be run more than once, but even then the run time of the tactic block itself should be negligible compared to that spent inside the invoked tactics.

counterparts. Only with the completely reworked system described in Chapter 4 did we achieve a satisfactory level of expressiveness for these kinds of extensions, indeed a level of expressiveness that is equal to that of builtins by erasing the distinction between builtins and user extensions.

2.3 Advanced Tactic Programming

On top of the primitives discussed in the previous section, Lean 3 provides *quotations* to construct kernel terms: `` `(t) `` is syntax for the reflected representation of a term `t` in the inductive type `expr` of kernel terms reflected into the Lean language. This makes it easy to pass compound terms to tactics, as in the following example where the goal is closed by passing a term of the appropriate type to the `apply` tactic.

```
example : true ∧ true :=
by do exact `(and.intro trivial trivial)
```

As `expr` values are fully elaborated kernel terms, elaboration of `t` must occur when the quotation is elaborated, strictly before the tactic is executed. This can be limiting if successful elaboration of the quoted term is dependent on further context supplied by the tactic, e.g. unification with the goal type by `exact` or access to (tactic-)local variables. Note that as a quotation is a pure term of type `expr`, it does not have access to e.g. the current metavariable context in order to register new metavariables to be solved by later unification.

In order to provide tactics with this extra flexibility, Lean 3 exposes the Lean type of unelaborated *pre-terms* `pexpr`, a reflection of the internal type (identical to that of `expr` on the C++ side) passed from the parser to the elaborator, together with its own quotation syntax ``` ``(t) ```. Pre-terms can be turned into full terms by the reflected elaboration function `to_expr : pexpr → tactic expr`.

```
example (p : Prop) : p → p :=
by do to_expr ``(id) >>= exact
```

As `to_expr` lives in the `tactic` monad, it can extend the metavariable context with a new hole for `id`'s type parameter, which is subsequently solved to `p` when the `to_expr` output is unified with the current goal by `exact`.

While `pexpr` originally was a completely opaque type to Lean code as described in [Ebner et al., 2017], I later unified it with `expr` into a new `expr` type parameterized over a `(elaborated : bool)` flag[5].

```
meta inductive expr (elaborated : bool := tt)
| ...
meta def pexpr := expr ff
```

Not only does this redefinition of `pexpr` make it inspectable by Lean code, it also allows sharing of common operations on both pre-terms and terms as in the following, while avoiding the C++-side issue of type confusion between terms and pre-terms via the distinguishing new type parameter.

```
meta def get_app_fn : expr elab → expr elab
| (app f a) := get_app_fn f
| a         := a
```

While elaboration of pre-term quotations is delayed to the point where they are converted to full terms, name analysis is still done early. Thus Lean 3 would have found a typo of `id` above even before the tactic was run, which is especially useful when used in a reusable tactic that is not immediately run. On the other hand, we just as often *want* to reference bindings such as local hypotheses in a quotation, in which case name resolution cannot be done by the parser but must be part of elaboration of the pre-term. The original ` ``(t) ` did just that, but to support the more eager parser name resolution when possible, I moved delayed name resolution to a new ` ```(t) ` syntax (such that increasing number of backticks signifies decreasing number of compile-time checks).

```
example (p : Prop) : p → p ∨ false :=
by do intro `h, to_expr ```(or.inl h) >>= exact
```

Here `` `h `` is a literal for Lean 3's `name` type of reflected names for declarations, free variables, etc., which is passed to the `intro` tactic to introduce a hypothesis `h : p`, which is then referenced by the quoted term to eventually solve the remaining goal.

Finally, I will discuss creating *user-defined attributes* as another example of advanced metaprogramming that I implemented, as well as further

5 `tt` and `ff` are Lean 3's Boolean *true* and *false* values, renamed to the more prevalent `true` and `false` in Lean 4.

```
meta def mk_name_set_attr (attr_name : name) : tactic unit :=
do let t := `(caching_user_attribute name_set),
   let v := `({name     := attr_name,
               descr    := "name_set attribute",
               mk_cache := λ ns, pure (name_set.of_list ns),
               dependencies := [] }
               : caching_user_attribute name_set),
   add_meta_definition attr_name [] t v,
   register_attribute attr_name

meta def get_name_set_for_attr (attr_name : name) : tactic name_set :=
do let cnst := expr.const attr_name [],
   attr ← eval_expr (caching_user_attribute name_set) cnst,
   caching_user_attribute.get_cache attr
```

Figure 2.2: Generating new attributes using metaprogramming.

forms of reflection necessary for their effective use. The code in Fig. 2.2 declares a new attribute of a given name that caches a set of declarations the attribute has been applied to. Metaprograms like `mk_name_set_attr` that manipulate the current environment can be run via `run_cmd`.

```
run_cmd mk_name_set_attr `no_rsimp
attribute [no_rsimp] or.comm
run_cmd get_name_set_for_attr `no_rsimp >>= trace  -- {or.comm}
```

This attribute is used in the implementation of `rsimp` in [Ebner et al., 2017] for blacklisting simplification lemmas, for which it relies on the fast membership checking of the native-backed `name_set` type. The `name_set` will be cached as long as the set of declarations the attribute or one of the `dependencies` attributes (of which there are none in this case) have been applied to remains unchanged.

The implementation of `mk_name_set_attr` works by dynamically constructing a term of type `caching_user_attribute name_set`, adding it to the environment as a new declaration, and registering that declaration in the attribute manager. When implementing the retrieval function `get_name_set_for_attr`, we encounter a new problem: Given a reflected term that describes a value of type `caching_user_attribute name_set`, we would like to evaluate it at run time so that we can pass it to the following function.

```
meta constant caching_user_attribute.get_cache {α : Type} :
    caching_user_attribute α → tactic α
```

As can be seen in Fig. 2.2, we can solve this issue by calling the primitive `eval_expr`:

```
eval_expr (α : Type) [reflected α] (e : expr) : tactic α
```

The type class parameter `[reflected α]` is necessary to provide a safe implementation (but see discussion below): in order for `eval_expr` to check that `e` does indeed describe a term of type α, it also needs a reflected description of the latter. (Indeed, a value of type `Type`, like α, does not even have a run-time representation.) The `reflected` type class is an opaque container for a term reflecting a known value. It is special in that the elaborator will synthesize the value of a parameter `[reflected α]` from the term passed for α, as long as this term is either closed (as in Fig. 2.2) or its free variables have `reflected` instances themselves. We actually have made implicit use of the latter feature in `mk_name_set_attr` — we were able to use the local variable `attr_name` in the quotation because the elaborator was able to find an instance of `reflected attr_name`. We did not need to demand such an instance by parameter because for simple types such as `name` we can construct a universal instance via dependent pattern matching and the primitive `reflected.subst ... : reflected f → reflected a → reflected (f a)`.

```
meta def reflect {α : Type} (a : α) [h : reflected a] : reflected a
    := h

meta instance name.reflect : Π (n : name), reflected n
| anonymous         := reflect anonymous
| (mk_string  s n) := (reflect (λ n, mk_string  s n)).subst
    (name.reflect n)
| (mk_numeral i n) := (reflect (λ n, mk_numeral i n)).subst
    (name.reflect n)
```

Only for reflections of non-computational types such as sorts, propositions, and types containing these do we need to pass along an instance as with `eval_expr`.

Discussion. Out of all the Lean 3 features described in this chapter, the ones from this section have seen the most user-visible changes in

Lean 4. Quotations had to change fundamentally since we removed abstract syntax tree/kernel-like pre-terms in Lean 4 in favor of concrete syntax trees faithfully representing the surface language as a foundation of Lean 4's macro system that I will discuss in much more detail in Chapter 4. The most significant restriction of Lean 3's pre-terms compared to Lean 4's syntax trees is already apparent in the name — while allowing convenient quoting of surface-level syntax and some introspection for unelaborated *terms*, there is no corresponding representation of other syntactic "categories" such as top-level commands, tactics, universe levels, and so on. In fact, Lean 3 does provide a special-purpose syntax `` `[t, ...] `` for quoting a sequence of tactics such that repeated parts of a tactic script can be factored out into a new tactic, but because of the lack of internal abstract representation, the sequence is translated immediately to a term of type `tactic unit`, preventing any introspection of the original sequence. This is in great contrast to Lean 4's tactic interpreter I have mentioned above consuming surface-level tactic syntax as is; we will come back to other issues with Lean 3 tactic quotations in Section 4.6.

For top-level commands in particular, Lean 3 does not provide any quotation mechanism at all, preventing users from reusing surface-level command syntax unless it is specifically exposed as a metaprogramming function. The reason for this design in Lean 3 is that it must mirror the internal implementation, in which pre-terms are used as a common interface between the parser and the term elaborator, but commands are implemented as direct function calls from the parser into a respective subsystem such as the inductive elaborator, with no common syntactic representation of commands present. Only a complete redesign of the elaborator and these interfaces in terms of always consuming a common syntax tree type at any elaboration level, which we have done in Lean 4 as part of its reimplementation in the Lean language, was able to lift these restrictions and guarantee convenient metaprogramming in any syntactic category.

Notably, there is no quotation syntax for *elaborated* terms in Lean 4 either, mostly because the need for it did not come up when implementing the generic tactics and other metaprograms included with Lean. However, it may still be useful for more domain-specific tactics developed by users, and

so I am excited about the `quote4` library[6] by Gabriel Ebner (the first author of [Ebner et al., 2017]), which advances the state of the art compared to Lean 3 by implementing *type-safe* expression quotations, making great use of Lean 4's extended metaprogramming capabilities as in the following example taken from the library.

```
def mkPairwiseEquality {α : Q(Sort u)} : List Q($α) → Q(Prop)
  | [a, b]        => q($a = $b)
  | a :: b :: cs => q($a = $b ∧ $(mkPairwiseEquality (b :: cs)))
  | _             => q(True)
```

Given a list of quoted terms of the same type α (*spliced* into the type-level quote `Q` via the `$` operator), `mkPairwiseEquality` produces a quoted proposition representing the pairwise equality of the terms (using the term-level quote `q` such that, slightly simplified, we have `q(t) : Q(α)` whenever `t : α`). These expression quotations can thus be seen as incorporating Lean 3's `reflected` typeclass, which does not exist as a built-in in Lean 4, and whose ad hoc extension of the typeclass system we regretted in hindsight. While the `quote4` library does not guarantee to the same degree as `reflected` that the carried term indeed was quoted and not constructed manually, this is not as useful in practice as it may first seem. In particular, there is no guarantee that in a call `eval_expr bool e`, the type `bool` from the *compile-time* environment in which the metaprogram is elaborated and compiled is identical to `bool` in the *run-time* environment carried by the `tactic` monad, so type confusion is still a possibility.

User-defined attributes finally do exist in Lean 4 as well, though in a variety of shapes optimized for various use cases that came up during the implementation of Lean 4 itself, as built-in attributes now use the same API. Attribute parameters are stored as syntax trees as usual in favor of the previous `reflected`-based design.

2.4 Interactive Proving

As we have seen, Lean 3 lets us execute arbitrary terms of the tactic monad using the `by` keyword. To help construct such tactic terms, we have provided common syntactic sugar such as the `do` notation and quotation

[6] `https://github.com/gebner/quote4`

literals. While this *programmatic* view is adequate for defining new tactics, end users are accustomed to more *declarative* representations, such as a sequence of tactics, each with its own convenient syntax, that can be stepped through and inspected interactively.

In order to gain this level of convenience, I introduced a special variant of `by` that takes a single tactic name and parses its arguments according to syntax rules encoded in the tactic's signature.

```
by super with mul_assoc mul_one
```

This is desugared to the following regular term, in which the extra parentheses deactivate the special handling:

```
by (super [] [`mul_assoc, `mul_one])
```

For executing a sequence of tactics, we may use `begin ... end` blocks.

```
begin intro h, refine or.inl h end
```

This proof is equivalent to the following expanded form:

```
by do intro `h, refine ```(or.inl h)
```

Using `begin ... end` has the added benefit that Lean will record the proof state at the beginning of each tactic so that it can be inspected by editors.

Let us now take a look at the detailed signature of `super`:

```
meta def tactic.interactive.super (extra_clause_names : parse
    ident*) (extra_lemma_names : parse with_ident_list) : tactic
    unit
```

By default, when parsing a `by` or a `begin ... end` block, the parser will look for interactive tactics in the namespace `tactic.interactive`. Users can switch to a different namespaces by using e.g. `begin[smt] ... end`, which will instead search the namespace `smt_tactic.interactive`.

When parsing the arguments of an interactive tactic, we handle parameters of type `parse p` specially by giving over control to the user-defined parser `p`. In this case, we parse a sequence of identifiers, optionally followed by the keyword `with` and another sequence of identifiers. Parsers can be built from a few exported primitives, which are described in Figure 2.3, using the standard applicative and monadic combinators. I later reused the same parser monad to implement the previously mentioned user-defined syntax at the expression level and commands at the top level:

```
/-- An opaque type representing Lean's native parser. -/
meta constant parser : Type → Type
meta constant parser.monad : monad parser
meta constant parser.alternative : alternative parser
/-- Parse an identifier and produce it as a quoted name. -/
meta constant ident : parser name
/-- Parse the given token. `tk` must be a registered token. -/
meta constant tk (tk : string) : parser unit
/-- Parse an unelaborated expression using the given right-binding
    power. -/
meta constant qexpr (rbp := std.prec.max) : parser pexpr
/-- Parse `with` followed by a list of identifiers. -/
meta def with_ident_list := (tk "with" *> ident*) <|> return []
meta def parse {α : Type} [has_reflect α] (p : parser α) : Type := α
```

Figure 2.3: Exposing Lean's native parser as a monadic parser combinator. `*>` and `*` are notations for the `applicative` combinators `seq_right` and `many`, respectively. Note that e.g. `parse with_ident_list` is definitionally equal to `list name`, so that as far as the definition of super is concerned, the argument is just a list of names.

```
@[user_notation] meta def format_macro (_ : parse $ tk "format!") (s
    : parse expr_p) : parser expr := ...

#eval format!"1 + 1 = {1 + 1}" -- "1 + 1 = 2"

@[user_command] meta def coinductive_cmd (meta_info :
    decl_meta_info) (_ : parse $ tk "coinductive") : parser unit :=
    ...

coinductive all_stream {α : Type u} (s : set α) : stream α → Prop
| step : ∀{a : α} {ω : stream α}, a ∈ s → all_stream ω →
    all_stream (a :: ω)
```

Discussion. The introduction of the user-extensible parsing interface described in this section greatly enhanced the expressivity of user-defined tactics. Out of the 117 mathlib tactics mentioned previously, many make creative use of this interface, such as

```
meta def opt_dir_with : parser (option (bool × name)) :=
(tk "with" *> ((λ arrow h, (option.is_some arrow, h)) <$> (tk
    "←")? <*> ident))?

meta def set (h_simp : parse (tk "!")?) (a : parse ident) (tp :
    parse ((tk ":") *> texpr)?) (_ : parse (tk ":=")) (pv :
    parse texpr) (rev_name : parse opt_dir_with) : tactic unit
    := ...
```

User-defined commands were also embraced by the community with 32 uses in mathlib at the time of writing, whereas user-defined notations were only used three times for variations of the above formatting macro. Much as with metaprogramming functions, the main limitation was that only a select few parsing functions were exposed from the C++ implementation, whereas in Lean 4 users have access to all intermediate parsers used for implementing the syntax of Lean in itself. Thus, while Lean 3 allowed extensions such as the `coinductive` parser above by specifically exposing a parser that parses the standard `inductive` syntax without the leading keyword, more invasive variations of built-in syntax were not realistic because of the implementation language barrier.

While the `parse` type offered a simple-to-use, strongly-typed interface for extending Lean 3's surface syntax, it did so in a very roundabout way, necessitated by integration into the existing design and code: whenever a tactic was parsed, for each of its `parse p` parameters `p` would be compiled to bytecode and interpreted to a `parser` runtime value, which would then be called to parse the corresponding tactic argument, yielding a value that would be reflected into a kernel term (via the `has_reflected` typeclass instance embedded in the `parse` application, itself compiled and interpreted on the fly) that could be embedded into the pre-term representation of the tactic block, eventually to be evaluated back into its runtime value when the tactic was actually run. But even with all these back-and-forth transformations and the overhead they introduced, the `parse` design still conflated parsing and evaluation, providing no concrete or abstract syntax tree at any of these stages that could be used by tooling such as refactoring of tactic scripts. This is in great contrast to how parsing and interpretation is strictly separated in our clean-slate design of Lean 4 including, as we will see in Section 4.6, in the case of tactic scripts. Lean 4 reduces the

transformations to a minimum by parsing tactic invocations once into a concrete syntax tree by the tactic's parser, which has been precompiled into an intermediate representation or native code (Section 3.6). At tactic execution time, the syntax tree is then consumed by the tactic interpreter mentioned in the previous section and tactic arguments are passed by it as is without further transformation to the specific tactic macro. In place of the strongly-typed values returned by `parse`, Lean 4 users can utilize extensive syntax patterns (Section 4.3.1) to analyze tactic arguments in a structured way.

2.5 Related Work

Metaprogramming in the form of tactic programming is present in many different theorem provers, using a variety of designs, languages, and implementations. I have mentioned the Ltac tactic language [Delahaye, 2000, Delahaye, 2002] of Coq before as an example of a quite different approach from the Lean 3 & 4 tactic languages. Ltac is a domain-specific language, i.e. in fact not identical to either Coq's implementation language or its language of terms and propositions. Apart from the syntax, the semantics as well are quite different from a generic functional programming language: the unification-based pattern matching is more akin to that of Prolog, and built-in backtracking support provides flexible control flow tailor-made for automation. Eisbach [Matichuk et al., 2016], a more recent tactic language for Isabelle, follows the same principles. As mentioned above, should the need for tactic programming at this abstraction level arise in the Lean community, I believe that Lean 4 gives us all the tools we need to emulate the desired primitives, ideally lightly embedded in the existing monadic tactic framework and notations instead of as a separate language.

A design aspect shared by both Ltac and Lean's tactic framework is the lack of compile-time type information on the proposition to be proved — the `tactic` type does not statically tell us anything about what goals it is applicable to, or even whether it solves any goals at all or introduces new ones or otherwise manipulates the list of goals. While this limitation is not all that relevant when writing tactic scripts that are executed immediately anyway, a more statically typed approach can be beneficial when implementing reusable automation, similarly to how statically typed languages can ease (but also complicate) development compared to dynamically

typed languages.[7] Mtac [Ziliani et al., 2015] and Mtac2 [Kaiser et al., 2018] follow this approach by use of a metaprogramming monad `M` such that running a metaprogram of type `M A`, if successful, is guaranteed to return a Coq term of type `A`. As with Ltac, this approach requires a special interpreter for metaprograms since a type `A` is usually inhabited not by terms but a more restricted class of *values* at run time. In fact, the related MetaCoq project [Sozeau et al., 2020] defines a similar interpreted monad for well-typed metaprogramming, but then falls back to an untyped monad similar to Lean's for the purpose of native code generation.

The metaprogramming approach taken by Lean 3 is quite similar to that of *elaborator reflection* [8] in Idris [Christiansen and Brady, 2016] published shortly before. There are some technical and performance differences discussed in [Ebner et al., 2017], but the primary difference, I would argue, is in presentation. Lean 3, as discussed, was focused primarily on tactic metaprogramming and provided rich syntax sugar for tactic scripts so that it did not actually feel like monadic metaprogramming. Lean 4 inherits and extends this tactic domain-specific language while greatly improving support for other forms of metaprogramming, in particular in the form of macros. Elaborator reflection in Idris on the other hand is strictly focused on the generic metaprogramming use case, with no additional syntax sugar hiding the monadic details other than `do` notation. This difference in focus is not surprising given that Idris is described primarily as a (dependently typed) programming language, not an interactive theorem prover.

[7] In particular, for implementing general-purpose automation such as a simplifier or generic proof search tactic, an untyped goal representation appears to be a better fit.

[8] Note that while the term "elaborator reflection" can reasonably be applied to Lean 3's implementation of metaprogramming as well, though perhaps misleadingly so because of its primary focus on tactic metaprogramming in favor of more generic elaboration interfacing, this does not seem appropriate for describing Lean 4's implementation as there is no primitive monad exporting functions from an underlying implementation language in Lean 4; elaboration instead is a regular function completely defined in the same Lean language and phase that tactics and metaprograms are written in.

What's new?
– Conor McBride, *Epigram: Practical Programming with Dependent Types* [McBride, 2005]

3 An Overview of Lean 4

Lean 4 is a reimplementation of the Lean interactive theorem prover in Lean itself, addressing many shortcomings I described in the previous chapter. It is the first version of Lean that can be described as both a theorem prover and a general-purpose programming language, opening up many new avenues for metaprograms as well as regular programs written in it. Its development is the main output of my Ph.D. as joint work with Leonardo de Moura, spanning from my internship at Microsoft Research in my second year during which we set the plan for many parts of the architecture described below to the first stable release of Lean 4 that will likely happen not too long after the completion of this thesis.

Contributions. In this chapter, I document the design and implementation of the central components of Lean 4 that I made significant contributions to: the module system, parser, elaborator, code generator, and user interface. I also give the first formal description of the type theory of Lean 4, building on top of previous work for Lean 3 [Carneiro, 2019].

Acknowledgements. This chapter is loosely based on the Lean 4 system description by Leonardo de Moura and me [de Moura and Ullrich, 2021], though it can more accurately be seen as complementary: I focus on topics that are not described in detail in that work nor in the remainder of this thesis. The implementation of the kernel changes and most parts of the elaborator and code generator are due to Leo, while a significant amount of the initial language server were developed by Marc Huisinga and Wojciech Nawrocki.

3.1 Architecture

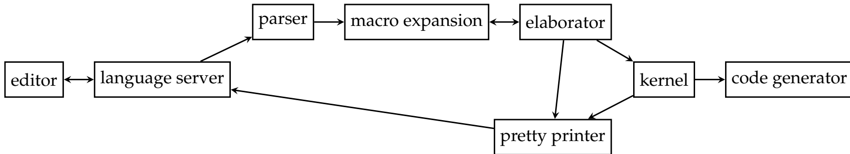


Figure 3.1: Schematic overview of the components of the Lean 4 implementation and control flow of a single processing step

The implementation of Lean 4 is made up of different components that the following sections will take a closer look at and whose interaction is sketched in Fig. 3.1: in interactive use of Lean inside an editor, the language server (Section 3.7) is the component that interfaces with the user's editor. In command-line ("batch") mode, it is replaced with a simple text interface that reads in a given file. In either case, we accept input as a string, which is processed into a concrete syntax tree by the parser (Section 3.4). Not pictured here is additional input that is loaded from other (already compiled) Lean files specified as *imports*, which is the task of the module system (Section 3.3). The concrete syntax tree describes a program in the *surface language* of Lean, i.e. the language exposed to users. It may contain syntax sugar and more complex *macros* that must be unfolded, which we will talk about in much more detail in Chapter 4, and ultimately has to be translated to the *core language* of Lean by the elaborator (Section 3.5), filling in many details left implicit by the user. If successful, the core language output is then checked by the kernel (Section 3.2), which is an independent component in order to minimize the Trusted Code Base. Any errors returned by the elaborator or kernel are reported to the user via the *pretty printer*, which is responsible for translating the core language of the erroneous parts back to readable surface language. If there are no errors and the elaborated command is a definition, it is also passed to the *code generator* (Section 3.6) in order to create executable code, both for extracting stand-alone programs and for extracting metaprograms that are then run as part of the respective component for subsequent input. Except for the kernel and code generator, these are the user-facing and -interacting components considered the *frontend*.

3.2 The Kernel and Type Theory

The desire for a small core theory and kernel, particularly when compared to other implementations of dependent type theory, that is simple to reimplement in external checkers remains an important aspect of Lean 4. Nevertheless, we took the Lean 4 rewrite as a chance to reevaluate and enrich some details of the type system.

A simple addition is *opaque* declarations already mentioned in Section 2.1. A declaration opaque $c_{\bar{u}} : \alpha := e$ is type-checked just like a definition, but afterwards is irreducible like an axiom — the body e exists merely as a witness of α being inhabited. Building on top of the framework and notations of [Carneiro, 2019] who formalized the type theory of Lean 3, we can formally state this by introducing the exact same typing and conversion rules Carneiro gives for axioms (called constants in Lean 3) for any such $c_{\bar{u}}$ that fulfills $\vdash e : \alpha$ where the universe variables of e and α are contained in $\bar{u}$.

$$\frac{}{\vdash c_{\bar{\ell}} : \alpha[\bar{\ell}/\bar{u}]} \qquad \frac{\ell_1 \equiv \ell'_1 \ \ldots \ \ell_n \equiv \ell'_n}{\vdash c_{\bar{\ell}} \equiv c_{\bar{\ell}'}} \qquad \frac{\ell_1 \equiv \ell'_1 \ \ldots \ \ell_n \equiv \ell'_n}{\vdash c_{\bar{\ell}} \Leftrightarrow c_{\bar{\ell}'}}$$

The same argument Carneiro gives for definitions being a conservative extension applies to opaque declarations as well: we can always replace them by their body in a correct typing derivation.

3.2.1 Internalizations

Various previously derivable expressions have been added as primitives for optimizing the implementation without changing expressivity of the theory or observable semantics of the implementation: structure projections and literals for strings and natural numbers. Literals are straightforward extensions where terms previously expressed using nested applications are now represented by a single, optimized runtime value. For example, a syntactic literal of type `Nat` is now represented in the core term language by an arbitrary-precision numerical data structure instead of a series of applications representing a binary encoding of the number. We can formalize this by introducing a new kind of expression l_n for any (meta-level) natural number $n \in \mathbb{N}$.

$$e ::= \ldots \mid l_n \qquad \Gamma \vdash l_n : \mathsf{Nat}$$

Here Nat is an inductive type with zero and successor constructors $z_{\mathsf{Nat}} : \mathsf{Nat}$ and $s_{\mathsf{Nat}} : \mathsf{Nat} \to \mathsf{Nat}$ like in [Carneiro, 2019], except that I use the name Nat to avoid confusion with the meta-level set $\mathbb{N}$. The implementation must ensure that the type Nat in the current environment indeed has this shape. In particular, it must not be an empty type for the above typing rule to be consistent.

In order to ensure that there is no observable change, we add conversion rules translating between the literal and constructor representation.

$$\Gamma \vdash l_0 \equiv z_{\mathsf{Nat}} \qquad \Gamma \vdash l_{n+1} \equiv s_{\mathsf{Nat}}\ l_n$$

We can use the same rules for the algorithmic version of conversion checking described by Carneiro, which notably lacks the transitivity rule of the formal conversion rules, though the great advantage of the new representation is that we can additionally provide more optimized reduction rules such as

$$\mathsf{mul}\ l_n\ l_m \rightsquigarrow l_{nm}$$

for the standard definition of mul (which again the implementation must verify), replacing its recursive reduction with optimized computation in the arbitrary-precision library (which therefore becomes part of the *Trusted Code Base*). String literals similarly are represented by a single, contiguous byte array (a UTF-8 encoding) and come with rules for translating from and to the constructor form. A compact representation of strings is especially important for writing programs in Lean, which are still elaborated to the core term language and checked by the kernel.

Finally, the third expression shape that has been internalized as a primitive is *structure projections*. For a non-mutual inductive type P with non-fixed parameters (*indices*) a and a single constructor c with parameters or *fields* b and type τ, written

$$P = \mu t : (\forall a :: \alpha.\ \mathsf{U}_\ell).\ (c : \forall b :: \beta.\ \tau)$$

in Carneiro's framework where :: represents a *telescope* of dependent binders, we can define the projection function π_i^P extracting the i-th field as

$$\pi_i^P \ := \ \mathsf{rec}_P\ C\ (\lambda b :: \beta.\ b_i)$$

$$\text{where } C = \lambda a :: \alpha.\ \lambda z : P\ a.\ \beta_i[(\pi_j^P\ a\ z)/b_j]_{j<i}$$

In the recursor *motive* C, which describes the result type, we recursively need to substitute preceding fields b_j for their projections in case β_i is dependent on them. Note however that π_i^P is not type correct if we try to project a (non-index) field whose type lives in a data-relevant universe $\mathsf{U}_{\ell'+1}$ out of a structure in the proof-irrelevant universe $\mathbb{P}$: indeed if we have $\ell = 0$, Carneiro's recursor typing rule tells us that C must map into $\mathsf{U}_0 = \mathbb{P}$ as *large elimination* is not applicable in this case.

As an example, for the dependent pair type

```
structure Sigma (α : Type u) (β : α → Type u) where
  a : α
  b : β a
```

or, formally,

$$\Sigma \;:=\; \lambda\alpha : \mathsf{U}_{\ell+1}.\, \lambda\beta : \alpha \to \mathsf{U}_{\ell+1}.\, \mu\mathsf{S} : \mathsf{U}_{\ell+1}.\, (\mathsf{mk} : \forall a : \alpha.\, \beta\, a \to \mathsf{S})$$

we have

$$\begin{aligned}
\pi_1^{\Sigma\,\alpha\,\beta} &\equiv \mathsf{rec}_{\Sigma\,\alpha\,\beta}\,(\lambda z : \Sigma\,\alpha\,\beta.\, \alpha)\,(\lambda a : \alpha.\, \lambda b : \beta\, a.\, a)\\
\pi_2^{\Sigma\,\alpha\,\beta} &\equiv \mathsf{rec}_{\Sigma\,\alpha\,\beta}\,(\lambda z : \Sigma\,\alpha\,\beta.\, \beta\,(\pi_1^{\Sigma\,\alpha\,\beta}\, z))\,(\lambda a : \alpha.\, \lambda b : \beta\, a.\, b)
\end{aligned}$$

Lean 3 autogenerates projection definitions `Sigma.a` and `Sigma.b` from the above terms.

```
def Sigma.a (z : Sigma α β) : α := ...
def Sigma.b (z : Sigma α β) : β (Sigma.a z) := ...
```

However, as the term size of every π_i^P is linear in the number of fields and there are as many projections as fields, this in sum makes the time and space overhead of declaring a structure type quadratic in the number of fields, leading to performance issues with big structures in practice. Thus Lean 4 introduces projections as primitive terms of constant size.

$$e \;:=\; \ldots \mid \pi_i^P\, e$$

The proper typing rule can be derived by substituting the above definition of π_i^P into the rules of [Carneiro, 2019]. Note however that the primitive π_i^P does not take the index values a as explicit parameters anymore as they can be inferred from the only argument.

$$\frac{\Gamma \vdash z : P\, a \qquad \ell \geq 1 \vee \Gamma \vdash \beta_i[(\pi_j^P\, z)/b_j]_{j<i} : \mathbb{P}}{\Gamma \vdash \pi_i^P\, z : \beta_i[(\pi_j^P\, z)/b_j]_{j<i}}$$

$$\text{where } P = \mu t : (\forall a :: \alpha.\ \mathsf{U}_\ell).\ (c : \forall b :: \beta.\ \tau)$$

We finally add formal and algorithmic reduction and congruence rules.

$$\frac{\Gamma \vdash \pi_i^P\ (c_P\ b) : \xi}{\Gamma \vdash \pi_i^P\ (c_P\ b) \equiv b_i} \qquad \frac{\Gamma \vdash e \equiv e'}{\Gamma \vdash \pi_i^P\ e \equiv \pi_i^P\ e'}$$

$$\frac{}{\Gamma \vdash \pi_i^P\ (c_P\ b) \rightsquigarrow b_i} \qquad \frac{\Gamma \vdash e \Leftrightarrow e'}{\Gamma \vdash \pi_i^P\ e \Leftrightarrow \pi_i^P\ e'}$$

One might also expect rules for compatibility with the recursor representation of a projection, but fortunately the conversion rule introduced in the next subsection will make these unnecessary.

User-facing projection functions in the structure's namespace are still generated on top of these primitive projections, but the total overhead is linear as long as the sum of term sizes of the projections' *types* is linear, i.e. there are not too many type dependencies between them, which is not usually a concern in practice.

Primitive projections are also supported in Coq using the flag `Primitive Projections`, which is documented with the note "the design of primitive projections is still evolving" [Inria, CNRS and contributors, 2021]. The flag disables the pattern matching primitive for affected types, while in Lean 4 the recursor is unaffected.

3.2.2 η-Conversion for Structure Types

Perhaps the most noticeable change for users is the addition of η-conversion for structures. This feature addresses various issues encountered by mathematicians in Lean 3 that can be summarized as "unpacking and re-packing something should yield an identical (definitionally equal) object". More precisely, applying the constructor of a structure type to the projections of a value of that structure in the right order should be definitionally equal to the original value, as with the product type in the following trivial example.

```
example (p : α × β) : (p.fst, p.snd) = p := rfl
```

Note that the reverse direction of "packing and unpacking" as in `(x, y).fst = x` has always been definitionally true in Lean as it does not require eta but simple projection reduction.

The lack of eta conversion for the product type in Lean 3 led to people avoiding eagerly destructuring pairs as in `fun (a, b) => ... a ... b ...` in favor of applying projections as late as possible as in `fun p => ... p.fst ... p.snd ...` in order to not block reduction unnecessarily, even though the former style usually allows for more meaningful variable names. In Lean 4, both styles are now definitionally equal and thus the more readable variant can be preferred without drawbacks.

```
example (p : α × β) :
  (match p with | (a, b) => (b, a)) = (p.snd, p.fst) := by rfl
```

Perhaps surprisingly, though, structure eta is even more useful for the trivial case of a structure with exactly one field, which is a common pattern in Lean similar to Haskell's `newtype`s that can be used to give a new interpretation to an existing type such as in the construction of the opposite category C^{op}:

```
structure Opposite (C : Type u) where op ::
  unop : C

instance [Category C] : Category (Opposite C) where ...
```

Here `Opposite` is a structure with constructor `op` and a single field `unop` that form a bijection between objects of a category `C` and its opposite category. The structure wrapper ensures we cannot confuse elements of `C` and `Opposite C`, while the addition of eta conversion ensures composing `op` and `unop` in either order is a definitional identity. In particular, every element `y : Opposite C` is known to be of the form `op x` for some `x : C`, again ensuring we do not unnecessarily unblock reduction. Note though that `x` and `op (op x)` are still distinct elements in distinct categories.

We can give a formal conversion rule for structure eta as follows.

$$\frac{t \text{ not free in } \beta \quad \Gamma \vdash z : P}{\Gamma \vdash c_P\ (\pi_1^P\ z)\ \ldots\ (\pi_{|\beta|}^P\ z) \equiv z}$$

$$\text{where } P = \mu t : \mathsf{U}_{\ell+1}.\ (c : \forall b :: \beta.\ \tau)$$

To avoid termination issues, we restrict ourselves to *non-recursive* structures, i.e. t may not be used in the field types. This also means that indices are not very useful as they cannot actually vary, so type families are not considered for structure eta. Similarly, eta conversion for propositional structures is

not very interesting for us as it is subsumed by proof irrelevance. The formal conversion rule can be used as a rule for algorithmic conversion as well, except that we have to consider that the arguments of c_P may merely be definitionally equal to the respective projection due to lack of a transitivity rule, and similarly we need to ensure symmetry manually.

$$\frac{t \text{ not free in } \beta \qquad \Gamma \vdash z : P \qquad \Gamma \vdash b_i \Leftrightarrow \pi_i^P\ z\ \forall i}{\Gamma \vdash c_P\ b \Leftrightarrow z} \qquad \frac{\dots}{\Gamma \vdash z \Leftrightarrow c_P\ b}$$

Finally, we need one more rule to cover transitivity in the case of field-less structure types such as $\mathsf{Unit} := \mu U : \mathsf{U}_1.\ (\star : U)$ where for any $u, v : \mathsf{Unit}$ we can formally derive $u \equiv \star_{\mathsf{Unit}} \equiv v$:

$$\frac{\Gamma \vdash u, v : \mu t : \mathsf{U}_{\ell+1}.\ (c : t)}{\Gamma \vdash u \Leftrightarrow v}$$

η-conversion for structure types is not especially novel; it also exists in Agda as well as in Coq when using the `Primitive Projections` flag mentioned above. While Lean 3 does not have structure eta, [Carneiro, 2019] in fact introduces primitive projections π_1 and π_2 on a primitive sigma type with the expected eta rule $(\pi_1\ x, \pi_2\ x) \equiv x$ and similar for a primitive universe lifting type purely to simplify the reduction to W-types as part of the soundness proof. Carneiro's argument that strengthening these provable equalities into definitional ones does not affect soundness applies to our extension of Lean's type system just as well.

3.2.3 Mutual Inductive Types

Lean 2 had built-in support for mutually dependent inductive types in the kernel, which was removed in Lean 3 in favor of compiling them down to single inductive types in order to further simplify the kernel. However, this encoding is quite complex in practice and turned out to be hard to maintain in the face of many edge cases. Thus we reintroduced mutual inductive types into the kernel in Lean 4.

A formal model of mutual inductive types via indexed W-types, analogously to Carneiro's model of single inductive types as W-types, is described in [Kaposi and von Raumer, 2020]. Thus I will focus on presenting Lean's kernel rules for them in Carneiro's framework in the following, replacing the original rules for single inductive types.

We define a mutual inductive specification $M = \overline{t : F.\ K}$, also called a *bundle* in the following, as a sequence of type variables $t : F$ with signatures $F = \forall a :: \alpha.\ \mathsf{U}_\ell$ and sums K of constructors $(c : e)$.

$$\boxed{\Gamma \vdash M\ \mathsf{spec}} \qquad \frac{\Gamma \vdash a_i :: \alpha_i \quad \Gamma; \overline{t_i : \forall a_i :: \alpha_i.\ \mathsf{U}_\ell}^{\,i} \vdash_i e_{ij}\ \mathsf{ctor}}{\Gamma \vdash \overline{t_i : \forall a_i :: \alpha_i.\ \mathsf{U}_\ell.\ \sum_j (c_{ij} : e_{ij})}^{\,i}\ \mathsf{spec}}$$

While the indices a_i of each t_i may be different, the resulting universe level ℓ must the same as inductive types may not depend on types in higher universes.

The specification of constructors is essentially unchanged from the single inductive case except that eventually the expected t_i must be returned, while recursive arguments can be of any simultaneously defined type t_j.

$$\boxed{\Gamma; \overline{t : F} \vdash_i \tau\ \mathsf{ctor}} \qquad \frac{\Gamma \vdash e :: \alpha_i}{\Gamma; \overline{t : F} \vdash_i t_i\ e\ \mathsf{ctor}}$$

$$\frac{\Gamma \vdash \beta : \mathsf{U}_{\ell'} \quad \mathrm{imax}(\ell', \ell) \le \ell \quad \Gamma, y : \beta; \overline{t : F} \vdash_i \tau\ \mathsf{ctor}}{\Gamma; \overline{t : F} \vdash_i \forall y : \beta.\ \tau\ \mathsf{ctor}}$$

$$\frac{\Gamma \vdash \gamma :: \mathsf{U}_{\ell'} \quad \Gamma, z :: \gamma \vdash e :: \alpha_j \quad \mathrm{imax}(\ell', \ell) \le \ell \quad \Gamma; \overline{t : F} \vdash_i \tau\ \mathsf{ctor}}{\Gamma; \overline{t : F} \vdash_i (\forall z :: \gamma.\ t_j\ e) \to \tau\ \mathsf{ctor}}$$

$$\text{where } F = \forall a :: \alpha.\ \mathsf{U}_\ell$$

Note that any well-formed single inductive $\Gamma; t : F \vdash K\ \mathsf{spec}$ by the old rules is still well-formed as in $\Gamma \vdash t : F.\ K\ \mathsf{spec}$.

Type formers, constructors, and recursors must now reference the index i of the inductive type of the bundle they want to perform on, though we will allow omitting the index for bundles of size one in order to preserve the old syntax.

$$e ::= \cdots \mid \mu_i M \mid c_{\mu_i M} \mid \mathsf{rec}_{\mu_i M}$$

$$\frac{\Gamma \vdash M\ \mathsf{spec}}{\Gamma \vdash \mu_i M : F_i} \qquad \frac{\Gamma \vdash M\ \mathsf{spec} \quad (c : \alpha) \in K_i}{\Gamma \vdash c_{\mu_i M} : \alpha[\mu_j M / t_j]_j}$$

$$\text{where } M = \overline{t : F.\ K}$$

The *large elimination* predicate LE, which controls which inductive types can be eliminated into an arbitrary universe, is essentially unchanged

since all types of the bundle live in the same universe. The special case of *subsingleton elimination* of certain inductive predicates in particular remains applicable to single inductive types only, with an unchanged predicate LE ctor.

$$\boxed{\Gamma \vdash M \; \mathsf{LE}}$$

$$\frac{1 \leq \ell}{\Gamma \vdash \overline{t : \forall a :: \alpha.\ \mathsf{U}_\ell.\ K} \; \mathsf{LE}} \qquad \frac{}{\Gamma \vdash t : F.\ 0 \; \mathsf{LE}} \qquad \frac{\Gamma; t : F \vdash \alpha \; \mathsf{LE\ ctor}}{\Gamma \vdash t : F.\ (c : \alpha) \; \mathsf{LE}}$$

Finally, given a bundle

$$M = \overline{t_i : \forall a_i :: \alpha_i.\ \mathsf{U}_\ell.\ \textstyle\sum_j (c_{ij} : \forall b_{ij} :: \beta_{ij}.\ t_i\ p_{ij})}^i$$

we can state a typing rule for the recursor as follows.

$$\frac{\Gamma \vdash M \; \mathsf{spec}}{\Gamma \vdash \mathsf{rec}_{\mu_{i_0} M} : \forall C :: \kappa.\ \forall e :: \varepsilon.\ \forall a_{i_0} :: \alpha_{i_0}.\ \forall z : t_{i_0}\ a_{i_0}.\ C_{i_0}\ a_{i_0}\ z}$$

where as parameters we have

- one *motive* C_i of type $\kappa_i = \forall a_i :: \alpha_i.\ \mu_i M\ a_i \to \mathsf{U}_u$ for each type t_i where u is a fresh universe variable if $\Gamma \vdash M$ LE, or otherwise $u = 0$. The motive represents the recursion/induction result type, which may be dependent on the input value and can vary for each inductive type in the bundle.

- one *minor premise* e_{ij} of type $\varepsilon_{ij} = \forall b :: \beta_{ij}.\ \forall v :: \delta.\ C_i\ p_{ij}\ (c_{ij}\ b)$ for each constructor c_{ij}. The minor premise describes a recursion/induction *case*, that is, how to compute the corresponding motive from the constructor's arguments b as well as inductive hypotheses v, whose types are constructed for each c_{ij} as follows: let $g :: \gamma \subseteq b_{ij} :: \beta_{ij}$ be the subsequence of recursive arguments of constructor c_{ij}, i.e. $\gamma_k = \forall x :: \xi.\ \mu_l M\ a'$ is an argument type resulting in some type $\mu_l M$ of the bundle. Then for each γ_k we obtain an inductive hypothesis $\delta_k = \forall x :: \xi.\ C_l\ a'\ (g_i\ x)$ that given the same parameters produces a value of the corresponding motive for the given indices a' and the recursive argument's value $g_i\ x$ at these parameters.

- a single *major premise* $z : t_{i_0}\ a_{i_0}$, which is the starting value of the recursion/induction.

The recursor returns a value of the corresponding motive for the major premise.

As an example, for the mutual inductive predicates

```
mutual
  inductive Even : Nat → Prop where
    | z : Even 0
    | s : Odd n → Even (n + 1)
  inductive Odd : Nat → Prop where
    | s : Even n → Odd (n + 1)
end
```

we formally have

$$\begin{aligned} M = \mathsf{Even} &: \mathsf{Nat} \to \mathbb{P}.\, (\mathsf{z} : \mathsf{Even}\ z_{\mathsf{Nat}}) + \\ &\qquad\qquad (\mathsf{s} : \forall n : \mathsf{Nat}.\ \mathsf{Odd}\ n \to \mathsf{Even}\ (s_{\mathsf{Nat}}\ n)), \\ \mathsf{Odd} &: \mathsf{Nat} \to \mathbb{P}.\, (\mathsf{s} : \forall n : \mathsf{Nat}.\ \mathsf{Even}\ n \to \mathsf{Odd}\ (s_{\mathsf{Nat}}\ n)) \end{aligned}$$

and, using $\mathsf{Even} := \mu_1 M, \mathsf{Odd} := \mu_2 M$, can derive the recursors

$$\begin{aligned} \Gamma &\vdash \mathsf{rec}_{\mathsf{Even}} : \forall C :: \kappa.\ \forall e :: \varepsilon.\ \forall n : \mathsf{Nat}.\ \forall z : \mathsf{Even}\ n.\ C_1\ n\ z \\ \Gamma &\vdash \mathsf{rec}_{\mathsf{Odd}}\ \ : \forall C :: \kappa.\ \forall e :: \varepsilon.\ \forall n : \mathsf{Nat}.\ \forall z : \mathsf{Odd}\ n.\ C_2\ n\ z \end{aligned}$$

where

$$\begin{aligned} C &:: \kappa = \mathsf{C}_{\mathsf{Even}} : \forall n : \mathsf{Nat}.\ \mathsf{Even}\ n \to \mathbb{P}, \mathsf{C}_{\mathsf{Odd}} : \forall n : \mathsf{Nat}.\ \mathsf{Odd}\ n \to \mathbb{P} \\ e &:: \varepsilon = \\ &e_{\mathsf{z}_{\mathsf{Even}}} : \mathsf{C}_{\mathsf{Even}}\ z_{\mathsf{Nat}}\ \mathsf{z}_{\mathsf{Even}}, \\ &e_{\mathsf{s}_{\mathsf{Even}}} : \forall n : \mathsf{Nat}.\ \forall o : \mathsf{Odd}\ n.\ \mathsf{C}_{\mathsf{Odd}}\ n\ o \to \mathsf{C}_{\mathsf{Even}}\ (s_{\mathsf{Nat}}\ n)\ (\mathsf{s}_{\mathsf{Even}}\ o), \\ &e_{\mathsf{s}_{\mathsf{Odd}}} : \forall n : \mathsf{Nat}.\ \forall e : \mathsf{Even}\ n.\ \mathsf{C}_{\mathsf{Even}}\ n\ o \to \mathsf{C}_{\mathsf{Odd}}\ (s_{\mathsf{Nat}}\ n)\ (\mathsf{s}_{\mathsf{Odd}}\ e) \end{aligned}$$

The computation rules then follow as in [Carneiro, 2019]. In particular, K-like reduction applies only to single inductive types and thus can be represented by the same rule.

Nested inductive types

In the specification of mutual inductive types we still demand recursive arguments of constructors to be of the form $\forall - :: -.\ t_i\ -$, which excludes recursive occurrences of the types to be defined *nested* inside other inductive types such as in the common definition of arbitrarily-branching *rose trees*.

```
inductive Tree (α : Type) where
  | node : α → List (Tree α) → Tree α
```

However, it is possible to "flatten" an occurrence of a constructor t_i nested inside another bundle M' as in $\mu_j M'[t_i\ e]$ (which can happen from substitution of a parameter, as above) into a direct type application $t'\ e$ by extending the current bundle M with the instantiation $M'[t_i\ e]$ where occurrences t'_j of types defined by M' have been appropriately adjusted. As the transformation is technically laborious, I will merely demonstrate it on the above example here.

```
mutual
  inductive Tree' (α : Type) where
    | node : α → ListTree α → Tree' α
  inductive ListTree (α : Type) where
    | nil : ListTree α
    | cons : Tree' α → ListTree α → ListTree α
end
```

For the new bundle to be well-formed, the arguments e of the nested occurrence must not depend on constructor parameters, i.e. $\Gamma \vdash e :: -$, and $t_i\ e$ must occur strictly positively in $\mu_j M'$ as demanded by the ctor rules.

This transformation can be done outside of the kernel and type system, which was the case in previous versions of Lean. Lean 4 relies on it as well, but incorporates it into the kernel to solve one particular issue: some definitional reductions are lost during the transformation. As users should not be exposed to the instantiation $M'[t_i\ e]$, all constructors and recursors are ultimately redefined in terms of the original M' such that the interface to the user is that of the original nested inductive type.

```
def Tree := Tree'
def ListTree.ofList : List (Tree α) → ListTree α := ...
def ListTree.toList : ListTree α → List (Tree α) := ...
def Tree.node (a : α) (ts : List (Tree α)) : Tree α :=
  Tree'.node a (ListTree.ofList ts)
def Tree.rec {C1 : Tree α → Sort u} {C2 : List (Tree α) → Sort u}
  (eNode : (a : α) → (ts : List (Tree α)) → C2 ts → C1
    (Tree.node a ts))
  (eNil : C2 [])
  (eCons : (head : Tree α) → (tail : List (Tree α)) → C1 head →
    C2 tail → C2 (head :: tail))
  (t : Tree α) → C1 t := ...
```

However, as the copy `ListTree` is not definitionally equal to the original `List (Tree α)` (neither in Lean's nominal `inductive` declarations nor in the structural μ encoding used above), we need to define and employ conversion functions such as `ListTree.ofList` used in `Tree.node`. In particular, in order to define the recursor `Tree.rec`, we have to transport terms such as values of the motives that are dependent on `List (Tree α)` values across the non-definitional equality `ListTree.ofList (ListTree.toList ts) = ts` and its reverse. Finally, when we try to reduce an open term like `Tree.rec C1 C2 eNode ... (Tree.node a ts)`, it turns out that it cannot be reduced to `eNode a ts` ... as we would hope, but before that gets stuck on the transport, which is reduced away only when the list structure of `ts` is fully known. By moving the transformation into the kernel but still checking the well-formedness of the result as a mutual inductive type, Lean 4 can postulate these expected conversion rules in exchange for only a modest increase in the Trusted Code Base.

3.3 The (Future of the) Module System

Lean 4's module system, as in a system for organizing and reusing code, has not fundamentally changed from the simplistic design of prior versions: a module is a single Lean file, and importing modules makes the union of their declarations available, transitively. Name clashes are avoided by use of the orthogonal namespace system producing hierarchical names such as `Std.RBMap.empty`, which requires a certain degree of rigor of and/or cooperation between authors to make sure their independently developed modules can be imported together. Typeclasses remain the preferred solution for abstracting code over a concept in favor of parameterized modules as found in say Standard ML [MacQueen, 1984].

And yet there are on-going discussions about changing this in the future. The primary motivation is to ensure that Lean remains scalable for the foreseeable future. At the time of writing, the mathlib repository has reached a size of more than one million lines of Lean code, requiring multiple hours for a full build, and no-one expects this growth to stop any time soon unless Lean itself stops being a viable tool for such a project size. In particular, I believe we will need to address the following two aspects in the future with an improved module system for *recompilation avoidance*:

- *Export less*. There are many kinds of changes to a module that should not be observable by, and thus cause recompilation of, downstream modules. These include superficial changes such as to whitespace or comments (though *docstrings* and declaration positions still need to be accessible to at least the language server) as well as module-internal changes such as changing the proof but not proposition of a theorem, as imported theorems are not rechecked at the default trust level. Ideally, changing the body of a definition or adding, changing, or removing private declarations should not trigger recompilation either, but two aspects of the Lean language complicate restricting the public *signature* of a module in this way: first, the type theory assumes that δ-reduction, or unfolding of definitions, is always possible, which makes dependent type theory inherently anti-modular. In practice though, it is common and good style to prove abstract properties about a concrete definition in the same module and then to rely only on these properties in dependent modules. For example, the particular chosen construction of the real numbers is usually inconsequential for users that instead rely on their abstract algebraic properties. Thus restricting definitional unfolding, at least outside the originating module, might be a reasonable limitation in exchange for improved recompilation avoidance. While not focused on separate compilation, [Gratzer et al., 2022] have recently made interesting advances in this direction by developing and implementing a type theory where definitional unfolding is not available by default but must be opted in per definition and scope.

  The second complication is metaprogramming, as compile-time execution of an imported definition is equivalent to unfolding it. Here a limitation to the current module is not practical; instead, we have to accept that changing metaprograms leads to downstream recompilation but should not accept that the same is true for definitions never used as or by metaprograms. Thus a *phase separation* system would be necessary to cleanly separate these worlds as for example implemented in Racket [Flatt, 2002] and more recently explored by [Sterling and Harper, 2021] in the context of an ML-style module system based on Martin-Löf type theory. In fact, Lean 4 already features a very limited form of phase separation in the form of the `initialize` and `builtin_initialize` commands which both take an `IO`

action to be executed on module load, but in the latter case restricted to the run time phase of the generated executable where it is used to e.g. register built-in syntax that should become active only in the next stage of the compiler. However, the name resolution algorithm is not aware of these phases yet as it should be in a proper phase separation system.

- *Import less*, by which I primarily refer to limiting the prevalence of transitive imports. Viewing the set of additionally needed modules necessary for loading a certain module as part of its module signature, this point can be seen as a special case of the previous aspect. While limiting the amount of exported data from a module may be less interesting to mathematicians than to programmers as purely internal changes to a theorem or definitions used for modeling mathematics are much rarer than to programs, I strongly believe that avoiding transitive imports would be to the immediate benefit of any user of a system like Lean. Taking the previous example of the real numbers, there is a good amount of modules involved in constructing the model, but once that is done, there is no reason that users of the type of real numbers should load these modules, or be recompiled when they change without affecting the abstract interface of the type. Making an import non-transitive can be seen as turning all of its exports private to any external users of the current module, and so similar checks to references to private declarations not escaping into the module signature would be necessary for a sound implementation.

For now though, we have focused on optimizing the implementation of the existing simplistic module system, yielding some significant performance gains that would be preserved by any of the changes sketched above. As a first step, Leo replaced the previous hand-written module serializer with a generic Lean object graph serializer. The serializer can persist a Lean object of any type and the objects referred to by it, transitively, as long as there is no closure object in the graph. As the module data is now a pure-data Lean type, module serialization can be done with this generic system, significantly simplifying the implementation and maintenance of it. The generic serializer also implements *maximum sharing* or *hash consing* [Ershov, 1958], i.e. structurally equal Lean objects are deduplicated. Finally, the serializer in fact preserves the in-memory layout of Lean objects, merely copying them and rewriting pointers for maximum sharing as

well as to make them relative to the start address of the serialized block. On deserialization, the relative pointers are rewritten back to absolute addresses according to the new start address of the loaded block. While a more compact representation that e.g. elides padding and uses smaller pointer representations would be possible, this aspect is crucial for the next step inspired by [Yang et al., 2015]: after the new module serializer was in place, I adjusted it to store pointers not relative to the start of the block, but absolute according to a virtual base address derived from the hash of the module name. Thus if we can load the block into memory at this address in downstream compilations, no pointers need to be adjusted.[1] In fact, there is no need to change the loaded block from its on-disk representation at all, which means that we can *memory map* it directly from disk into the memory of the Lean process, which is an operating system function available on all platforms supported by Lean. The advantage of using a read-only memory mapping instead of standard allocated memory is that imported modules effectively no longer contribute to Lean's *working set size*: they can be loaded partially, on-demand by the operating system, can be shared among different Lean processes loading the same module, and can be discarded in low-memory situations without having to write them to swap space. Indeed, when activating this extension we have seen great savings on *mathport*[2], a version of Lean 3 mathlib binary-translated to the Lean 4 module format: the time for importing these, at the time, 1871 modules fell from 2.3 to 1.5 seconds[3], and more importantly the total heap size of all allocations fell from 1.2GB to 55MB. Thus I feel safe to say that this part of the module system at least is future-proof even in the face of drastically increased import closure sizes, which nevertheless we should aim to keep limited by the *import less* mission statement.

The performance of this new scheme depends on the ability to map modules at the requested addresses without conflict. If there is a conflict

[1] This approach is comparable to the use of *preferred base addresses* in shared library loading such as in Windows [Richter and Nasarre, 2008], though the advent of Address Space Layout Randomization (ASLR) has largely rendered this scheme obsolete; as we only map non-executable data, ASLR is not a concern for our purposes.

[2] `https://github.com/leanprover-community/mathport`

[3] Lean still gathers information such as a map of all declaration from the imports, so it cannot be 0; Leo later optimized the import time further independently of memory mapping.

with another memory mapping, be it a another Lean module or a loaded native library or some other kind of memory region, we have to fall back to adjusting all pointers according to the new base address, in which case this region cannot be shared with other processes anymore. However, as objects in other modules are referenced not by direct pointers but e.g. by declaration name, relocating an imported module does not affect the ability to memory map dependent modules. In practice, collisions are unlikely given the immense size of modern address spaces. We can make this more precise by applying the birthday problem: on modern 64-bit architectures, user-space addresses are usually limited to the lower 47 bits. At the time of writing, mathport and Lean itself consist of 2995 and 617 Lean modules, respectively, with an average file size of 320 and 892 KB, and a 99th percentile of 2.24 and 6.46 MB. If we semi-conservatively approximate this by saying we want to uniformly place n modules of size 2 MB (2^{21} bytes) at addresses aligned to 2 MB, then there are $d = 2^{47}/2^{21} = 2^{26}$ possible slots and we can conclude by a folklore formula for the birthday paradox (which can e.g. be derived from equation (3.4) of [Feller, 1950]) that we would need to load

$$n \approx \sqrt{2d \ln \frac{1}{1-p}} \approx 9645$$

modules to reach a probability of $p = 50\%$ of even one collision. This ball-park calculation greatly overestimates the average module size while at the same time ignoring collisions from overlaps because of the smaller in-practice alignment, non-uniformity of the chosen hash function, and a negligible amount of other mappings in the address space. Thus while the possibility of collisions cannot be excluded and must be addressed using a fall-back implementation, it is quite negligible in practice, and again would benefit from the mantra *import less*.

3.4 The Parser

The parser is the component with perhaps the most dramatic technological shift from Lean 3 to Lean 4: in previous versions, it was a hand-written recursive-descent implementation with precedence and (limited) extensibility of term notations handled by a dynamic parsing table based on Pratt

parsing [Pratt, 1973]. A syntax tree, even an abstract one, did not exist of the whole file: terms were parsed into kernel-structure pre-terms we have already seen in Section 2.3, while the only output of parsers of top-level commands was their side effects such as additions to the environment, i.e. elaboration did not exist as a separate step from parsing.

Quite in contrast to these versions, we designed the Lean 4 parser from scratch with the following primary goals in mind:

flexibility Lean 4's lexical and syntactical grammar is complex and includes indentation and other whitespace sensitivity. It should be possible to introduce such custom "tweaks" locally without having to adjust the fundamental parsing approach.

extensibility Lean's grammar can be extended dynamically within a Lean file, and with Lean 4 we want to extend this to cover embedding domain-specific languages that may look nothing like Lean, down to using a separate set of tokens.

losslessness The parser should not only produce a concrete syntax tree that can be used in the macro system, but also preserve all whitespace and other "sub-token" information for use in tooling. The exact original input should be recoverable from the syntax tree.

performance Despite the above goals, the overhead of the parser building blocks, and the overall parser performance on average-complexity input, should be comparable with that of the previous parser hand-written in C++.

While many parser implementations in theorem provers are similarly manual as Lean 3's, there are some exceptions such as Isabelle's use of an Earley parser [Earley, 1970] for parsing its inner syntax even in absence of precedence annotations, potentially generating multiple, overlapping syntax tree interpretations. For the purpose of embedding domain-specific languages, we especially looked at *scannerless generalized LR-parsing* (SGLR) [Visser, 1997] as used in the implementation of the Syntax Definition Formalism SDF [Bravenboer et al., 2006]. However, both of these algorithms require a context-free grammar in a sufficiently analyzable representation. A more flexible, dynamic approach is presented by [Swierstra and Duponcheel, 1996] who enrich parser combinators with

just enough static structure to efficiently parse LL(1) grammars without backtracking. In order to attain our goals of flexibility and performance, we thus combined this approach with Pratt precedence parsing that had already proven itself in previous versions of Lean, but added the possibility of backtracking when necessary so as not to prevent user extensions that in combination may not be LL(1). The backtracking combinator is backed by a cache inspired by packrat parsing [Ford, 2002] in order to avoid exponential run times.

The goal of losslessness finally is not as much a question of parser implementation as of its output. I have implemented syntax objects in Lean 4 as an inductive type of *nodes* (or nonterminals), *atoms* (or terminals), and, as a special case of terminals (which will become important in Chapter 4), *identifiers*.

```
inductive Syntax where
  | node  (kind : Name) (args : Array Syntax)
  | atom  (info : SourceInfo) (val : String)
  | ident (info : SourceInfo) (rawVal : String) ...
  | missing
```

An additional constructor represents *missing* parts from syntax error recovery. Using a uniform, generic constructor `node` for inner nodes of arbitrary arity that is identified merely by an attached name ensures that the syntax tree is arbitrarily extensible. Tokens (atoms and identifiers) are annotated with source location metadata unless generated by a macro.

```
inductive SourceInfo where
  | original (leading : Substring) (pos : String.Pos) (trailing :
    Substring) (endPos : String.Pos)
  | synthetic ...
  | none
```

By capturing both the start and end position of a token as well as its leading and trailing whitespace/comments, we make sure to preserve all lexical information a consumer of the syntax tree may need. Keeping the information locally in the syntax leaves means that it will not be lost or become desynchronized when other parts of the tree are transformed.

Users of Lean for the most part do not interact with the parser combinators or the definition of `Syntax` directly. A convenient shorthand allows for writing down new syntax in a style reminiscent of (Extended) Backus-Naur Form.

```
syntax "ℕ" : term  -- atoms
syntax "if" term "then" term "else" term : term  -- sequences
syntax "{" ident (":" term)? "//" term "}" : term  -- optionals
syntax "[" term,* "]" : term  -- repetition, with separator

-- implementing the syntax of `syntax` in itself
declare_syntax_cat stx2
syntax "syntax2" stx2* ":" ident : command
syntax ident : stx2
syntax str : stx2
syntax "(" stx2* ")" : stx2
syntax stx2 ("?" <|> "*" <|> ",*" <|> ...) : stx2
...
```

A `syntax` command registers a given syntax specification with a syntactic category, such as `term` and `command` above, given after the colon. A syntactic category can be thought of as a kind of extensible nonterminal in the grammar, with `syntax` registering new grammar rules for that nonterminal. The `declare_syntax_cat` command can be used to introduce new syntax categories. Every `syntax` command by default generates a unique name that is used as the kind in a `Syntax.node` object generated when the grammar rule is applied. The name is usually irrelevant as syntax objects are mostly consumed and transformed by use of syntax *quasiquotations* as described in detail in Chapter 4. We will see many more examples of declaring and using syntax there as well as in Chapter 5.

3.5 The Elaborator

Elaboration, in short, can be thought of as taking a fully macro-expanded ("desugared") syntax tree and producing a corresponding kernel representation, i.e. a fully specified lambda term in the case of term elaboration; in truth, macro expansion and elaboration are intertwined in Lean (Section 4.5). For the purpose of this section though we will focus on the traditional tasks of the elaborator requiring type information without considering macro expansion, which include:

- inference, by unification plus conversion, of elided subterms from e.g. implicit parameters or explicit holes (_) in the syntax tree
- insertion of coercions on type conflicts

- typeclass inference
- tactic execution (now part of macro expansion: Section 4.6)
- resolving ambiguous notation and overloaded references

While these tasks are relatively unchanged across Lean versions, there are some fundamental differences in their implementation. Lean 2 for example put great focus on approximating higher-order unification by constraint solving [de Moura et al., 2015a]. This approach was replaced in Lean 3 by a simpler and far more efficient scheme, never gathering constraints but resolving them on the spot, failing otherwise. Only typeclass inference and tactic execution were potentially delayed until the expected type at that position was sufficiently known. Thus the elaboration order for the most part was fixed, proceeding in practice from left to right in the term, which can greatly limit the effectiveness of type inference. For example, in

```
List.map (fun x => x.fst) xs
```

elaboration of the projection notation `x.fst` will fail unless the type of `x` is sufficiently known, which requires elaborating the second argument `xs` first.

Possibly the biggest elaboration change from Lean 3 to Lean 4 is the lifting of this strict ordering. Instead of a fixed ordering of nested elaboration calls that either succeed or fail, each elaboration handler for a specific kind of term syntax can request to be *postponed*, in which case its result is replaced with a fresh metavariable. The elaboration procedure is eventually rerun to resolve the metavariable; if all currently postponed elaborators keep requesting postponement because of some type checking dependency cycle, elaboration fails. This approach was explored in parallel to Lean 4 development in the Klister language [Barrett et al., 2022], refining upon our implementation by continuing the postponed elaborator at the point of the suspension instead of rerunning it from the beginning. After initial experiments with basing elaboration on a continuation monad transformer, we decided against this approach because the transformer would have increased overall overhead using our current code generator. We have not noticed significant overhead from rerunning elaborators so far.

3.6 The Code Generator

Lean 4 currently features a single backend for native compilation of Lean code: a C code generator. The resulting code can be compiled with a standard C compiler and linked against the Lean runtime implemented in C++. C was an obvious choice as a first target because of its simple interfacing with C++ while being sufficiently expressive to translate our Lean intermediate representation (IR) to in a straightforward transformation, and being slightly faster to compile with standard compilers than C++. Another strong advantage of C compared to perhaps more typical choices such as LLVM IR is that we can check the generated code into the Lean 4 repository and so bootstrap Lean 4 on any system that provides standard C and C++ compilers. A direct LLVM backend is being developed by Siddharth Bhat at the time of writing, but we will still want to keep the C backend around for the purpose of bootstrapping. We support both the *programming* use case of creating a standalone executable from Lean code (as is the case for the executables distributed with Lean 4) and the *metaprogramming* use case of compiling e.g. tactics into native shared libraries for efficient execution.

Before being emitted as C code, Lean definitions go through two major intermediate representations:

- λ_{pure} is a pure, strict, untyped language used for general optimizing transformations common to functional languages and in particular based on those used in GHC [Peyton Jones, 1996].

- λ_{RC} extends λ_{pure} with explicit instructions for reference counting, thus losing purity. We will discuss it in much greater detail in Chapter 6, in which I also present formal semantics for both IRs.

Both IRs as well as their transformations are implemented in Lean itself, replacing an initial implementation in C++ for bootstrapping.

Lean 4 also comes with an interpreter for λ_{RC} I have implemented, which allows for rapid incremental development and testing right from inside the editor. Whenever the interpreter calls a function for which native, ahead-of-time compiled code is available, which in particular is true for all functions from the standard library, it will switch to that instead. Thus the interpretation overhead is negligible as long as e.g. all expensive tactics are precompiled.

In detail, *precompilation* consists of the build system compiling the tactics and other relevant Lean code into a shared library, which can then be loaded into both the interactive Lean server and the batch compiler via a command line flag. Then, before entering a function for the first time, the interpreter will check if the mangled name of that function is available as a symbol in the current process via `dlsym` or similar platform functions. In implementing this approach, I specifically took care in making this setup as invisible and painless as possible for users. Users should not even have to think about what has to happen in the background to make their tactics fast, and in particular the setup should not be dependent on or be likely break from external factors. For this purpose I have extended the Lean binary releases with a self-contained LLVM toolchain consisting of a C compiler, basic headers, a linker, and other tools necessary for native compilation, inspired by a similar setup for the Zig language [Kelley, 2020]. A critical component not provided by LLVM is interfacing with the system C library, which depending on the operating system requires installing an SDK with download size and permission requirements that may be prohibitive for some users of Lean, such as those using restricted university machines. Following Zig, we solve this issue by distributing necessary files bundled with Lean directly.

Windows requires installation of the Windows SDK for native compilation in the standard setup. Because Lean and its runtime can currently only be built with the alternative, Unix-like MSYS2 toolchain[4] in any case, we replace required *import libraries* for linking from the SDK with those from MSYS2.

macOS usually requires installation of the *Xcode Command Line Tools* to acquire stub files necessary for linking against the C library. We ship Lean with copies of those stub files to skip this step.

Linux distributions should not require additional components because they usually allow linking against the system C shared library directly. However, a similar but separate issue is that doing so may embed *versioned symbols* of this specific library version into the binary, unnecessarily complicating distribution of Lean programs

[4] `https://www.msys2.org/`

platform	compr. size	uncompressed size		
	(zstd)	basic Lean files	LLVM	libc interface
macOS	108MB	384MB	116MB	1.6MB
Linux	112MB	406MB	119MB	3.9MB
Windows	116MB	407MB	180MB	3.8MB

Figure 3.2: Example of the composition of a Lean release[5] per platform.

(including Lean itself) to other Linux machines. We resolve this issue by distributing a sufficiently old version of `glibc` with Lean, which we link Lean program against (but still use the system `glibc` at run time).

The result is a truly stand-alone compilation toolchain for all platforms supported by Lean. Fig. 3.2 gives an impression of the space overhead of this solution.

Bootstrapping. A purely internal but fundamental issue we faced as soon as we started reimplementing parts of Lean in Lean itself is how to *bootstrap* the system given this cyclic build dependency. Standard solutions to this problem implemented by other self-hosting compilers include:

- Use of a previous release. GHC for example can be compiled using the two previous major releases [GHC, 2020] while building the Rust compiler requires and automatically downloads the latest beta release [Rust, 2021]. This is a sensible option for mature languages, since it constrains at least the first built stage to the features available in the previous release (however, Rust takes the liberty of enabling otherwise inaccessible unstable features in the release via a special environment variable). Later stages, if they implement more functionality than the first stage, may make use of language features introduced after the previous release via conditional compilation.

- Bundling the bootstrap compiler with the code. This is a simple solution that allows for updating the bootstrap compiler at any time

[5] `https://github.com/leanprover/lean4-nightly/releases/tag/nightly-2022-01-31`

> and as often as desired between releases. However, version control systems may not scale well when checking in large binaries, especially when doing so for every supported platform. The OCaml compiler partially avoids this issue by checking in a platform-independent bytecode executable of the compiler that is run by an interpreter written in C.[6] Idris 2, which coincidentally became a bootstrapped dependently typed language at roughly the same time as Lean, bundles generated bootstrap files for Racket and Chez Scheme.[7]

Because we expected frequent changes to the Lean 4 language during development, we decided to go with an approach inspired by OCaml, but avoiding interpretation overhead when building the first stage: as mentioned above, Lean code is currently compiled into C, and this code is platform-independent because all platform-specific functionality is part of the runtime written in C++. Thus it is sufficient to check in this C code in order to derive a bootstrap compiler on any supported platform running at native speed, dependent only on a C/C++ compiler toolchain.

3.7 The User Interface

Lean 4 aims to provide editor integration on par with regular programming languages via the Language Server Protocol (LSP)[8]. The LSP was not designed for theorem proving languages and thus needs to be extended to provide additional information such as a current-goal display, but we still strived for implementing the standard LSP requests whenever possible in order to leverage existing support in editors. At the time of writing, there are mature Lean 4 editor extensions for VS Code[9], Emacs[10], and Neovim[11].

One particular aspect that was influenced by our experiences from Lean 3 is the server process architecture. While Lean 3 did not implement the LSP, like most language server implementations it used a single process to serve information about the set of Lean files in a project, with changes in

6 https://github.com/ocaml/ocaml/tree/trunk/boot
7 https://github.com/idris-lang/Idris2/tree/main/bootstrap/idris2_app
8 https://microsoft.github.io/language-server-protocol/
9 https://github.com/leanprover/vscode-lean4
10 https://github.com/leanprover/lean4-mode
11 https://github.com/Julian/lean.nvim

one file being automatically propagated to dependent open files. While powerful, the concurrent architecture necessary for this approach turned out to be complex to build and maintain. In Lean 4, we decided for a simpler approach where every open Lean file is managed by a dedicated server process, which brings with it other advantages:

- Crashes and side effects are limited to the current file, making sure we do not lose other files' states. This is especially important in the case of Lean as user-defined meta code may trigger arbitrary panics or stack overflows.

- The management of module memory mappings (Section 3.3) is simplified: by making sure that files are represented by different processes that exchange data only via the file system, we know that we can map all imports from disk into memory regardless of whether they are opened in the editor, with the sharing of mappings providing the same memory deduplication benefits as the more complex single-process architecture. By restarting the file-dedicated process when the file's imports have changed, we can completely avoid the complication of when to free the mappings by relying on the operating system's cleanup at process exit, and we ensure that we end up with a consistent state that is unaffected by the server's previous state.

As editors expect a single server process to communicate with via LSP, these file worker processes are hidden behind and managed by a *watchdog* process. The watchdog also accumulates and serves project-wide data such as for the find-all-references request [Mennicken, 2022].

When file contents are processed in a server worker, we create a *snapshot* of the entire parser and elaboration state after each command, which is inexpensive as these are persistent Lean data structures. Every version of the opened document is associated with a lazy list of such snapshots that is extended asynchronously until the end of the file is reached. When an incoming change results in a new document version, the prefix of the snapshot list that is unaffected by the change is reused without re-elaboration and the still-running elaboration tasks, if any, are replaced with elaboration tasks of the new contents. Incoming requests retrieve the lazy snapshot list of the current document version once and then wait until the list is sufficiently progressed to answer the request. This design

of using different versions of immutable lists, which may share a prefix of nodes, avoids race conditions between document changes and execution of requests.

The main component that drives processing of semantic requests is called the *info tree*, which is a result of elaboration and stored in the associated snapshot. The info tree mirrors the call tree of elaboration, with each node containing metadata and the result of the invocation of a specific elaborator, such as:

- the Lean function name of the elaborator, which is used to e.g. jump to the definition of a notation,
- the syntax tree input, which can be used to locate info tree nodes relevant to the current cursor position,
- the global and local context at that point,
- for term elaboration, the expected type and the resulting kernel term, used e.g. for jumping to the declaration of a referenced constant,
- for tactic elaboration, the goals before and after execution of the tactic, shown in the editor as the tactic state.

While the concept of the snapshot list was established in Lean 3, the info tree is a novel central component in Lean 4, replacing one of my more questionable but quite effective hacks in Lean 3: as the Lean 3 elaborator had not been designed with such metadata output in mind, requests would instead execute re-elaboration of the command at point, triggering an exception when the cursor location had been reached, attaching the necessary metadata to the exception object while it bubbled up through the elaborator. The info tree is a welcome improvement over this approach, improving request latency and providing a cleaner separation of elaborator and language server.

I found I could not say what it was I understood; that it was in fact on the level of meaning above language, a level we like to believe scarcely exists

– Gene Wolfe, *The Book of the New Sun*

4 A Macro System for Theorem Provers

In interactive theorem provers, extensible syntax is not only crucial to lower the cognitive burden of manipulating complex mathematical objects, but plays a critical role in developing reusable abstractions in libraries. *Mixfix* notation systems have become an established part of many modern ITPs for attaching terse and familiar syntax to functions and predicates of arbitrary arity.

```
_⊢_:_ = Typing                                          Agda
Notation "Ctx ⊢ E : T" := (Typing Ctx E T).             Coq
notation typing ("_ ⊢ _ : _")                           Isabelle
notation Γ `⊢` e `:` τ := Typing Γ e τ                  Lean 3
```

As a further extension, all shown systems also allow *binding names* inside mixfix notations.

```
syntax ∃ A (λ x → P) = ∃[ x ∈ A ] P                     Agda
Notation "∃ x , P" := (exists (fun x => P)).            Coq
notation exists (binder "∃")                            Isabelle
notation `∃` binder `,` r:(scoped P, Exists P) := r     Lean 3
```

While these extensions differ in the exact syntax used, what is true about all of them is that at the time of the notation declaration, the system already, statically knows what parts of the term are bound by the newly introduced variable. This is in stark contrast to *macro systems* in Lisp and related languages where the expansion of a macro (a syntactic substitution) can be specified not only by a *template expression* with placeholders like above, but also by arbitrary *syntax transformers*, i.e. code evaluated at compile

time that takes and returns a syntax tree.[1] As we move to more and more expressive notations and ideally remove the boundary between built-in and user-defined syntax, I argue that we should no more be limited by the static nature of existing notation systems and should instead introduce syntax transformers to the world of ITPs.

However, as usual, with greater power comes greater responsibility. By using arbitrary syntax transformers, we lose the ability to statically determine what parts of the macro template can be bound by the macro input. Thus it is no longer straightforward to avoid *hygiene* issues (i.e. accidental capturing of identifiers; [Kohlbecker et al., 1986]) by automatically renaming identifiers. In this chapter, I propose to learn from and adapt the macro hygiene systems implemented in the Scheme family of languages for interactive theorem provers in order to obtain more general but still well-behaved notation systems.

After giving a practical overview of the new, macro-based notation system Leonardo de Moura and I implemented in Lean 4 in Section 4.1, I describe the issue of hygiene and the general hygiene algorithm, which should be just as applicable to other ITPs, in Section 4.2. Section 4.3 gives a detailed description of the implementation of this algorithm, and macros in general, in Lean 4. In Section 4.5, I show how to extend the use case of macros from mere syntax substitutions to type-aware elaboration. Finally, we have already encountered hygiene issues in Lean 3 in a different part of the system: the tactic framework. I discuss how these issues are inevitable when implementing reusable tactic scripts and how the macro system can be applied to this hygiene problem as well in Section 4.6.

Contributions. In this chapter, I present a system for hygienic macros optimized for theorem proving languages as implemented in Lean 4.

- I describe a novel, efficient hygiene algorithm I developed for employing macros in ITP languages at large: a combination of a white-box, effect-based approach for detecting newly introduced identifiers and an efficient encoding of scope metadata.
- I show how such a macro system can be seamlessly integrated into existing elaboration designs to support type-directed expansion even

[1] These two macro declaration styles are commonly referred to as *pattern-based* vs. *procedural*.

if they are not based on homogeneous source-to-source transformations.

- I show how hygiene issues also manifest in tactic languages and how they can be solved with the same macro system. To the best of my knowledge, the tactic language in Lean 4 is the first tactic language in an established theorem prover that is automatically hygienic in this regard.

Acknowledgements. The macro system described in this chapter is joint work with Leonardo de Moura [Ullrich and de Moura, 2022a], itself an extended journal version of [Ullrich and de Moura, 2020] we were invited to submit. I developed the macro system itself after thorough discussions with Leo about our needs, performance requirements and, last but not least, available complexity budget. I also designed and implemented the syntax tree data structure described in Section 3.4 as part of this work as well as related language features such as quotations and the antiquotation language (Section 4.3.1) while Leo was the main implementer of users of these such as the elaborator and tactic interpreter (Section 4.6). Section 4.4 describes novel work I have implemented after the above publications.

4.1 Lean 4 Macro System by Example

Lean's previous notation system as shown above is still supported in Lean 4, but based on a much more general macro system; in fact, the `notation` keyword itself has been reimplemented as a macro, more specifically as a *macro-generating macro* making use of a tower of abstraction levels. The corresponding Lean 4 command[2] for the example above

```
notation Γ "⊢" e ":" τ => Typing Γ e τ
```

expands to the macro declaration

```
macro Γ:term "⊢" e:term ":" τ:term : term => `(Typing $Γ $e $τ)
```

where the syntactic category (`term`) of placeholders and of the entire macro is now specified explicitly. The right-hand side uses an explicit

[2] All examples including full context can be found in [Ullrich and de Moura, 2023].

syntax quasiquotation to construct the syntax tree, with syntax placeholders (*antiquotations*) prefixed with `$`. As suggested by the explicit use of a quotation, the right-hand side may now be an arbitrary Lean term computing a syntax object; in other words, there is no distinction between pattern-based and procedural macros in Lean 4. In contrast to the term-specific notation systems listed in the introduction, we can now use this abstraction level to implement simple macros in syntactic categories other than `term`, such as for a definition-like command macro that automatically wraps the given value in a lazily-evaluated thunk.

```
macro "defthunk" id:ident ":=" e:term : command =>
  `(def $id := Thunk.mk (fun _ => $e))
defthunk big := mkArray 100000 true
```

The expansion of the above command is

```
def big := Thunk.mk (fun _ => mkArray 100000 true)
```

`macro` itself is another command-level macro that, for our `notation` example, expands to two commands

```
syntax term "⊢" term ":" term : term
macro_rules
  | `($Γ ⊢ $e : $τ) => `(Typing $Γ $e $τ)
```

that is, a pair of parser extension (Section 3.4) and syntax transformer. The reason we decided to ultimately separate these two concerns is that we can now obtain a well-structured syntax tree pre-expansion, i.e. a *concrete* syntax tree, for the tooling reasons described in Section 3.4.

Both `syntax` and `macro_rules` are in fact further macros for regular Lean definitions encoding procedural metaprograms, though users should rarely need to make use of this lowest abstraction level explicitly. Both commands can only be used at the top level; we are not currently planning support for local macros.

There is no more need for the complicated `scoped` syntax of Lean 3 since the desired translation can now be specified naturally, without any need for further annotations.

```
notation "∃" b "," P => Exists (fun b => P)
```

The lack of static restrictions on the right-hand side ensures that this works just as well with custom binding notations, even ones whose translation

cannot statically be determined before substitution as in the following flexible *set comprehension* notation.

```
syntax "{" term "|" term "}" : term
macro_rules
  | `({$x ∈ $s | $e}) => `(preimage (fun $x => $e) $s)
  | `({$x      | $e}) => `({$x ∈ univ | $e})

notation "⋃" b "," e => Union {b | e}
```

The new notation allows us to build sets such as `{n | n + 1}` and `{n ∈ primes | n * 2}` via the set function `preimage f s` that encodes the preimage of the set `s` over the function `f`. Because both binding forms `$x ∈ $s` and `$x` are terms syntactically, we can abstract over them uniformly in derived syntax such as the big union operator above.

In this example we explicitly made use of the `macro_rules` abstraction level for its convenient syntactic pattern matching syntax. `macro_rules` are "open" in the sense that multiple transformers for the same `syntax` declaration can be defined; they are tried up to the first match, starting with the newest `macro_rules` declaration (though this can be customized using explicit priority annotations, as with most metaprogramming extension points in Lean). Thus the following extension will not be shadowed by the `$x` default case above:

```
macro_rules
  | `({$x ≤ $hi | $e}) => `({$x ∈ setOf (fun x => x ≤ $hi) | $e})
```

As an extended example of grammatic reuse, I present a partial reimplementation of the arithmetic "bigop" notations found[3] in Coq's Mathematical Components library [Mahboubi and Tassi, 2021] such as

```
\sum_ (i ← [0, 2, 4] | i != 2) i
```

for summing over a filtered sequence of elements. The specific bigop notations are defined in terms of a single `\big` fold operator; however, because Coq's notation system is unable to abstract over the indexing syntax, every specific bigop notation has to redundantly repeat every specific index notation before delegating to `\big`. In total, the 12 index notations for `\big` are duplicated for 3 different bigops in the file.

[3] `https://github.com/math-comp/math-comp/blob/master/mathcomp/ssreflect/bigop.v`

```
Notation "\sum_ ( i ← r ) F"     := (\big[addn/0]_(i ← r) F).
Notation "\sum_ ( i ← r | P ) F" := (\big[addn/0]_(i ← r | P) F).
...
Notation "\prod_ ( i ← r ) F"     := (\big[muln/1]_(i ← r) F).
Notation "\prod_ ( i ← r | P ) F" := (\big[muln/1]_(i ← r | P) F).
...
```

In contrast, in Lean 4, we can introduce a new syntactic category for index notations, interpret it once in \big, and define new bigops on top of it without any redundancy.

```
declare_syntax_cat index
syntax ident "←" term : index
syntax ident "←" term "|" term : index
syntax "\\big[" term:max "/" term "]_(" index ")" term : term
...

macro_rules
  | `(\big[$op/$idx]_($i:ident ← $r)      $F) => ...
  | `(\big[$op/$idx]_($i:ident ← $r | $p) $F) => ...
...

macro "Σ" "(" idx:index ")" F:term : term =>
  `(\big[Add.add/0]_($idx) $F)

macro "Π" "(" idx:index ")" F:term : term =>
  `(\big[Mul.mul/1]_($idx) $F)
```

The full example as implemented by Leo is included in the original paper's supplement. Please note that this is not merely a showcase of a parsing extension, but that abstracting over binding syntax in this manner is fundamentally incompatible with any static approach of ensuring hygiene. The dynamic nature of Scheme-like macros allows us to always apply such factorings without being burdened by static restrictions while still preserving hygiene.

We had decided early on that these examples were the minimum of generality the new macro system should support. What was less clear was how to do so in a hygienic manner given that previous approaches based on static analysis were not feasible anymore. Instead we looked at the Scheme family of languages and in particular Racket's new *Sets of Scopes* [Flatt, 2016] hygiene algorithm, which is described by its author as

being both simpler and more uniform than previous algorithms based on explicit renaming.

4.2 Hygiene Algorithm

In this section, I will give a mostly self-contained description of my algorithm for automatic hygiene applied to a simple recursive macro expander; I postpone comparisons to existing hygiene algorithms to Section 4.8.

Hygiene issues occur when transformations such as macro expansions lead to an unexpected capture (rebinding) of identifiers. For example, we would expect the notation

```
notation "const" e =>
  fun x => e
```

to always produce a constant function regardless of the specific `e`. We would not expect the term `const x` to be the identity function `fun x => x` because intuitively there is no `x` in scope at the argument position of `const`; that the *implementation* of the macro makes use of the name internally should be of no concern to the macro *user*.

Thus hygiene issues can also be described as a *confusion of scopes* when syntax parts are removed from their original context and inserted into new contexts, which makes name resolution strictly after macro expansion (such as in a compiler preceded by a preprocessor) futile. Instead we need to track scopes *as metadata* before and during macro expansion so as not to lose information about the original context of identifiers. Specifically: [4]

1. When an identifier captured in a syntax quotation matches one or (in the case of overloading) more top-level symbols, the identifier is annotated with a list of these symbols as *top-level* scopes to preserve its *extra-macro* context (which, because of the lack of local macros, can only contain top-level bindings). Here *top-level* is to be understood as in the *global context* in Section 2.2: the set of Lean declarations. For example, the identifier `map` may resolve to the

[4] Lean allows overloaded top-level bindings whereas local bindings are shadowing.

symbol list `{List.map,Option.map}` when the respective namespaces are opened.

2. When a macro is expanded, all identifiers freshly introduced by the expansion are annotated with a new *macro* scope to preserve the *intra-macro* context. In particular, different expansions of the same macro introduce different annotations. Macro scopes are appended to a list, meaning the list is implicitly ordered by expansion time as we only append fresh macro scopes, ensuring we do not have to worry about ordering when comparing two such lists. This full "history of expansions" is necessary to treat macro-producing macros correctly, as we shall see in Section 4.2.2.

Thus, the expansion of the above term `const x` should be (an equivalent of) `fun x.1 => x` where `1` is a fresh macro scope appended to the macro-introduced `x`, preventing it from capturing the `x` from the original input. In general, I will present hygienic identifiers in the following as $\texttt{n.msc}_1\texttt{.msc}_2\texttt{.}\dots\texttt{.msc}_n\texttt{\{tsc}_1\texttt{,}\dots\texttt{,tsc}_n\texttt{\}}$ where `n` is the original name, `msc` are macro scopes, and `tsc` top-level scopes, eliding the braces if there are no top-level scopes as in the example above. I use this dot notation to suggest both the ordered nature of macro scopes and their eventual implementation in Section 4.3. I will now describe how to implement these operations in a standard macro expander.

4.2.1 Expansion Algorithm

A Scheme-style macro expander takes a syntax tree as input and produces a fully expanded tree, that is, where all macro uses have been reduced to *core forms* that cannot be described as macros and are instead handled by the later stages, such as a direct interpreter or an elaborator. Important special cases are *binding* core forms, i.e. core forms such as Lean's lambda syntax `fun x =>` … that introduce new identifiers, and potential *references* to these bindings, that is, identifiers in any non-binding position. The expander should rename bindings and their references where necessary to avoid hygiene issues such that later stages do not have to know anything about the implementation of hygiene, or indeed that it was applied at all.

Given a *global context* (a set of symbols), the expansion algorithm does so by a conventional top-down expansion, keeping track of an initially-empty *local context* (another set of symbols). When a binding core form

is encountered, the local context is extended with the bound symbol(s); existing top-level scopes on the binding identifier are discarded at this step since they are only needed for references. Thus I will formally define a symbol as an identifier together with a list of macro scopes, such as `x.1` above. As we shall see in Section 4.3, this definition of symbol is covered by the pre-existing one in Lean, so later stages indeed do not have to concern themselves with it.

When a reference is encountered, it is resolved according to the following rules:

1. If the local context has an entry for the same symbol, the reference binds to the corresponding local binding; any top-level scopes are again discarded.
2. Otherwise, if the identifier is annotated with one or more top-level scopes or matches one or more symbols in the global context, it binds to all of these (to be disambiguated by the elaborator).
3. Otherwise, the identifier is unbound and an error is generated.

In the common incremental compilation mode of ITPs, every command is fully processed before subsequent commands. Thus, an expander for such a system will never extend the global context by itself, but pass the fully expanded command to the next compilation stage before being called again with the next command's unexpanded syntax tree and a possibly extended global context.

Notably, the expander does not introduce macro scopes by itself, either, much in contrast to other expansion algorithms. We instead delegate this task to the macro's implementation, though in a completely transparent way for all pattern-based and for conventional procedural macros. I claim that a macro should in fact be interpreted as an *effectful* computation since two expansions of the same identifier-introducing macro should not return the same syntax tree to avoid unhygienic interactions between them. Thus, as a *side effect*, it should apply a fresh macro scope to each newly introduced identifier. In particular, a syntax quotation should not merely be seen as a datum, but as an effectful value that obtains and applies this fresh scope to all the identifiers captured by it to immediately ensure hygiene for pattern-based macros. Procedural macros producing identifiers not originating from syntax quotations might need to obtain and make use of the fresh macro scope explicitly.

As an aside, note that forgoing to do so is not sufficient to reliably implement *anaphoric* or other hygiene-bending macros that make an identifier (conventionally `it` in Lisp languages) available in the scope of the macro caller, as discussed in [Barzilay et al., 2011]. Instead, I believe that the correct translation of anaphoric macros to Lean is to change such identifiers to *keywords* that do not participate in hygiene at all, analogous to the *syntax parameters* of [Barzilay et al., 2011]. An example for this is the `this` keyword introduced by tactics such as `have` that can be used to refer to the just-proved fact.[5]

I give a specific monad-based implementation of effectful syntax quotations as a regular macro in Section 4.3. But before that, let us look at some examples that illustrate the workings of the algorithm.

4.2.2 Examples

Given the following input,

```
def x := 1
def e := fun y => x
notation "const" e => fun x => e
def y := const x
```

a Lean-like system using the presented expansion algorithm should incrementally parse, expand, and elaborate each declaration before advancing to the next one. For a first, trivial example, let us focus on the expansion of the second line. At this point, the global context contains the symbol `x` (plus any default imports that we will ignore here). Descending into the right-hand side of the definition, the expander first adds `y` to the local context. The reference `x` does not match any local definitions, so it binds to the matching top-level definition.

In the next line, the built-in `notation` macro expands to the two declarations

```
syntax "const" term : term
macro_rules
  | `(const $e) => `(fun x => $e)
```

[5] `https://github.com/leanprover/lean4/blob/6d0c91c/src/Init/Notation.lean#L204-L207`

When a top-level macro application unfolds to multiple declarations, we expand and elaborate these incrementally as well to ensure that declarations are in the global context of subsequent declarations from the same expansion. When recursively expanding the `macro_rules` declaration (we will assume for this example that `macro_rules` itself is a core form) in the global context `{x, e}`, we first visit the syntax quotation on the left-hand side. The identifier `e` inside of it is in an antiquotation and thus not captured by the quotation. Instead it is a pattern variable, so we add `e` to the local context before visiting the right-hand side, where we find the quotation-captured identifier `x` and annotate it with the matching top-level definition of the same name; we do not yet know that it is in a binding position. When visiting the reference `e`, we see that it matches a local binding and do not add top-level scopes.

```
macro_rules
  | `(const $e) => `(fun x{x} => $e)
```

Visiting the last line

```
def y := const x
```

with the global context `{x, e}`, we descend into the right-hand side. We expand the `const` macro given a fresh macro scope `1`, which is applied to any captured identifiers.

```
def y := fun x.1{x} => x
```

We add the symbol `x.1` (discarding the top-level scope `x`) to the local context and finally visit the reference `x`. The reference does not match the local binding `x.1` but does match the top-level binding `x`, so it binds to the latter.

```
def y := fun x.1 => x
```

Now let us briefly look at a more complex example of a macro-macro, that is, a macro generating another macro, demonstrating use of the macro scopes stack:

```
macro "m" n:ident : command => `(
  def f := 1
  macro "mm" : command => `(
    def $n := f + 1
    def f := $n + 1))
```

If we use this macro as in `m f`, we apply a fresh macro scope 1 to all captured identifiers, then incrementally process the two new declarations.

```
def f.1 := 1
macro "mm" : command => `(
  def f := f.1{f.1} + 1
  def f.1{f.1} := f + 1)
```

If we use the new macro `mm`, we apply one more macro scope 2.

```
def f.2 := f.1.2{f.1} + 1
def f.1.2{f.1} := f.2 + 1
```

When processing these new definitions, we see that the scopes ensure the expected name resolution.

```
def f.1 := 1
...
def f.2 := f.1 + 1
def f.1.2 := f.2 + 1
```

In particular, we now have global declarations `f.1`, `f.2`, and `f.1.2` that show that storing only a single macro scope would have led to a collision.

4.3 Implementation

The implementation of the macro system is based on the `Syntax` type, which we have seen in Section 3.4. The most relevant constructor for hygiene is `Syntax.ident`.

```
inductive Syntax where
  | ident (info : SourceInfo) (rawVal : String) (val : Name)
    (preresolved : List (Nat × List String))
  | ...
```

Identifiers carry macro scopes inline in their `Name` while top-level scopes are held in a separate list `preresolved`. The additional `Nat` is an implementation detail of Lean's hierarchical name resolution.

The type `Name` of hierarchical names precedes the implementation of the macro system (indeed, we have seen its Lean 3 equivalent referenced before in Section 2.3) and is used throughout Lean's implementation for referring to (namespaced) symbols.

```
1  partial def expand : Syntax → ExpanderM Syntax
2    | `($id:ident) => do
3      let val : Name := getIdentVal id
4      let gctx ← getGlobalContext
5      let lctx ← getLocalContext
6      if lctx.contains val then
7        pure (mkIdent val)
8      else match resolve gctx val ++ getPreresolved id with
9        | []        => throw ("unknown identifier " ++ toString val)
10       | [(id, _)] => pure (mkIdent id)
11       | ids       => pure (mkOverloadedIds ids)
12   | `(fun ($id : $ty) => $e) => do
13     let val := getIdentVal id
14     let ty ← expand ty
15     let e ← withLocal val (expand e)
16     `(fun ($(mkIdent val) : $ty) => $e)
17   | ...  -- other core forms
18   | _ => do
19     let t ← getTransformerFor stx.getKind
20     let stx ← withFreshMacroScope (t stx)
21     expand stx
```

Figure 4.1: Abbreviated implementation of a recursive expander for the macro system

```
inductive Name where
  | anonymous
  | str (base : Name) (s : String)
  | num (base : Name) (n : Nat)
```

The syntax `` `a.b `` is a literal of type `Name` for use in meta-programs, just like in Lean 3. The numeric part of `Name` is not accessible from the surface syntax and reserved for internal names; similar designs are found in other ITPs. By reusing `Name` for storing macro scopes, but not top-level scopes, we ensure that the new definition of *symbol* from Section 4.2.1 coincides with the existing Lean type and no changes to the implementation of the local or global context are necessary for adopting the macro system.

A Lean 4 implementation of the expansion algorithm described in the previous section is given in Fig. 4.1; the full, executable implementation

including examples is included in [Ullrich and de Moura, 2023]. As a generalization, syntax transformers in the full implementation have the type `Syntax → TransformerM Syntax` where the `TransformerM` monad gives access to the global context and a fresh macro scope per macro expansion. The expander itself uses an extended `ExpanderM` monad based on `TransformerM` that also stores the local context and the set of registered macros. We use the Lean equivalent of Haskell's `do` notation (see also Chapter 5) to program in these monads.

As described in Section 4.2.1, the expander in Fig. 4.1 has built-in knowledge of some "core forms" (lines 2-16) with special expansion behavior, while all other forms are assumed to be macros and expanded recursively (lines 19-21). Identifiers form one base case of the recursion. As described in the previous section, their symbol is first looked up in the local context (line 6; recall that the `Name` of an identifier includes macro scopes), then as a fall back in the global context plus its own top-level scopes (line 8). `mkIdent : Name → Syntax` creates an identifier without source information or top-level scopes, which are not needed after expansion. `mkOverloadedIds` implements the Lean special case of overloaded symbols to be disambiguated by elaboration; systems without overloading support should throw an ambiguity error instead in this case.

As an example of a core binding form, the expansion of a single-parameter `fun` is shown in lines 12-16 of Fig. 4.1. It recursively expands the given parameter type, then expands the body in a new local context extended with the symbol value of `id`. Here `getIdentVal : Syntax → Name` in particular implements the discarding of top-level scopes from binders.

Finally, in the macro case, we look up the syntax transformer for the given syntax kind, run it in a new context with a fresh current macro scope, and recurse on the expansion result.

Syntax quotations are one example of a macro: they do not have built-in semantics but transform into code that constructs the appropriate syntax tree (`expandStxQuot` in Fig. 4.2). More specifically, a syntax quotation will, at run time (of the surrounding macro), query the current macro scope `msc` from the surrounding `TransformerM` monad (code generated by `expandStxQuot`) and apply it to all captured identifiers (code generated by `quoteSyntax`). `quoteSyntax` recurses through the quoted syntax tree, reflecting its constructors. Basic datatypes such as `String` and `Name` are turned into `Syntax` via the typeclass method `quote`. For antiquotations, we return their contents unreflected. In the case of identifiers, we resolve

```
1  partial def quoteSyntax : Syntax → TransformerM Syntax
2    | Syntax.ident info rawVal val preresolved => do
3      let gctx ← getGlobalContext
4      let preresolved := resolve gctx val ++ preresolved
5      `(Syntax.ident SourceInfo.none $(quote rawVal)
6          (addMacroScope $(quote val) msc) $(quote preresolved))
7    | stx@(Syntax.node k args) =>
8      if isAntiquot stx then pure (getAntiquotTerm stx)
9      else do
10       let args ← args.mapM quoteSyntax
11       `(Syntax.node $(quote k) $(quote args))
12   | Syntax.atom info val => `(Syntax.atom SourceInfo.none $(quote val))
13   | Syntax.missing => pure Syntax.missing
14
15 def expandStxQuot (stx : Syntax) : TransformerM Syntax := do
16   let stx ← quoteSyntax (stx.getArg 1)
17   `(do msc ← getCurrMacroScope; pure $stx)
```

Figure 4.2: Simplified syntax transformer for syntax quotations

possible global references at compile time and reflect them, while `msc` is applied at run time. Thus a quotation `` `(a + $b) `` inside a global context where the symbol `a` matches declarations `a.a` and `b.a` is transformed to the equivalent of

```
do let msc ← getCurrMacroScope
   pure (Syntax.node `plus [
     Syntax.ident SourceInfo.none "a" (addMacroScope `a msc)
       [`a.a, `b.a],
     Syntax.atom SourceInfo.none "+",
     b])
```

This implementation of syntax quotations itself makes use of syntax quotations for simplicity and thus is dependent on its own implementation in the previous stage of the compiler (Section 6.4). Indeed, the helper variable `msc` must be renamed should the name already be in scope and used inside an antiquotation.[6] Note that `quoteSyntax` is allowed to reference

6 As long as no such case exists, a hygienic implementation of syntax quotations can be

the same `msc` as `expandStxQuot` because they are part of the same macro call and the current macro scope is unchanged between them. While alternative approaches that use fresh macro scopes on function calls *within* a macro are thinkable, we prefer the presented behavior, which matches that of the Scheme family, because it preserves referential transparency: if `quoteSyntax` is inlined into `expandStxQuot`, the behavior is unchanged.

4.3.1 Extended Quasiquotations

Automatic hygiene can greatly simplify development of macros, but a convenient way for constructing and destructing syntax is at least as important. Before we get to more complex macro examples below, I will describe some syntactic extensions to quotations and antiquotations I have implemented that will come in useful.

A first obvious such extension is to allow quotations including antiquotations as patterns such as after `match` or `fun`.

```
fun
  | `(())        => ...
  | `(($e))      => ...
  | `(($e, $f)) => ...
```

Because every use of patterns eventually unfolds to a `match` in Lean, this is in fact implemented as a macro that expands `match` terms with quotation patterns into ones without such patterns. Note also that because no new syntactic identifiers are generated while matching against a quotation, there is no issue of hygiene with quotation patterns.

For syntax with repeated parts, antiquotation *splices* enable us to match or introduce these parts as a whole. For example, a recursive macro for n-tuple syntax can be written as

```
macro_rules
  | `(())             => `(Unit.unit)
  | `(($e))           => e
  | `(($e, $es,*)) => `(Prod.mk $e ($es,*))
```

Here `$es,*` matches/introduces the remaining elements of the tuple, including its separators. Analogous splicing syntax exists for other separators,

bootstrapped from an unhygienic one, which is what I did in the case of Lean.

as well as `$x*` for separator-less iteration. At most one splice can be used per sequence in the case of syntax patterns, but it can be pre- and suffixed with an arbitrary (but fixed) number of other elements.

In `$x*`, `x` has type `Array Syntax`, the same type as the second argument of the `node` constructor. For `$x,*` and similar we instead use the dependent wrapper type `SepArray ","` that provides convenience access functions for the sequence both with and without separator elements. Finally, we also provide implicit coercions between these types that automatically insert/remove/replace the separators accordingly.

While exposing splices as typed values in this way ensures that we can comfortably process or synthesize them procedurally as well, it is often more convenient to inspect splice contents immediately as part of the quotation. For this we support extended splices `$[...]*` etc. where the splice content is parsed like an element of the sequence and can contain nested antiquotations. If used as a pattern, the match succeeds if and only if the nested pattern matches every element, in which case the contained antiquotations are each bound to an `Array` of all corresponding element-wise matches.

```
match stx with
| `(match $discr with $[| $patss,* => $branches]*) =>
  -- discr : Syntax
  -- patss : Array (SepArray ",")
  -- branches : Array Syntax
  ...
```

By default, quotations are parsed as either terms or top-level commands, since these syntactic categories are both commonly used and should usually be disjoint. Other syntactic categories, e.g. the category of universe `levels` that heavily overlaps with `term`, can be specified explicitly at the beginning of a quotation. Similarly, antiquotations can be suffixed with a colon followed by a category or syntax kind where otherwise ambiguous.

```
match levelStx with
| `(level| $id:ident) => ...  -- a universe variable
| `(level| _)         => ...  -- a universe placeholder
| `(level| $l)        => ...  -- any (other) universe term
```

When no kind is given, the parser always defaults to the outermost applicable meaning in the grammar, e.g. `$l:level` above and not the more specific `$l:ident` from a nested grammar rule.

```
1 macro_rules
2   | `(fun
3         | $ps1,* => $rhs1
4         $alts:matchAlt*) => do
5     let discrs ← ps1.getElems.mapM (fun _ => withFreshMacroScope `(x))
6     `(fun $discrs* =>
7         match $[$discrs:ident],* with
8         | $ps1,* => $rhs1
9         $alts:matchAlt*)
```

Figure 4.3: A macro rule for expanding a combined fun-match syntax.

For a full example of using these and other features, we can look at a macro rule unfolding syntax such as

```
fun
  | some a, some b => some (a + b)
  | _,      _      => none
```

into

```
fun x.1 x.2 =>
  match x.1, x.2 with
  | some a, some b => some (a + b)
  | _,      _      => none
```

The macro rule (Fig. 4.3) derives the number of *discriminants* (`x.1, x.2` in the example) from the number of patterns of the first alternative; if other alternatives have differing number of patterns, it will lead to an elaboration error in `match` later on. The macro then introduces a lambda abstraction over a sequence of fresh variable names of this number and subsequently matches on them using the given patterns.

It does so by matching on the first alternative of the match in detail, then capturing the remaining ones in `alts : Array Syntax`. The left-hand side `ps1` of the first alternative is a `SepArray ","`, so we use `getElems` to access its elements and generate a fresh variable for each of them, which we do by running the single quotation `` `(x) `` repeatedly under `withFreshMacroScope`, annotating the variable with a unique macro scope each time.[7]

[7] This is comparable to a call to the `gensym` function found in many Lisp systems.

With `discrs : Array Syntax` generated, we insert it into the final quotation, once as a straight sequence after `fun` and once separated by commas after `match`, followed by the alternatives copied from the input without changes. Note that `$discrs:ident*` would not have worked in this case because identifiers are merely a special case of the more general `match` discriminant syntax that allows prefixing a discriminant with `h:`, where the identifier `h` will then hold the proof that the discriminant matched the corresponding pattern. Thus there is no direct identifier sequence to insert and we have to instead say that we insert a sequence of general discriminants, each one built up of an identifier without a proof variable prefix, which internally will wrap each element in an additional syntax tree node of the `matchDiscr` kind. This necessary disambiguation of overlapping syntax sadly is a price we have to pay for our preference of such syntax over more regular but verbose one such as S-expressions.

Finally, for the sake of completeness I will mention the *token antiquotation* syntax `%$x` that any token can be suffixed with to extract/set its `SourceInfo` metadata. This kind of antiquotation is mostly useful for displaying errors on specific tokens and preserving metadata in transformations.

```
| `(tactic| case $tag =>%$arrTk $tac) => do
  ...
  reportUnsolvedGoalsAt arrTk

...

  case cons => skip
          --^ unsolved goals displayed here
```

4.4 Typed Syntax

In the previous section, I mentioned how overlapping syntax can complicate macro development compared to e.g. S-expression-based systems. As one major refinement after publishing of the conference and journal paper, I have sought to rectify this downside by introduction of a more type-safe version of the homogeneous `Syntax` type and its integration into the macro system.

For an even more drastic example of this issue, take the following macro implementation.

```
syntax "mk_0" ident : command
macro_rules
  | `(mk_0 $id) => `(def $id := 0)
```

The macro looks innocent enough: given an identifier, it expands to a new declaration of that name that evaluates to zero. Unfortunately, it in fact creates an invalid syntax tree in the macro system described so far, and to understand why, we would need to look deep into the grammar of `def`, where we would find a parser declaration roughly equivalent to the following:

```
syntax declId := ident (".{" ident,+ "}")?
syntax defCmd := "def" declId ...
```

Thus `def` is not actually followed just by an identifier but a "declaration identifier", which is an identifier optionally followed by a declaration of universe levels of the form `id.{l, ...}`. As a missing optional part is still encoded as an empty `node` constructor in the syntax tree, the syntax tree structure of `declId` is always different from `ident` and it is never correct to pass a syntax tree of one of these kinds where the other is expected as the macro does above; instead, syntax kind annotations have to be applied carefully as in `def $id:ident` in order to insert the syntax tree at the correct location and make the parser emit the empty node for the missing optional part. The only reason the macro did not lead to some elaboration failure in previous versions of Lean 4 is that the mistake was so common that the `def` elaborator explicitly checked for this case and accepted the raw identifier in place of the `declId`.

Obviously a more robust solution was needed. As this issue is similar to a traditional type confusion error even if syntax kinds are not directly Lean types, I have implemented a new component of the macro system that lifts them to the type level via a wrapper structure over the homogeneous `Syntax` type.

```
structure TSyntax (kinds : List Name) where
  raw : Syntax
```

A value of type `TSyntax ks` is supposed to contain a syntax node of any kind listed in `ks`. Most often this list consists of a single element, and there is a coercion eliding the list for this common case as in ``TSyntax `declId``. While the new type can by itself be helpful for giving more precise type signatures such as ``getIdentVal : TSyntax `ident → Name``, the crucial part

is integration into the quasiquotation elaborator, which provides the main ways of producing and consuming typed syntax: a quasiquotation with or without kind annotation, both inside a pattern and outside, now always is of type `TSyntax ks` where `ks` contains the applicable syntax kinds at this position in the input, following the same *outermost* rule as described above. Thus in `` `($e) `` we still end up with `` e : TSyntax `expr `` and not a list containing `` `expr `` as well as all specific `expr` kinds, which would be a large unhelpful list that because of extensibility could not be exhaustive in any case. A non-singleton kind list mostly occurs from uses of the non-extensible `<|>` alternation operator instead of an extensible syntax category.

```
syntax "strOrNum" (str <|> num) : term
macro_rules
  | `(strOrNum $x) => ...  -- x : TSyntax [`str, `num]
```

Finally, splices analogously have been changed to use kind-safe types representing sequences of `TSyntax` objects with and without separators.

With this system in place, our original macro now raises a type error instead of silently creating invalid syntax trees when executed.

```
macro_rules
  | `(mk_0 $id) => `(def $id := 0)       argument
                                           id
                                         has type
                                           TSyntax `ident : Type
                                         but is expected to have type
                                           TSyntax `declId : Type
```

However, we can do even better: instead of using types merely to reject invalid programs, we can use them to fix our programs as well. As the given macro implementation is still intuitively reasonable to us, we can make sure it is accepted as is with Lean taking care of the uninteresting detail of kind conversion by registering a coercion implementing that part, which is now part of the Lean core library:

```
instance : Coe (TSyntax `ident) (TSyntax `declId) where
  coe id := Unhygienic.run `(declId| $id:ident)
```

Here `Unhygienic.run` allows us to run the effectful quotation in a pure context, which is safe to do as long as there are no identifiers introduced

by the quotation like in this case. While it would be possible to derive such coercions automatically from the grammar, it is not clear to us whether that is always desirable, and we found that in practice a few key coercions are sufficient to reduce the need for explicit kind annotations.

Our experience with the new typed syntax system has been very positive. It has successfully prevented users from falling into kind confusion traps as above and removed the need for reflexively specifying kind annotations for all antiquotations out of an abundance of caution because the previous rules were too subtle to rely on in practice.

4.5 Integrating Macros into Elaboration

The macro system as described so far can handle most syntax sugars of previous versions of Lean except for (then built-in) ones requiring type information. For example, the *anonymous constructor* `⟨e, ...⟩` is sugar for the application `(c e ...)` if the expected type of the expression is known and it is an inductive type with a single constructor `c`. While trivial to parse, there is no way to implement this syntax as a macro if expansion is done strictly prior to elaboration. A more complex example is the *structure instance notation* `{ field1 := e, ... }` that must analyze the definition of the given or inferred structure type in order to expand to the correct constructor call. To the best of my knowledge, none of the ITPs listed in the introduction support hygienic elaboration extensions of this kind, but I will show how to extend their common elaboration scheme in that way in this section.

Elaboration[8] can be thought of as a function `elabTerm : Syntax → ElabM Expr` in an appropriate monad `ElabM`[9] from a (concrete or abstract) surface-level syntax tree type `Syntax` to a fully-specified core term type `Expr`. The particular definition of `Expr` is not important in the following. While such an elaboration system could readily be composed with a type-insensitive macro expander such as the one presented in Section 4.2, we would rather like to *intertwine* the two to support type-sensitive but still hygienic-by-default macros (henceforth called *elaborators*) without having to reimplement macros of the kind discussed so far. Indeed, these can automatically

[8] at the term level; elaboration of other syntactic categories works analogously but with different output types

[9] or some other encoding of effects

be adapted to the new type given an adapter between the two monads, similarly to the adaption of macros to *expanders* in [Dybvig et al., 1986]:

```
def transformerToElaborator (t : Syntax → TransformerM
 Syntax) :
Syntax → ElabM Expr :=
fun stx => do
let stx' ← (transformerMToElabM t) stx
elabTerm stx'
```

Because most parts of the new hygiene system are implemented by the expander for syntax quotations, the only changes to an elaboration system necessary for supporting hygiene are storing the current macro scope in the elaboration monad (to be passed to the expansion monad in the adapter) and allocating a fresh macro scope whenever a macro or elaborator is invoked. Thus elaborators immediately benefit from hygiene as well whenever they use syntax quotations to construct unelaborated helper syntax objects to pass to `elabTerm`. In order to support syntax quotations in these two and other monads, I generalized their implementation to a new monad typeclass implemented by both monads.

```
class MonadQuotation (m : Type → Type) where
  getCurrMacroScope : m MacroScope
  withFreshMacroScope : m α → m α
```

The second operation is not used by syntax quotations directly, but can be used by procedural macros and elaborators to manually enter new macro call scopes.

As an example, the following is a simplified implementation of the anonymous constructor syntax mentioned above.

```
elab_rules <= expectedType
  | `(⟨$args,*⟩) => do
    match Expr.getAppFn expectedType with
    | Expr.const constName _ => do
      let ctors ← getCtors constName
      match ctors with
      | [ctor] => do
        let stx ← `($(mkCIdent ctor) $args*)
        elabTerm stx
```

`elab_rules` is analogous to `macro_rules`: instead of a transformer, it expects a computation in the elaboration monad returning `Expr` (when given a term syntax). Additionally, the expected type can be requested as an additional elaboration input after <=. If the type is not sufficiently known when the elaborator is invoked, it is automatically postponed (Section 3.5). The function `mkCIdent : Name → Syntax` synthesizes a hygienic reference to the given constant name by storing it as a top-level scope and applying a reserved macro scope to the constructed identifier. Note the monadic binding of the syntax quotation, and that the separators of `$args,*` are implicitly discarded when it is used as a plain sequence `$args*`.

4.6 Tactic Hygiene

As we have seen in Chapter 2, Lean 3 includes a tactic framework that, much like macros, allows users to write custom automation either procedurally inside a tactic monad or "by example" using tactic language quotations, or in a mix of both. For example, Lean 3 uses a short tactic block to prove injection lemmas for data constructors.

```
def mkInjEq : TacticM Unit :=
  `[intros; apply propext; apply Iff.intro; ...]
```

Unfortunately, this code unexpectedly broke in Lean 3 when used from a library for homotopy type theory that defined its own `propext` and `Iff.intro` declarations;[10] in other words, Lean 3 tactic quotations are unhygienic and required manual intervention in this case. Just like with macros, the issue with tactics is that binding structure in such embedded terms is not known at declaration time. Only at tactic run time do we know all local variables in the current context that preceding tactics may have added or removed, and therefore the scope, if any, of each identifier captured in the quotation.

Arguably, the Lean 3 implementation also exhibits a lack of hygiene in the handling of tactic-introduced identifiers: it does not prevent users from referencing such an identifier outside of the scope it is declared in.

```
def myTac : TacticM Unit := `[intro h]
lemma triv (p : Prop) : p → p := begin myTac; exact h end
```

[10] https://github.com/leanprover/lean/pull/1913

Coq's similar Ltac tactic language [Delahaye, 2000] exhibits the same issue and users are advised not to introduce fixed names in tactic scripts but to generate fresh names using the `fresh` tactic first,[11] which can be considered a manual hygiene solution.

Lean 4 instead extends its automatically hygienic macro implementation to tactic scripts by allowing regular macros in the place of tactic invocations.

```
macro "myTac" : tactic => `(intro h; exact h)
theorem triv (p : Prop) : p → p := by myTac
```

By the same hygiene mechanism described above, introduced identifiers such as `h` are renamed so as not to be accessible outside of their original scope, while references to global declarations are preserved as top-level scope annotations. Thus Lean 4's tactic framework resolves both hygiene issues discussed here without requiring manual intervention by the user. Expansion of tactic macros in fact does not precede but is integrated into the *tactic interpreter* `evalTactic : Syntax → TacticM Unit` such that recursive macro calls are expanded lazily, allowing for combinators like `repeat` that would otherwise lead to infinite recursion during expansion.

```
syntax "repeat" tactic : tactic
macro_rules
  | `(tactic| repeat $t) => `(tactic| try ($t; repeat $t))
```

Note that the `macro` shorthand cannot be used in this case because the parser for `repeat` would not yet be available in the right-hand side. When `$t` eventually fails, the recursion is broken without visiting and expanding the subsequent `repeat` macro call. The `try` tactical is used to ignore this eventual failure.

While we believe that macros will cover most use cases of Lean 3's tactic quotations in Lean 4, their use within larger `TacticM` metaprograms can be recovered by passing such a quotation to `evalTactic`:

```
def myTac2 : TacticM Unit := do
  let stx ← `(tactic| intro h; exact h)
  evalTactic stx
```

`TacticM` implements the `MonadQuotation` typeclass for this purpose.

[11] https://github.com/coq/coq/issues/9474

4.7 Best-Effort Eager Name Analysis in Macros

The dynamic nature of binding in macros has enabled us to implement many Lean language features as macros, with hygiene guaranteeing that bindings within the macro do not interfere with ones outside of it. However, while knowing that a mistyped identifier in a macro will not be accidentally be bound by bindings at the use site is great in theory, in practice it would be even better to be told about the typo immediately while writing the macro! This is especially true when renaming a declaration, either manually where we might accidentally miss an occurrence of it inside a macro and then must track name binding errors at use sites back to the responsible macro, or automatically where refactoring tools must treat macros as black boxes. After all, a static view of a notation's binding structure is exactly what we gave up in this chapter's introduction in exchange for the ability to arbitrarily abstract over bindings. For example, there is no general way to statically analyze whether `x` inside the following quotation is well-scoped given an arbitrary computation resulting in `stx`:

```
let stx ← ...
`(fun $stx => x)
```

With this theoretical limitation in mind, and given that this is more of a practical issue of maintenance, perhaps a practical, best-effort solution is sufficient as long as we retain all hygiene guarantees. I have done so with an *opt-in*, partial but extensible approach to eager name resolution in macros based on a variant of our quotation syntax.[12]

```
``(fun x => x + $y + id z)  -- error: unknown identifier 'z'
```

A *double-backtick* quotation eventually unfolds to the basic, single-backtick version and thus retains all its semantics. Before that, however, it recursively checks for identifiers that can statically be assumed to be unbound using the following heuristics:

1. If there is a special *precheck* hook registered for the syntax kind in question, we use it. Precheck hooks can signal binding errors as well as recursively continue the precheck on nested syntax, possibly with

[12] Note that, in contrast to Lean 3, the meaning of the number of backticks before a quotation or name literal is consistent in Lean 4: in both cases, the second backtick adds static name resolution.

an extended (untyped) *quotation context*, which is initially empty. For example, we provide a precheck hook for `fun x => e` that recurses into `e` after adding `x` to the quotation context. The central *identifier* precheck hook ultimately raises an error if an identifier is reached that is neither in the global, extra-macro context, nor in the quotation context. Other examples for precheck hooks I have added are for `match` and function application.

2. Otherwise, if there are no identifiers in the quoted syntax, we assume that there is no risk of unbound ones, and the check succeeds. In particular, antiquotations (which contain *unquoted* identifiers only) are always skipped.

3. Otherwise, if the quoted syntax is a macro, we unfold it and precheck the result. Here we assume that the macro is sufficiently good-natured: while macros are pure functions by definition in Lean, their behavior and in particular binding structure could in theory change drastically enough in the presence of antiquotations that it could lead to false positives of this analysis. If that is the case, the user either has to provide a custom precheck hook for it, or fall back to the basic quotation syntax.

   In contrast to macros, we cannot typically run elaborators directly during this check because they usually depend on type information that is not yet available, and are not guaranteed to be pure.

4. Otherwise the check fails since we do not know how to analyze the syntax at hand. Again, users would either have to provide a precheck hook, or use the single-backtick quotation syntax.

This approach is clearly best-effort with many opportunities for unhandled cases. However, since it is guaranteed that after the check the semantics are the same as for the basic quotation syntax, including unchanged hygiene guarantees, we believe that the practical advantages of using the syntax where possible are significant. In particular, we have observed that quotations capturing global identifiers, which are most at risk of breaking during refactorings, are usually quite simple while complicated quotations that are used to translate one syntax into a more general one often do not reference any identifiers at all, and thus are trivially accepted by rule 2 from above.

This is especially true for notation right-hand sides, which are usually exceedingly simple in structure, most often consisting of nothing but an application of a global function symbol to the notation arguments, which is covered by the identifier and application precheck hooks. Thus I have modified the `notation` macro to use double-backtick quotations by default in order to make users immediately aware of any unbound identifiers inside of them, restoring its Lean 3 behavior (where the absence of macros naturally allowed for such a check).

```
notation "∃" x "," e => Exits.intro (fun x => e)  -- error: unknown
    identifier 'Exits.intro'
```

We provide an option to disable this change, though all notations in the standard library passed the additional check without modification. The check did on the other hand find a notation example in our documentation that contained an accidentally unbound identifier.

4.8 Related Work

The main inspiration behind the hygiene implementation presented in this chapter was Racket's *Sets of Scopes* hygiene algorithm [Flatt, 2016]. Much like in the approach described above, Racket annotates identifiers both with scopes from their original context as well as with additional macro scopes when introduced by a macro expansion. However, there are some significant differences: Racket stores both types of scopes in a homogeneous, unordered set and does name resolution via a maximum-subset check. For both simplicity of implementation and performance, I have reduced scopes to the bare minimal representation using only strict equality checks, which we can easily encode in our existing `Name` implementation. In particular, we only apply scopes to matching identifiers and only inside syntax quotations. This optimization is of special importance because top-level declarations in Lean and other ITPs are not part of a single, mutually recursive scope as in Racket, but they each open their own scope over all subsequent declarations, which would lead to a total number of scope annotations quadratic in the number of declarations using the Sets of Scopes algorithm. Finally, Racket detects macro-introduced identifiers using a "black-box" approach without the macro's cooperation following the marking approach of [Kohlbecker et al., 1986]: a fresh macro scope is applied to all identifiers

in the macro input, then inverted on the macro output. While elegant, a naive implementation of this approach can again result in quadratic runtime compared to unhygienic expansion and requires further optimizations in the form of lazy scope propagation [Dybvig et al., 1993], which is more difficult to implement in a pure language like Lean. Lean's "white-box" approach based on the single primitive of an effectful syntax quotation, while slightly easier to escape from in procedural syntax transformers, is simple to implement, incurs minimal overhead, and is equivalent for pattern-based macros.

The idea of automatically handling hygiene in the macro, and not in the expander, was introduced in [Clinger and Rees, 1991], though only for pattern-based macros. MetaML [Taha and Sheard, 2000] refined this idea by tying hygiene more specifically to syntax quotations, which Template Haskell [Sheard and Peyton Jones, 2002] interpreted as monadic computations requiring access to a fresh-names generator, much like in our design. However, both of the latter systems should perhaps be characterized more as metaprogramming frameworks than Scheme-like macro systems: there are no "macro calls" but only explicit splices and so only built-in syntax with known binding semantics can be captured inside syntax quotations. Thus the question of which captured identifiers to rename becomes trivial again, just like in the basic notation systems discussed in this chapter's introduction.

While the vast majority of research on hygienic macro systems has focused on S-expression-based languages, there have been previous efforts on marrying that research with non-parenthetical syntax, with different solutions for combining syntax tree construction and macro expansion. The Dylan language requires macro syntax to use predefined terminators and eagerly scans for the end of a macro call using this knowledge [Bachrach et al., 1999], while in Honu [Rafkind and Flatt, 2012] the syntactic structure of a macro call is discovered during expansion by a process called "enforestation". The Fortress language [Allen et al., 2005] strictly separates the two concerns into grammar extensions and transformer declarations, much like we do. Dylan and Fortress are restricted to pattern-based macro declarations and thus can make use of simple hygiene algorithms while Honu uses the full generality of the Racket macro expander. On the other hand, Honu's authors "explicitly trade expressiveness for syntactic simplicity" [Rafkind and Flatt, 2012]. In order to express the full Lean language and desirable extensions in a macro

```
method my_allI =
  rule_tac allI, rename_tac escaped

lemma "∀ x. x = x"
  apply my_allI
  apply (rule_tac t = escaped in refl)
```

Figure 4.4: Unstructured Eisbach proof method that unhygienically introduces an identifier escaped that can be used outside of its original scope.

system, we require both unrestricted syntax of macros and procedural transformers.

Many of the antiquotation extensions presented in Section 4.3.1 have been inspired by similar syntax in Rust's pattern-based macro language, though we have opened their use to procedural macros as well, using appropriate type representations.

Many theorem provers such as Agda, Coq, Idris, and Isabelle not already based on a macro-powered language provide restricted syntax extension mechanisms, circumventing hygiene issues by statically determining binding as seen in this chapter's introduction. Extensions that go beyond that do not come with automatic hygiene guarantees. Agda's macros,[13] for example, operate on the de Bruijn index-based core term level and are not hygienic.[14] The ACL2 prover in contrast uses a subset of Common Lisp as its input language and adapts the hygiene algorithm of [Dybvig et al., 1993] based on renaming [Eastlund and Felleisen, 2010]. The experimental Cur theorem prover [Chang et al., 2019] is a kind of dual to Lean's approach: it takes an established language with hygienic macros, Racket, and extends it with a dependent type system and theorem proving tools. ACL2 does not support tactic scripts, while in Cur they can be defined via regular macros. However, this approach does not currently provide tactic hygiene as defined in Section 4.6.[15] The Eisbach proof method language [Matichuk et al., 2016] for Isabelle is notable in that while it

[13] https://agda.readthedocs.io/en/v2.6.0.1/language/reflection.html#macros
[14] https://github.com/agda/agda/issues/3819
[15] https://github.com/wilbowma/cur/issues/104

allows for reusable proof methods (comparable to Lean's tactic macros) to be abstracted over terms, these terms are analyzed and typechecked before being passed to the proof method, and there is no interaction between names in different proof methods either when using the "structured" proof style, making the question of hygiene moot. Using the "unstructured" proof style, the same hygiene issue as noted for other systems can be encountered (Fig. 4.4).

I seek to stretch your imaginations towards what programming can be if we choose to make it so
– Conor McBride, *Epigram: Practical Programming with Dependent Types* [McBride, 2005]

5 An Imperative Extension of *do* Notation

The success story of Haskell's perhaps most well-known abstraction, the monad as popularized by [Wadler, 1990a], is by now invariably linked with its ubiquitous syntax sugar, the `do` notation. Together they can express imperative sequencing on a more general, well-behaved abstraction level while retaining a terse, familiar, and suggestive syntax. They are in fact so commonly linked, and taught, together that programmers tend to use them even when weaker, potentially more performant abstractions would suffice [Marlow et al., 2016].

It is then surprising that no serious attempts to add more imperative control flow techniques over just sequencing seem to have been made, unless one considers that most of these only carry their weight in the presence of mutable variables. For example, there is not much reason to introduce an `if _ then _` syntax without an `else` branch to Haskell when there is already the `when` combinator that can be used to the same effect. Mutable variables in turn are of course seen as an antithesis to purely functional programming's core tenet of referential transparency.

On the other hand, even imperative languages have started to rein in mutability and make it the exception, not the rule. For example, in Rust [Matsakis and Klock, 2014], `let`-bound variables are immutable unless explicitly marked `mut`, and mutable references are non-shareable to prevent "spooky actions at a distance". Ultimately, mutable variables in imperative languages (excluding mutable references) are in fact commonly compiled down to static single assignment form [Rosen et al., 1988], which is known to be equivalent to a subset of continuation passing style [Kelsey, 1995], meaning it is certainly possible to transform them to

pure code.

Starting with mutable variables, I thus explore embracing imperative language features as part of the do notation in this chapter. I do so in the context of Lean 4, though the implementation can readily be adapted to any other functional language with support for monads and monad transformers. We will consider three extensions in total, which at this point I merely want to tease using suggestive side-by-side examples of Rust and Lean code:

- the mentioned mutable variables (Section 5.1),

```
let mut x = read_int();
if x != 0 {
  x = f(x);
}
...
```

```
do let mut x ← readInt
   if x != 0 then
     x := f x
   ...
```

- early return (Section 5.2),

```
let line = read_line();
if line == "" {
  return
}
...
```

```
do let line ← readLine
   if line = "" then
     return ()
   ...
```

- and finally iteration in the form of for loops (Section 5.3).

```
let lines = read_lines();
let mut sum = 0;
for line in lines {
  if line == "END" {
    break
  }
  sum += parse_int(line);
}
...
```

```
do let lines ← readLines
   let mut sum := 0
   for line in lines do
     if line = "END" then
       break
     sum := sum + parseInt
    line
   ...
```

In each section, I will motivate the respective extension using examples, discuss possible syntax and semantics, focusing on those that are least surprising to both imperative and functional programmers, and give formal translation rules into purely functional code. For validation of the translation, I present an incremental, one-to-one implementation using Lean's flexible macro system in Appendix A and discuss it in Section 5.4.1. I also discuss a more extensive implementation built into Lean itself (Section 5.4.2). I show that the code produced by the translation can still be analyzed and reasoned over using the same proof construction tools we use for pure code (Section 5.5), and formally prove in Lean that the translation preserves the input's static and dynamic semantics specified as a simple type system and natural semantics (Section 5.6; the full proof can be found in Appendix B).

Contributions. In this chapter, I extend the `do` domain-specific language with support for mutable variables as well as the familiar imperative statements `return`, `for`, `break`, and `continue`, usable in both monadic and pure computations. I give a formal translation of these extensions, backed by a reference implementation and a formal equivalence proof in Lean, down to well-known combinators so that the core language does not have to be changed. Throughout, I discuss issues of syntax, semantics, compilation, and proving.

Acknowledgements. The work described in this chapter is joint work with Leonardo de Moura [Ullrich and de Moura, 2022b]. I designed the language extension after discussions with Daniel Selsam about avoiding boilerplate in monadic programming in Lean. Leo later added further extensions such as try-catch blocks not described here. I authored the formal translation rules below, the reference implementation in Appendix A, and the semantics preservation proof in Appendix B; the full implementation in Lean (Section 5.4.2) on the other hand is largely due to Leo.

5.1 Local Mutation

One incentive for introducing mutable variables to Lean was the proliferation of the following "conditional update" pattern in the Lean codebase:

```
do ...
   x ← if b then f x else pure x
   ...
```

One cannot help but notice that the `else` branch feels redundant: nothing of interest is happening in it.[1] While the boilerplate in this abstract case is still minimal, the issue compounds in the presence of multiple variables with names of more realistic length.

```
do ...
   (aVar, anotherVar) ← if someCondition then do
       aVar ← transform1 aVar
       anotherVar ← transform2 aVar anotherVar
       pure (aVar, anotherVar)
     else pure (aVar, anotherVar)
   ...
```

This code is still easy enough to scan and comprehend, but we would be hard-pressed to defend it to e.g. a programmer coming from Rust, who might expect something more like

```
do let mut aVar ...
   ...
   if someCondition then
     aVar ← transform1 aVar
     anotherVar ← transform2 aVar anotherVar
   ...
```

No matter how much we extoll the virtues of purely functional code to the programmer, it is unlikely we can persuade them, or ourselves, that the latter code is not easier to read, comprehend, and ultimately maintain. At the same time, the boilerplate is mostly about bindings and not data so that we cannot profitably abstract it into a new (higher-order) combinator. Thus we finally relent and instead perform the before-unthinkable: we will try to assign the above code semantics that are reasonable in the eyes of both functional and imperative programmers.

Let us start by copying the careful approach of Rust and other languages of introducing separate binding syntax for mutable variables: `let mut x`

[1] Note that monadic bindings in both Haskell and Lean are non-recursive, thus shadowing works as expected in this case.

`:= ...`, and `let mut x ← ...` for the monadic form. This way, we can make sure that users will not accidentally make use of local mutation without even knowing it exists. For a thus introduced variable `x`, `x := ...` seems like the obvious choice of syntax for reassigning the variable to the value of a pure term. Unfortunately, the corresponding syntax with a monadic ← is already taken. Since distinguishing between declaration and reassignment is certainly a good idea, for Lean 4 we have decided on the drastic step of realigning the monadic binding syntax to the pure one and changing the syntax for introducing an immutable binding to `let x ← ...` as seen in the examples in the introduction[2]; we have seen the older Lean 3, Haskell-like syntax without `let` in Section 2.2. Thus a variable-with-value definition in Lean 4 is uniformly signified by the `let` keyword.

With syntactic questions out of the way, let us now turn to the expected semantics, starting with the question of scope. Declaring a mutable variable should grant us access to its value in the subsequent code just like with immutable ones, but reassignment will have to be more limited: if there is another `do` block nested anywhere inside the first one, say within an argument to some monadic combinator, there is no way in general to propagate reassignments back to the outer block without true mutation. Neither changing the result type nor introducing a new monadic layer to do the propagation is guaranteed to work (or even typecheck) in all contexts of the inner block. Thus we will sensibly restrict reassignment of a variable to the same `do` block it was declared in using `let mut`. This also resolves the question as to how mutable variables should behave when the block is executed multiple times: in any single execution, the variable is first freshly declared and then mutated, so the executions are independent. However, a similar but more subtle problem exists for some specific monads, those that may execute the >>= right-hand side more than once, such as the list/nondeterminism and the continuation monad:

```
do let mut x := 0
   let y ← choose [0, 1, 2, 3]
   x := x + 1
   guard (x < 3)
   pure (x + y)
```

[2] In fact, the "new" syntax is reminiscent of the original monadic binding notation (let $x \Leftarrow e$ in e') in [Moggi, 1991].

In usual implementations of nondeterminism, this program will bind `y` to the values 0 to 3 in turn, execute the remainder of the block with each of them, and then collect all the results from the different executions in a list. The `guard` function discards the "strands" of execution for which the given condition is false. Should the reassignment of `x` then persist into further executions, creating the output `[1, 3]`, or should it be limited to the current execution only, yielding `[1, 2, 3, 4]`? I argue that the more intuitive (and implementable) semantics is the latter one, where local mutation is interpreted as a *local state effect on top* of the underlying monad. Thus re-running a nondeterministic computation or captured continuation is still "pure": mutable variables will start out with the same values as in the first run. With the alternative semantics, it would not even be the case that our initial examples from the beginning of the section are indeed equivalent in all monads. If we wanted to implement these "impure" semantics, we would to introduce the state effect *below* the nondeterminism effect (or rely on existing arbitrary state from the base monad such as `IO` or `ST`), which is not an option in general as not all monad transformers commute.

Now that the desired semantics are established, let us start the discussion of the formal translation of these semantics with a basic syntax and desugaring of do similar to that in the Haskell 98 report [Peyton Jones, 2003] (Figure 5.1). Here do is followed by one of two kinds of *statements*, which are defined inductively: either a plain expression, or a monadic binding followed by another statement. The value of an expression as a statement is that of the expression (D1), while a monadic binding desugars to an application of the monadic bind operator >>= (D2). We do not directly rewrite occurrences of do into other do terms but do so using a recursive helper function *D*, which will become useful starting with the next section. For the sake of presentation, let us also introduce a syntax for *pure* let bindings as an abbreviation for monadic bindings of applications of the `pure` function (A1). In the translation rules, I will assume that all abbreviations have already been unfolded. A more direct translation of pure let bindings is certainly possible[3], but equivalent in Lean under the standard monad laws (Figure 5.2), so I will use this shortcut to minimize the number of

[3] and necessary in languages like Haskell where pure and monadic bindings have different shadowing/recursive semantics

Syntax

$$
\begin{aligned}
x, y &\in \mathit{Var} \\
e \in \mathit{Expr} &::= \texttt{do}\ s \\
&\quad \mid x \mid \texttt{fun}\ x\ \texttt{=>}\ e \mid \texttt{let}\ x\ \texttt{:=}\ e\ \texttt{in}\ e' \mid e\ e' \mid e\ \texttt{>>=}\ e' \mid \ldots \\
s \in \mathit{Stmt} &::= e \\
&\quad \mid \texttt{let}\ x \leftarrow s\texttt{;}\ s'
\end{aligned}
$$

Translation

$$
\texttt{do}\ s \quad \rightsquigarrow \quad D(s) \tag{5.1}
$$

$$
\begin{aligned}
&D : \mathit{Stmt} \to \mathit{Expr} \\
&D(e) = e && \text{(D1)} \\
&D(\texttt{let}\ x \leftarrow s\texttt{;}\ s') = D(s)\ \texttt{>>=}\ \texttt{fun}\ x\ \texttt{=>}\ D(s') && \text{(D2)}
\end{aligned}
$$

Abbreviations

$$
\begin{aligned}
\texttt{let}\ x\ \texttt{:=}\ e\texttt{;}\ s &\equiv \texttt{let}\ x \leftarrow \texttt{pure}\ e\texttt{;}\ s && \text{(A1)} \\
s\texttt{;}\ s\texttt{'} &\equiv \texttt{let x} \leftarrow s\texttt{;}\ s\texttt{'} && \text{(A2)}
\end{aligned}
$$

Figure 5.1: A basic do translation

translation cases that we need to consider. Similarly, we will restrict ourselves to variable bindings instead of more complex pattern bindings, but the translation naturally extends to the latter, as is done in the full implementation in Lean 4 (Section 5.4.2). Finally, we introduce statement sequencing s; s' not as a primitive but as yet another abbreviation, in this case for a binding to a fresh variable name `x` (contrasted by its monospace font from *metavariables* in italics) (A2). For simplicity, I will assume in the following that the underlying rewriting system is hygienic as is the case for my reference implementation (Section 5.4.1) utilizing the macro system presented in Chapter 4 in order to avoid conflicts between such identifiers introduced by rewriting rules and user-specified names.

I note that the grammar as presented has been optimized for translation,

Operations

$$\texttt{pure} : \alpha \to m\ \alpha$$
$$(\texttt{>>=}) : m\ \alpha \to (\alpha \to m\ \beta) \to m\ \beta$$

Laws

$$(x \texttt{ >>= } f) \texttt{ >>= } g \quad = \quad x \texttt{ >>= } (\texttt{fun } a \texttt{ => } f\ a \texttt{ >>= } g) \tag{5.2}$$
$$\texttt{pure } a \texttt{ >>= } f \quad = \quad f\ a \tag{5.3}$$
$$f \texttt{ >>= pure} \quad = \quad f \tag{5.4}$$

Figure 5.2: Basic monadic operations and their laws. `pure` is usually defined in the more general `Applicative` typeclass, which comes with even more operations and laws, but I will focus on monads for this chapter. In the first law and its applications in the text we assume tacitly, w.l.o.g., that a is not free in g.

not parsing, and is thus ambiguous regarding the associativity of semicolons. I will tacitly assume that they always associate to the right and will use parentheses where necessary, i.e. $\texttt{let}\ x \leftarrow s_1\texttt{;}\ s_2\texttt{;}\ s_3$ represents the same statement as $\texttt{let}\ x \leftarrow s_1\texttt{;}\ (s_2\texttt{;}\ s_3)$, not $\texttt{let}\ x \leftarrow (s_1\texttt{;}\ s_2)\texttt{;}\ s_3$ nor $(\texttt{let}\ x \leftarrow s_1\texttt{;}\ s_2)\texttt{;}\ s_3$. In the reference implementation, I resolve the issue using parsing precedences and a curly braces notation for grouping statements. The full implementation in Lean also supports a Haskell-inspired indentation-sensitive syntax as seen in this chapter's introduction, which I will use in examples.

The syntax from Figure 5.1 already diverges from that known from Haskell or previous versions of Lean in two distinct ways: firstly, we denote both kinds of bindings with a leading `let` keyword as discussed above. Secondly, our monadic binding binds not just a term but another statement. With the grammar at hand, this isn't very useful yet because nested bindings can always be floated out by the monad associativity law

Syntax

$$
\begin{aligned}
s \in \mathit{Stmt} ::= {} & \ldots \\
& \mid \texttt{let mut } x \texttt{ := } e\texttt{; } s \\
& \mid x \texttt{ := } e \\
& \mid \texttt{if } e \texttt{ then } s
\end{aligned}
$$

Abbreviations

$$\texttt{let mut } x \leftarrow s\texttt{; } s' \equiv \texttt{let y} \leftarrow s\texttt{; let mut } x \texttt{ := y; } s' \tag{A3}$$

$$x \leftarrow s \equiv \texttt{let y} \leftarrow s\texttt{; } x \texttt{ := y} \tag{A4}$$

Figure 5.3: Syntax of a do with local mutation

as follows:

$$
\begin{aligned}
& D(\texttt{let } x \leftarrow \texttt{(let } y \leftarrow s\texttt{; } s'\texttt{); } s'') \\
\overset{\text{(D2)}}{=} & D(\texttt{let } y \leftarrow s\texttt{; } s') \texttt{ >>= fun } x \texttt{ => } D(s'') \\
\overset{\text{(D2)}}{=} & (D(s) \texttt{ >>= fun } y \texttt{ => } D(s')) \texttt{ >>= fun } x \texttt{ => } D(s'') \\
\overset{\text{(5.2)}}{=} & D(s) \texttt{ >>= (fun } y \texttt{ => } D(s') \texttt{ >>= fun } x \texttt{ => } D(s'')) \\
\overset{\text{(D2)}}{=} & D(\texttt{let } y \leftarrow s\texttt{; } (D(s') \texttt{ >>= fun } x \texttt{ => } D(s''))) \\
\overset{\text{(D2)}}{=} & D(\texttt{let } y \leftarrow s\texttt{; let } x \leftarrow s'\texttt{; } s'')
\end{aligned}
$$

Nested bindings become a more interesting option when we add control flow statements, which we do together with adding syntax for local mutation in Figure 5.3: $\texttt{let mut } x \texttt{ := } e\texttt{; } s$ introduces a mutable variable x that can later (inside s) be *reassigned* using $x \texttt{ := } e$. We introduce monadic equivalents $\texttt{let mut } x \leftarrow s\texttt{; } s'$ and $x \leftarrow s$ by desugaring them to the sequence of a temporary, non-mutable monadic binding and the respective pure statement (A3, A4), thus simplifying our translation by keeping $\texttt{let } x \leftarrow s\texttt{; } s'$ as the only primitive monadic binding.

In straight-line code, local mutation is not very interesting: in $\texttt{let mut } x \leftarrow s\texttt{; ...; } x \leftarrow s'\texttt{; ...}$, we could achieve the expected semantics by

replacing both bindings with `let` x ← ... and relying on shadowing for "mutation".

Thus we also introduce the one-sided conditional statement `if` e `then` s so local mutation becomes meaningful:

```
do let mut a ← f
   if b a then
     a ← g a
   pure a
```

We could now introduce an abbreviation for two-sided `if` e `then` s_1 `else` s_2 statements as is done in the reference and full implementation, but I will refrain from doing so here to avoid possibly confusing overlap between our source and target language.

By the informal semantics discussed above, we expect the translation of the above code block to be equivalent to the term

```
f >>= fun a => if b a then g a else pure a
```

Let us start by unfolding all abbreviations we have introduced.

```
do let y ← f
   let mut a := y
   let x ←
     if b a then
       let y' ← g a
       a := y'
   pure a
```

We can easily eliminate the mutable variable by passing the updated state outwards, much like we did manually in our very first example in this section:

```
do let y ← f
   let a := y
   let (x, a) ←
     if b a then
       let y' ← g a
       let a := y'
       pure ((), a)
     else
       pure ((), a)
   pure a
```

By the monad laws, this is in fact equivalent to the expected term. The

Lean compiler as well manages to simplify this code in practice using well-known functional optimizations such as *case of case* and *case of constructor* [Peyton Jones, 1996] for specific instances of `pure` and >>= that can be inlined. For example, for the `Reader` monad, the generated code for the example is the IR equivalent of

```
fun r =>
  let a := f r
  match b a with
  | false => a
  | true  => g a r
```

Experienced purely functional programmers might notice a familiar pattern in the state-returning code: putting the "return value" and state into a pair and then extracting them at >>= mirrors the implementation of the `StateT` monad transformer (Figure 5.4). Indeed, we can rewrite the code using it:

```
do let y ← f
   let a := y
   StateT.run' (do
     let x ←
       if b a then
         let y' ← StateT.lift (g a)
         set y'
       else
         StateT.lift (pure ())
     let a ← get
     pure a) a
```

Here we introduce a *state effect* for the mutable variable `a` using `StateT.run'`, then `get` and `set` the current value inside. All existing monadic actions are lifted to the base monad using `StateT.lift` (which is a no-op for the specific case of `pure`).

For this simple example, the `StateT` code is not exactly simpler than the previous version. However, the main motivation for abstracting translation of mutable variables into a separate effect is that it does not only simplify the presentation of that translation, but it will also ensure *modularity* of our extension with others defined similarly: by separating every extension into its own effect, we can layer them naturally without having to manually reconcile their interaction. In other words, while we might need to add

Operations

$$
\begin{array}{ll}
\texttt{StateT.run'} & : \texttt{StateT}\ \sigma\ m\ \alpha \to \sigma \to m\ \alpha \\
\texttt{StateT.lift} & : m\ \alpha \to \texttt{StateT}\ \sigma\ m\ \alpha \\
\texttt{get} & : \texttt{StateT}\ \sigma\ m\ \sigma \\
\texttt{set} & : \sigma \to \texttt{StateT}\ \sigma\ m\ \texttt{Unit}
\end{array}
$$

Properties

$$
\begin{array}{lcl}
\texttt{StateT.run' (pure}\ a\texttt{)}\ s & = & \texttt{pure}\ a \\
\texttt{StateT.run' (StateT.lift}\ x\ \texttt{>>=}\ f\texttt{)}\ s & = & x\ \texttt{>>= fun a =>} \\
 & & \quad \texttt{StateT.run' (}f\ \texttt{a)}\ s \\
\texttt{StateT.run' (get >>=}\ f\texttt{)}\ s & = & \texttt{StateT.run' (}f\ s\texttt{)}\ s \\
\texttt{StateT.run' (set}\ s\texttt{' >>=}\ f\texttt{)}\ s & = & \texttt{StateT.run' (}f\ \texttt{())}\ s\texttt{'}
\end{array}
$$

Figure 5.4: `StateT` monad transformer operations and relevant properties for reasoning about them. Note that Lean's `set` method is called `put` in Haskell.

new translation rules for local mutation when introducing new `do` syntax in later sections, we will not have to modify existing rules for them. We will not have to worry about how to preserve the state on `break`, nor about how `return` inside loops has to be handled.

Figure 5.5 gives a formal translation of mutable variables to state effects: when encountering the definition of a mutable variable y, we use the helper function S_y to lift the following statements, i.e. y's scope, into the state transformer and rewrite them appropriately (D3). For monadic actions e, we do this with `StateT.lift` (S1). We use the shadowing approach for binding the current value of y, starting with its initial value in (D3), so we need to rebind it after reassignments as well as at possible control flow join points. In our reduced grammar, this can only be the case between the two statements in `let` $x \leftarrow s$`;` s', so we add a binding `let` $y \leftarrow$ `get` in between them (S2). This step could of course be elided if there are no reassignments of y in s. Furthermore, it only makes sense if x and y are distinct variables, which we enforce. We also do so for `let mut` x, meaning shadowing of mutable variables is disallowed in general. Apart from

Translation

$$D(\texttt{let mut } x \texttt{ := } e\texttt{; } s) = \texttt{let } x \texttt{ := } e\texttt{; StateT.run' } D(S_x(s))\ x \quad \text{(D3)}$$

$$D(\texttt{if } e \texttt{ then } s) = \texttt{if } e \texttt{ then } D(s) \texttt{ else pure ()} \quad \text{(D4)}$$

$$S : \mathit{Var} \to \mathit{Stmt} \to \mathit{Stmt}$$

$$S_y(e) = \texttt{StateT.lift } e \quad \text{(S1)}$$

$$S_y(\texttt{let } x \leftarrow s\texttt{; } s') = \texttt{let } x \leftarrow S_y(s)\texttt{; let } y \leftarrow \texttt{get; } S_y(s') \quad \text{if } x \neq y \quad \text{(S2)}$$

$$S_y(\texttt{let mut } x \texttt{ := } e\texttt{; } s) = \texttt{let mut } x \texttt{ := } e\texttt{; } S_y(s) \quad \text{if } x \neq y \quad \text{(S3)}$$

$$S_y(x \texttt{ := } e) = x \texttt{ := } e \quad \text{if } x \neq y \quad \text{(S4)}$$

$$S_y(y \texttt{ := } e) = \texttt{set } e \quad \text{(S5)}$$

$$S_y(\texttt{if } e \texttt{ then } s) = \texttt{if } e \texttt{ then } S_y(s) \quad \text{(S6)}$$

Figure 5.5: Translation of a `do` with local mutation

simplifying the translation rules, another reason for this restriction is to avoid any confusion on the user's side between mutable and immutable bindings. Finally, on a reassignment y `:=` e of the variable in question, we `set` the new value (S5).

$S_y(s)$ will thus eliminate any reassignments of y in s. If any remaining reassignments are encountered in the main translation function D, that must mean that no such mutable variable is in scope, and an appropriate error should be generated.

5.2 Early Return

Now that we have support for basic imperative control flow in `do`, it makes sense to talk about supporting `return` as well. While not without its own controversies, the programming pattern of *early return* seems to generally be well regarded in imperative programming for quickly discharging

Operations

$$
\begin{aligned}
\texttt{runCatch} &: \texttt{ExcepT}\ \alpha\ m\ \alpha \rightarrow\ m\ \alpha \\
\texttt{ExcepT.lift} &: m\ \alpha\ \rightarrow \texttt{ExcepT}\ \varepsilon\ m\ \alpha \\
\texttt{throw} &: \varepsilon\ \rightarrow \texttt{ExcepT}\ \varepsilon\ m\ \alpha
\end{aligned}
$$

Properties

$$
\begin{aligned}
\texttt{runCatch (pure}\ a\texttt{)} &= \texttt{pure}\ a \\
\texttt{runCatch (ExceptT.lift}\ x\ \texttt{>>=}\ f\texttt{)}\ s &= x\ \texttt{>>= fun}\ a\ \texttt{=> runCatch (}f\ a\texttt{)} \\
\texttt{runCatch (throw}\ a\ \texttt{>>=}\ f\texttt{)} &= \texttt{pure}\ a
\end{aligned}
$$

Figure 5.6: `ExceptT` monad transformer operations and relevant properties for reasoning about them

trivial/pathological cases in the beginning of a function without introducing indentation creep from nested conditionals. This matches our experience with it in Lean such as in the following example.

```
def isDefEqSingleton (structName : Name) (s : Expr) (v : Expr) :
    MetaM Bool := do
  let ctorVal ← getStructureCtor structName
  if ctorVal.numFields != 1 then
    return false  -- not a structure with a single field
  let s ← whnf s
  if !s.isMVar then
    return false  -- not an unsolved metavariable
  ...
```

This code taken from the Lean unifier and slightly simplified for presentation tries to reduce a unification problem `p s` $\stackrel{?}{=}$ `v` where `p` is the projection of a single-field structure type to `s` $\stackrel{?}{=}$ `c v` where `c` is the constructor of that type, using early return to quickly abort when the problem is not of the expected shape. As we shall see in the next section, `return` becomes even more useful when combined with iteration.

As with mutation, the reasonable semantics we can implement without changing code outside the `do` block is local: we will implement `return` *e* to abort execution *of the current* `do` *block* and have it return the value of *e*. I

Syntax

$$s \in \mathit{Stmt} ::= \dots$$
$$\quad | \ \texttt{return}\ e$$

Translation

$$\texttt{do}\ s \ \rightsquigarrow\ \texttt{runCatch}\ D(R(s)) \tag{5.1$'$}$$

$$R : \mathit{Stmt} \rightarrow \mathit{Stmt}$$

$$R(\texttt{return}\ e) = \texttt{throw}\ e \tag{R1}$$
$$R(e) = \texttt{ExceptT.lift}\ e \tag{R2}$$
$$R(\texttt{let}\ x \leftarrow s\texttt{;}\ s') = \texttt{let}\ x \leftarrow R(s)\texttt{;}\ R(s') \tag{R3}$$
$$R(\texttt{let mut}\ x\ \texttt{:=}\ e\texttt{;}\ s) = \texttt{let mut}\ x\ \texttt{:=}\ e\texttt{;}\ R(s) \tag{R4}$$
$$R(x\ \texttt{:=}\ e) = x\ \texttt{:=}\ e \tag{R5}$$
$$R(\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2) = \texttt{if}\ e\ \texttt{then}\ R(s_1)\ \texttt{else}\ R(s_2) \tag{R6}$$

Figure 5.7: A `do` with early return via exceptions

will discuss this decision more in Section 5.7.

Before we get to the implementation, a quick syntax consideration: in Lean 4, even before introducing the extended `do` block, we had already removed our analog of Haskell's `return` function in favor of the more general `pure`. Thus the decision to requisition the word as a keyword known from many imperative languages instead was a relatively easy one.

Programmatically, we could implement support for `return` by restructuring the entire `do` block into the noted nested conditionals. Even more so than in the previous section, however, it turns out that we can implement the desired semantics with relatively few additional rules by introducing a new effect (Figure 5.7), this time using the exception monad transformer `ExceptT` (Figure 5.6): we implement `return` e by raising e as an exception (R1), lifting all other monadic actions into the monad using `ExceptT.lift` (R2), and finally catching the exception, if any, and returning its captured value at the top level of the `do` block (5.1′) using `runCatch` (Figure 5.6). This way, we get the short-circuiting semantics for free from

`ExceptT`'s implementation of `>>=` introduced by our unchanged translation of `let` x `←` s`;` s`'`. The only existing rule we had to change was not that of an extension but the basic top-level translation rule (5.1). If a `do` block does not contain any `return` statements, we can of course fall back to the original rule instead. Note also that we did not have to extend S_y at all in this case because the only new syntax, `return` e, is eliminated before D, and therefore S_y, is ever run.

5.3 Iteration

One of the first things a functional programmer usually learns is that loops from imperative languages can be replaced by recursion. However, the mere fact of equivalence does not imply that the translation is always as readable or maintainable as the original. One issue with recursive helper definitions is that of textual locality: we have to define the recursion either before or after its (usually singular) use site even if it by itself is not a self-contained abstraction, moving it out of its surrounding context and hurting code comprehension compared to in-place usage of loops.

Focusing first on iteration over the elements of a collection, perhaps the most common kind of loop, I note that the locality issue can be avoided by use of folds, which allow in-place iteration using anonymous functions. The monadic fold function for lists, for example, is defined in Lean as

```
List.foldlM [Monad m] (f : δ → α → m δ) (init : δ) (xs : List α)
    : m δ
```

where `List α` is a list of elements of type α and δ is the accumulator type. In our experience from working on the Lean codebase, where fold-like traversal is pervasive, folds work relatively well for cases of simple control flow and one or two datums kept in the accumulator.

```
do ...
   let (x, y) ← zs.foldlM (init := (x, y)) fun (x, y) z => do
     let x' ← f x z
     if p x' then
       pure (x', g y z)
     else
       pure (x', y)
   ...
```

Lean's support for named parameters, the low precedence of `fun`, and extended dot notation avoids some confusion about parameter order as well as having to parenthesize the "loop body". Specifically, the term `zs.foldlM (init := a) f` is notation for `List.foldlM f a zs`. However, as soon as the number of "mutables" increases and/or the control flow inside the loop body gets more complex, handling and updating of the state tuple can quickly get onerous. In a few cases, we even resorted to manually introducing a temporary `StateT` layer exclusively for a single loop, in which case we can use the simpler combinator

```
List.forM [Monad m] : List α → (α → m Unit) → m Unit
```

instead:

```
do ...
   let (x, y) ← StateT.run' (x, y) do
     zs.forM fun z => do
       let (x, y) ← get
       let x' ← StateT.lift (f x z)
       if p x' then
         set (x', g y z)
       else
         set (x', y)
     get
   ...
```

With local mutation in hand, we would like to remove the need for such boilerplate code by adding a primitive syntax for iteration to `do` blocks with full support for mutable variables.

```
do let mut x := ...
   let mut y := ...
   ...
   for z in zs do
     x ← f x z
     if p x then
       y := g y z
   ...
```

Note that the second `do` keyword in the example above should not be regarded as a separate `do` block for purposes of `mut` and `return` scoping, but as part of the `for` statement syntax. Naturally, we would like to extend

support for `return` to `for` as well, providing us with a succinct way to reimplement some well-known combinators.

```
def List.findM? (p : α → m Bool) (xs : List α) : m (Option α) := do
  for x in xs do
    let b ← p x
    if b then
      return some x
  return none
```

Depending on the context and whether partial application can be used, it might even be more natural to inline these small definitions (adjusting the use of `return` if necessary) instead of remembering the combinator's name and calling it. As we will see later, we can prove by induction and simplification that these implementations are equivalent to the recursive definitions. Finally, in line with `return` we would also like to support `break` and `continue` with the expected semantics.

The formal syntax and translation supporting all these features is given in Figure 5.8. The main translation rule (D5) introduces two exception effects using `runCatch`: one for `break` outside, and one for `continue` inside a call to a collection-polymorphic `forM`, meaning both the loop body and the overall loop term return `Unit` in their respective monad. The loop body s is then rewritten appropriately, starting with the outer effect (so that its actions will be lifted by the subsequent rewrite for the inner effect) using the helper function B. B rewrites any `break` statement to `throw ()` (B1) and lifts all other monadic actions (B3), much like R. However, it should not rewrite `break` in nested loops (B8), so at that point we switch to another helper function L that merely lifts the loop into the correct monad. The helper function C for the inner `continue` effect is defined analogously to B.

Finally, we adapt the existing extensions to the newly introduced syntax: for R, this is merely recursive traversal (R7-R9), but for S_y, we need to introduce a `get` call at the loop entrance since it is a control flow join point (S9).

Before continuing, let us ensure that the order of effects indeed make sense: in the most general case, a `for` loop can contain all of `return`, reassignments to outer mutable variables, `break`, `continue`, and inner mutable variables, which correspond to effects introduced on top of the base monad in this order. While state effects and exception effects commute with instances of the same kind, this is not true when combining

Syntax

$$
\begin{aligned}
s \in \mathit{Stmt} ::= &\ \ldots \\
&|\ \texttt{break} \\
&|\ \texttt{continue} \\
&|\ \texttt{for}\ x\ \texttt{in}\ e\ \texttt{do}\ s
\end{aligned}
$$

Translation

$$
\begin{aligned}
&D(\texttt{for}\ x\ \texttt{in}\ e\ \texttt{do}\ s) = \\
&\qquad \texttt{runCatch (forM}\ e\ \texttt{(fun}\ x\ \texttt{=> runCatch}\ D(C(B(s))))) && \text{(D5)}
\end{aligned}
$$

$$
\begin{aligned}
&B : \mathit{Stmt} \to \mathit{Stmt} \\
&B(\texttt{break}) &&= \texttt{throw ()} && \text{(B1)} \\
&B(\texttt{continue}) &&= \texttt{continue} && \text{(B2)} \\
&B(e) &&= \texttt{ExceptT.lift}\ e && \text{(B3)} \\
&B(\texttt{let}\ x \leftarrow s;\ s') &&= \texttt{let}\ x \leftarrow B(s);\ B(s') && \text{(B4)} \\
&B(\texttt{let mut}\ x\ \texttt{:=}\ e;\ s) &&= \texttt{let mut}\ x\ \texttt{:=}\ e;\ B(s) && \text{(B5)} \\
&B(x\ \texttt{:=}\ e) &&= x\ \texttt{:=}\ e && \text{(B6)} \\
&B(\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2) &&= \texttt{if}\ e\ \texttt{then}\ B(s_1)\ \texttt{else}\ B(s_2) && \text{(B7)} \\
&B(\texttt{for}\ x\ \texttt{in}\ e\ \texttt{do}\ s) &&= \texttt{for}\ x\ \texttt{in}\ e\ \texttt{do}\ L(s) && \text{(B8)}
\end{aligned}
$$

$$
\begin{aligned}
&L : \mathit{Stmt} \to \mathit{Stmt} \\
&L(\texttt{break}) &&= \texttt{break} && \text{(L1)} \\
&L(\texttt{continue}) &&= \texttt{continue} && \text{(L2)} \\
&L(e) &&= \texttt{ExceptT.lift}\ e && \text{(L3)} \\
&L(\texttt{let}\ x \leftarrow s;\ s') &&= \texttt{let}\ x \leftarrow L(s);\ L(s') && \text{(L4)} \\
&L(\texttt{let mut}\ x\ \texttt{:=}\ e;\ s) &&= \texttt{let mut}\ x\ \texttt{:=}\ e;\ L(s) && \text{(L5)} \\
&L(x\ \texttt{:=}\ e) &&= x\ \texttt{:=}\ e && \text{(L6)} \\
&L(\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2) &&= \texttt{if}\ e\ \texttt{then}\ L(s_1)\ \texttt{else}\ L(s_2) && \text{(L7)} \\
&L(\texttt{for}\ x\ \texttt{in}\ e\ \texttt{do}\ s) &&= \texttt{for}\ x\ \texttt{in}\ e\ \texttt{do}\ L(s) && \text{(L8)}
\end{aligned}
$$

Figure 5.8: A do with support for iteration

$$S_y(\texttt{break}) = \texttt{break} \tag{S7}$$
$$S_y(\texttt{continue}) = \texttt{continue} \tag{S8}$$
$$S_y(\texttt{for}\ x\ \texttt{in}\ e\ \texttt{do}\ s) = \texttt{for}\ x\ \texttt{in}\ e\ \texttt{do}\ (\texttt{let}\ y \leftarrow \texttt{get};\ S_y(s)) \tag{S9}$$

$$R(\texttt{break}) = \texttt{break} \tag{R7}$$
$$R(\texttt{continue}) = \texttt{continue} \tag{R8}$$
$$R(\texttt{for}\ x\ \texttt{in}\ e\ \texttt{do}\ s) = \texttt{for}\ x\ \texttt{in}\ e\ \texttt{do}\ R(s) \tag{R9}$$

Figure 5.8: A `do` with support for iteration (cont.)

the two different kinds: an exception monad transformer on top of a state transformer will preserve the current state on an exception, whereas it will be lost when stacked in reverse. Thus the state of inner mutable variables, but not outer ones, will be lost on `continue` or `break`, while all state will be lost on `return`, which matches our intuitive understanding of these imperative concepts.

Figure 5.9 contains the translation of a small `do` block as an example. We can see that the top-most exception effect is used in place of `continue`, the one below for `break`, and that all other actions are lifted to the base monad below both of them.

I note that while the translation functions B and C could readily be fused into a single pass, separating them enables us to conditionally execute only one of the two passes (or neither of them) depending on the presence of the respective command in the loop. If a pass is not needed, the respective `runCatch` call should be removed as well. Finally, the `get` call can also be elided if there is no reassignment of the mutable variable in question inside the loop body.

In the encoding implemented in Lean 4, the collection-polymorphic `forM` is parameterized by the type class `ForM γ α`, where γ is some container type of elements of type α.

```
forM [Monad m] [ForM γ α] : γ → (α → m Unit) → m Unit
```

A function with a similar signature exists in the `Foldable` t typeclass of

```
do let mut s := 0
   for x in xs do
     if x % 2 = 0 then
       continue
     if x > 5 then
       break
     s := s + x
   IO.println s
```

```
runCatch
  (let s := 0 in
   StateT.run' (do
       runCatch (forM xs (fun x => runCatch (do
         let s ← ExceptT.lift (ExceptT.lift get)
         if x % 2 = 0 then throw ()
         else ExceptT.lift (ExceptT.lift (StateT.lift (ExceptT.lift
    (pure ())))))
         let s ← ExceptT.lift (ExceptT.lift get)
         if x > 5 then ExceptT.lift (throw ())
         else ExceptT.lift (ExceptT.lift (StateT.lift (ExceptT.lift
    (pure ())))))
         let s ← ExceptT.lift (ExceptT.lift get)
         ExceptT.lift (ExceptT.lift (set (s + x))))))
       let s ← get
       StateT.lift (ExceptT.lift (IO.println s)))
     s)
```

Figure 5.9: Translation example, using basic do notation instead of >>= for readability

the Haskell base library, where our γ corresponds to $t\ \alpha$. Alternatively, the `MonoFoldable` typeclass of the `mono-traversible` package[4] provides a function that is closer to the above signature and as such also allows for iterating over monomorphic collection types as well as ones with more than one type parameter, such as the key-value pairs of a map, without going through a temporary list of pairs.

Compared to other looping constructs, using fold-like traversal as a primitive is particularly interesting for total languages such as Lean because it delegates the issue of termination to the combinator. Indeed, if we opt into Lean's support for non-total functions, we can introduce the `repeat` and `while` statements as two syntax abbreviations

```
repeat s          ≡  for u in Loop.mk do s
while c do s      ≡  repeat ((if ¬c then break); s)
```

where `Loop` is an auxiliary type containing a single constructor `Loop.mk : Loop`, and its `ForM Loop Unit` instance is defined using the function

```
partial def loopForever [Monad m] (f : Unit → m Unit) :
  m Unit := f () >>= fun _ => loopForever f
```

The `partial` keyword, as mentioned in Chapter 2, ensures soundness by checking that the function type is inhabited (in this case by `fun f => pure ()`) and then turning the function into an opaque constant, making its body inaccessible to proofs.

5.4 Implementation

We have implemented two versions of the extended do notation described above: Section 5.4.1 describes a reference implementation, examples, and equivalence proofs. The reference implementation relies on Lean's hygienic macro system described in the previous chapter, and is a direct encoding of the presented translation rules. Languages with similarly expressive macro systems should allow for easy adaptation of the implementation,

[4] https://hackage.haskell.org/package/mono-traversable-1.0.15.1/docs/Data-MonoTraversable.html

but a built-in implementation manually ensuring hygiene is also conceivable. Section 5.4.2 describes the actual implementation in Lean 4 and the extensions it adds on top of the reference implementation.

5.4.1 Reference Implementation

The full reference implementation is given in Appendix A, of which I will give an overview in the following.

For defining the syntax of the extended do notation, we start by introducing a new syntactic category `stmt` of do statements, which is used in a new term notation `do'` *stmt*. We use the keyword `do'` instead of `do` to distinguish the reference implementation from Lean's full implementation.

```
declare_syntax_cat stmt

syntax "do'" stmt : term
```

Abbreviations such as A2 can now be implemented as simple macros.

```
macro:0 s₁:stmt ";" s₂:stmt : stmt => `(let x ← $s₁; $s₂)
```

Because the second `stmt` is not restricted to a precedence level, Lean will greedily associate the notation to the right as desired; note that the first occurrence of `stmt` is eliminated during elimination of left-recursion. By restricting nested statements in trailing positions to precedence levels greater than 0, we can force them not to contain a semicolon unless wrapped in curly braces, which use maximum precedence applied to syntax starting and ending with a token by default.

```
syntax "if" term "then" stmt:1 : stmt
macro "{" s:stmt "}" : stmt => `($s)
```

Thus `if b then c; d` unfolds to the same syntax tree as `{if b then c}; d`, not `if b then {c; d}`.

We represent the translation function D using the following auxiliary notation

```
syntax "d!" stmt : term  -- `d! s` corresponds to `D(s)`
```

and encode rule D2 as follows:

```
macro_rules
  | `(d! let $x ← $s; $s') => `((d! $s) >>= fun $x => (d! $s'))
```

Because macro_rules declarations may extend existing macros with new cases at any point, we can in fact introduce our language extensions modularly just like in the formal translation, with each section of Appendix A corresponding to one of these extensions.

Similarly to the above rule but using a generic expansion framework, we can encode function S_y using the auxiliary notations

```
declare_syntax_cat expander
syntax "expand!" expander "in" stmt : stmt
syntax "mut" ident : expander
-- `expand! mut y in s` corresponds to `S_y(s)`
```

and encode rule S1 as follows

```
macro_rules
  | `(stmt| expand! mut $y in $e:term) => `(stmt| StateT.lift $e)
```

where the `stmt|` prefix adjusts the syntactic category parsed by the quasiquotations. We can use the abstract `expander` syntax category above to factor out common traversing rules at functions S_y, R, B, and L. We will use the following syntax declarations for the latter three functions:

```
syntax "return" : expander
syntax "break"  : expander
syntax "lift"   : expander
```

Then, we can encode rules S6, R6, B7, and L7 using a single rule

```
macro_rules
  | `(stmt| expand! $exp in if $e then $s₁ else $s₂) =>
    `(stmt| if $e then expand! $exp in $s₁ else expand! $exp in $s₂)
```

As a simple optimization over the formal translation rules, the reference implementation avoids introducing monadic layers that are not actually used. If the number of nested occurrences of the keyword in question (return, break, or continue) has not changed after applying the respective helper function, it throws away the transformation result and does not emit the respective `runCatch` code. Lean's full implementation uses an abstract syntax tree of do commands that allows for more efficient implementations of such analyses.

As a further extension, the reference implementation produces meaningful error messages when side conditions in the formal rules are violated. For example, rule (S2) is implemented as follows

```
macro_rules
  | `(stmt| expand! mut $y in let $x ← $s; $s') =>
    if x == y then
      throw <| Macro.Exception.error x s!"cannot shadow 'mut'
     variable '{x.getId}'"
    else
      `(stmt| let $x ← expand! mut $y in $s;
              let $y ← get;
              expand! mut $y in $s')
```

5.4.2 Full Implementation

I have written the reference implementation described above with the primary goals of conciseness and simplicity in mind. Thus it intentionally does not implement some features of the full implementation built into Lean.

For one, the full implementation allows pattern bindings such as `do let (a, b) ← s; ...` that I kept out of the formal rules and the reference implementation as mentioned above. The full implementation also supports a `do` equivalent of Lean's `match` term where the right-hand side of each alternative is a sequence of statements.

```
do let mut n := 0
   for x in xs do
     match x with
     | none => n := n + 1
     | _    => pure ()
   IO.println n
```

The implementation of `match` statements is analogous to the one for `if` statements. Indeed, the full implementation also supports the Rust-inspired abbreviation `if let none := x then n := n + 1` of the above `match` block mixing the two styles.

For efficiency, the full implementation also implements variants such as `let mut x ← s; ...` instead of first expanding the code to `let y ← s; let mut x := y; ...`.

A fundamental restriction of a purely syntactic solution such as in the formal rules and reference implementation is that we cannot reject some undesirable inputs such as changing a variable's type when reassigning

it. Fortunately, we have seen in the previous chapter that our macro system allows for a seamless transition to type-aware *elaborators*, which the full implementation uses to generate an error message on e.g. the input `do let mut x := 0; x := true; ...`.

```
error: invalid reassignment, value has type
  Bool
but is expected to have type
  Nat
```

The full implementation also supports two useful features that are not directly inspired by imperative languages features, but nevertheless can help in avoiding boilerplate in monadic programs absent from equivalent imperative ones, so I will mention them here for the sake of completeness: nested actions and automatic monadic lifting. The first feature allows users to embed terms of the form `← a` in expressions, and is inspired to the `!a` notation available in Idris. For example,

```
do if (← get).x >= 0 then
     action
```

expands to

```
do let s ← get
   if s.x >= 0 then
     action
```

The main difference to Idris' design is that I aimed to make the scope of nested actions more predictable: instead of being lifted "as high as possible" [Brady, 2014], they are always lifted to the enclosing `do` block, i.e. to the same scope as the `return` statement. Using the notation outside of a `do` block is an error.

The second feature, automatic monadic lifting, is not directly tied to the `do` notation but part of Lean's general coercions system. The monad lifting typeclass

```
class MonadLift (m : Type → Type) (n : Type → Type) where
  monadLift : m α → n α
```

which is based on the typeclass of the same name available in the `Control.Monad.Layer` Haskell package, is automatically used by the coercions system to remedy type errors when using one monad inside another one. This feature is particularly useful in the Lean codebase because

we prefer using appropriate concrete monads over monad-polymorphic functions where possible in order to reduce code specialization overhead.

5.5 Reasoning

One of the stated advantages of purely functional programming is ease of reasoning. We can use Lean's theorem proving aspect to show that the output of our formal translation, while (sometimes unnecessarily) verbose, can still be analyzed using the same tools as corresponding functional code not using the extensions and indeed shown to be equivalent to it. Appendix A includes such equivalence proofs. The proofs are parameterized by an arbitrary monad `m` with an instance of the typeclass `LawfulMonad` that encodes the monadic laws of Figure 5.2. These equivalence proofs are straightforward using Lean tactics such as `cases`, `induction`, and `simp`. The simplifier in particular helps with applying the monadic, `StateT` (Figure 5.4), and `ExceptT` (Figure 5.6) laws automatically wherever applicable.

To take one simple example, let us show that a monadic program using the reference `do'` notation containing a conditional mutable update is equivalent to one using only basic syntax.

```
theorem simple [Monad m] [LawfulMonad m] (b : Bool) (ma ma' : m α) :
    (do' let mut x ← ma;
         if b then { x ← ma' };
         pure x)
    = (ma >>= fun x => if b then ma' else pure x) :=
  by cases b <;> simp
```

The tactic `cases b` splits the proof into two cases `(b = true)` and `(b = false)`, which is sufficient to solve the remainder by simplification. Appendix A also contains more complex equivalence proofs such as ones about programs using `for`, which for a specific iteratee type such as lists we can do by induction over the list.

5.6 Formalization

While the previous section demonstrated that specific examples of the extended `do` notation can be shown to correspond to their expected semantics,

that does not conclusively prove that our translation produces sensible — or even type-correct — output in all cases. Most of the translation rules are relatively straightforward, but there are subtle details around variable binding and shadowing as well as layering and lifting of monad transformers so that we should not a priori exclude the possibility of mistakes in them.

In order to increase trust in the translation's correctness, I have formulated an operational semantics of extended `do` notation that gives an alternative, dynamic and even simpler view of the expected behavior (Figure 5.10) as well as a corresponding type system that formalizes the expected static semantics of the notation (Figure 5.11) such that we expect the following correctness theorem to hold.

Theorem. *For any well-typed* `do` *block* $\Gamma \vdash$ `do` $s : \tau$*, there exists a unique (up to* α*-equivalence) translation* `do` $s \rightsquigarrow e$*, which is of the same type* ($\Gamma \vdash e : \tau$) *and equivalent under evaluation* ($\forall v.$ `do` $s \Rightarrow v \longleftrightarrow e \Rightarrow v$)*.*

Proof. See below for a strictly stronger statement for the case of Lean as the base language of terms, formalized and proved in Lean.

I necessarily restricted the given semantics to the special case of the identity monad and iteration over lists (arbitrarily presented as strict, like in the Lean language), as we would not be able to formulate operational semantics as such in presence of arbitrary monad and `ForM` instances. Since the translation functions are parametric over these instances, I believe this is only a minor limitation, and we will revisit the base monad decision below. The semantics and type system are otherwise generic over those of the base language, and thus a proof of the correctness theorem is also dependent on the base semantics, including an implementation of all monad transformer functions that fulfills the previously presented equations.

In order to focus on the translation and semantics of the `do` notation instead of these details of the base language, I decided to verify the correctness theorem in Lean using the Lean language itself and its existing implementation of monad transformers as the representation of terms (Appendix B), that is, a *shallow embedding* of Lean terms (abstracted over the contexts Γ and Δ) inside of a *deep embedding* of `do` statements as an inductive datatype. More specifically, statements are represented *intrinsically typed* by an inductive family `Stmt m ω Γ Δ b c α` such that the presented typing rules are fulfilled by definition (Figure 5.12). The parameters of the type

$$
\begin{array}{lll}
v \in \mathit{Val} & ::= \texttt{fun}\ x\ \texttt{=>}\ e \mid \texttt{()} \mid \texttt{true} \mid \texttt{false} \mid \texttt{nil} \mid \texttt{cons}\ v_1\ v_2 \mid \ldots & \subseteq \mathit{Expr} \\
n \in \mathit{Neut} & ::= v \mid \texttt{return}\ v \mid \texttt{break} \mid \texttt{continue} & \subseteq \mathit{Stmt} \\
\sigma \in \mathit{State} & \equiv \mathit{Var} \rightharpoonup \mathit{Val} &
\end{array}
$$

$\boxed{e \Rightarrow v}$

$$
\ldots \qquad \frac{\langle s, \emptyset \rangle \Rightarrow \langle n, \emptyset \rangle \qquad n \in \{v, \texttt{return}\ v\}}{\texttt{do}\ s \Rightarrow v}
$$

$\boxed{\langle s, \sigma \rangle \Rightarrow \langle n, \sigma' \rangle}$

$$
\frac{e[\sigma] \Rightarrow v}{\langle e, \sigma \rangle \Rightarrow \langle v, \sigma \rangle} \qquad \frac{\langle s, \sigma \rangle \Rightarrow \langle v, \sigma' \rangle \qquad \langle s'[v/x], \sigma' \rangle \Rightarrow \langle n, \sigma'' \rangle}{\langle \texttt{let}\ x\ \leftarrow\ s;\ s', \sigma \rangle \Rightarrow \langle n, \sigma'' \rangle}
$$

$$
\frac{\langle s, \sigma \rangle \Rightarrow \langle n, \sigma' \rangle \qquad n \notin \mathit{Val}}{\langle \texttt{let}\ x\ \leftarrow\ s;\ s', \sigma \rangle \Rightarrow \langle n, \sigma' \rangle} \qquad \frac{x \in \sigma \qquad e[\sigma] \Rightarrow v}{\langle x\ \texttt{:=}\ e;\ s, \sigma \rangle \Rightarrow \langle \texttt{()}, \sigma[x \mapsto v] \rangle}
$$

$$
\frac{x \notin \sigma \qquad e[\sigma] \Rightarrow v \qquad \langle s, \sigma[x \mapsto v] \rangle \Rightarrow \langle n, \sigma' \rangle}{\langle \texttt{let mut}\ x\ \texttt{:=}\ e;\ s, \sigma \rangle \Rightarrow \langle n, \sigma'[x \mapsto \bot] \rangle}
$$

Figure 5.10: Extending a natural semantics $e \Rightarrow v$ reducing expressions to values with a rule for `do` block evaluation for the special case of the identity monad. A helper relation $\langle s, \sigma \rangle \Rightarrow \langle n, \sigma' \rangle$ reduces statements to *neutral* statements under a mutable state context (a partial map from variables to values, updated via the notation $\cdot[\cdot \mapsto \cdot]$). Immutable bindings are evaluated by immediate substitution in the remainder statement, while the mutable context is substituted ($e[\cdot]$) just before evaluating any nested term e. The given value type and semantics are *strict* (see `cons`), but could easily be changed to be lazy.

$$\boxed{\langle s, \sigma \rangle \Rightarrow \langle n, \sigma' \rangle}$$

$$\frac{e[\sigma] \Rightarrow \texttt{true} \qquad \langle s, \sigma \rangle \Rightarrow \langle n, \sigma' \rangle}{\langle \texttt{if}\ e\ \texttt{then}\ s, \sigma \rangle \Rightarrow \langle n, \sigma' \rangle} \qquad \frac{e[\sigma] \Rightarrow \texttt{false}}{\langle \texttt{if}\ e\ \texttt{then}\ s, \sigma \rangle \Rightarrow \langle \texttt{()}, \sigma \rangle}$$

$$\frac{e[\sigma] \Rightarrow v}{\langle \texttt{return}\ e, \sigma \rangle \Rightarrow \langle \texttt{return}\ v, \sigma \rangle} \qquad \frac{e[\sigma] \Rightarrow \texttt{nil}}{\langle \texttt{for}\ x\ \texttt{in}\ e\ \texttt{do}\ s, \sigma \rangle \Rightarrow \langle \texttt{()}, \sigma \rangle}$$

$$\frac{\begin{array}{cc} e[\sigma] \Rightarrow \texttt{cons}\ v_1\ v_2 & \langle s[v_1/x], \sigma \rangle \Rightarrow \langle n, \sigma' \rangle \\ n \in \{\texttt{()}, \texttt{continue}\} & \langle \texttt{for}\ x\ \texttt{in}\ v_2\ \texttt{do}\ s, \sigma' \rangle \Rightarrow \langle n, \sigma'' \rangle \end{array}}{\langle \texttt{for}\ x\ \texttt{in}\ e\ \texttt{do}\ s, \sigma \rangle \Rightarrow \langle n, \sigma'' \rangle}$$

$$\frac{e[\sigma] \Rightarrow \texttt{cons}\ v_1\ v_2 \qquad \langle s[v_1/x], \sigma \rangle \Rightarrow \langle \texttt{break}, \sigma' \rangle}{\langle \texttt{for}\ x\ \texttt{in}\ e\ \texttt{do}\ s, \sigma \rangle \Rightarrow \langle \texttt{()}, \sigma' \rangle}$$

$$\frac{e[\sigma] \Rightarrow \texttt{cons}\ v_1\ v_2 \qquad \langle s[v_1/x], \sigma \rangle \Rightarrow \langle \texttt{return}\ v, \sigma' \rangle}{\langle \texttt{for}\ x\ \texttt{in}\ e\ \texttt{do}\ s, \sigma \rangle \Rightarrow \langle \texttt{return}\ v, \sigma' \rangle}$$

Figure 5.10: Extending a natural semantics with a rule for `do` block evaluation (cont.)

$$\boxed{\Gamma \vdash e : \tau}$$

$$\cdots \quad \frac{\Gamma \,|\, \cdot \vdash_{\texttt{nofor}} s : m\ \alpha \hookrightarrow \alpha}{\Gamma \vdash \texttt{do}\ \ s : m\ \alpha} \qquad \frac{}{\Gamma \vdash \texttt{()}\ : \texttt{Unit}}$$

$$\frac{}{\Gamma \vdash \texttt{nil} : \texttt{List}\ \alpha} \qquad \frac{\Gamma \vdash e : \alpha \qquad \Gamma \vdash e' : \texttt{List}\ \alpha}{\Gamma \vdash \texttt{cons}\ \ e\ \ e' : \texttt{List}\ \alpha}$$

$$\boxed{\Gamma \,|\, \Delta \vdash_f s : m\ \alpha \hookrightarrow \omega}$$

$$\frac{\Gamma, \Delta \vdash e : m\ \alpha}{\Gamma \,|\, \Delta \vdash_f e : m\ \alpha \hookrightarrow \omega}$$

$$\frac{x \notin \Delta \qquad \Gamma \,|\, \Delta \vdash_f s : m\ \alpha \hookrightarrow \omega \qquad \Gamma, x : \alpha \,|\, \Delta \vdash_f s' : m\ \beta \hookrightarrow \omega}{\Gamma \,|\, \Delta \vdash_f \texttt{let}\ \ x\ \leftarrow\ s;\ \ s' : m\ \beta \hookrightarrow \omega}$$

$$\frac{x \notin \Gamma, \Delta \qquad \Gamma, \Delta \vdash e : \alpha \qquad \Gamma \,|\, \Delta, x : \alpha \vdash_f s : m\ \beta \hookrightarrow \omega}{\Gamma \,|\, \Delta \vdash_f \texttt{let mut}\ \ x\ \ \texttt{:=}\ e;\ \ s : m\ \beta \hookrightarrow \omega}$$

$$\frac{\Gamma, \Delta, x : \alpha \vdash e : \alpha}{\Gamma \,|\, \Delta, x : \alpha \vdash_f x\ \ \texttt{:=}\ e : m\ \ \texttt{Unit} \hookrightarrow \omega}$$

$$\frac{\Gamma, \Delta \vdash e : \omega}{\Gamma \,|\, \Delta \vdash_f \texttt{return}\ \ e : m\ \alpha \hookrightarrow \omega}$$

Figure 5.11: Extending an expression typing relation $\Gamma \vdash e : \tau$ with a rule for typing `do` blocks via a statement typing relation $\Gamma \,|\, \Delta \vdash_f s : m\ \alpha \hookrightarrow \omega$ over some monad m. Δ is an additional context of mutable variables, initially empty. ω is the *return type* expected inside `return` statements, initially equal to α but may diverge from it in `let` bindings. . . .

$$\boxed{\Gamma \,|\, \Delta \vdash_f s : m\;\alpha \hookrightarrow \omega}$$

$$\frac{\Gamma, \Delta \vdash e : \texttt{Bool} \qquad \Gamma \,|\, \Delta \vdash_f s : m\;\text{Unit} \hookrightarrow \omega}{\Gamma \,|\, \Delta \vdash_f \texttt{if}\;\, e\;\, \texttt{then}\;\, s : m\;\alpha \hookrightarrow \omega}$$

$$\frac{x \notin \Delta \qquad \Gamma, \Delta \vdash e : \texttt{List}\;\alpha \qquad \Gamma, x : \alpha \,|\, \Delta \vdash_{\texttt{for}} s : m\;\texttt{Unit} \hookrightarrow \omega}{\Gamma \,|\, \Delta \vdash_f \texttt{for}\;\, x\;\, \texttt{in}\;\, e\;\, \texttt{do}\;\, s : m\;\texttt{Unit} \hookrightarrow \omega}$$

$$\frac{}{\Gamma \,|\, \Delta \vdash_{\texttt{for}} \texttt{break} : m\;\alpha \hookrightarrow \omega} \qquad \frac{}{\Gamma \,|\, \Delta \vdash_{\texttt{for}} \texttt{continue} : m\;\alpha \hookrightarrow \omega}$$

Figure 5.11: . . . $f \in \{\texttt{for}, \texttt{nofor}\}$ controls occurrences of `break` and `continue`.

constructor correspond to the variables of the same name in Figure 5.11, except that we split f into `b, c : Bool` that separately control `break` and `continue`, respectively, for reasons that will become clear in the next paragraph. As is common with formalizations, we additionally do not use named variables, but choose de Bruijn notation so that Γ and Δ are lists of types and variable references indices into those lists, trivializing α-equivalence. Bindings outside of the `do` block are not part of Γ but represented as regular Lean bindings so that Γ is initially empty. Thus the statement

```
let x ← ...; let mut y := ...; y := pure (x + y)
```

is represented by the Lean term

```
Stmt.bind ... (Stmt.letmut ... (Stmt.assg 0 (fun ([x]) ([y]) =>
  pure (x + y))))
```

where the context assignments are destructed into the individual variables by pattern matching on the heterogeneous lists. The full definitions and proofs in Appendix B are written in a literate style explaining further details and exported to LaTeX via the Alectryon document generator [Pit-Claudel, 2020] and its Lean 4 frontend LeanInk [Bülow, 2022] whose development I supervised.

This very precise representation of statements not only guarantees that the translation functions always produce unique, type-correct statements

```
notation:30 Γ " ⊢ " α => Assg Γ → α

inductive Stmt (m : Type → Type) (ω : Type) :
    (Γ Δ : List Type) → (b c : Bool) → (α : Type) → Type where
  | expr (e : Γ ⊢ Δ ⊢ m α) : Stmt m ω Γ Δ b c α
  | bind (s : Stmt m ω Γ Δ b c α) (s' : Stmt m ω (α :: Γ) Δ b c β) :
      Stmt m ω Γ Δ b c β
  | letmut (e : Γ ⊢ Δ ⊢ α) (s : Stmt m ω Γ (α :: Δ) b c β) :
      Stmt m ω Γ Δ b c β
  | assg (x : Fin Δ.length) (e : Γ ⊢ Δ ⊢ Δ.get x) :
      Stmt m ω Γ Δ b c Unit
  | if (e : Γ ⊢ Δ ⊢ Bool) (s : Stmt m ω Γ Δ b c Unit) :
      Stmt m ω Γ Δ b c Unit
  | ret (e : Γ ⊢ Δ ⊢ ω) : Stmt m ω Γ Δ b c α
  | for (e : Γ ⊢ Δ ⊢ List α)
        (s : Stmt m ω (α :: Γ) Δ true true Unit) :
      Stmt m ω Γ Δ b c Unit
  | break : Stmt m ω Γ Δ true c α
  | cont : Stmt m ω Γ Δ b true α
```

Figure 5.12: Inductive family representing intrinsically typed do statements with shallowly embedded terms. The notation $\Gamma \vdash \Delta \vdash \alpha$ stands for the type of functions mapping assignments (heterogeneous lists) of the contexts Γ and Δ to a value of α. Each constructor encodes, in order, a typing rule from Figure 5.11. Of particular note are constructors changing indices of the type family: `bind` and `for` extend the immutable context while `letmut` extends the mutable context, which is accessed by `assg` (assignment), representing the target variable as an index into the context, i.e. a de Bruijn index.

and terms as long as they are themselves type-correct Lean functions, but it also tells us much, and gives guarantees, about their working just by looking at their signatures:

```
S [Monad m] : Stmt m ω Γ (Δ ++ [α]) b c β →
              Stmt (StateT α m) ω (α :: Γ) Δ b c β
R [Monad m] : Stmt m ω Γ Δ b c α →
              Stmt (ExceptT ω m) Empty Γ Δ b c α
L [Monad m] : Stmt m ω Γ Δ b c α →
              Stmt (ExceptT Unit m) ω Γ Δ b c α
```

```
B [Monad m] : Stmt m ω Γ Δ b c α →
              Stmt (ExceptT Unit m) ω Γ Δ false c α
C [Monad m] : Stmt m ω Γ Δ false c α →
              Stmt (ExceptT Unit m) ω Γ Δ false false α
D [Monad m] : Stmt m Empty Γ ∅ false false α → (Γ ⊢ m α)
```

We can immediately see that `S` transforms a mutable variable (always the outermost one, so there is no need for the parameter y with de Bruijn notation) to an immutable variable, that `R` eliminates `return` statements (by setting their expected type to the uninhabited type `Empty`[5]), and B and C `break` and `continue` statements, respectively, as well what monadic layers they each introduce. Finally, `D` transforms a statement free of `return` and unbounded occurrences of mutable variables, `break`, and `continue` to a term of the expected type.

The choice of shallowly embedded terms obviates the dependency on a presentation of the natural semantics for the base language, and analogously we can express evaluation of statements directly as a *definitional interpreter* $[\![\cdot]\!]$ `: Do Id α → α` (where `Do m α` is an abbreviation of `Stmt m α ∅ ∅ false false α`) instead of an inductive predicate. Lean again guarantees that evaluation of any well-typed statement is unambiguous and terminating this way, which are desirable properties that we would expect to hold as long as the base language also fulfills them. Moreover, the implementation as a function makes it trivial to lift the restriction on the base monad and strengthen $[\![\cdot]\!]$ into the type `[Monad m] → Do m α → m α`. This generalization is in fact crucial for the Lean proof of the correctness theorem because it allows us to verify each translation function individually and modularly, finally composing the correctness proofs of `R` and `D` into the following succinct generalization of the correctness theorem over any monad obeying the laws from Figure 5.2.

```
def Do.trans [Monad m] (s : Do m α) : m α :=
  runCatch (D (R s) ∅)  -- (5.1′)
theorem Do.trans_eq_eval [Monad m] [LawfulMonad m] :
  ∀ s : Do m α, Do.trans s = ⟦s⟧ := ...  -- (B.5)
```

5 More explicit encodings such as ω `: Option Type` would be possible; it is an interesting property of the chosen shallow embedding of terms and types though that while `return` statements returning a value of `Empty` can certainly exist syntactically in an inconsistent context, the translation function may use this same inconsistent context to prove that the statement cannot exist and refuse to translate it.

Proving its correctness is not the only application of the formalization `Do.trans` of the translation rules, however. While the choice of mixed deep/shallow embedding by design blurs the distinction between the translated language and the implementation language, we can still see two separate *stages* in the signature of the main translation function, `D`: it takes a `Stmt`, i.e. compile-time information, and returns a function that then takes an assignment of the immutable context `Γ`, that is, run-time information. Indeed, it is possible to *partially* evaluate `D` and the other translation functions given a `Stmt` but not the context assignment (or values of any other variables free in expressions nested in `Stmt`) such that the `Stmt` is completely erased. While Lean does not natively support this kind of multi-stage programming, I demonstrate in Appendix B.6 how the built-in simplifier can be used as a partial evaluator for this purpose. Thus the formal translation could be used to replace the macro implementation while providing more static guarantees, though the latter is still more desirable in terms of efficiency and modularity.

5.7 Evaluation

I designed the extended `do` notation described above for use in Lean's own, monad-heavy implementation. It has been readily adopted by the Lean developers, with existing code being gradually refactored to make use of the new features as well. The feedback by users has also been very positive, and the extended `do` notation is used in many applications and packages developed by the Lean community: out of 92 GitHub repositories written in Lean with the topic "lean4"[6] in January 2023, 57 repositories by 38 different authors make use of at least `let mut` and `for`. Thus I feel safe to claim that the use of extended `do` notation in Lean 4 programming has by now become ubiquitous.

At the time of writing, the Lean 4 codebase itself contains 708 occurrences of `let mut` declarations, and 762 occurrences of `for`. The codebase also contains 3323 occurrences of `return`, though many of them are equivalent to `pure`, i.e. they do not actually short-circuit evaluation but merely avoid the need for parentheses around the return value.

[6] `https://github.com/topics/lean4?l=lean`

```
do let x := a
  f (fun y => do
    if p y then
      return x
```

Figure 5.13: Highlighting the `do` keyword belonging to a `return` statement under the cursor using the standard "document highlight" request of the Language Server Protocol, shown here using Visual Studio Code

So far we have not heard of specific anti-patterns or "foot-guns" users encountered with the notation. This is in contrast to our first implementation of the notation, which lacked the `mut` modifier. This resulted in non-obvious bugs in code containing nested `do` blocks such as

```
do let x := a
   ...
   f (fun y => do
      ...
      x := b
      ...)
   ...
```

The developer's intention in the above code was to reassign the variable `x` from the outer `do` block, but in reality the reassignment was scoped to the independent inner `do` block, resulting in a new kind of "scope confusion" between the language semantics and the user's intentions. The introduction of the `mut` modifier makes sure that this kind of code raises an appropriate compile-time error. It should also be remarked that the `mut` keyword dramatically simplifies the implementation by making scope checks a simple decision local to `do` notation as manifested in the helper function S_y.

It is conceivable that similar scope confusion could also occur when using `return` in nested blocks. In practice, this does not seem to be an issue since the expected type for the nested `do` block is often different from the outer one. Still, as a precaution, I have implemented highlighting of the corresponding `do` keyword when users place their cursor on a `return` statement in any editor supporting the Lean 4 language server (Figure 5.13).

We were pleasantly surprised to find that users are also using the

extended do notation in pure code via the identity monad. For example, the wrapAt? function in the Cli package[7] uses the following for statement.

```
Id.run do ...
   let mut line := line
   let mut result := #[]
   for i in [:resultLineCount] do
     result := result.push (line.take maxWidth)
     line := line.drop maxWidth
   return "\n".intercalate result.toList
```

The notation #[] denotes the empty array, and [:resultLineCount] is a range of natural numbers from 0 up to resultLineCount (exclusively).

Users also seem to prefer the for statement even when it corresponds to an existing combinator such as foldl. For example, the translation verifier reopt-vcg[8] contains the following code fragment.

```
do ...
   let mut stats : GoalStats := GoalStats.init
   for r in results do
     stats := stats.addResult r
   pure stats
```

As a final example, the function hasBadParamDep? from the Lean 4 code base demonstrates combining nested iteration, nested actions, and early return.

```
def hasBadParamDep? (ys : Array Expr) (indParams : Array Expr)
    : MetaM (Option (Expr × Expr)) := do
  for p in indParams do
    let pType ← inferType p
    for y in ys do
      if (← dependsOn pType y) then
        return some (p, y)
  return none
```

[7] https://github.com/mhuisi/lean4-cli

[8] https://github.com/GaloisInc/reopt-vcg

5.8 Related Work

I am not aware of similar formal work exploring imperative extensions to a purely functional language, but there are multiple existing libraries with some overlap. Perhaps the library closest to the presented work is the proof-of-concept Haskell package *ImperativeHaskell*[9], which via creative use of custom operators encodes mutable variables, `for` counting loops, and early return. The implementation is based on mutable references (`IORef`) and limited to `IO` as the base monad, which is a significant restriction in practice that in particular would severely complicate verification of programs using it. It is my belief that any implementation of these features generalized to arbitrary monads would require desugaring more expressive than custom operators, such as presented in this chapter. The *early* library[10] makes use of a GHC plugin to provide a syntax for early return when binding particular values, inspired by a similar syntax in Rust. The *control-monad-loop* library[11] provides a looping function with `continue` and `break` functionality encoded via a continuations-carrying monad, but none of the other effects. It also supports returning values from `break`s, which I have considered but discarded for the time being for lack of convincing use cases.

Apart from these imperative extensions, I am aware of three further extensions of Haskell's `do` notation: [Marlow et al., 2016] optimize its desugaring so that some blocks can be run using only `Applicative` instead of `Monad` operations, making `do` blocks both more general and potentially more efficient. [Erkök and Launchbury, 2002] add an `mdo` variant of the notation that changes the semantics of monadic bindings to allow recursion. Both extensions have since been implemented as language extensions of the GHC compiler, and should be compatible with Lean's imperative extensions. [Paterson, 2001] adapts the notation to *arrows*, a generalization of monads, which we have not explored in Lean so far.

Outside of Haskell, the Scala library *effectful*[12] translates `for` loops in a `do`-like macro to applications of `traverse`, but does not combine this with support for further control flow like `break`, `continue`, and `return`, or

[9] https://hackage.haskell.org/package/ImperativeHaskell

[10] https://github.com/inflex-io/early

[11] https://hackage.haskell.org/package/control-monad-loop-0.1

[12] https://github.com/pelotom/effectful

for local mutation. Idris features an extended do notation [Brady, 2014] that allows giving "alternative" patterns for a binding, which if matched determine the result of the whole block without executing the remaining statements:

```
do Just x_ok ← readNumber | Nothing => pure Nothing
   Just y_ok ← readNumber | Nothing => pure Nothing
   pure (Just (x_ok, y_ok))
```

This can be seen as an implementation of early return, though without nesting in further control flow statements such as `if` or `for`. A similar syntax was later added to Agda and is also available in Lean, though only with a single default pattern in the case of Lean. The Koka language [Leijen, 2014] is a function-oriented language with built-in effects and as such does not employ monads or do notation. However, I will note that it has support for both mutable variables and multi-shot effects such as nondeterminism, for the combination of which it has assigned the same "strand-local" semantics as I discussed in Section 5.1.

[Gibbons and dos Santos Oliveira, 2009] identify *mapping* and *accumulating* as the core aspects of imperative iteration, and show that the `traverse` operator can represent both of them simultaneously in functional code. For Lean, we focused on the more restrictive folds, which embody only the accumulating part, since they cover most use cases where local mutation or extended control flow can profitably be applied in our experience. However, it is possible to extend our approach to `traverse`, e.g. with a new `for mut x in xs do s` syntax that for a mutable variable `xs` allows `x` to be reassigned in the loop body and eventually reassigns the thus mapped collection to `xs`.

I have previously explored the idea of rewriting code using mutable variables into equivalent pure code to make it amenable to formal verification in my master's thesis [Ullrich, 2016] in the context of translating a subset of Rust to Lean, which served as direct inspiration for the presented work. As in the translation above, mutation in straight-line code is translated to shadowing in the thesis, though for conditional statements, the continuation is duplicated, which is less of an issue when compilation of the translated code is not a goal. Similarly, both terminating and non-terminating loops are supported via a fold-like monadic loop combinator, but the combinator is not computable (i.e. executable) because it employs classical logic to "decide" termination. Termination must instead be proved or disproved

after translation. The translation also handles some advanced cases like turning mutable references into lenses [Foster et al., 2007], which we have no plans of supporting in Lean's `do` notation. [Ho and Protzenko, 2022] introduced a similar but more general approach for verification of Rust code.

[Nipkow, 1998] utilizes a formalization approach very similar to the mixed deep/shallow embedding I used above, calling it "taking the semantic view". The lack of dependent types in Isabelle/HOL, however, would complicate representing a heterogeneous context like I did.

these techniques are cleverly dreadful, rather than dreadfully clever
– Conor McBride, *Epigram: Practical Programming with Dependent Types* [McBride, 2005]

6 An Efficient Reference Counting Scheme for Functional Programming

In the previous chapter, we have seen how a relatively simple extension of the Lean language allows for an imperative programming style where beneficial. The syntactic support for e.g. mutable variables however could be of limited use, and even actively misleading, if the *data structures* stored in these variables did not behave like users might expect from imperative languages. For example, it would be quite deceiving if the loop

```
let mut result := #[]
for i in [:resultLineCount] do
  result := result.push (line.take maxWidth)
  line := line.drop maxWidth
```

from Section 6.7 accumulating an array `result` did not run in linear time. And yet purely functional languages, in order to preserve referential transparency, usually have to resort to conservative copying of the array, choosing a different representation such as a linked list or tree, or modeling the in-place update as an explicit effect [Launchbury and Peyton Jones, 1995].

However, while discussing how to improve on the simplistic runtime system presented in Chapter 2, an additional alternative presented itself to Leo and me: if we stuck with reference counting for Lean 4 as well, which we were quite willing to do considering the usual complexity of a tracing garbage collector, we could make use of this reference counter for checking uniqueness of references *dynamically*, allowing for destructive updates of objects without violating referential transparency.

This chapter describes the optimized reference counting system, along with relevant compiler phases and runtime properties, implemented in Lean 4 that resulted from these considerations. Going far beyond our expectations, this system contributes to the fact that the run time of Lean programs can not only compete with but even outperform binaries implemented in established languages based on tracing garbage collection (Section 6.7). The fundamentals of the reference counting system have since been adopted [Reinking et al., 2021] by the Koka language, where the very fitting name of "Functional But In-Place" was introduced for pure algorithms that make use of destructive updates in this way, as well as by the Roc language [Feldman, 2021]. Combined with the imperative-inspired extensions from the previous chapter, we obtain a new paradigm for writing programs in Lean, which I have named *pure imperative programming*: using imperative syntax for writing pure functional programs that nevertheless can achieve the run time asymptotics usually expected from impure languages only.

Contributions. In this chapter, I present a reference counting system optimized for purely functional languages and used by Lean 4.

- I describe the core optimization of introducing sound destructive updates, and the related optimization of using borrowed references.
- I formally define a compiler that implements the introduction of reference counting instructions as well as these optimizations.
- I give a formal reference-counting semantics and prove the compiler correct relative to it (Appendix C).
- I compare the actual implementation of this compiler in Lean 4 with other compilers for functional languages and show its competitiveness.

Acknowledgements. The presented reference counting system is joint work with Leonardo de Moura [Ullrich and de Moura, 2019a]. Leo and I designed the intermediate representation and optimizations together during my internship at Microsoft Research. Leo then went on to implement them in Lean, while I authored the formal semantics and correctness proof (which was previously published as [Ullrich and de Moura, 2019b] as a

separate appendix). The formal descriptions of the compilation steps were developed by us in tandem.

6.1 IR Syntax

For the source language λ_{pure} of the compiler in this chapter, we will use a simple untyped functional intermediate representation (IR) in the style of A-normal form [Flanagan et al., 1993]. While the actual IR implemented in Lean 4 has subsequently acquired additional instructions and a restricted type system, λ_{pure} captures all features relevant for this chapter.

$$\begin{aligned}
w, x, y, z &\in \mathit{Var} \\
c &\in \mathit{Const} \\
e &\in \mathit{Expr} &&::= c\ \overline{y} \mid \mathbf{pap}\ c\ \overline{y} \mid x\ y \mid \mathbf{ctor}_i\ \overline{y} \mid \mathbf{proj}_i\ x \\
F &\in \mathit{FnBody} &&::= \mathbf{ret}\ x \mid \mathbf{let}\ x = e;\ F \mid \mathbf{case}\ x\ \mathbf{of}\ \overline{F} \\
f &\in \mathit{Fn} &&::= \lambda\ \overline{y}.\ F \\
\delta &\in \mathit{Program} &&= \mathit{Const} \rightharpoonup \mathit{Fn}
\end{aligned}$$

Expressions in this language are made up of various kinds of applications. The applied function is a constant c (with partial applications marked with the keyword **pap**), a variable x, the i-th constructor of an erased datatype, or the special function $\mathbf{proj}_i$, which returns the i-th argument of a constructor application. All arguments of function applications are variables. Function bodies always end with evaluating and **ret**urning a variable. They can be chained with (non-recursive) **let** statements and branch using **case** statements, which evaluate to their i-th arm given an application of $\mathbf{ctor}_i$. As further detailed in Section 6.4.3, we consider calls of the form $\mathbf{let}\ r = c\ \overline{x};\ \mathbf{ret}\ r$ to be tail calls. A program is a partial map from constant names to their implementations. The body of a constant's implementation may refer back to the constant, which we use to represent recursion, and analogously mutual recursion. In examples as seen in the previous section, we use $f\ \overline{x} = F$ as syntax sugar for $\delta(f) = \lambda\ \overline{x}.\ F$.

As an intermediate representation, we can and should impose restrictions on the structure of λ_{pure} to simplify working with it. We will assume that, as partially expressed in the IR syntax,

- all constructor applications are fully applied, by eta-expanding them.

- no constant applications are over-applied, that is, called with more arguments than their static arity suggests because they in turn return a function value. Instead, the additional arguments are applied to the returned function value as separate applications as below so as to homogenize the structure of constant applications.
- all variable applications take only one argument, again by splitting them into separate applications where necessary. While this simplification can introduce additional allocations of intermediary partial applications, it greatly simplifies the presentation of the operational semantics. All presented program transformations can be readily extended to a system with n-ary variable applications, which are handled analogously to n-ary constant applications.
- every function abstraction has been lambda-lifted to a top-level constant c.
- trivial bindings **let** $x = y$ have been eliminated through copy propagation.
- all dead **let** bindings have been removed.
- all parameter and **let** names of a function are mutually distinct. Thus we do not have to worry about shadowing.

One major extension of the IR as implemented in Lean 4 over this reference language is the addition of *join points* [Maurer et al., 2017] for representing common parts of code, much like basic blocks with multiple predecessors in imperative IRs. While the presence of join points does complicate the presented analyses, there is no fundamental change and an IR without join points is just as expressive, at the cost of redundant code.

Our target language λ_{RC} is an impure extension of λ_{pure} with operations specific to reference counting:

$$\begin{aligned} e \in \mathit{Expr} &::= \ldots \mid \mathbf{reset}\ x \mid \mathbf{reuse}\ x\ \mathbf{in}\ \mathbf{ctor}_i\ \overline{y} \\ F \in \mathit{FnBody} &::= \ldots \mid \mathbf{inc}\ x;\ F \mid \mathbf{dec}\ x;\ F \end{aligned}$$

I will use the subscripts $_{pure}$ or $_{RC}$ (e.g., Expr_{pure} or Expr_{RC}) to refer to the base or extended syntax, respectively, where otherwise ambiguous.

The new instructions **inc** and **dec** modify the reference counter of an object. **reset** and **reuse** work together to reuse memory used to store constructor values and simulate destructive updates in constructor values, as further detailed in the following sections.

6.2 IR by Example

Before formally introducing the semantics of the IR in the next section and the compiler transformations in the subsequent sections, let us start by looking at representative examples using these instructions and give an intuitive understanding of the reference counting model.

A helpful mental model for thinking about reference-counted objects in the following examples and further on is to view each such counter as a collection of virtual tokens (which do not exist as actual objects in the intermediate representation or at run time). Then an object should be held alive as long as someone (the token *owner*) holds at least one of its tokens. The **inc** instruction creates a new token while **dec** destroys one token. A freshly created object starts with one token passed to the creator. When its last token is destroyed, the allocation is freed after one token of each contained field is destroyed, which recursively can lead to further deallocations. By default, functions take arguments as owned references, i.e. they also get passed a corresponding token to ensure that the lifetime of the passed reference is not dependent on the caller. The function is responsible for eventually consuming the token, which it may do not only by using the **dec** instruction, but also by storing it in a newly allocated object, returning it, or passing it to another function that takes an owned reference.

As a first trivial example, the identity function *id* does not require any token manipulations when it takes and returns its argument as an owned reference. It is represented in the reference counting IR as

$id\ x\ =\ \mathbf{ret}\ x$

The function *mkPairOf* takes an x and returns the pair (x, x).

$mkPairOf\ x\ =\ \mathbf{inc}\ x\ ;\ \mathbf{let}\ p\ =\ \mathbf{ctor}_{\mathsf{Prod.mk}}\ x\ x\ ;\ \mathbf{ret}\ p$

Now an **inc** instruction is required because two tokens for x are consumed

by the `Prod.mk` constructor[1] (we will also say that "x is consumed" twice) in order to make sure that consumers of the created pair can use both components independently. The function *fst* on the other hand, which takes two arguments x and y and returns x, uses a **dec** instruction for consuming the unused y.

fst x y = **dec** y ; **ret** x

The examples above suggest that we do not need any RC operations when we take arguments as owned references and consume them exactly once, i.e. use them linearly. Contrast this with a function that only inspects its argument: *isNil* returns true if the given list is empty and false otherwise. If the parameter is taken as an owned reference, the compiler generates the following code

isNil *xs* = **case** *xs* **of**
  (*nil* → **dec** *xs* ; **ret** *true*)
  (*cons* → **dec** *xs* ; **ret** *false*)

dec is again used to consume all unused owned parameters; note that **case** itself does not consume a token because, in contrast to arbitrary function calls, we know that it will not hold on to the passed reference.

As an alternative to owned references, a function can accept a *borrowed* reference that does not come with a token, much like for the operand of **case**. The function can assume that at least one token for the referenced object must have existed at the time of the call, and that this token will persist until the call is finished, but no more. In particular, it may not return the reference (directly or wrapped in a new object) without creating a new token for it. We can generate the following compact code for *isNil* that fulfills these conditions if we take a borrowed reference.

isNil *xs* = **case** *xs* **of** (*nil* → **ret** *true*) (*cons* → **ret** *false*)

Note that *xs* being a borrowed reference is not directly reflected in the IR since borrowing does not exist as a concept in the formal semantics of the IR; instead it is merely a convention the compiler introduces, a part of the Lean calling convention, and of the formal type system in Appendix C.

[1] The actual IR as defined above uses indices for constructor references in **case** and **ctor** as well as field references in **proj**, though I will use suggestive names in most examples as here instead.

As a less trivial example, let us take a look at the function *hasNone* that checks whether a given list of optional values contains a *none* value, which in Lean we could define as

```
def hasNone : List (Option α) → Bool
  | []            => false
  | none   :: _   => true
  | some _ :: xs => hasNone xs
```

Similarly to *isNil*, *hasNone* only inspects its argument, though via recursion. Nevertheless, when considering the parameter as a borrowed reference, the compiler can produce the following code free of reference counting instructions for it

```
hasNone xs = case xs of
  (nil → ret false)
  (cons → let h = proj_head xs ; case h of
    (none → ret true)
    (some → let t = proj_tail xs ; let r = hasNone t ; ret r))
```

Note that the **case** instruction does not introduce binders. Instead, we use explicit projections $\mathbf{proj}_i$ for accessing the head and tail of the *cons* cell. The borrow inference heuristic discussed in Section 6.4 correctly infer *xs* as a borrowed parameter.

When using owned references only, we know at run time whether a value is uniquely referenced or not simply by checking its reference counter. We can leverage this information and minimize the number of allocations using the other set of extensions in λ_{RC}:

- **let** y = **reset** x effectively consumes a token like **dec** x and assigns a special dummy value to y to mark it as "unset" from this point on; however, if this was the last reference to x, the allocation is not actually freed but assigned to y instead of the dummy value. The reference counters of each field are still decremented and the referenced objects possibly freed, so y should be thought of pointing not to an actual Lean object but a "raw allocation" in this case.

- **let** z = (**reuse** y **in** $\mathbf{ctor}_i\ \overline{w}$) then creates the constructor value $\mathbf{ctor}_i\ \overline{w}$ either (when y is set) reusing the raw allocation y, or otherwise using a fresh allocation.

Let us explore the use of these instructions in the compiler output of the list function *map* as an example.

```
map f xs = case xs of
  (nil → ret xs)
  (cons →
   let x = proj_head xs ; inc x ;
   let s = proj_tail xs ; inc s ;
   let w = reset xs ;
   let y = f x ; let ys = map f s ;
   let r = (reuse w in ctor_cons y ys) ; ret r)
```

If the input list *xs* is *nil*, *map* just returns it. Otherwise, it extracts the head x and tail s from *xs* and then uses the **inc** instruction to create owned references to these two values. Then, **reset** *xs* consumes *xs*, possibly storing a raw allocation in y that is reused to create the new *cons* cell after mapping the head and, recursively, the tail of *xs*.

If no cell of the list referenced by *xs* is shared, the code above will not allocate any memory, entirely reusing the existing allocations. This would not be true if we did not have separate **reset** and **reuse** instructions: if we removed the **reset** instruction and directly used *xs* in an alternative version of **reuse**, it would be kept alive during the recursive call, preventing allocation reuses in the tail s as a second reference would still be stored in *xs*. Note that removing the **inc** s instruction instead would be incorrect when *xs* is a shared value. Although the **reset** and **reuse** instructions can in general be used for reusing memory between two otherwise unrelated values, in examples like *map* where the reused value has a close semantic connection to the reusing value, we will use common functional vocabulary and say that the list is being *destructively updated* (up to the first shared cell).

6.3 Semantics of the Reference-Counting IR

Before discussing the compiler transformations that can produce the presented output, we should make our reference counting model more precise. We define the semantics of λ_{RC} (Figures 6.1 and 6.2) using a big-step relation $\rho \vdash \langle F, \sigma \rangle \Downarrow \langle l, \sigma' \rangle$ that maps the body F and a mutable *heap* σ under a context ρ to a *location* and the resulting heap. The context ρ maps variables to locations. A heap σ is a mapping from locations to

pairs of values and reference counters. A value is a constructor value or a partially-applied constant. The reference counters of live values should always be positive; dead values are removed from the heap map.

$$\begin{aligned}
l &\in \mathit{Loc} \\
\rho &\in \mathit{Ctxt} &&= \mathit{Var} \rightharpoonup \mathit{Loc} \\
\sigma &\in \mathit{Heap} &&= \mathit{Loc} \setminus \{\blacksquare\} \rightharpoonup \mathit{Value} \times \mathbb{N}^+ \\
v &\in \mathit{Value} &&::= \mathtt{ctor}_i\ \bar{l} \mid \mathtt{pap}\ c\ \bar{l}
\end{aligned}$$

The first rule for saturated applications in Fig. 6.1 demonstrates the general use of all these parts: it looks up the called function's implementation in the program map and the arguments' locations in the context. Then it constructs a fresh context from the function's bound parameters and these locations (which we implicitly take as an assertion that these two lists have the same length), with which it finally executes the called function's body, resulting in a new heap and return value location. If, on the other hand, a function is partially applied, the function's name and the arguments' locations are stored in a `pap` cell in a fresh location.

`pap` cells are retrieved from the heap by the rules for application of variables. We have to be careful to increment the partial application arguments' reference counters when copying them out of the `pap` cell, and to decrement the cell afterwards, which we do with helper functions defined in Fig. 6.2.[2] We cannot do so via explicit reference counting instructions because the number of arguments in a `pap` cell is not known statically. Similarly, the `ctor` instruction allocates a new `ctor` cell, while `case` and `proj` inspect such a cell (without consuming it) and select the appropriate function continuation or constructor field, respectively.

Continuing with the reference counting instructions in Fig. 6.2, `inc` is a direct manipulation of the pointee's reference counter. `dec`rementing a unique reference additionally removes the value from the heap and recursively decrements its components. `reset`, when used on a unique reference, eagerly decrements the components of the referenced value, replaces them with the special location ■,[3] and returns the location of the

[2] If the `pap` reference is unique, the two steps could be coalesced so that the arguments do not have to be touched.

[3] which can be represented by any unused pointer value such as the null pointer in a real

Const-App-Full

$$\frac{\delta(c) = \lambda\,\overline{y_c}.\,F \qquad \bar{l} = \overline{\rho(y)} \qquad [\overline{y_c} \mapsto \bar{l}] \vdash \langle F, \sigma\rangle \Downarrow \langle l', \sigma'\rangle}{\rho \vdash \langle c\,\overline{y}, \sigma\rangle \Downarrow \langle l', \sigma'\rangle}$$

Const-App-Part

$$\frac{\delta(c) = \lambda\,\overline{y_c}.\,F \qquad \bar{l} = \overline{\rho(y)} \qquad |\,\bar{l}\,| < |\,\overline{y_c}\,| \qquad l' \notin \mathrm{dom}(\sigma)}{\rho \vdash \langle \mathbf{pap}\; c\,\overline{y}, \sigma\rangle \Downarrow \langle l', \sigma[l' \mapsto (\mathbf{pap}\; c\,\bar{l}, 1)]\rangle}$$

Var-App-Full

$$\frac{\begin{array}{c}\sigma(\rho(x)) = (\mathbf{pap}\; c\,\bar{l}, _) \qquad \delta(c) = \lambda\,\overline{y_c}.\,F \\ l_y = \rho(y) \qquad [\overline{y_c} \mapsto \bar{l}\,l_y] \vdash \langle F, \mathrm{dec}(\rho(x), \mathrm{inc}(\bar{l}, \sigma))\rangle \Downarrow \langle l', \sigma'\rangle\end{array}}{\rho \vdash \langle x\,y, \sigma\rangle \Downarrow \langle l', \sigma'\rangle}$$

Var-App-Part

$$\frac{\begin{array}{c}\sigma(\rho(x)) = (\mathbf{pap}\; c\,\bar{l}, _) \\ \delta(c) = \lambda\,\overline{y_c}.\,F \qquad l_y = \rho(y) \qquad |\,\bar{l}\,l_y\,| < |\,\overline{y_c}\,| \qquad l' \notin \mathrm{dom}(\sigma)\end{array}}{\rho \vdash \langle x\,y, \sigma\rangle \Downarrow \langle l', \mathrm{dec}(\rho(x), \mathrm{inc}(\bar{l}, \sigma))[l' \mapsto (\mathbf{pap}\; c\,\bar{l}\,l_y, 1)]\rangle}$$

Ctor-App

$$\frac{\bar{l} = \overline{\rho(y)} \qquad l' \notin \mathrm{dom}(\sigma)}{\rho \vdash \langle \mathbf{ctor}_i\,\overline{y}, \sigma\rangle \Downarrow \langle l', \sigma[l' \mapsto (\mathbf{ctor}_i\,\bar{l}, 1)]\rangle}$$

Case

$$\frac{\sigma(\rho(x)) = (\mathbf{ctor}_i\,\bar{l}, _) \qquad \rho \vdash \langle F_i, \sigma\rangle \Downarrow \langle l', \sigma'\rangle}{\rho \vdash \langle \mathbf{case}\; x\; \mathbf{of}\; \overline{F}, \sigma\rangle \Downarrow \langle l', \sigma'\rangle}$$

Proj

$$\frac{\sigma(\rho(x)) = (\mathbf{ctor}_j\,\bar{l}, _) \qquad l' = \bar{l}_i}{\rho \vdash \langle \mathbf{proj}_i\; x, \sigma\rangle \Downarrow \langle l', \sigma\rangle}$$

Return

$$\frac{\rho(x) = l}{\rho \vdash \langle \mathbf{ret}\; x, \sigma\rangle \Downarrow \langle l, \sigma\rangle}$$

Let

$$\frac{\rho \vdash \langle e, \sigma\rangle \Downarrow \langle l, \sigma'\rangle \qquad \rho[x \mapsto l] \vdash \langle F, \sigma'\rangle \Downarrow \langle l', \sigma''\rangle}{\rho \vdash \langle \mathbf{let}\; x = e;\; F, \sigma\rangle \Downarrow \langle l', \sigma''\rangle}$$

Figure 6.1: λ_{RC} semantics: the λ_{pure} fragment

$$\textsc{Inc}\quad \frac{\rho \vdash \langle F, \mathrm{inc}(\rho(x), \sigma)\rangle \Downarrow \langle l', \sigma' \rangle}{\rho \vdash \langle \mathbf{inc}\ x;\ F, \sigma \rangle \Downarrow \langle l', \sigma' \rangle} \qquad \textsc{Dec}\quad \frac{\rho \vdash \langle F, \mathrm{dec}(\rho(x), \sigma)\rangle \Downarrow \langle l', \sigma' \rangle}{\rho \vdash \langle \mathbf{dec}\ x;\ F, \sigma \rangle \Downarrow \langle l', \sigma' \rangle}$$

$$\mathrm{inc}(l, \sigma) = \sigma[l \mapsto (v, i+1)] \quad \text{if } \sigma(l) = (v, i)$$

$$\mathrm{inc}(l\ \overline{l'}, \sigma) = \mathrm{inc}(\overline{l'}, \mathrm{inc}(l, \sigma))$$

$$\mathrm{dec}(l, \sigma) = \begin{cases} \sigma & \text{if } l = \blacksquare \\ \sigma[l \mapsto (v, i-1)] & \text{if } \sigma(l) = (v, i), i > 1 \\ \mathrm{dec}(\overline{l'}, \sigma[l \mapsto \bot]) & \text{if } \sigma(l) = (\mathbf{pap}\ c\ \overline{l'}, 1) \\ \mathrm{dec}(\overline{l'}, \sigma[l \mapsto \bot]) & \text{if } \sigma(l) = (\mathbf{ctor}_i\ \overline{l'}, 1) \end{cases}$$

$$\mathrm{dec}(l\ \overline{l'}, \sigma) = \mathrm{dec}(\overline{l'}, \mathrm{dec}(l, \sigma))$$

$$\textsc{Reset-Uniq}\quad \frac{\rho(x) = l \qquad \sigma(l) = (\mathbf{ctor}_i\ \overline{l'}, 1)}{\rho \vdash \langle \mathbf{reset}\ x, \sigma \rangle \Downarrow \langle l, \mathrm{dec}(\overline{l'}, \sigma[l \mapsto (\mathbf{ctor}_i\ \blacksquare^{|\overline{l'}|}, 1)]) \rangle}$$

$$\textsc{Reset-Shared}\quad \frac{\rho(x) = l \qquad \sigma(l) = (_, i) \qquad i \neq 1}{\rho \vdash \langle \mathbf{reset}\ x, \sigma \rangle \Downarrow \langle \blacksquare, \mathrm{dec}(l, \sigma) \rangle}$$

$$\textsc{Reuse-Uniq}\quad \frac{\rho(x) = l \qquad \sigma(l) = (\mathbf{ctor}_j\ \blacksquare^{|\overline{y}|}, 1) \qquad \overline{\rho(y)} = \overline{l''}}{\rho \vdash \langle \mathbf{reuse}\ x\ \mathbf{in}\ \mathbf{ctor}_i\ \overline{y}, \sigma \rangle \Downarrow \langle l, \sigma[l \mapsto (\mathbf{ctor}_i\ \overline{l''}, 1)] \rangle}$$

$$\textsc{Reuse-Shared}\quad \frac{\rho(x) = \blacksquare \qquad \rho \vdash \langle \mathbf{ctor}_i\ \overline{y}, \sigma \rangle \Downarrow \langle l', \sigma' \rangle}{\rho \vdash \langle \mathbf{reuse}\ x\ \mathbf{in}\ \mathbf{ctor}_i\ \overline{y}, \sigma \rangle \Downarrow \langle l', \sigma' \rangle}$$

Figure 6.2: λ_{RC} semantics: the impure extensions

now-invalid cell. This value is intended to be used only by **reuse** or **dec**. The former reuses it for a new constructor cell, asserting that its size is compatible with the old cell. The latter frees the cell, ignoring the replaced children.

If **reset** is used on a shared, non-reusable reference, it behaves like **dec** and returns $\blacksquare$, which instructs **reuse** to behave like **ctor**. Note that we cannot simply return the reference in both cases and do another uniqueness check in **reuse** because other code between the two expressions may have altered its reference count.

6.4 A Compiler from λ_{pure} to λ_{RC}

Following the actual implementation of the compiler in Lean, I will discuss a compiler from λ_{pure} to λ_{RC} in three steps:

1. inserting **reset**/**reuse** pairs (Section 6.4.1)
2. inferring borrowed parameters (Section 6.4.2)
3. inserting **inc**/**dec** instructions (Section 6.4.3)

Only the last step is obligatory for obtaining correct λ_{RC} programs. I describe a correctness proof of these steps relative to the semantics presented in the previous section in Appendix C.

6.4.1 Inserting Destructive Update Operations

As a first step, we will construct a heuristics-based function

$$\delta_{reuse} : Const \to Fn_{RC}$$

that amends a given pure program $\delta : Const \to Fn_{pure}$ by inserting **reset**/**reuse** instructions.

We define δ_{reuse} as

$$\delta_{reuse}(c) = \lambda\ \overline{y}.\ R(F) \quad \text{if } \delta(c) = \lambda\ \overline{y}.\ F$$

implementation. In the actual implementation in Lean, we avoid these memory writes by introducing a **del** instruction that behaves like **dec** but ignores the constructor fields.

$$R : FnBody_{pure} \rightarrow FnBody_{RC}$$
$$R(\texttt{let}\ x = e;\ F) = \texttt{let}\ x = e;\ R(F)$$
$$R(\texttt{ret}\ x) = \texttt{ret}\ x$$
$$R(\texttt{case}\ x\ \texttt{of}\ \overline{F}) = \texttt{case}\ x\ \texttt{of}\ \overline{D(x, n_i, R(F_i))}$$
where n_i = #fields of x in i-th branch

$$D : Var \times \mathbb{N} \times FnBody_{RC} \rightarrow FnBody_{RC}$$
$$D(x, n, \texttt{case}\ y\ \texttt{of}\ \overline{F}) = \texttt{case}\ y\ \texttt{of}\ \overline{D(x, n, F)}$$
$$D(x, n, \texttt{ret}\ y) = \texttt{ret}\ y$$
$$D(x, n, \texttt{let}\ y = e;\ F) = \texttt{let}\ y = e;\ D(x, n, F)$$
if $x \in e$ or $x \in F$
$$D(x, n, F) = \texttt{let}\ w = \texttt{reset}\ x;\ S(w, n, F)$$
otherwise, if $S(w, n, F) \neq F$ for a fresh w
$$D(x, n, F) = F \quad \text{otherwise}$$

$$S : Var \times \mathbb{N} \times FnBody_{RC} \rightarrow FnBody_{RC}$$
$$S(w, n, \texttt{let}\ x = \texttt{ctor}_i\ \overline{y};\ F) = \texttt{let}\ x = \texttt{reuse}\ w\ \texttt{in}\ \texttt{ctor}_i\ \overline{y};\ F$$
if $|\ \overline{y}\ | = n$
$$S(w, n, \texttt{let}\ x = e;\ F) = \texttt{let}\ x = e;\ S(w, n, F)$$
otherwise
$$S(w, n, \texttt{ret}\ x) = \texttt{ret}\ x$$
$$S(w, n, \texttt{case}\ x\ \texttt{of}\ \overline{F}) = \texttt{case}\ x\ \texttt{of}\ \overline{S(w, n, F)}$$

Figure 6.3: Inserting `reset`/`reuse` pairs

where $R(F)$ (Figure 6.3) uses a simple heuristic for replacing $\texttt{ctor}_i\ \overline{y}$ expressions occurring in F with $\texttt{reuse}\ w\ \texttt{in}\ \texttt{ctor}_i\ \overline{y}$ where w is a fresh variable introduced by R as the result of a new **reset** operation: for every operation $\texttt{case}\ x\ \texttt{of}\ \overline{F}$, R attempts to insert **reset** instructions for x. It does so by scanning each subsequent control path for a point where x becomes *dead* (unused) via the helper function D. At each such point, a further helper function S scans forward for possible *substitution* points where x could be reused: **ctor** expressions that have the same arity as the constructor value referenced by x in the respective branch of the original **case**.[4] If S does not find a matching substitution, D leaves this **case** branch unchanged. As an example, let us try to recreate the **reset**/**reuse** result for the function *map* from Section 6.2.

```
map f xs = case xs of
  (nil → ret xs)
  (cons →
   let x = proj_head xs;
   let s = proj_tail xs;
   let y = f x; let ys = map f s;
   let r = ctor_cons y ys; ret r)
```

Applied to the body of *map*, R will invoke D on both **case** branches. There is no change in the first branch, but in the second branch the discriminant *xs* becomes unused, and so S is invoked to find a constructor instruction of arity 2, the arity of the matched constructor *cons*. r binds such a constructor, so S injects a $\texttt{reuse}\ w\ \texttt{in}$, which leads to D to inject $\texttt{let}\ w = \texttt{reset}\ xs$ before the recursive call, resulting in the output we saw in Section 6.2 except for **inc** instructions inserted later.

As a second example, consider a function *swap* that swaps the first two elements of a list.

```
def swap : List α → List α
  | []           => []
  | [x]          => [x]
  | x :: y :: zs => y :: x :: zs
```

The λ_{pure} representation of this function is

[4] which I assume here as ambient information for simplicity; in the actual implementation, we save this information for each branch when we compile the typed frontend language into λ_{pure}

```
swap xs = case xs of
  (nil → ret xs)
  (cons → let t₁ = proj_tail xs ; case t₁ of
    (nil → ret xs)
    (cons →
     let h₁ = proj_head xs ;
     let h₂ = proj_head t₁ ; let t₂ = proj_tail t₁ ;
     let r₁ = ctor_cons h₁ t₂ ;
     let r₂ = ctor_cons h₂ r₁ ; ret r₂))
```

By applying R to *swap*, we obtain

```
swap xs = case xs of
  (nil → ret xs)
  (cons → let t₁ = proj_tail xs ; case t₁ of
    (nil → ret xs)
    (cons →
     let h₁ = proj_head xs ; let w₁ = reset xs ;
     let h₂ = proj_head t₁ ; let t₂ = proj_tail t₁ ; let w₂ = reset t₁ ;
     let r₁ = (reuse w₂ in ctor_cons h₁ t₂) ;
     let r₂ = (reuse w₁ in ctor_cons h₂ r₁) ; ret r₂))
```

As with *map*, this code is allocation-free if the relevant list cells are uniquely referenced.

6.4.2 Inferring Borrowing Signatures

The reuse optimization discussed in the previous section is the most important RC-based optimization as it avoids costly allocations; see also Section 6.7 for a quantitative comparison of the impact of the various optimizations. Inferring borrowing signatures in comparison is a secondary optimization that does not avoid allocations, but can avoid unnecessary RC modifications. As either optimization can potentially be applied to the same value, preventing the other optimization from applying, we make sure to run the more important reuse optimization first and to preserve its output during borrow inference.

Formally we will model the task of inferring borrowing signatures as a mapping $\beta : Const \rightharpoonup \{\mathbb{O}, \mathbb{B}\}^*$, which for every function should return a list describing each parameter of the function as either $\mathbb{O}$wned or $\mathbb{B}$orrowed — as mentioned in Section 6.2, borrow information is not part of the actual program. Marking a parameter as borrowed means that the caller of the

$$
\begin{aligned}
&\texttt{collect}_{\mathbb{O}} \;:\; \mathit{FnBody}_{\mathsf{RC}} \to 2^{\mathit{Vars}}\\
&\texttt{collect}_{\mathbb{O}}(\mathbf{let}\ z = \mathbf{ctor}_i\ \overline{x}\,;\ F) = \texttt{collect}_{\mathbb{O}}(F)\\
&\texttt{collect}_{\mathbb{O}}(\mathbf{let}\ z = \mathbf{reset}\ x\,;\ F) = \texttt{collect}_{\mathbb{O}}(F) \cup \{x\}\\
&\texttt{collect}_{\mathbb{O}}(\mathbf{let}\ z = \mathbf{reuse}\ x\ \mathbf{in}\ \mathbf{ctor}_i\ \overline{x}\,;\ F) = \texttt{collect}_{\mathbb{O}}(F)\\
&\texttt{collect}_{\mathbb{O}}(\mathbf{let}\ z = c\ \overline{x}\,;\ F) = \texttt{collect}_{\mathbb{O}}(F) \cup \{x_i \in \overline{x} \mid \beta(c)_i = \mathbb{O}\}\\
&\texttt{collect}_{\mathbb{O}}(\mathbf{let}\ z = x\ y\,;\ F) = \texttt{collect}_{\mathbb{O}}(F) \cup \{x, y\}\\
&\texttt{collect}_{\mathbb{O}}(\mathbf{let}\ z = \mathbf{pap}\ c_{\mathbb{O}}\ \overline{x}; F) = \texttt{collect}_{\mathbb{O}}(F) \cup \{\overline{x}\}\\
&\texttt{collect}_{\mathbb{O}}(\mathbf{let}\ z = \mathbf{proj}_i\ x\,;\ F) = \texttt{collect}_{\mathbb{O}}(F) \cup \{x\}\ \texttt{if}\ z \in \texttt{collect}_{\mathbb{O}}(F)\\
&\texttt{collect}_{\mathbb{O}}(\mathbf{let}\ z = \mathbf{proj}_i\ x\,;\ F) = \texttt{collect}_{\mathbb{O}}(F)\ \texttt{if}\ z \notin \texttt{collect}_{\mathbb{O}}(F)\\
&\texttt{collect}_{\mathbb{O}}(\mathbf{ret}\ x) = \emptyset\\
&\texttt{collect}_{\mathbb{O}}(\mathbf{case}\ x\ \mathbf{of}\ \overline{F}) = \textstyle\bigcup_{F_i \in \overline{F}} \texttt{collect}_{\mathbb{O}}(F_i)
\end{aligned}
$$

Figure 6.4: Collecting variables that should not be marked as borrowed

function will guarantee that the referenced object is alive for the entire call. Note that this view makes partially applying constants with borrowed parameters problematic because we do not, in general, know when the applied function will actually be called. Therefore we will extend the program δ_{reuse} from the previous section to a program δ_β by defining a trivial wrapper constant $c_{\mathbb{O}} := c$ (we will assume that this name is fresh) for any such constant c, set $\beta(c_{\mathbb{O}}) := \overline{\mathbb{O}}$, and replace any occurrence of **pap** $c\ \overline{y}$ with **pap** $c_{\mathbb{O}}\ \overline{y}$. The compiler step given in the next subsection will, as part of the general transformation, insert the necessary **inc** and **dec** instructions into $c_{\mathbb{O}}$ to convert between the two signatures.

Informally, the current heuristic for computing β says that a parameter should be borrowed if neither it nor any projections of it are used in **reset** or passed as an owned reference to a function (which itself may try to **reset** the corresponding parameter). This is sufficient to guarantee that borrow inference does not prevent reuse; in other cases like for **ret** the best choice is less clear as forcing parameters used in **ret** to be owned can avoid **inc** instructions, but may lead to more instructions in other branches of the function. The formal definition of the current heuristic that opportunistically marks parameters as borrowed in such cases is given in Fig. 6.4. This implementation depends on β already being defined appropriately for all constants referenced by the given function body. In the case of (mutual) recursion, we start with the approximation $\beta(c) = \mathbb{B}^{|\overline{y}|}$ for any involved constant c with $\delta(c) = \lambda\overline{y}.\ b$, then iteratively update each $\beta(c)$

using $collect_{\mathbb{O}}(b)$ until a fixed point is reached. By applying this heuristic to the *hasNone* function described before, we obtain $\beta(hasNone) = \mathbb{B}$. That is, in an application *hasNone xs*, *xs* is taken as a borrowed reference.

6.4.3 Inserting Reference Counting Operations

Given any well-formed definition of β and δ_β, I finally give a procedure for correctly inserting **inc** and **dec** instructions.[5]

$$\begin{aligned}
&\delta_{RC}(c) : Const \to Fn_{RC} \\
&\delta_{RC}(c) = \lambda\, \overline{y}.\, \mathbb{O}^-(\overline{y}, C(F, \beta_l)) \qquad \text{where} \quad \delta_\beta(c) = \lambda\, \overline{y}.\, F, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\;\; \beta_l = [\overline{y} \mapsto \beta(c), \ldots \mapsto \mathbb{O}]
\end{aligned}$$

The map $\beta_l : Var \to \{\mathbb{O}, \mathbb{B}\}$ keeps track of the borrow status of each local variable. For simplicity, I default all missing entries to $\mathbb{O}$.

In general, variables should be incremented prior to being used in an *owned context* that consumes an RC token, unless that is the last use of the variable. Variables used in any other (*borrowed*) context do not need to be incremented. The definition of the core compilation function C (Fig. 6.5) makes use of the following two helper functions to conditionally add RC instructions in these contexts:

- $\mathbb{O}^+_x$ prepares x for usage in an owned context by incrementing it. The increment can be omitted on the last use of an owned variable, with V representing the set of live variables after the use.

$$\begin{aligned}
\mathbb{O}^+_x(V, F, \beta_l) &= F && \text{if } \beta_l(x) = \mathbb{O} \land x \notin V \\
\mathbb{O}^+_x(V, F, \beta_l) &= \mathbf{inc}\ x;\ F && \text{otherwise}
\end{aligned}$$

- $\mathbb{O}^-_x$ decrements x if it is both owned and dead.

$$\begin{aligned}
\mathbb{O}^-_x(F, \beta_l) &= \mathbf{dec}\ x;\ F && \text{if } \beta_l(x) = \mathbb{O} \land x \notin \mathrm{FV}(F) \\
\mathbb{O}^-_x(F, \beta_l) &= F && \text{otherwise}
\end{aligned}$$

[5] I will tersely say that a variable x “is incremented/decremented” when an **inc**/**dec** operation is applied to it, i.e. the RC of the referenced object is incremented/decremented at runtime.

$$
\begin{aligned}
&C \ : \ \mathit{FnBody}_{\mathsf{RC}} \times (\mathit{Var} \to \{\mathbb{O}, \mathbb{B}\}) \to \mathit{FnBody}_{\mathsf{RC}} \\
&C(\mathbf{ret}\ x,\ \beta_l) = \mathbb{O}^+_x(\emptyset,\ \mathbf{ret}\ x,\ \beta_l) \\
&C(\mathbf{case}\ x\ \mathbf{of}\ \overline{F},\ \beta_l) = \mathbf{case}\ x\ \mathbf{of}\ \overline{\mathbb{O}^-(\overline{y}, C(F,\beta_l),\beta_l)} \\
&\qquad\qquad \text{where } \{\overline{y}\} = \mathrm{FV}(\mathbf{case}\ x\ \mathbf{of}\ \overline{F}) \\
&C(\mathbf{let}\ y = \mathbf{proj}_i\ x;\ F,\ \beta_l) = \mathbf{let}\ y = \mathbf{proj}_i\ x;\ \mathbf{inc}\ y;\ \mathbb{O}^-_x(C(F,\beta_l),\beta_l) \\
&\qquad\qquad \text{if } \beta_l(x) = \mathbb{O} \\
&C(\mathbf{let}\ y = \mathbf{proj}_i\ x;\ F,\ \beta_l) = \mathbf{let}\ y = \mathbf{proj}_i\ x;\ C(F,\ \beta_l[y \mapsto \mathbb{B}]) \\
&\qquad\qquad \text{if } \beta_l(x) = \mathbb{B} \\
&C(\mathbf{let}\ y = \mathbf{reset}\ x;\ F,\ \beta_l) = \mathbf{let}\ y = \mathbf{reset}\ x;\ C(F,\ \beta_l) \\
&C(\mathbf{let}\ z = c\ \overline{y};\ F,\ \beta_l) = C_{\mathsf{app}}(\overline{y}, \beta(c), \mathbf{let}\ z = c\ \overline{y};\ C(F,\beta_l),\beta_l) \\
&C(\mathbf{let}\ z = \mathbf{pap}\ c\ \overline{y};\ F,\ \beta_l) = C_{\mathsf{app}}(\overline{y}, \overline{\mathbb{O}}, \mathbf{let}\ z = \mathbf{pap}\ c\ \overline{y};\ C(F,\beta_l),\beta_l) \\
&C(\mathbf{let}\ z = x\ y;\ F,\ \beta_l) = C_{\mathsf{app}}(x\ y, \mathbb{O}\ \mathbb{O}, \mathbf{let}\ z = x\ y;\ C(F,\beta_l),\beta_l) \\
&C(\mathbf{let}\ z = \mathbf{ctor}_i\ \overline{y};\ F,\ \beta_l) = C_{\mathsf{app}}(\overline{y}, \overline{\mathbb{O}}, \mathbf{let}\ z = \mathbf{ctor}_i\ \overline{y};\ C(F,\beta_l),\beta_l) \\
&C(\mathbf{let}\ z = \mathbf{reuse}\ x\ \mathbf{in}\ \mathbf{ctor}_i\ \overline{y};\ F,\ \beta_l) = \\
&\quad C_{\mathsf{app}}(\overline{y}, \overline{\mathbb{O}}, \mathbf{let}\ z = \mathbf{reuse}\ x\ \mathbf{in}\ \mathbf{ctor}_i\ \overline{y};\ C(F,\beta_l),\beta_l)
\end{aligned}
$$

$$
\begin{aligned}
&C_{\mathsf{app}} : \mathit{Var}^* \times \{\mathbb{O}, \mathbb{B}\}^* \times \mathit{FnBody}_{\mathsf{RC}} \times (\mathit{Var} \to \{\mathbb{O}, \mathbb{B}\}) \to \mathit{FnBody}_{\mathsf{RC}} \\
&C_{\mathsf{app}}(y\ \overline{y'}, \mathbb{O}\ \overline{b}, \mathbf{let}\ z = e;\ F, \beta_l) = \\
&\qquad \mathbb{O}^+_y(\overline{y'} \cup \mathrm{FV}(F), C_{\mathsf{app}}(\overline{y'}, \overline{b}, \mathbf{let}\ z = e;\ F, \beta_l), \beta_l) \\
&C_{\mathsf{app}}(y\ \overline{y'}, \mathbb{B}\ \overline{b}, \mathbf{let}\ z = e;\ F, \beta_l) = \\
&\qquad C_{\mathsf{app}}(\overline{y'}, \overline{b}, \mathbf{let}\ z = e;\ \mathbb{O}^-_y(F, \beta_l), \beta_l) \\
&C_{\mathsf{app}}([], [], \mathbf{let}\ z = e;\ F, \beta_l) = \mathbf{let}\ z = e;\ F
\end{aligned}
$$

Figure 6.5: Inserting `inc`/`dec` instructions

$\mathbb{O}^-(\overline{x}, F, \beta_l)$ decrements multiple variables, which may be needed at the start of a function or `case` branch.

$$
\begin{aligned}
\mathbb{O}^-(x\ \overline{x'}, F, \beta_l) &= \mathbb{O}^-(\overline{x'}, \mathbb{O}^-_x(F, \beta_l), \beta_l) \\
\mathbb{O}^-([], F, \beta_l) &= F
\end{aligned}
$$

More specifically, `ret` is an owned context and the set of live variables after it is empty. `case` is not an owned context and any owned variables that are dead in any of the branches should be decremented. `proj` itself is not an owned context, but if the input is owned, we unconditionally increment the output to make it owned as well such that we do not have to track the

lifetime relationship between the two variables and we can decrement the input if it is now unused. If a borrowed variable is projected, we do not need to insert RC instructions, but we do have to update β_l.

Applications are handled by a separate function, recursing over the arguments and parameter borrow annotations in parallel; for partial, variable and constructor applications, the latter is set to all-$\mathbb{O}$. For any owned parameter, we increment the passed variable unless it is owned and not part of the remaining program *or* the remaining arguments. For a borrowed parameter, we merely make sure to decrement the passed variable after the call if it is owned and now unused. Note the subtle interplay between passing the same owned variable as both an owned and as a borrowed parameter: if it is passed as owned first, the latter occurrence in the argument list will trigger an increment that ensures the variable is live for the entirety of the call even after consuming one RC token for the owned parameter, and the borrowed parameter later will ensure to free this extra token via a decrement after the call if the variable is now unused. If instead the variable is passed as borrowed first, it first triggers the decrement, if any, after the call, after which the variable is now considered live after the call and any further use of it as an owned parameter will trigger an increment as expected. Similarly, if an owned variable is passed as a borrowed parameter multiple times in the same call, a decrement, if any, is generated only once because after the first such decrement the variable is considered live after the call. Thus in total, if the same variable is passed n times as an owned parameter in a call, it will be incremented $n-1$ times before the call, and once more if it must be live after the call because it is either syntactically used after it or because it is passed as borrowed at least once. Correspondingly, it will be decremented once after the call if it was borrowed at least once but is unused after that.

Examples

Let us validate the behavior of the compiler on two application special cases. The value of β_l is constant in these examples and left implicit in applications.

1. Consuming the same argument multiple times

$$\beta(\mathsf{c}) := \mathbb{O}\ \mathbb{O}$$

$$
\begin{aligned}
&\beta_l := [\mathsf{y} \mapsto \mathbb{O}] \\
&C(\mathbf{let}\ \mathsf{z = c\ y\ y};\ \mathbf{ret}\ \mathsf{z}) \\
&\quad = C_{app}(\mathsf{y\ y}, \mathbb{O}\ \mathbb{O}, \mathbf{let}\ \mathsf{z = c\ y\ y};\ C(\mathbf{ret}\ \mathsf{z})) \\
&\quad = C_{app}(\mathsf{y\ y}, \mathbb{O}\ \mathbb{O}, \mathbf{let}\ \mathsf{z = c\ y\ y};\ \mathbf{ret}\ \mathsf{z}) \\
&\quad = \mathbb{O}^+_{\mathsf{y}}(\{\mathsf{y}, \mathsf{z}\}, C_{app}(\mathsf{y}, \mathbb{O}, \mathbf{let}\ \mathsf{z = c\ y\ y};\ \mathbf{ret}\ \mathsf{z})) \\
&\quad = \mathbb{O}^+_{\mathsf{y}}(\{\mathsf{y}, \mathsf{z}\}, \mathbb{O}^+_{\mathsf{y}}(\{\mathsf{z}\}, C_{app}([], [], \mathbf{let}\ \mathsf{z = c\ y\ y};\ \mathbf{ret}\ \mathsf{z}))) \\
&\quad = \mathbb{O}^+_{\mathsf{y}}(\{\mathsf{y}, \mathsf{z}\}, \mathbb{O}^+_{\mathsf{y}}(\{\mathsf{z}\}, \mathbf{let}\ \mathsf{z = c\ y\ y};\ \mathbf{ret}\ \mathsf{z})) \\
&\quad = \mathbb{O}^+_{\mathsf{y}}(\{\mathsf{y}, \mathsf{z}\}, \mathbf{let}\ \mathsf{z = c\ y\ y};\ \mathbf{ret}\ \mathsf{z}) \\
&\quad = \mathbf{inc}\ \mathsf{y};\ \mathbf{let}\ \mathsf{z = c\ y\ y};\ \mathbf{ret}\ \mathsf{z}
\end{aligned}
$$

Because y is dead after the call, it needs to be incremented only once, *moving* its last token into c instead.

2. Borrowing and consuming the same argument

$$
\begin{aligned}
&\beta(\mathsf{c}) := \mathbb{B}\ \mathbb{O} \\
&\beta_l := [\mathsf{y} \mapsto \mathbb{O}] \\
&C(\mathbf{let}\ \mathsf{z = c\ y\ y};\ \mathbf{ret}\ \mathsf{z}) \\
&\quad = C_{app}(\mathsf{y\ y}, \mathbb{B}\ \mathbb{O}, \mathbf{let}\ \mathsf{z = c\ y\ y};\ C(\mathbf{ret}\ \mathsf{z})) \\
&\quad = C_{app}(\mathsf{y\ y}, \mathbb{B}\ \mathbb{O}, \mathbf{let}\ \mathsf{z = c\ y\ y};\ \mathbf{ret}\ \mathsf{z}) \\
&\quad = C_{app}(\mathsf{y}, \mathbb{O}, \mathbf{let}\ \mathsf{z = c\ y\ y};\ \mathbb{O}^-_{\mathsf{y}}(\mathbf{ret}\ \mathsf{z})) \\
&\quad = C_{app}(\mathsf{y}, \mathbb{O}, \mathbf{let}\ \mathsf{z = c\ y\ y};\ \mathbf{dec}\ \mathsf{y};\ \mathbf{ret}\ \mathsf{z}) \\
&\quad = \mathbb{O}^+_{\mathsf{y}}(\{\mathsf{y}, \mathsf{z}\},\ C_{app}([], [], \mathbf{let}\ \mathsf{z = c\ y\ y};\ \mathbf{dec}\ \mathsf{y};\ \mathbf{ret}\ \mathsf{z})) \\
&\quad = \mathbb{O}^+_{\mathsf{y}}(\{\mathsf{y}, \mathsf{z}\}, \mathbf{let}\ \mathsf{z = c\ y\ y};\ \mathbf{dec}\ \mathsf{y};\ \mathbf{ret}\ \mathsf{z}) \\
&\quad = \mathbf{inc}\ \mathsf{y};\ \mathbf{let}\ \mathsf{z = c\ y\ y};\ \mathbf{dec}\ \mathsf{y};\ \mathbf{ret}\ \mathsf{z}
\end{aligned}
$$

Even though the owned parameter comes after the borrowed parameter, the presence of y in the **dec** instruction emitted when handling the first parameter makes sure we do not accidentally move ownership when handling the second parameter, but copy y by emitting an **inc** instruction.

6.4.4 Preserving Tail Calls

An important caveat with the RC insertion algorithm given above is that it can destroy *tail calls*, which in the Lean IR have the shape **let** $r = c\,\overline{x}$; **ret** r, as seen the in the last example above. As recursive tail calls can be implemented as direct jumps without nesting stack frames, it is highly desirable to preserve them throughout the compiler. In the actual implementation in Lean, this is done with a small adjustment to the borrow inference algorithm outlined above: if a parameter involved in a recursive tail call is marked as borrowed but passed an owned variable, we mark it as owned instead. If the parameter is marked as borrowed and passed as borrowed, no change is necessary.

6.5 Optimizing Functional Data Structures for reset/reuse

In the previous examples, we have seen that the reuse optimization automatically handles operations over nested data structures like lists quite well. Let us now look at a slightly more complex example of a functional data structure, a red-black tree, as a more advanced example.

```
inductive Color where
  | r | b

inductive Tree α where
  | leaf : Tree α
  | node : Color → Tree α → α → Tree α → Tree α
```

Following [Okasaki, 1999], a red-black tree can be represented as an inductive type with two constructors: a nullary constructor for leaf nodes, and a constructor for interior nodes holding the color red or black, a left child, a generic value, and a right child.

As with the list operations, reuse analysis just works for many operations on such trees such as the following function for painting a node black.

```
def setBlack : Tree → Tree
  | .node _ a x b => .node .b a x b
  | t             => t
```

As long as the input reference is unshared, this function will perform an in-place update of it. Note that we can use the convenient "prefix dot" notation to reference an inductive type's constructor without prefixing it with the full type name, in which case the namespace will be inferred from the expected type at that location, while still clearly differentiating between constructor and variable names.

A more interesting operation is the following function for rebalancing the tree after an insertion.

```
def balance1 : Tree α → α → Tree α → Tree α
  | .node .r (.node .r a x b) y c, z, d
  | .node .r a x (.node .r b y c), z, d =>
    .node .r (.node .b a x b) y (.node .b c z d)
  | a, x, b => .node .b a x b

def ins [Ord α] (v : α) : Tree α → Tree α
  | .node .b a x b =>
      if v < x then
        balance1 (ins v a) x b else ...
  | ...
```

`balance1` corresponds to the first half of `balance` in [Okasaki, 1999], which is concerned with recoloring sequences of red nodes (a violation of the red-black invariant) resulting from insertion into the left child of a black node. Note that in Lean we can give multiple patterns separated by `|` that share a common right-hand side after =>.

Considering possible reuse, we see that in each case, the right-hand side contains one more `node` constructor reference than the corresponding pattern, so a priori at least one new node will be allocated on each call to `balance1`. However, should the compiler decide to inline the function into `ins`, which in Lean we can force it to do using an `@[inline]` annotation, another `node` pattern from `ins` comes into view and we can achieve insertion into sufficiently unshared red-black trees with only one allocation (the new node holding the inserted value) after all. Thus the efficacy of the reuse optimization may depend on other optimizations implemented by the compiler.

If we want to avoid depending on the inliner in this manner, all is not lost, however. We can refactor the code to create an intermediary `node` object in `ins`, reusing the passed `node` object if possible, and pass it to `balance1`, where we now have as many `node` patterns as constructor calls.

```
def balance1 : Tree α → Tree α
  | .node .b (.node .r (.node .r a x b) y c) z d
  | .node .b (.node .r a x (.node .r b y c)) z d =>
    .node .r (.node .b a x b) y (.node .b c z d)
  | a => a

def ins [Ord α] (v : α) : Tree α → Tree α
  | .node .b a x b =>
      if v < x then
        balance1 (.node .b (ins v a) x b) else ...
  | ...
```

Thus we again can achieve reusing all but one newly allocated node for insertion. This example shows that programming with reuse in mind might lead to different code patterns compared to languages lacking the optimization, where the second version could lead to additional allocations absent inlining. In fact, it could be argued that the second version, while being further from [Okasaki, 1999], is more natural: we use the recursive result of `ins` to assemble a temporary new tree that may violate the red-black invariant, then call upon `balance1` to transform it into a valid tree.

6.6 Runtime Considerations

Apart from the static optimizations described above, Lean 4 features multiple reference counting-related optimizations in its runtime. One of the most expensive features to support in a reference-counted runtime is multithreading: usually when values of a program may potentially be used by multiple threads at the same time, all reference-counting operations have to be upgraded to atomic versions, which introduces considerable overhead even for the usual majority of values that are accessed by only one thread after all. The Lean runtime improves upon this all-or-nothing approach by tagging runtime values with one of three states: *single-threaded*, *multi-threaded*, or *persistent*. Persistent values are the simplest: they are never freed and thus all reference-counting operations can be implemented as no-ops without any need for cross-thread synchronization. Persistent values may only reference other persistent values, and they are never considered for reuse. Persistent values are currently created in two ways:

the compiler marks “constant” values whose lifetime is the entire program run time as persistent. Additionally, Lean’s internal object graph serializer used to persist the results of checking a Lean module stores objects to disk as persistent values. Thus when this information is imported into subsequent Lean processes, we can access it without paying for reference-counting operations apart from a conditional jump on the object state, and in particular we can map the file into the process memory as shared, read-only pages as described in Section 3.3.

Multi-threaded values may reference persistent or other multi-threaded values and always use thread-safe reference-counting operations. Single-threaded values may reference values in any state and use unsynchronized reference-counting operations. Apart from the persistent cases mentioned above, a newly allocated value is marked as single-threaded. Before a single-threaded value is stored in a location that may potentially be accessed by multiple threads, which primarily consists of the closure passed to the thread creation primitive of the Lean runtime, as well as special mutable cells if they themselves are marked as multi-threaded, it is converted to the multi-threaded state, which is sufficient for establishing correct happens-before relations between reference-counting operations on the value both before and after conversion. Converting a value to multi-threaded implies converting all contained single-threaded values as well to preserve the invariant mentioned above. Both single-threaded and multi-threaded values can participate in reuse: if one thread observes a reference count of 1 on a multi-threaded value, there can be no other threads referencing the value and potentially conflicting with the reuse.

As mentioned there is a real cost of additional branches for checking the value state in each reference-counting operation. Note, however, that accessing and modifying the value state does not require atomic operations as all state transitions happen in single-threaded contexts. We will take a look at the real cost and benefit of this dynamic check compared to a simpler always-atomic implementation in Section 6.7.

One more aspect of the Lean runtime deserves to be mentioned here: apart from arbitrary inductive types, the runtime has special support for dynamically sized types such as contiguous arrays and strings. While reuse optimization as presented in this chapter aims to reuse existing allocations of inductive types, even when there is no direct semantic connection between the source and target of the reuse, the situation is quite different in case of the dynamically sized types, which are transformed not by pattern

matching and constructor applications but specific functions with built-in runtime implementations (that is, implementations not written in Lean) such as

```
Array.set : (a : Array α) → Fin a.size → Array α
String.push : String → Char → String
```

where the first operation implements proven-in-bounds array writes and the second operation implements string appends. Unlike with the reuse heuristic, there is a clear source for potential reuse in these cases. The runtime implementations of these functions dynamically check whether the input array or string is uniquely referenced, and if so perform a destructive update and return the changed value. Thus on unshared inputs the array write takes constant time and the string append takes asymptotically constant time (using a constant growth factor on reallocation) as expected. There is no need for a distinct **reset** step in these cases, which on a value referencing arbitrarily many other values would be counter-productive in any case.

6.7 Experimental Evaluation

All RC optimizations described in the previous sections are implemented in the Lean 4 compiler, written in Lean itself.[6] To test the efficiency of the compiler and RC optimizations, Leo and I have created and/or ported a number of benchmarks[7] that aim to replicate common tasks performed in compilers and proof assistants. All timings reported below are arithmetic means of 20 runs as reported by the `temci` benchmarking tool [Bechberger, 2016], executed on a PC with an i9-10900 Intel CPU (2.8GHz–5.2GHz, 20 hardware threads) and 32 GB RAM running Ubuntu 22.04, using Clang 14.0.1 for compiling the Lean runtime library as well as the C code emitted by the Lean compiler.

- `deriv` and `const_fold` are implementations of symbolic differentiation and constant folding of arithmetic expressions, respectively, exemplifying the kind of tree transformations that are found through-

6 https://github.com/leanprover/lean4/tree/IFL19/library/init/lean/compiler/ir
7 https://github.com/leanprover/lean4/tree/master/tests/bench

out the implementation of an interactive theorem prover and in particular of automation.

- `rbmap` stress tests a red-black tree implementation, parts of which I discussed in Section 6.6. The benchmarks `rbmap_10`, `rbmap_2`, and `rbmap_1` are variants where every 10th, second, or every tree operation, respectively, is prevented from performing destructive updates by retaining an additional reference to the tree. Thus `rbmap_1` presents the worst case for an implementation optimized for reuse, though as we will see in the results it still benefits from reuse in other parts of the benchmark.

- `parser` benchmarks the Lean parser (Section 3.4) on a single Lean file.

- `qsort` is an implementation of the quicksort sorting algorithm, using destructive updates of unshared arrays for an efficient yet simple and pure implementation in Lean.

- `binarytrees` is taken from the Computer Languages Benchmarks Game[8]. This benchmark is a simple adaption of Hans Boehm's `GCBench` benchmark[9]. The Lean version is a translation of the then fastest, parallelized Haskell solution.[10]

- `unionfind` implements the union-find algorithm, which is frequently used to implement decision procedures in automated reasoning, again making use of destructive array updates in the Lean implementation.

I have tested the impact of each optimization described in this chapter by selectively disabling it and comparing the resulting runtime with the base runtime (Fig. 6.6, Fig. 6.7):

[8] `https://benchmarksgame-team.pages.debian.net/benchmarksgame/program/binarytrees-ghc-6.html`

[9] `http://hboehm.info/gc/gc_bench/`

[10] Faster Haskell implementations using finer-grained parallelism and compact regions have since been submitted, but I have kept the version that is comparable to the versions in other languages.

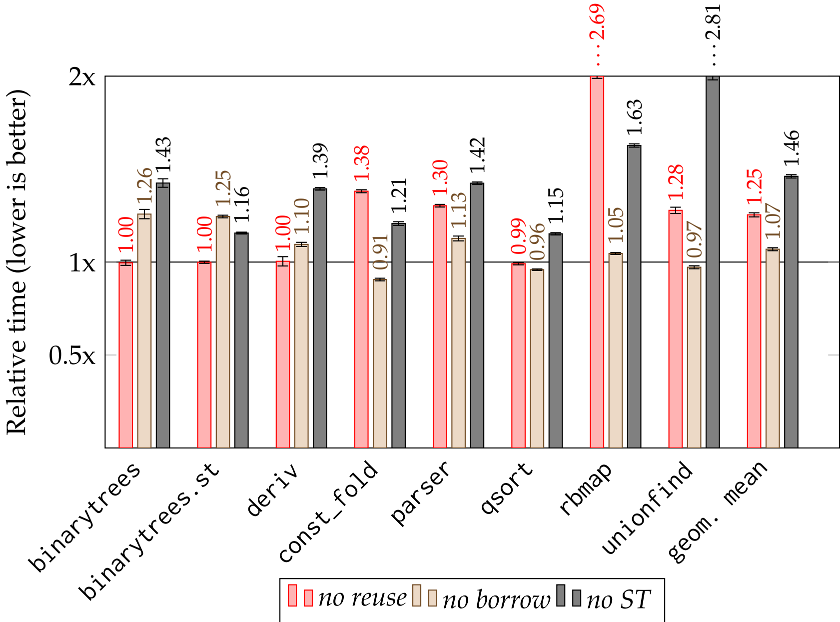


Figure 6.6: Lean variant benchmarks, normalized by the base run time. Error bars signify one standard deviation.

- *no reuse* disables the insertion of **`reset`**/**`reuse`** operations. Built-in functions as described in Section 6.6 are unaffected, i.e. at most constant-time overhead is introduced per operation. If built-ins such as array updates were disabled as well, array-based benchmarks would not terminate in any reasonable amount of time from the change in asymptotics.

- *no borrow* disables borrow inference, assuming that all parameters are owned. Note that the compiler must still honor borrow annotations on builtins, which are unaffected.

- *no ST* uses atomic RC operations for all values. Destructive updates are unaffected.

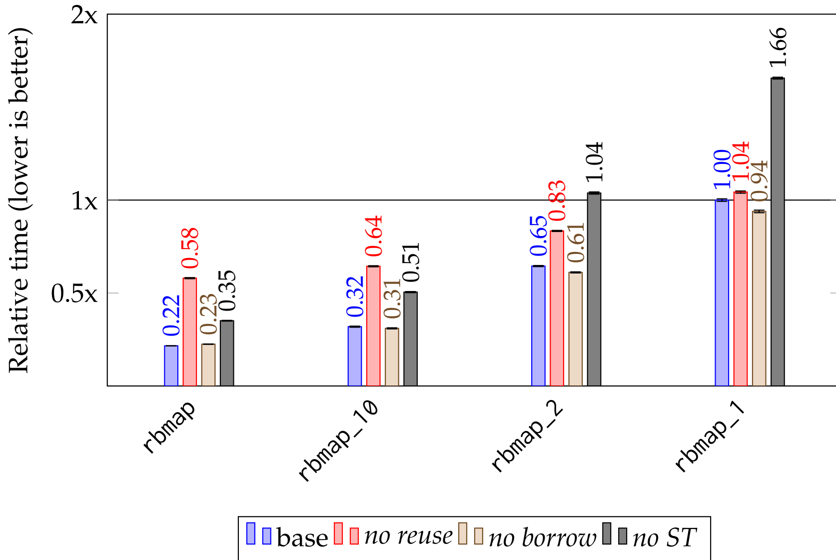


Figure 6.7: `rbmap` variation benchmarks by increasing frequency of retaining intermediate values, normalized by the base run time of `rbmap_1`.

The results show that the reuse analysis significantly improves performance in the benchmarks `const_fold`, `parser`, `rbmap`, and `unionfind`. `qsort` notably is unaffected because while it heavily benefits from destructive array updates, these are not part of reuse analysis per se as mentioned above. Borrow inference provides significant speedups in the benchmarks `binarytrees` and `deriv`, though it also leads to slight slowdowns in other benchmarks, suggesting that the heuristic could further be refined in the future. Avoiding atomic RC operations significantly speeds up every benchmark, including the heavily parallelized benchmark `binarytrees`.

Leo and I have also directly translated some of these programs to other statically typed, functional languages: Haskell, OCaml, and Standard ML (Fig. 6.8, Fig. 6.9). For the latter we selected the compilers MLton [Weeks, 2006], which performs whole program optimization and can switch between multiple GC schemes at runtime, and MLKit, which combines region inference and garbage collection [Hallenberg et al., 2002]. While not primarily a functional language, we have also included Swift

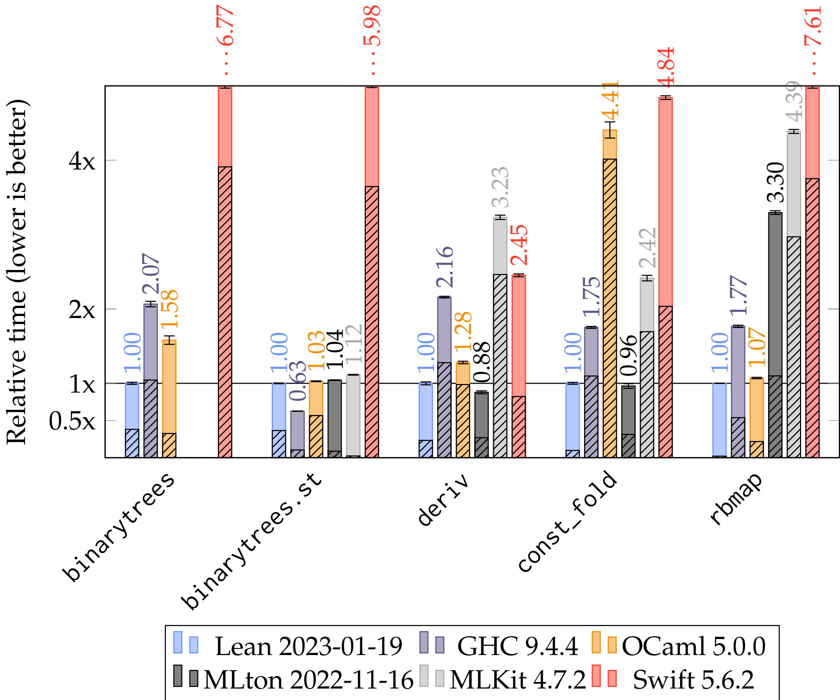


Figure 6.8: Cross-language benchmarks by wall-clock time, normalized by the Lean run time. Error bars signify one standard deviation, the striped portion signifies an approximation of GC time as reported by the respective compiler where available. For Swift, we measure time spent in inc, dec, and deallocation runtime functions as GC time using `perf`. For Lean, the "fast path" of a single-threaded inc or dec is inlined, so we can only measure atomic RC adjustment time and object deletion time.

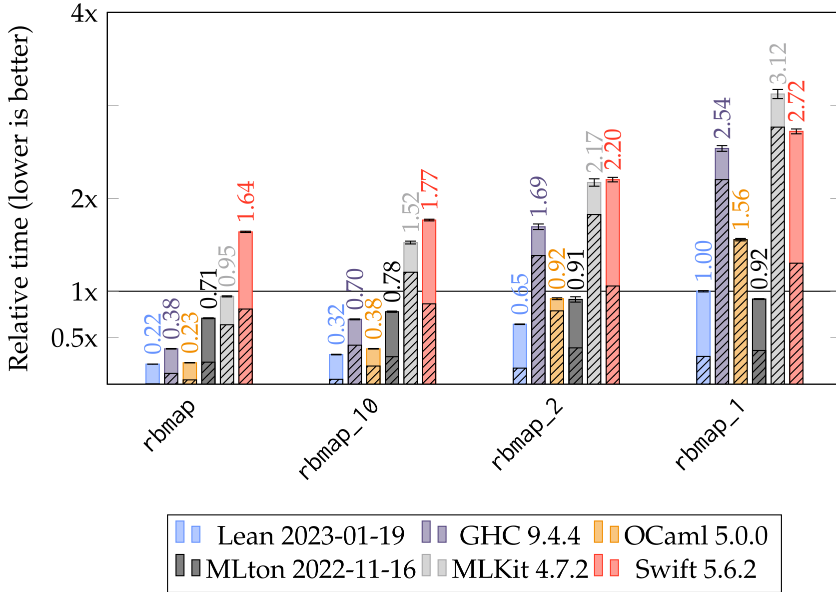


Figure 6.9: `rbmap` variation benchmarks by increasing frequency of retaining intermediate values, normalized by the Lean run time for `rbmap_1`.

as a popular statically typed language using reference counting. For `binarytrees`, we have used the mentioned Haskell implementation and comparable versions: For Swift, we used the second-fastest, safe Benchmark Game implementation, which is much more comparable to the other versions than the fastest one completely depending on unsafe code. The parallel OCaml 5.0 implementation is taken from the Sandmark benchmark suite[11]. We did not find a comparable SML implementation.

While the absolute runtimes in Fig. 6.8 and Fig. 6.9 are influenced by many factors other than the implementation of garbage collection that make direct comparisons difficult, the results still signify that both Lean's garbage collection and the overall runtime and compiler implementation

[11] `https://github.com/ocaml-bench/sandmark/blob/62a71ee/benchmarks/multicore-numerical/binarytrees5_multicore.ml`

are very competitive. The time Lean spends on garbage collection in particular is usually smaller than for other compilers, sometimes drastically so, never exceeding 50% of the total time. A notable exception is `binarytrees.st` where the allocation pattern of independent trees does not lend itself to reuse but seems to benefit some generational garbage collection implementations, though the advantage does not translate to the multi-threaded version in the case of GHC. Further machine-specific tweaking of GC parameters can alleviate a significant part of the overhead[12], but a significant benefit of our simple reference counting scheme is that there are no parameters to tweak.

In general, the results demonstrate that the differences in implementation are at least as impactful as the fundamental differences between reference counting and tracing garbage collection. Lean and Swift are both reference counted, but the lack of non-atomic reference counting updates in Swift introduces immense overhead in this kind of benchmarks. Fig. 6.9 shows how both GC schemes are impacted by increased retention, but the specific impact varies significantly, with MLton scaling much better than other implementations with a tracing GC, which eventually spend most of the time in the garbage collector.

6.8 Related Work

The idea of representing RC operations as explicit instructions so as to optimize them via static analysis is described as early as [Barth, 1977]. [Schulte, 1994] describes a system with many features similar to the ones I described above. In general, Schulte's language is much simpler than the Lean IR, with a single list type as the only non-primitive type, and no higher-order functions, nor is there a description of its formal dynamic semantics. Schulte gives an algorithm for inserting RC instructions that, like the algorithm from this chapter, has an on-the-fly optimization for omitting **`inc`** instructions if a variable is already dead and would immediately be decremented afterwards. Schulte briefly discusses how RC operations can be minimized by treating some parameters as "nondestructive" in the sense of our borrowed references. In contrast to our inference of

[12] `https://gitlab.haskell.org/ghc/ghc/-/issues/14981#note_474988`

borrow annotations, Schulte proposes to create one copy of a function for each possible destructive/nondestructive combination of parameters (i.e. exponential in the number of (non-primitive) parameters) and to select an appropriate version for each call site of the function. Our approach never duplicates code.

Introducing destructive updates into pure programs has traditionally focused on primitive operations like array updates [Hudak and Bloss, 1985], particularly in the functional array languages SISAL [McGraw et al., 1983] and SAC [Scholz, 1994]. [Grelck and Trojahner, 2004] propose an instruction `alloc_or_reuse` for SAC that can select one of multiple array candidates for reuse, but do not describe heuristics for when to use the instruction. [Férey and Shankar, 2016] describe how functional update operations explicit in the source language can be turned into destructive updates using the reference counter. In contrast, [Schulte, 1994] presents a "reusage" optimization that has an effect similar to the one obtained with our **`reset`**/**`reuse`** instructions. In particular, it is independent of a specific surface-level update syntax. However, his optimization (named transformation *T14*) is more restrictive and is only applicable to a branch of a **`case`** x if x is dead at the beginning of the branch. His optimization cannot handle the simple *swap* described earlier, let alone more complex functions such as the red black tree re-balancing function $balance_1$.

While not a purely functional language, the Swift programming language[13] has directly influenced many parts of the work described in this chapter. To the best of my knowledge, Swift was the first non-research language to use an intermediate representation with explicit RC instructions, as well as the idea of (safely) avoiding RC operations via "borrowed" parameters (which are called "+0" or "guaranteed" in Swift), in its implementation. While Swift's primitives may also elide copies when given a unique reference, no speculative destructive updates are introduced for user-defined types, but this may not be as important for an impure language as it is for Lean. Parameters default to borrowed in Swift, but the compiler may locally change the calling convention inside individual modules.

[Baker, 1994] describes optimizing reference counting by use of *two* pointer kinds, a standard one and a *deferred increment* pointer kind. The

[13] `https://developer.apple.com/swift/`

latter kind can be copied freely without adding RC operations, but must be converted into the standard kind by incrementing it before storing it in an object or returning it. The two kinds are distinguished at runtime by pointer tagging. Our borrowed references can be viewed as a static refinement of this idea. Baker then describes an extended version of deferred-increment he calls *anchored* pointers that store the stack level (i.e. the lifetime) of the standard pointer they have been created from. Anchored pointers do not have to be converted to the standard kind if returned from a stack frame above this level. In order to statically approximate this extended system, we would need to extend the IR type system with support for some kind of *lifetime annotations* on return types as featured in Cyclone [Jim et al., 2002] and Rust.

[Ungar et al., 2017] optimize Swift's reference counting scheme by using a single bit to tag objects possibly shared between multiple threads, much like Lean's approach. However, because of mutability, every single store operation must be intercepted to (recursively) tag objects before becoming reachable from an already tagged object. [Choi et al., 2018] remove the need for tagging by extending every object header with the ID of the thread T that allocated the value, and two reference counters: a shared one that requires atomic operations, and another one that is only updated by T. Again thanks to immutability, we can make use of the simpler scheme for Lean without introducing store barriers during normal code generation. Object tagging instead only has to be done in threading primitives and explicitly mutable cells.

After publication of the original paper ([Ullrich and de Moura, 2019a]) this chapter is based on, the presented approach has been adapted and further extended in context of the Koka language [Leijen, 2014]: [Reinking et al., 2021] describe Koka's RC algorithm "Perceus" based on a linear resource calculus that is more general and higher-level than the ANF presentation in this chapter[14], and further extend the presented work with *drop* and *reuse specialization*, which replace uses of the generic instructions corresponding to Lean's **dec** and **reuse**, respectively, with specialized code for the given type and reuse context. Work is underway on a new Lean IR

[14] In particular, it does not directly support n-ary function applications, which are a significant source of complication as we have seen in Section 6.4.3, allocating $n-1$ intermediary closures instead.

with more type information that would allow adopting this optimization. [Lorenzen and Leijen, 2022] introduce *drop-guided, frame-limited reuse* for Koka, which improves on the reuse algorithm presented above by both limiting the memory overhead reuse analysis can introduce by holding on to allocations and introducing new opportunities for reuse. The basic idea is that instead of running reuse analysis before RC insertion as described in this chapter, turning the order around allows one to use the lifetime information encoded by the inserted RC instructions in the reuse analysis for the mentioned goals. However, borrow inference is not part of this work and the approach for it described in this chapter is not directly adaptable as it is designed to run after reuse analysis but before RC insertion. As of this writing, we have not incorporated these improvements back into Lean, partially because the performance of the initial design described here proved sufficient for our use cases, but we are interested in revisiting them in the future.

7 Conclusion

Here my pen shall halt, reader, though I do not
– Gene Wolfe, *The Book of the New Sun*

I have carried you, as Gene Wolfe would have said, from gate to gate — from the parser to the depths of the runtime system, from the theory of the kernel to the practice of syntax design. After more than four years of development, we have achieved an unprecedented state of extensibility with Lean 4. The macro system I have developed is the central piece of this work, powering everything from simple to understand notations to powerful type-aware elaboration procedures as part of a unified architecture. It is a direct continuation of the decades of work on such systems in the Lisp family, taking their learned lessons and applying them to the peculiarities of interactive theorem provers. The extended `do` notation is a prime example for how the macro system enables users to embed non-trivial domain-specific languages into Lean naturally, without compromising the ability to reason about it and its output. But far from a mere example, as the syntactic foundation of the pure imperative programming paradigm it has by now become a fundamental aspect of how programming in Lean is done by beginners and experts alike. Meanwhile, the optimized reference counting scheme as the semantic foundation of the paradigm has far exceeded the expectations of Leo and me, both in the demonstrated competitiveness compared to established functional compilers that made us double-check our results as well as in the quick adoption by Koka and the resulting series of publications uncovering further refinements.

In time with the conclusion of my PhD, Lean 4 has now reached a degree of maturity where it is starting to be embraced by programmers, lecturers, and mathematicians porting over Lean 3 mathlib at an incredible speed. It is now up to these users to utilize the flexibility we have provided and together with us explore new applications of Lean and interactive theorem proving at large.

7.1 Future Work

There is, however, only so much a group of, for the majority of the time, two people can achieve in four years. We have not yet come around to research, design, and implement everything we had planned, and had to set aside interesting subproblems for a lack of time.

Kernel and Type System. I have elided a more formal description of nested inductive types from Section 3.2, which would likely take at least as much space as the description of the other extensions combined. Other properties such as decidability of algorithmic reduction and type checking have not been investigated for either Lean 3 or Lean 4 so far. An interesting approach for doing this is to formalize the algorithm in the prover itself assuming correctness of the theory and extract a certified, executable type checker from it [Sozeau et al., 2019].

An unintended aspect of the type system that did not change between Lean 3 and 4 is the undecidability of definitional equality and non-transitivity of algorithmic equality [Carneiro, 2019]. Different solutions to this problem have been proposed, including an explicit move towards the undecidability of extensional type theory, but will have to be evaluated on their practical impact, especially as we are not aware of such impact resulting from the theoretic issue.

Syntax and Macros. For the most part, the current syntax system has been shaped by the needs of the Lean implementation itself. We have seen people successfully apply it to building their own embedded languages, but also notice current limitations such as not being able to introduce custom token parsers. Will our approach scale to their needs, will we need deeper grammar analysis for optimization and introspection? Only time will tell.

The macro system as well features some deliberate limitations such as the lack of local macros. It is not unimaginable that with more and more sophisticated macros written by users, we will want to integrate even more ideas from the Lisp world of macros and extend our hygiene algorithm.

Finally, a topic I was quite interested in researching, but which I abandoned during my PhD in favor of focusing on the presented topics, is the theory side of macros and hygiene. Given Lean's capabilities as a theorem

prover, it would be tempting to try and prove that its hygienic macro system is actually so, as well as proving some equivalence between the Lean and Racket hygiene systems on Lean's more limited macro language. It is not immediately clear whether existing formal definitions of macro hygiene [Adams, 2015, Pombrio et al., 2017] would be sufficiently applicable to this language.

Pure Imperative Programming. The extended `do` notation as described features a minimum viable set of imperative primitives to make the extension worthwhile, but even more extensions have (such as `try` ... `catch`/`finally`) and surely will be integrated into Lean in the future. A particular problem we have not tackled so far is the question of how to extend `do` blocks and their effects across sufficiently generic higher-order functions as well as local helper functions. With more extensions, however, there is always the danger of introducing confusion for users regarding the expected semantics and control flow, so more research of the usability side for people from different backgrounds will be necessary.

On the runtime side, the convenience of implicitly inserted allocation reuse is also its main limitation currently. In order for the execution time of a Lean program to be more robust in the face of refactorings or simple oversights, it would be desirable to statically guarantee reuse at important locations. At the time of writing, Marc Huisinga is completing a master's thesis on such a uniqueness type system [Smetsers et al., 1994] for a limited set of Lean programs under my supervision, but further work will be necessary to cover more of the Lean language and to integrate it into Lean and expose it to users in a convenient fashion.

A

Macro Implementation of *do* Notation

This appendix contains the complete reference implementation of the extended `do` notation from Chapter 5. `macro`s and `macro_rules` are annotated with their corresponding translation/abbreviation name.

A.1 Basic do Notation

```
open Lean

declare_syntax_cat stmt
syntax "do'" stmt : term

-- Prevent `return/let/...` from being parsed as a term
syntax (priority := low) term : stmt
syntax "let" ident "←" stmt:1 ";" stmt : stmt
macro "{" s:stmt "}" : stmt => `($s)

syntax "d!" stmt : term  -- corresponds to `D(s)`

macro_rules
  | `(do' $s) => `(d! $s)  -- (1)

-- helper function; see usage below
def expandStmt (s : TSyntax `stmt) : MacroM (TSyntax `stmt) := do
  let s' ← expandMacros s
  if s == s' then
    Macro.throwUnsupported
```

```
  else
    -- There is no static guarantee that `expandMacros` stays in
    -- the `stmt` category, but it is true for all our macros
    return TSyntax.mk s'

macro_rules
/-(D1)-/ | `(d! $e:term) => `($e)
/-(D2)-/ | `(d! let $x ← $s; $s') =>
           `((d! $s) >>= fun $x => (d! $s'))
         | `(d! $s) => do
  -- fallback rule: try to expand abbreviation
  let s' ← expandStmt s
  `(d! $s')

/-(A1)-/ macro "let" x:ident ":=" e:term ";" s:stmt : stmt => `(let
    $x ← pure $e; $s)
-- priority `0` prevents `;` from being used in trailing contexts
    without braces (see e.g. `:1` above)
/-(A2)-/ macro:0 s₁:stmt ";" s₂:stmt : stmt => `(let x ← $s₁; $s₂)
```

A.2 Mutable Variables

```
import Do.Basic

open Lean

syntax "let" "mut" ident ":=" term ";" stmt : stmt
syntax ident ":=" term : stmt
syntax "if" term "then" stmt:1 : stmt

declare_syntax_cat expander
-- generic syntax for traversal-like functions S_y/R/B/L
syntax "expand!" expander "in" stmt:1 : stmt
syntax "mut" ident : expander  -- corresponds to `S_y`

-- generic traversal rules
macro_rules
  -- subsumes (R3, B4, L4)
  | `(stmt| expand! $exp in let $x ← $s; $s') => `(stmt| let $x ←
    expand! $exp in $s; expand! $exp in $s')
  -- subsumes (R4, B5, L5)
```

```
  | `(stmt| expand! $exp in let mut $x := $e; $s') => `(stmt| let
    mut $x := $e; expand! $exp in $s')
  -- subsumes (R5, B6, L6)
  | `(stmt| expand! $_ in $x:ident := $e) => `(stmt| $x:ident := $e)
  -- subsumes (S6, R6, B7, L7)
  | `(stmt| expand! $exp in if $e then $s₁) => `(stmt| if $e then
    expand! $exp in $s₁)
  | `(stmt| expand! $exp in $s) => do
    let s' ← expandStmt s
    `(stmt| expand! $exp in $s')


macro_rules
/-(D3)-/ | `(d! let mut $x := $e; $s) => `(let $x := $e; StateT.run'
    (d! expand! mut $x in $s) $x)
         | `(d! $x:ident := $_:term) =>
             throw <| Macro.Exception.error x s!"variable '{x.getId}'
    is not reassignable in this scope"
/-(D4)-/ | `(d! if $e then $s₁) => `(if $e then d! $s₁ else pure ())
                                       -- (D4)

macro_rules
/-(S1)-/ | `(stmt| expand! mut $_ in $e:term) => `(stmt| StateT.lift
    $e)
/-(S2)-/ | `(stmt| expand! mut $y in let $x ← $s; $s') =>
             if x == y then
               throw <| Macro.Exception.error x s!"cannot shadow
    'mut' variable '{x.getId}'"
             else
               `(stmt| let $x ← expand! mut $y in $s; let $y ← get;
    expand! mut $y in $s')
/-(S3)-/ | `(stmt| expand! mut $y in let mut $x := $e; $s') =>
             if x == y then
                 throw <| Macro.Exception.error x s!"cannot shadow
    'mut' variable '{x.getId}'"
               else
                 `(stmt| let mut $x := $e; expand! mut $y in $s')
         | `(stmt| expand! mut $y in $x:ident := $e) =>
             if x == y then
/-(S5)-/       `(stmt| set $e)
             else
```

```
/-(S6)-/      `(stmt| $x:ident := $e)

/-(A3)-/ macro:0 "let" "mut" x:ident "←" s:stmt:1 ";" s':stmt :
     stmt => `(let y ← $s; let mut $x := y; $s')
/-(A4)-/ macro:0 x:ident "←" s:stmt:1 : stmt => `(let y ← $s;
     $x:ident := y)
-- a variant of (A4) since we technically cannot make the above
     macro a `stmt`
macro:0 x:ident "←" s:stmt:1 ";" s':stmt : stmt => `(let y ← $s;
     $x:ident := y; $s')

/- Examples -/

variable [Monad m]
variable (ma ma' : m α)

-- mark `map_eq_pure_bind : f <$> x = x >>= pure (f a)` as a
     simplification lemma.
attribute [local simp] map_eq_pure_bind

-- The typeclass `LawfulMonad` encodes the monad laws (6.2-6.4)
example [LawfulMonad m] :
    (do' let mut x ← ma;
         pure x : m α)
    =
    ma
:= by simp

example [LawfulMonad m] :
    (do' let mut x ← ma;
         x ← ma';
         pure x)
    =
    (ma >>= fun _ => ma')
:= by simp

--  `#check_failure` succeeds if the given term fails to be elaborated
#check_failure do'
  let mut x ← ma;
  let x ← ma';  -- cannot shadow 'mut' variable 'x'
  pure x
```

```
#check_failure do'
  x ← ma;  -- variable 'x' is not reassignable in this scope
  pure ()

variable (b : Bool)

-- The following equivalence is true even if `m` does not satisfy
     the monad laws
example :
    (do' if b then {
           discard ma
         })
    =
    (if b then discard ma else pure ())
:= rfl

theorem simple [LawfulMonad m] :
    (do' let mut x ← ma;
         if b then {
           x ← ma'
         };
         pure x)
    =
    (ma >>= fun x => if b then ma' else pure x)
:= by cases b <;> simp

-- nondeterminism example from Section 6.2
def choose := @List.toLazy

def ex : LazyList Nat := do'
  let mut x := 0;
  let y ← choose [0, 1, 2, 3];
  x := x + 1;
  guard (x < 3);
  pure (x + y)

-- Generate all solutions
#eval ex.toList  -- [1, 2, 3, 4]
```

A.3 Early Return

```
import Do.Mut

open Lean

def runCatch [Monad m] (x : ExceptT α m α) : m α :=
  ExceptT.run x >>= fun
    | Except.ok x => pure x
    | Except.error e => pure e

/-- Count syntax nodes satisfying `p`. -/
partial def Lean.Syntax.count (stx : Syntax) (p : Syntax → Bool) :
    Nat :=
  stx.getArgs.foldl (fun n arg => n + arg.count p) (if p stx then 1
    else 0)

syntax "return" term : stmt

syntax "return" : expander

macro_rules
/-(6.1')-/ | `(do' $s) => do
  -- optimization: fall back to original rule (1) if now `return`
    statement was expanded
  let s' ← expandStmt (← `(stmt| expand! return in $s))
  if s'.raw.count ·( matches `(stmt| return $_)) == s.raw.count ·(
    matches `(stmt| return $_)) then
    `(d! $s)
  else
    `(ExceptCpsT.runCatch (d! $s'))

macro_rules
/-(R1)-/ | `(stmt| expand! return in return $e) => `(stmt| throw $e)
/-(R2)-/ | `(stmt| expand! return in $e:term) => `(stmt|
    ExceptCpsT.lift $e)

/- Examples -/

variable [Monad m]
variable (ma ma' : m α)
```

```
variable (b : Bool)

example [LawfulMonad m] :
    (do' let x ← ma;
         return x)
    = ma
:= by simp

example : Id.run
    (do' let x := 1; return x)
    = 1
:= rfl

example [LawfulMonad m] :
     (do' if b then {
            let x ← ma;
            return x
          };
          ma')
     =
     (if b then ma else ma')
:= by cases b <;> simp
```

A.4 Iteration

```
import Do.Return

open Lean

syntax "for" ident "in" term "do'" stmt:1 : stmt
syntax "break " : stmt
syntax "continue " : stmt

syntax "break" : expander
syntax "continue" : expander
syntax "lift" : expander

macro_rules
  -- subsumes (S7, R7, B2, L1)
  | `(stmt| expand! $_   in break) => `(stmt| break)
  -- subsumes (S8, R8, L2)
```

```
  | `(stmt| expand! $_   in continue) => `(stmt| continue)
  -- subsumes (L8, R9)
  | `(stmt| expand! $exp in for $x in $e do' $s) => `(stmt| for $x
    in $e do' expand! $exp in $s)

macro_rules
  -- (D5), optimized like (6.1')
  | `(d! for $x in $e do' $s) => do
    let mut s := s
    let sb ← expandStmt (← `(stmt| expand! break in $s))
    let hasBreak := sb.raw.count ·( matches `(stmt| break)) <
    s.raw.count ·( matches `(stmt| break))
    if hasBreak then
      s := sb
    let sc ← expandStmt (← `(stmt| expand! continue in $s))
    let hasContinue := sc.raw.count ·( matches `(stmt| continue)) <
    s.raw.count ·( matches `(stmt| continue))
    if hasContinue then
      s := sc
    let mut body ← `(d! $s)
    if hasContinue then
      body ← `(ExceptCpsT.runCatch $body)
    let mut loop ← `(forM $e (fun $x => $body))
    if hasBreak then
      loop ← `(ExceptCpsT.runCatch $loop)
    pure loop
  | `(d! break%$b) =>
    throw <| Macro.Exception.error b "unexpected 'break' outside
    loop"
  | `(d! continue%$c) =>
    throw <| Macro.Exception.error c "unexpected 'continue' outside
    loop"

macro_rules
/-(B1)-/ | `(stmt| expand! break in break) => `(stmt| throw ())
/-(B3)-/ | `(stmt| expand! break in $e:term) => `(stmt|
    ExceptCpsT.lift $e)
/-(B8)-/ | `(stmt| expand! break in for $x in $e do' $s) => `(stmt|
    for $x in $e do' expand! lift in $s)
         | `(stmt| expand! continue in continue) => `(stmt| throw ())
```

```
        | `(stmt| expand! continue in $e:term) => `(stmt|
    ExceptCpsT.lift $e)
        | `(stmt| expand! continue in for $x in $e do' $s) =>
    `(stmt| for $x in $e do' expand! lift in $s)

macro_rules
/-(L3)-/ | `(stmt| expand! lift in $e:term) => `(stmt|
    ExceptCpsT.lift $e)

macro_rules
/-(S9)-/ | `(stmt| expand! mut $y in for $x in $e do' $s) => `(stmt|
    for $x in $e do' { let $y ← get; expand! mut $y in $s })

/- Examples -/

variable [Monad m]
variable (ma ma' : m α)
variable (b : Bool)
variable (xs : List α) (act : α → m Unit)

attribute [local simp] map_eq_pure_bind

example [LawfulMonad m] :
    (do' for x in xs do' {
          act x
        })
    =
    xs.forM act
:= by induction xs <;> simp_all!

def ex2 (f : β → α → m β) (init : β) (xs : List α) : m β := do'
  let mut y := init;
  for x in xs do' {
    y ← f y x
  };
  return y

example [LawfulMonad m] (f : β → α → m β) :
    ex2 f init xs = xs.foldlM f init := by
  unfold ex2; induction xs generalizing init <;> simp_all!
```

```
@[simp] theorem List.find?_cons {xs : List α} : (x::xs).find? p =
     if p x then some x else xs.find? p := by
  cases h : p x <;> simp_all!

example (p : α → Bool) : Id.run
    (do' for x in xs do' {
           if p x then {
             return some x
           }
         };
         pure none)
    =
    xs.find? p
:= by induction xs with
      | nil => simp [Id.run, List.find?]
      | cons x => cases h : p x <;> simp_all [Id.run]

variable (p : α → m Bool)

theorem byCases_Bool_bind (x : m Bool) (f g : Bool → m β) (isTrue
     : f true = g true) (isFalse : f false = g false) : (x >>= f) =
     (x >>= g) := by
  have : f = g := by
    funext b
    cases b with
    | true  => exact isTrue
    | false => exact isFalse
  rw [this]

theorem eq_findM [LawfulMonad m] :
    (do' for x in xs do' {
           let b ← p x;
           if b then {
             return some x
           }
         };
         pure none)
    =
    xs.findM? p
:= by induction xs with
      | nil => simp!
```

```
    | cons x xs ih =>
      rw [List.findM?, ← ih]; simp
      apply byCases_Bool_bind <;> simp

def ex3 [Monad m] (p : α → m Bool) (xss : List (List α)) : m
    (Option α) := do'
  for xs in xss do' {
    for x in xs do' {
      let b ← p x;
      if b then {
        return some x
      }
    }
  };
  pure none

theorem eq_findSomeM_findM [LawfulMonad m] (xss : List (List α)) :
    ex3 p xss = xss.findSomeM? (fun xs => xs.findM? p) := by
  unfold ex3
  induction xss with
  | nil => simp!
  | cons xs xss ih =>
    simp [List.findSomeM?]
    rw [← ih, ← eq_findM]
    induction xs with
    | nil => simp
    | cons x xs ih => simp; apply byCases_Bool_bind <;> simp [ih]

def List.untilM (p : α → m Bool) : List α → m Unit
  | []     => pure ()
  | a::as => p a >>= fun | true => pure () | false => as.untilM p

theorem eq_untilM [LawfulMonad m] :
  (do' for x in xs do' {
         let b ← p x;
         if b then {
           break
         }
       })
  =
  xs.untilM p
```

```
:= by induction xs with
      | nil => simp!
      | cons x xs ih =>
        simp [List.untilM]; rw [← ih]; clear ih
        apply byCases_Bool_bind <;> simp

/- Adding `repeat` and `while` statements -/

@[specialize] partial def loopForever [Monad m] (f : Unit → m Unit)
    : m Unit :=
  f () *> loopForever f

inductive Loop' where
  | mk : Loop'

instance : ForM m Loop' Unit where
  forM _ f := loopForever f

macro:0 "repeat" s:stmt:1 : stmt => `(stmt| for u in Loop'.mk do' $s)

#eval do'  -- 0 1 2 3
  let mut i := 0;
  repeat {
    if i ≥ 3 then {
      break
    };
    IO.println i;
    i := i + 1
  };
  return i

macro:0 "while" c:term "do'" s:stmt:1 : stmt =>
  `(stmt| repeat { if !$c then break; { $s } })

#eval do'  -- 0 1 2 3
  let mut i := 0;
  while (i < 3) do' {
    IO.println i;
    i := i + 1
  };
  return i
```

B

Formal Correctness of *do* Translation

```
import Do.For
import Lean
import Aesop
```

In this appendix, I present an intrinsically typed representation of the syntax of do statements from Chapter 5 as well of their translation functions and an equivalence proof thereof to a simple denotational semantics.

The appendix is generated from a Lean file in literate style using the documentation generator Alectryon [Pit-Claudel, 2020]. I make extensive use of the proof search tactic Aesop [Limperg and From, 2023] for automation.

B.1 Contexts

We represent contexts as lists of types and assignments of them as heterogeneous lists over these types. As is common with lists, contexts grow to the left in our presentation. The following encoding of heterogeneous lists avoids the universe bump of the usual inductive definition ([Altenkirch, 2010]).

```
def HList : List (Type u) → Type u
  | []      => PUnit
  | α :: αs => α × HList αs

@[matchPattern]
abbrev HList.nil : HList [] := ⟨⟩
@[matchPattern]
```

```
abbrev HList.cons (a : α) (as : HList αs) : HList (α :: αs) := (a, as)
```

We overload the common list notations `::` and `[e, ...]` for `HList` using the macro system. Note the recursive macro usage in the latter notation.

```
infixr:67 " :: " => HList.cons

syntax (name := hlistCons) "[" term,* "]" : term
macro_rules (kind := hlistCons)
  | `([])           => ``(HList.nil)
  | `([$x])         => ``(HList.cons $x [])
  | `([$x, $xs,*]) => ``(HList.cons $x [$xs,*])
```

Lean's very general, heterogeneous definition of ++ causes some issues with our overloading above in terms such as `a ++ [b]`, so we restrict it to the `List` interpretation in the following, which is sufficient for our purposes.

```
local macro_rules
  | `($a ++ $b) => ``(List.append $a $b)

abbrev Assg Γ := HList Γ
```

The following function updates a heterogeneous list at a given, guaranteed in-bounds index. It uses the subtype `Fin n` of natural numbers smaller than `n`.

```
def HList.set : {αs : List (Type u)} → HList αs →
    (i : Fin αs.length) → αs.get i → HList αs
  | _ :: _, _ :: as, ⟨0,      _⟩, b => b :: as
  | _ :: _, a :: as, ⟨n + 1, h⟩, b =>
    a :: set as ⟨n, Nat.le_of_succ_le_succ h⟩ b
  | [],     [],       _,            _ => []
```

We write ∅ for empty contexts and assignments and $\Gamma \vdash \alpha$ for the type of values of type α under the context Γ, that is, the function type from an assignment to α.

```
instance : EmptyCollection (Assg ∅) where
  emptyCollection := []
```

```
notation:30 Γ " ⊢ " α => Assg Γ → α

def Assg.drop : Assg (α :: Γ) → Assg Γ
  | _ :: as => as
```

In one special case, we will need to manipulate contexts from the right, i.e. the outermost scope.

```
def Assg.extendBot (x : α) : {Γ : _} → Assg Γ → Assg (Γ ++ [α])
  | [],      []      => [x]
  | _ :: _, a :: as => a :: extendBot x as

def Assg.dropBot : {Γ : _} → Assg (Γ ++ [α]) → Assg Γ
  | [],      _       => []
  | _ :: _, a :: as => a :: dropBot as
```

B.2 Intrinsically Typed Representation of do Statements

The type `Stmt` representing statements defined below is parameterized by

- `m`: base monad (fixed: to the left of the colon)
- ω: `return` type, `m` ω is the type of the entire do block (fixed)
- Γ: do-local immutable context
- Δ: do-local mutable context
- `b`: `break` allowed
- `c`: `continue` allowed
- α: local result type, `m` α is the type of the statement

The constructor signatures are best understood by comparing them with the corresponding typing rules in Section 5.6. Note that the choice of de Bruijn indices changes/simplifies some parts, such as obviating freshness checks (`x` $\notin$ Δ).

```
inductive Stmt (m : Type → Type u) (ω : Type) :
    (Γ Δ : List Type) → (b c : Bool) → (α : Type) → Type _ where
  | expr (e : Γ ⊢ Δ ⊢ m α) : Stmt m ω Γ Δ b c α
  | bind (s : Stmt m ω Γ Δ b c α)
      (s' : Stmt m ω (α :: Γ) Δ b c β) :
      Stmt m ω Γ Δ b c β  -- let _ ← s; s'
  | letmut (e : Γ ⊢ Δ ⊢ α) (s : Stmt m ω Γ (α :: Δ) b c β) :
      Stmt m ω Γ Δ b c β  -- let mut _ := e; s
  | assg (x : Fin Δ.length) (e : Γ ⊢ Δ ⊢ Δ.get x) :
      Stmt m ω Γ Δ b c Unit  -- x := e
  | «if» (e : Γ ⊢ Δ ⊢ Bool) (s : Stmt m ω Γ Δ b c Unit) :
      Stmt m ω Γ Δ b c Unit  -- if e then s
  | ret (e : Γ ⊢ Δ ⊢ ω) : Stmt m ω Γ Δ b c α   -- return e
  | «for» (e : Γ ⊢ Δ ⊢ List α)
      (s : Stmt m ω (α :: Γ) Δ true true Unit) :
      Stmt m ω Γ Δ b c Unit  -- for _ in e do s
  | «break» : Stmt m ω Γ Δ true c α  -- break
  | cont : Stmt m ω Γ Δ b true α  -- continue
```

Neutral statements are a restriction of the above type.

```
inductive Neut (ω α : Type) : (b c : Bool) → Type _ where
  | val (a : α) : Neut ω α b c
  | ret (o : ω) : Neut ω α b c
  | «break» : Neut ω α true c
  | cont : Neut ω α b true
```

We elide `Neut.val` where unambiguous.

```
instance : Coe α (Neut ω α b c) := ⟨Neut.val⟩
instance : Coe (Id α) (Neut ω α b c) := ⟨Neut.val⟩
```

We write `e[ρ][σ]` for the substitution of both contexts in `e`, a simple application in this encoding. `σ[x ↦ v]` updates σ at `x` (a de Bruijn index).

```
macro:max (priority := high)
  e:term:max noWs "[" ρ:term "]" "[" σ:term "]" : term => `($e $ρ $σ)
macro:max (priority := high)
  σ:term:max noWs "[" x:term " ↦ " v:term "]" : term =>
    ``(HList.set $σ $x $v)
```

B.3 Dynamic Evaluation Function

A direct encoding of the operational semantics from Section 5.6 as a definitional interpreter, generalized over an arbitrary monad. Note that the immutable context ρ is accumulated (`v :: ` ρ) and passed explicitly instead of immutable bindings being substituted immediately as that is a better match for the above definition of `Stmt`. Iteration over the values of the given list in the `for` case introduces a nested, mutually recursive helper function, with termination of the mutual bundle following from a size argument over the statement primarily and the length of the list in the `for` case secondarily.

```
@[simp] def Stmt.eval [Monad m] (ρ : Assg Γ) :
    Stmt m ω Γ Δ b c α → Assg Δ → m (Neut ω α b c × Assg Δ)
  | .expr e, σ => e[ρ][σ] >>= fun v => pure ⟨v, σ⟩
  | .bind s s', σ =>
    -- defining this part as a separate definition helps Lean
    -- with the termination proof
    let rec @[simp] cont val
      | ⟨.val v, σ'⟩ => val v σ'
      -- the `Neut` type family forces us to repeat these cases
      -- as the LHS/RHS indices are not identical
      | ⟨.ret o, σ'⟩ => pure ⟨.ret o, σ'⟩
      | ⟨.break, σ'⟩ => pure ⟨.break, σ'⟩
      | ⟨.cont, σ'⟩ => pure ⟨.cont, σ'⟩
    s.eval ρ σ >>= cont (fun v σ' => s'.eval (v :: ρ) σ')
  | .letmut e s, σ =>
    s.eval ρ (e[ρ][σ], σ) >>= fun ⟨r, σ'⟩ => pure ⟨r, σ'.drop⟩
  -- `x` is a valid de Bruijn index into `σ` by definition of `assg`
  | .assg x e, σ => pure ⟨(), σ[x ↦ e[ρ][σ]]⟩
  | .if e s, σ => if e[ρ][σ] then s.eval ρ σ else pure ⟨(), σ⟩
  | .ret e, σ => pure ⟨.ret e[ρ][σ], σ⟩
  | .for e s, σ =>
    let rec @[simp] go σ
      | [] => pure ⟨(), σ⟩
      | a::as =>
        s.eval (a :: ρ) σ >>= fun
          | ⟨(), σ'⟩ => go σ' as
          | ⟨.cont, σ'⟩ => go σ' as
          | ⟨.break, σ'⟩ => pure ⟨(), σ'⟩
          | ⟨.ret o, σ'⟩ => pure ⟨.ret o, σ'⟩
```

```
    go σ e[ρ][σ]
  | .break, σ => pure ⟨.break, σ⟩
  | .cont, σ => pure ⟨.cont, σ⟩
termination_by
  eval s _ => (sizeOf s, 0)
  eval.go as => (sizeOf s, as.length)
```

At the top-level statement, the contexts are empty, no loop control flow statements are allowed, and the return and result type are identical.

```
abbrev Do m α := Stmt m α ∅ ∅ false false α

def Do.eval [Monad m] (s : Do m α) : m α :=
  Stmt.eval ∅ s ∅ >>= fun
    | ⟨Neut.val a, _⟩ => pure a
    | ⟨Neut.ret o, _⟩ => pure o

notation "⟦" s "⟧" => Do.eval s
```

B.4 Translation Functions

We adjust the immutable context where necessary. The mutable context never has to be adjusted.

```
@[simp] def Stmt.mapAssg (f : Assg Γ' → Assg Γ) :
    Stmt m ω Γ Δ b c β → Stmt m ω Γ' Δ b c β
  | .expr e => .expr (e ∘ f)
  | .bind s₁ s₂ =>
    .bind (s₁.mapAssg f) (s₂.mapAssg (fun (a :: as) => (a :: f as)))
  | .letmut e s => .letmut (e ∘ f) (s.mapAssg f)
  | .assg x e => .assg x (e ∘ f)
  | .if e s => .if (e ∘ f) (s.mapAssg f)
  | .ret e => .ret (e ∘ f)
  | .for e s => .for (e ∘ f) (s.mapAssg (fun (a :: as) => (a :: f as)))
  | .break => .break
  | .cont => .cont
```

Let us write $\mathtt{f} \circ_e \mathtt{e}$ for the composition of $\mathtt{f} : \alpha \to \beta$ with $\mathtt{e} : \Gamma \vdash \Delta \vdash \alpha$, which we will use to rewrite embedded terms.

```
infixr:90 " ∘ₑ "  => fun f e => fun ρ σ => f e[ρ][σ]
```

The formalization of S presents some technical hurdles. Because it operates on the outer-most mutable binding, we have to operate on that context from the right, from which we lose some helpful definitional equalities and have to rewrite types using nested proofs instead.

The helper function shadowSnd is particularly interesting because it shows how the shadowing in translation rules (S2) and (S9) is expressed in our de Bruijn encoding: The context $\alpha :: \beta :: \alpha :: \Gamma$ corresponds, in this order, to the y that has just been bound to the value of get, then x from the respective rule, followed by the y of the outer scope. We encode the shadowing of y by dropping the third element from the context as well as the assignment. We are in fact forced to do so because the corresponding branches of S would not otherwise typecheck. The only mistake we could still make is to drop the wrong α value from the assignment, which (speaking from experience) would eventually be caught by the correctness proof.

```
@[simp] def S [Monad m] : Stmt m ω Γ (Δ ++ [α]) b c β →
    Stmt (StateT α m) ω (α :: Γ) Δ b c β
/-(S1)-/ | .expr e => .expr (StateT.lift ∘ₑ unmut e)
/-(S2)-/ | .bind s₁ s₂ =>
           .bind (S s₁) (.bind (.expr (fun _ _ => get))
                                (.mapAssg shadowSnd (S s₂)))
/-(S3)-/ | .letmut e s => .letmut (unmut e) (S s)
         | .assg x e =>
           if h : x < Δ.length then
/-(S4)-/     .assg ⟨x, h⟩ (fun (y :: ρ) σ => List.get_append_left .. ▶
                                  e ρ (Assg.extendBot y σ))
           else
/-(S5)-/     .expr (set (σ := α) ∘ₑ cast (List.get_last h) ∘ₑ unmut e)
/-(S6)-/ | .if e s => .if (unmut e) (S s)
         -- unreachable case; could be eliminated by a more precise
         -- specification of `ω`, but the benefit would be minimal
         | .ret e => .ret (unmut e)
/-(S7)-/ | .break => .break
/-(S8)-/ | .cont => .cont
/-(S9)-/ | .for e s =>
           .for (unmut e) (.bind (.expr (fun _ _ => get))
                                  (.mapAssg shadowSnd (S s)))
where
  @[simp] unmut {β} (e : Γ ⊢ Δ ++ [α] ⊢ β) : α :: Γ ⊢ Δ ⊢ β
```

```
    | y :: ρ, σ => e ρ (Assg.extendBot y σ)
  @[simp] shadowSnd {β} :
      Assg (α :: β :: α :: Γ) → Assg (α :: β :: Γ)
    | a’ :: b :: _ :: ρ => a’ :: b :: ρ
```

Formalizing the remaining translation functions is straightforward.

```
@[simp] def R [Monad m] :
    Stmt m ω Γ Δ b c α → Stmt (ExceptT ω m) Empty Γ Δ b c α
/-(R1)-/ | .ret e => .expr (throw ∘ₑ e)
/-(R2)-/ | .expr e => .expr (ExceptT.lift ∘ₑ e)
/-(R3)-/ | .bind s s’ => .bind (R s) (R s’)
/-(R4)-/ | .letmut e s => .letmut e (R s)
/-(R5)-/ | .assg x e => .assg x e
/-(R6)-/ | .if e s => .if e (R s)
/-(R7)-/ | .break => .break
/-(R8)-/ | .cont => .cont
/-(R9)-/ | .for e s => .for e (R s)

@[simp] def L [Monad m] :
    Stmt m ω Γ Δ b c α → Stmt (ExceptT Unit m) ω Γ Δ b c α
/-(L1)-/ | .break => .break
/-(L2)-/ | .cont => .cont
/-(L3)-/ | .expr e => .expr (ExceptT.lift ∘ₑ e)
/-(L4)-/ | .bind s s’ => .bind (L s) (L s’)
/-(L5)-/ | .letmut e s => .letmut e (L s)
/-(L6)-/ | .assg x e => .assg x e
/-(L7)-/ | .if e s => .if e (L s)
         | .ret e => .ret e
/-(L8)-/ | .for e s => .for e (L s)

@[simp] def B [Monad m] :
    Stmt m ω Γ Δ b c α → Stmt (ExceptT Unit m) ω Γ Δ false c α
/-(B1)-/ | .break => .expr (fun _ _ => throw ())
/-(B2)-/ | .cont => .cont
/-(B3)-/ | .expr e => .expr (ExceptT.lift ∘ₑ e)
/-(B4)-/ | .bind s s’ => .bind (B s) (B s’)
/-(B5)-/ | .letmut e s => .letmut e (B s)
/-(B6)-/ | .assg x e => .assg x e
/-(B7)-/ | .if e s => .if e (B s)
         | .ret e => .ret e
/-(B8)-/ | .for e s => .for e (L s)
```

```
-- (elided in the paper)
@[simp] def C [Monad m] : Stmt m ω Γ Δ false c α →
    Stmt (ExceptT Unit m) ω Γ Δ false false α
  | .cont => .expr (fun _ _ => throw ())
  | .expr e => .expr (ExceptT.lift ∘ₑ e)
  | .bind s s' => .bind (C s) (C s')
  | .letmut e s => .letmut e (C s)
  | .assg x e => .assg x e
  | .if e s => .if e (C s)
  | .ret e => .ret e
  | .for e s => .for e (L s)
```

The remaining function to be translated is `D`, which is straightforward as well except for its termination proof, as it recurses on the results of `S` (D3) and `C ∘ B` (D5). Because of rules (S2, S9) that introduce new bindings, `S` may in fact increase the size of the input, and the same is true for `C` and `B` for the `sizeOf` function automatically generated by Lean. Thus we introduce a new measure `numExts` that counts the number of special statements on top of basic `do` notation and prove that all three functions do not increase the size according to that measure. Because the rules (D3) and (D5) each eliminate such a special statement, it follows that `D` terminates because either the number of special statements decreases in each case, or it remains the same and the total number of statements decreases.

```
@[simp] def Stmt.numExts : Stmt m ω Γ Δ b c α → Nat
  | .expr _ => 0
  | .bind s₁ s₂ => s₁.numExts + s₂.numExts
  | .letmut _ s => s.numExts + 1
  | .assg _ _ => 1
  | .if _ s => s.numExts
  | .ret _ => 1
  | .for _ s => s.numExts + 1
  | .break => 1
  | .cont => 1

@[simp] theorem Stmt.numExts_mapAssg (f : Assg Γ' → Assg Γ)
    (s : Stmt m ω Γ Δ b c β) :
    numExts (mapAssg f s) = numExts s := by
  induction s generalizing Γ' <;> simp [*]
```

```
theorem Stmt.numExts_S [Monad m] (s : Stmt m ω Γ (Δ ++ [α]) b c β) :
    numExts (S s) ≤ numExts s := by
  -- `induction` does not work with non-variable indices, so we first
  -- generalize `Δ ++ [α]` into an explicit equation
  revert s
  suffices {Δ': _ } → (s : Stmt m ω Γ Δ' b c β) →
      (h : Δ' = (Δ ++ [α])) →
      numExts (S (h ▸ s : Stmt m ω Γ (Δ ++ [α]) b c β)) ≤ numExts s
    from fun s => this s rfl
  intro Δ' s h
  induction s generalizing Δ with
    subst h
  | bind _ _ ih₁ ih₂ => simp [Nat.add_le_add, ih₁ rfl, ih₂ rfl]
  | letmut _ _ ih =>
    simp [Nat.add_le_add, ih (List.cons_append ..).symm]
  | assg => aesop
  | «if» _ _ ih => simp [Nat.add_le_add, ih rfl]
  | «for» _ _ ih => simp [Nat.add_le_add, ih rfl]
  | _ => simp

theorem Stmt.numExts_L_L [Monad m] (s : Stmt m ω Γ Δ b c β) :
    numExts (L (L s)) ≤ numExts s := by
  induction s <;> simp [Nat.add_le_add, *]

theorem Stmt.numExts_C_B [Monad m] (s : Stmt m ω Γ Δ b c β) :
    numExts (C (B s)) ≤ numExts s := by
  induction s <;> simp [Nat.add_le_add, numExts_L_L, *]

-- Auxiliary tactic for showing that `D` terminates
macro "D_tac" : tactic =>
  `({simp_wf
     solve
      | apply Prod.Lex.left; assumption
      | apply Prod.Lex.right' <;> simp_arith })

@[simp] def D [Monad m] : Stmt m Empty Γ ∅ false false α → (Γ ⊢ m α)
  | .expr e => (e[·][∅])
  | .bind s s' => (fun ρ => D s ρ >>= fun x => D s' (x :: ρ))
  | .letmut e s =>
    -- for termination
    have := Nat.lt_succ_of_le <| Stmt.numExts_S (Δ := []) s
```

```
    fun ρ =>
      let x := e[ρ][∅]
      StateT.run' (D (S s) (x :: ρ)) x
  | .if e s => (fun ρ => if e[ρ][∅] then D s ρ else pure ())
  | .for e s =>
    -- for termination
    have := Nat.lt_succ_of_le <| Stmt.numExts_C_B (Δ := []) s
    fun ρ => runCatch
      (forM e[ρ][∅] (fun x => runCatch (D (C (B s)) (x :: ρ))))
  | .ret e => (nomatch e[·][∅])
termination_by _ s => (s.numExts, sizeOf s)
decreasing_by D_tac
```

Finally we compose `D` and `R` into the translation rule for a top-level statement (6.1′).

```
def Do.trans [Monad m] (s : Do m α) : m α := runCatch (D (R s) ∅)
```

B.5 Equivalence Proof

Using the monadic definitional interpreter, we can modularly prove for each individual translation function that evaluating its output is equivalent to directly evaluating the input, modulo some lifting and adjustment of resulting values. After induction on the statement, the proofs are mostly concerned with case splitting, application of congruence theorems, and simplification. We can mostly offload these tasks onto Aesop.

```
attribute [local simp] map_eq_pure_bind ExceptT.run_bind
attribute [aesop safe apply] bind_congr

variable [Monad m] [LawfulMonad m]

theorem eval_R (s : Stmt m ω Γ Δ b c α) :
    (R s).eval ρ σ = (
      ExceptT.lift (s.eval ρ σ) >>= fun x =>
        match (generalizing := false) x with
        | (.ret o, _) => throw o
        | (.val a, σ) => pure (.val a, σ)
        | (.cont, σ) => pure (.cont, σ)
        | (.break, σ) => pure (.break, σ)
```

```
      : ExceptT ω m (Neut Empty α b c × Assg Δ)) := by
  apply ExceptT.ext
  induction s with
  | «for» e =>
    simp only [Stmt.eval, R]
    induction e ρ σ generalizing σ <;>
      aesop (add norm unfold Stmt.eval.go)
  | _ => aesop (add unsafe cases Neut)
             (erase Aesop.BuiltinRules.destructProducts)

@[simp] theorem eval_mapAssg (f : Assg Γ' → Assg Γ) :
    ∀ (s : Stmt m ω Γ Δ b c β),
      Stmt.eval ρ (.mapAssg f s) σ = Stmt.eval (f ρ) s σ := by
  intro s
  induction s generalizing Γ' with
  | «for» e s ih =>
    simp only [Stmt.eval, Stmt.mapAssg, Function.comp]
    induction e (f ρ) σ generalizing σ <;>
      aesop (add norm unfold Stmt.eval.go)
  | _ => aesop (add unsafe cases Neut)
```

We need one last helper function on context bottoms to be able to state the invariant of `S`, and then prove various lemmas about their interactions.

```
def Assg.bot : {Γ : _} → Assg (Γ ++ [α]) → α
  | [],      [a]      => a
  | _ :: _, _ :: as => bot as

@[simp] theorem Assg.dropBot_extendBot (a : α) (σ : Assg Γ) :
    Assg.dropBot (Assg.extendBot a σ) = σ := by
  induction Γ <;> cases σ <;> simp [dropBot, extendBot, *]
@[simp] theorem Assg.bot_extendBot (a : α) (σ : Assg Γ) :
    Assg.bot (Assg.extendBot a σ) = a := by
  induction Γ <;> cases σ <;> simp [bot, extendBot, *]
@[simp] theorem Assg.extendBot_bot_dropBot (σ : Assg (Γ ++ [α])) :
    Assg.extendBot (Assg.bot σ) (Assg.dropBot σ) = σ := by
  induction Γ <;> cases σ <;> simp [dropBot, bot, extendBot, *]
  rfl

@[simp] theorem Assg.dropBot_set_extendBot_init (a : α) (σ : Assg Γ)
    (h : i.1 < Γ.length) {b} :
```

```
    Assg.dropBot ((Assg.extendBot a σ).set i b) =
      σ.set ⟨i.1, h⟩ (List.get_append_left .. ▸ b) := by
  induction Γ with
  | nil => contradiction
  | cons _ _ ih =>
    cases σ
    have ⟨i, h'⟩ := i
    cases i <;> simp [HList.set, dropBot]
    rw [ih]

@[simp] theorem Assg.bot_set_extendBot_init (a : α) (σ : Assg Γ)
    (h : i.1 < Γ.length) {b} :
    Assg.bot ((Assg.extendBot a σ).set i b) = a := by
  induction Γ with
  | nil => contradiction
  | cons _ _ ih =>
    cases σ
    have ⟨i, h'⟩ := i
    cases i <;> simp [HList.set, dropBot, bot]
    rw [ih]; apply Nat.lt_of_succ_lt_succ h

@[simp] theorem Assg.dropBot_set_extendBot_bottom (a : α) (σ : Assg Γ)
    (h : ¬ i.1 < Γ.length) {b} :
    Assg.dropBot ((Assg.extendBot a σ).set i b) = σ := by
  induction Γ with
  | nil => rfl
  | cons _ _ ih =>
    cases σ
    have ⟨i, h'⟩ := i
    cases i
    · apply False.elim (h (Nat.zero_lt_succ _))
    · simp [HList.set, dropBot]
      rw [ih]
      intro h''
      apply False.elim (h (Nat.succ_lt_succ h''))

@[simp] theorem Assg.bot_set_extendBot_bottom (a : α) (σ : Assg Γ)
    (h : ¬ i.1 < Γ.length) {b} :
    Assg.bot ((Assg.extendBot a σ).set i b) =
      cast (List.get_last h) b := by
  induction Γ with
```

```
  | nil =>
    have ⟨i, h'⟩ := i
    cases i
    · simp [HList.set, extendBot, bot]; rfl
    · apply False.elim
      apply Nat.not_lt_zero _ (Nat.lt_of_succ_lt_succ h')
  | cons  _ _ ih =>
    cases σ
    have ⟨i, h'⟩ := i
    cases i
    · apply False.elim (h (Nat.zero_lt_succ _))
    · simp [bot]
      rw [ih]
      intro h''
      apply False.elim (h (Nat.succ_lt_succ h''))

theorem eval_S : ∀ (s : Stmt m ω Γ (Δ ++ [α]) b c β),
    StateT.run ((S s).eval (a :: ρ) σ) a =
      s.eval ρ (Assg.extendBot a σ) >>= fun
      | r :: σ => pure ((r :: Assg.dropBot σ), Assg.bot σ) := by
  suffices {Δ': _ } → (s : Stmt m ω Γ Δ' b c β) →
    (h : Δ' = (Δ ++ [α])) →
    StateT.run ((S (h ▸ s)).eval (a :: ρ) σ) a
      = s.eval ρ (h ▸ Assg.extendBot a σ) >>= fun
      | r :: σ => pure ((r :: Assg.dropBot (h ▸ σ)),
                        Assg.bot (h ▸ σ))
    from fun s => this s rfl
  intro Δ' s h
  induction s generalizing Δ a with
    subst h
  | bind s₁ s₂ ih₁ ih₂ =>
    have ih₁ := @ih₁ (h := rfl)
    have ih₂ := @ih₂ (h := rfl)
    aesop (add safe cases Neut)
  | letmut e s ih =>
    have ih := @ih (Δ := _ :: Δ) (h := rfl)
    aesop
  | «for» e s ih =>
    have ih := @ih (h := rfl)
    simp only [S, Stmt.eval, S.unmut]
    -- surgical generalization
```

```
    generalize h : a = a'
    conv =>
      pattern HList.cons a' _
      rw [← h]
    clear h
    induction e ρ _ generalizing σ a' <;> aesop (add safe cases Neut)
  | _ => aesop

theorem HList.eq_nil (as : HList ∅) : as = ∅ := rfl

attribute [local simp] ExceptT.run_bind

theorem eval_L (s : Stmt m ω Γ Δ b c α) :
    (L s).eval ρ σ = ExceptT.lift (s.eval ρ σ) := by
  apply ExceptT.ext
  induction s with
  | «for» e =>
    simp only [Stmt.eval, L]
    induction e ρ σ generalizing σ <;> aesop
  | _ => aesop (add safe cases Neut)

theorem eval_B (s : Stmt m ω Γ Δ b c α) :
    (B s).eval ρ σ =
      (ExceptT.lift (s.eval ρ σ) >>= fun x =>
        match (generalizing := false) x with
        | (.ret o, σ) => pure (.ret o, σ)
        | (.val a, σ) => pure (.val a, σ)
        | (.cont, σ) => pure (.cont, σ)
        | (.break, _) => throw ()
        : ExceptT Unit m (Neut ω α false c × Assg Δ)) := by
  apply ExceptT.ext
  induction s with
  | «for» e =>
    simp only [Stmt.eval, B]
    induction e ρ σ generalizing σ <;> aesop (add norm simp eval_L)
  | _ => aesop (erase Aesop.BuiltinRules.destructProducts)

@[simp] def throwOnContinue : (Neut ω α false c × Assg Δ) →
    ExceptT Unit m (Neut ω α false false × Assg Δ)
  | (.ret o, σ) => pure (.ret o, σ)
  | (.val a, σ) => pure (.val a, σ)
```

```
  | (.cont, _) => throw ()

theorem eval_C (s : Stmt m ω Γ Δ false c α) : (C s).eval ρ σ =
    ExceptT.lift (s.eval ρ σ) >>= throwOnContinue := by
  revert s
  suffices {b: _ } → (s' : Stmt m ω Γ Δ b c α) → (h : b = false) →
      let s : Stmt m ω Γ Δ false c α := h ▸ s'
      (C s).eval ρ σ = ExceptT.lift (s.eval ρ σ) >>= throwOnContinue
    from fun s => this s rfl
  intro b' s h
  induction s with
    (first | subst h | trivial)
  | «for» e =>
    simp only [Stmt.eval, C]
    induction e ρ σ generalizing σ <;>
      aesop (add norm simp eval_L, unsafe apply ExceptT.ext)
  | _ => aesop (add unsafe apply ExceptT.ext)

theorem D_eq {m} [Monad m] [LawfulMonad m] :
    (s : Stmt m Empty Γ ∅ false false α) →
    D s ρ = s.eval ρ ∅ >>= fun (Neut.val a, _) => pure a
  | .expr e => by simp
  | .bind s₁ s₂ => by
    have ih₁ := @D_eq (s := s₁)
    have ih₂ := @D_eq (s := s₂)
    aesop
  | .letmut e s => by
    -- for termination
    have := Nat.lt_succ_of_le <| Stmt.numExts_S (Δ := []) s
    have ih := (D_eq (ρ := ·) (S s))
    aesop (add safe cases Neut, norm simp eval_S)
  | .if e s => by simp; split <;> simp [D_eq s]
  | .ret e => nomatch e ρ ∅
  | .for e s => by
    -- for termination
    have := Nat.lt_succ_of_le <| Stmt.numExts_C_B (Δ := []) s
    have ih := (D_eq (ρ := ·) (C (B s)))
    simp
    induction e ρ ∅ <;>
      aesop (add safe cases Neut, norm unfold runCatch,
             norm simp [eval_C, eval_B])
```

```
termination_by _ s => (s.numExts, sizeOf s)
decreasing_by D_tac
```

The equivalence proof follows from the invariants of D and R.

```
theorem Do.trans_eq_eval : ∀ s : Do m α, Do.trans s = ⟦s⟧ := by
  aesop (add norm simp [D_eq, eval_R],
         norm unfold [runCatch, Do.trans, Do.eval])
```

B.6 Partial Evaluation

We define a new term notation `simp [...] in e` that rewrites the term e using the given simplification theorems. This is an example of a simple term elaborator.

```
open Lean in
open Lean.Parser.Tactic in
open Lean.Meta in
open Lean.Elab in
elab "simp" "[" simps:simpLemma,* "]" "in" e:term : term => do
  -- construct goal `⊢ e = ?x` with fresh metavariable `?x`,
  -- simplify, solve by reflexivity, and return assigned value of `?x`
  let e ← Term.elabTermAndSynthesize e none
  let x ← mkFreshExprMVar (← inferType e)
  let goal ← mkFreshExprMVar (← mkEq e x)
  -- disable ζ-reduction to preserve `let`s
  Term.runTactic goal.mvarId! (← `(tactic|
    (simp (config := { zeta := false }) [$simps:simpLemma,*]
     rfl)))
  instantiateMVars x

-- further clean up generated programs
attribute [local simp] Assg.extendBot cast
attribute [-simp] StateT.run'_eq
```

We can now use this new notation to completely erase the translation functions from an invocation on the example `ex2` from `For.lean` (manually translated to `Stmt`).

```
/-
```

```
let mut y := init;
for x in xs do' {
  y ← f y x
};
return y
-/
def ex2' (f : β → α → m β) (init : β) (xs : List α) : Do m β :=
  .letmut (fun _ _ => init) <|
    .bind (
      .for (fun _ _ => xs) <|
      -- `y ← f y x` unfolded to `let z ← f y x; y := z` (A4)
      .bind
        (.expr (fun ([x]) ([y]) => f y x))
        (.assg ⟨0, by simp⟩ (fun ([z, x]) _ => z))) <|
    .ret (fun _ ([y]) => y)

def ex2'' (f : β → α → m β) (init : β) (xs : List α) : m β :=
  simp [Do.trans] in Do.trans (ex2' f init xs)
```

Compare the output of the two versions - the structure is identical except for unused monadic layers in the formal translation, which would be harder to avoid in this typed approach.

```
#print ex2
/-
def ex2 : ... → (β → α → m β) → β → List α → m β :=
fun {m} {α} [Monad m] {β} f init xs =>
  ExceptCpsT.runCatch
    (let y := init;
    StateT.run'
      (do
        forM xs fun x => do
            let y ← get
            let y ← StateT.lift (ExceptCpsT.lift (f y x))
            let _ ← get
            set y
        let y ← get
        StateT.lift (throw y))
      y)
-/
#print ex2''
```

```
/-
def ex2'' : ... → (β → α → m β) → β → List α → m β :=
fun {m} [Monad m] {β α} f init xs =>
  runCatch
    (let x := init;
    StateT.run'
      (do
        runCatch
            (forM xs fun x =>
              runCatch do
                let x_1 ← ExceptT.lift (ExceptT.lift get)
                let x ← ExceptT.lift
                  (ExceptT.lift (StateT.lift (ExceptT.lift (f x_1 x))))
                let _ ← ExceptT.lift (ExceptT.lift get)
                ExceptT.lift (ExceptT.lift (set x)))
        let x ← get
        StateT.lift (throw x))
      x)
-/
```

We can evaluate the generated program like any other Lean program, and prove that both versions are equivalent.

```
#eval ex2'' (m := Id) (fun a b => pure (a + b)) 0 [1, 2]

example (f : β → α → m β) :
    ex2'' f init xs = ex2 f init xs := by
  rw [ex2, ex2'']
  unfold runCatch
  induction xs generalizing init <;> simp_all! [StateT.run'_eq]
```

For one more example, consider `ex3` from `For.lean`.

```
/-
for xs in xss do' {
  for x in xs do' {
    let b ← p x;
    if b then {
      return some x
    }
  }
};
```

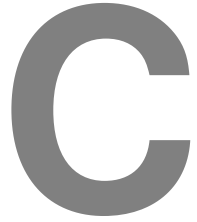
```
pure none
-/

def ex3' [Monad m] (p : α → m Bool) (xss : List (List α)) :
    m (Option α) :=
  simp [Do.trans] in Do.trans (
      .bind
        (.for (fun _ _ => xss) <|
          .for (fun ([xs]) _ => xs) <|
            .bind
              (.expr (fun ([x, xs]) _ => p x))
              (.if (fun ([b, x, xs]) _ => b)
                (.ret (fun ([b, x, xs]) _ => some x))))
        (.expr (fun _ _ => pure none)))

#print ex3
#print ex3'
#eval ex3' (m := Id) (fun n => n % 2 == 0) [[1, 3], [2, 4]]

example (p : α → m Bool) (xss : List (List α)) :
    ex3' p xss = ex3 p xss := by
  rw [ex3, ex3']
  unfold runCatch
  induction xss with
  | nil => simp!
  | cons xs xss ih =>
    simp
    induction xs <;> aesop (add safe apply byCases_Bool_bind)
```

While it would be possible to override our `do'` notation such that its named syntax is first translated to nameless `Stmt` constructors and then applied to `simp [Do.trans] in`, for demonstration purposes I decided to encode these examples manually. In practice, the macro implementation remains more desirable as mentioned in Section 5.6.



Formal Reference Counting Semantics & Proof of Correctness

This appendix contains the correctness proof of the compiler presented in Section 6.4. Parts of it have been formalized by [Huisinga, 2019] in Lean 3 under my supervision.

The proof is split into the following parts:

- a specification of the semantics of λ_{pure}
- a well-formedness predicate on pure programs that we assume holds for the compiler inputs
- well-formedness predicates on reset/reuse and borrow inference steps that abstract from the specific implementations
- a type system for λ_{RC}
- a proof that type-correct programs are semantics-preserving in that pure and RC semantics coincide
- a proof that the presented compiler produces type-correct output, and therefore preserves semantics

C.1 Pure Semantics

The following rules present a natural semantics of λ_{pure}, which correspond to the λ_{RC} semantics given in Section 6.3 after removing reference counts

and locations, resulting in a heap-less presentation where values directly reference other values and are never explicitly freed.

$$\begin{aligned} \rho \in \mathit{Ctxt} &= \mathit{Var} \rightharpoonup \mathit{Value} \\ v \in \mathit{Value} &::= \mathtt{ctor}_i\ \overline{v} \mid \mathtt{pap}\ c\ \overline{v} \end{aligned}$$

Const-App-Full
$$\frac{\delta(c) = \lambda\ \overline{y_c}.\ F \qquad \overline{v} = \overline{\rho(y)} \qquad [\overline{y_c} \mapsto \overline{v}] \vdash F \Downarrow v'}{\rho \vdash c\ \overline{y} \Downarrow v'}$$

Const-App-Part
$$\frac{\delta(c) = \lambda\ \overline{y_c}.\ F \qquad \overline{v} = \overline{\rho(y)} \qquad |\ \overline{v}\ | < |\ \overline{y_c}\ |}{\rho \vdash \mathtt{pap}\ c\ \overline{y} \Downarrow \mathtt{pap}\ c\ \overline{v}}$$

Var-App-Full
$$\frac{\rho(x) = \mathtt{pap}\ c\ \overline{v} \qquad \delta(c) = \lambda\ \overline{y_c}.\ F \qquad v' = \rho(y) \qquad [\overline{y_c} \mapsto \overline{v}\ v'] \vdash F \Downarrow v''}{\rho \vdash x\ y \Downarrow v''}$$

Var-App-Part
$$\frac{\rho(x) = \mathtt{pap}\ c\ \overline{v} \qquad \delta(c) = \lambda\ \overline{y_c}.\ F \qquad v' = \rho(y) \qquad |\ \overline{v}\ v'\ | < |\ \overline{y_c}\ |}{\rho \vdash x\ y \Downarrow \mathtt{pap}\ c\ \overline{v}\ v'}$$

Ctor-App
$$\frac{\overline{v} = \overline{\rho(y)}}{\rho \vdash \mathtt{ctor}_i\ \overline{y} \Downarrow \mathtt{ctor}_i\ \overline{v}}$$

Proj
$$\frac{\rho(x) = \mathtt{ctor}_j\ \overline{v} \qquad v' = v_i}{\rho \vdash \mathtt{proj}_i\ x \Downarrow v'}$$

Return
$$\frac{\rho(x) = v}{\rho \vdash \mathtt{ret}\ x \Downarrow v}$$

Let
$$\frac{\rho \vdash e \Downarrow v \qquad \rho[x \mapsto v] \vdash F \Downarrow v'}{\rho \vdash \mathtt{let}\ x = e;\ F \Downarrow v'}$$

Case
$$\frac{\rho(x) = \mathtt{ctor}_i\ \overline{v} \qquad \rho \vdash F_i \Downarrow v'}{\rho \vdash \mathtt{case}\ x\ \mathtt{of}\ \overline{F} \Downarrow v'}$$

We also trivially extend the pure semantics to λ_{RC} in order to express *semantic refinement* for the compiler passes that only change the RC semantics of a program.

$$\frac{\rho \vdash F \Downarrow v}{\rho \vdash \texttt{inc}\ x;\ F \Downarrow v}\ \textsc{Inc} \qquad \frac{\rho \vdash F \Downarrow v}{\rho \vdash \texttt{dec}\ x;\ F \Downarrow v}\ \textsc{Dec}$$

$$\frac{}{\rho \vdash \texttt{reset}\ x \Downarrow v}\ \textsc{Reset} \qquad \frac{\rho \vdash \texttt{ctor}_i\ \overline{y} \Downarrow v}{\rho \vdash \texttt{reuse}\ x\ \texttt{in}\ \texttt{ctor}_i\ \overline{y} \Downarrow v}\ \textsc{Reuse}$$

Definition 1. *We say δ_B* refines *δ_A in the pure semantics if for each constant c with $\delta_A(c) = \lambda\ \overline{y}.\ F$, we have $\delta_B(c) = \lambda\ \overline{y'}.\ F'$ and*

$$[\overline{y} \mapsto \overline{v}] \vdash F \Downarrow v' \iff [\overline{y'} \mapsto \overline{v}] \vdash F' \Downarrow v' \textit{ for all } \overline{v}, v'$$

Note that there are no assertions about constants in δ_B but not in δ_A (e.g. helper definitions introduced by compiler passes).

C.2 Well-Formedness

We will make the following reasonable assumptions about inputs of the compiler, formally described as a predicate $\vdash_{pure} \delta$:

- used variables are first defined (*definite assignment*)
- defined variables are used at least once (the input is *reduced*)
- full applications are of the correct arity
- bindings are fresh

Definition 2 (well-formedness of pure programs)**.**

$$\frac{\forall c \in \mathrm{dom}(\delta).\ \delta \vdash_{pure} c}{\vdash_{pure} \delta} \qquad \frac{\delta(c) = \lambda\,\overline{y}.\,F \qquad \overline{y} \vdash_{pure} F}{\delta \vdash_{pure} c}$$

$$\frac{}{\Gamma, x \vdash_{pure} \texttt{ret}\ x} \qquad \frac{\overline{\Gamma, x \vdash_{pure} F}}{\Gamma, x \vdash_{pure} \texttt{case}\ x\ \texttt{of}\ \overline{F}}$$

$$\frac{\Gamma \vdash_{pure} e \qquad z \in FV(F) \qquad z \notin \Gamma \qquad \Gamma, z \vdash_{pure} F}{\Gamma \vdash_{pure} \texttt{let}\ z = e;\ F}$$

$$\frac{\delta(c) = \lambda\,\overline{y_c}.\,F' \qquad |\,\overline{y}\,| = |\,\overline{y_c}\,|}{\Gamma, \overline{y} \vdash_{pure} c\ \overline{y}} \qquad \frac{}{\Gamma, \overline{y} \vdash_{pure} \texttt{pap}\ c\ \overline{y}} \qquad \frac{}{\Gamma, x, y \vdash_{pure} x\ y}$$

$$\frac{}{\Gamma, \overline{y} \vdash_{pure} \texttt{ctor}_i\ \overline{y}} \qquad \frac{}{\Gamma, x \vdash_{pure} \texttt{proj}_i\ x}$$

The free-variables function FV is defined as usual.

We will implicitly assume $\vdash_{pure} \delta$ for the pure input program δ in the following.

Theorem 1. *If* $\Gamma \vdash_{pure} F$*, then* $\mathrm{FV}(F) \subseteq \Gamma$.

Proof. By induction over F. Any free variables in a subterm are covered by a corresponding context extension, and the only point where the context is reduced is where that variable is bound and thus removed from FV. □

C.3 reset/reuse

Variables introduced by `reset` form a separate, *affine* context Δ, i.e. they may not be duplicated. They may be consumed by `reuse`. `dec` instructions introduced later will turn the context *linear*, as postulated by the full type system below.

Definition 3 (Well-formedness including reset/reuse instructions).

$$\frac{\forall c \in \mathrm{dom}(\delta).\ \delta \vdash_{reuse} c}{\vdash_{reuse} \delta} \qquad \frac{\delta(c) = \lambda\ \overline{y}.\ F \qquad \overline{y}; \cdot \vdash_{reuse} F}{\delta \vdash_{reuse} c}$$

$$\frac{}{\Gamma, x; \Delta \vdash_{reuse} \texttt{ret}\ x} \qquad \frac{\overline{\Gamma, x; \Delta \vdash_{reuse} F}}{\Gamma, x; \Delta \vdash_{reuse} \texttt{case}\ x\ \texttt{of}\ \overline{F}}$$

$$\frac{\Gamma \vdash_{pure} e \qquad z \in FV(F) \qquad z \notin \Gamma \cup \Delta \qquad \Gamma, z; \Delta \vdash_{reuse} F}{\Gamma; \Delta \vdash_{reuse} \texttt{let}\ z = e;\ F}$$

$$\frac{z \notin \Gamma \cup \Delta \qquad \Gamma; \Delta, z \vdash_{reuse} F}{\Gamma; \Delta \vdash_{reuse} \texttt{let}\ z = \texttt{reset}\ e;\ F}$$

$$\frac{\Gamma; \Delta \vdash_{reuse} F}{\Gamma; \Delta, x \vdash_{reuse} \texttt{let}\ z = \texttt{reuse}\ x\ \texttt{in}\ \texttt{ctor}_i\ \overline{y};\ F}$$

Theorem 2. *For the specific δ_{reuse} described in Chapter 6, we have $\vdash_{reuse} \delta_{reuse}$. Moreover, δ_{reuse} refines δ in the pure semantics.*

Proof. The first statement follows from the definition δ_{reuse} and the assumption $\vdash_{pure} \delta$. The second statement follows directly from the definition of δ_{reuse} and of the pure semantics of reset and reuse. □

Theorem 3. *If $\Gamma; \Delta \vdash_{reuse} F$, then $\mathrm{FV}(F) \subseteq \Gamma \cup \Delta$.*

Proof. As before. □

C.4 Borrow Inference

The borrow heuristic function β (Section 6.4.2) does not immediately affect the program, except that we introduce an *owned* wrapper for every function with a borrowed parameter, which may not directly be applied partially.

Definition 4 (Program after borrow inference). *We assume that for every constant $c \in \delta_{reuse}$ there exists an unused constant name $c_{\mathbb{O}}$. The program after borrow inference δ_β is defined as the extension*

$$\delta_\beta = \delta'_{reuse}[c_{\mathbb{O}} \mapsto \lambda\ \overline{y}.\ c\ \overline{y} \mid \mathbb{B} \in \beta(c), \delta_{reuse}(c) = \lambda\ \overline{y}.\ F]$$

where δ'_{reuse} is obtained from δ_{reuse} by replacing every occurrence of pap *$c\ \overline{y}$ where $\mathbb{B} \in \beta(c)$ by* pap *$c_{\mathbb{O}}\ \overline{y}$.*

Definition 5 (Well-formedness of borrow inference). *β is arity-correct. Partially applied constants do not have borrowed parameters.*

$$\frac{\vdash_{reuse} \delta \qquad \delta \vdash_\beta c \quad \forall c \in \mathrm{dom}(\delta)}{\vdash_\beta \delta} \qquad \frac{\delta(c) = \lambda\, \overline{y}.\, F \qquad |\, \overline{y}\, | = |\, \beta(c)\, | \qquad \vdash_\beta F}{\delta \vdash_\beta c}$$

$$\frac{\mathbb{B} \notin \beta(c) \qquad \vdash_\beta F}{\vdash_\beta \texttt{let}\ x = \texttt{pap}\ c\ \overline{y};\ F} \qquad \frac{}{\vdash_\beta \texttt{ret}\ x}$$

The rules for all other instructions proceed by direct induction on F or $\overline{F}$.

Theorem 4. *For δ_β from Definition 4 and any arity-correct β, we have $\vdash_\beta \delta_\beta$. Moreover, δ_β refines δ_{reuse} in the pure semantics.*

Proof. By definition of δ_β. □

C.5 A Type System for RC-Correct Programs

A program's behavior should not be changed by compiling it to (or optimizing it in) λ_{RC}. Before designing the compiler, it is helpful to capture the global, dynamic invariants necessary for this in a static type system that reasons about just the local context of a function.

Intuitively, a program is RC-correct if

1. owned variables are *locally count-correct*: every owned variable is ultimately consumed or `dec`ed on each control flow path, with every `inc` allowing and necessitating one more consumption,
2. borrowed parameters are not `dec`ed, as conceptually they do not carry their own RC token, and
3. values from `reset` are handled by exactly one `reuse` or `dec` on each control flow path, and are not used in any other context.

The type system formalizing these constraints is quite simple: since types have been erased from λ_{pure}, the only types are $\mathbb{O}$, $\mathbb{B}$, and $\mathbb{R}$ for owned, borrowed, and reset references, respectively.

$$\tau \in Ty ::= \mathbb{O} \mid \mathbb{B} \mid \mathbb{R}$$

The type system is *linear* to represent conditions (1) and (3); for *borrowed* references, we add the usual weakening and contraction rules from intuitionistic linear logic [Benton et al., 1993] to model their non-linear, or *intuitionistic*, semantics.

$$\textsc{Ty-Var}\ \frac{}{x : \tau \vdash_{RC} x : \tau} \qquad \textsc{Ty-Weaken}\ \frac{\Gamma \vdash_{RC} e : \tau}{\Gamma, x : \mathbb{B} \vdash_{RC} e : \tau}$$

$$\textsc{Ty-Contract}\ \frac{\Gamma, x : \mathbb{B}, x : \mathbb{B} \vdash_{RC} e : \tau}{\Gamma, x : \mathbb{B} \vdash_{RC} e : \tau} \qquad \textsc{Ty-Contract-F}\ \frac{\Gamma, x : \mathbb{B}, x : \mathbb{B} \vdash_{RC} F}{\Gamma, x : \mathbb{B} \vdash_{RC} F}$$

We define well-typed programs and constants in terms of well-typed function bodies; the return type is elided since it is always $\mathbb{O}$.

$$\frac{\vdash_\beta \delta \qquad \forall c \in \mathrm{dom}(\delta).\ \delta \vdash_{RC} c}{\vdash_{RC} \delta} \qquad \frac{\delta(c) = \lambda\, \overline{y}.\ F \qquad \overline{y : \beta(c)} \vdash_{RC} F}{\delta \vdash_{RC} c}$$

`inc` introduces a new owned reference from a borrowed or owned reference, `dec` consumes an owned or reset reference.

$$\textsc{Ty-Inc}\ \frac{\tau \in \{\mathbb{O}, \mathbb{B}\} \qquad \Gamma, x : \tau, x : \mathbb{O} \vdash_{RC} F}{\Gamma, x : \tau \vdash_{RC} \texttt{inc}\ x;\ F} \qquad \textsc{Ty-Dec}\ \frac{\tau \in \{\mathbb{O}, \mathbb{R}\} \qquad \Gamma \vdash_{RC} F}{\Gamma, x : \tau \vdash_{RC} \texttt{dec}\ x;\ F}$$

Note that the first rule can introduce the same variable with two different types. It may help to view this type system as a *capability* system: A hypothesis of type $\mathbb{O}$ or $\mathbb{R}$ grants (exactly) one consuming usage, while one of type $\mathbb{B}$ grants only non-consuming usage.

Return values must be owned, while non-consuming, immediate uses like in `case` can be owned or borrowed.

$$\textsc{Ty-Return}\ \frac{\Gamma \vdash_{RC} x : \mathbb{O}}{\Gamma \vdash_{RC} \texttt{ret}\ x} \qquad \textsc{Ty-Case}\ \frac{\tau \in \{\mathbb{O}, \mathbb{B}\} \qquad \overline{\Gamma, x : \tau \vdash_{RC} F}}{\Gamma, x : \tau \vdash_{RC} \texttt{case}\ x\ \texttt{of}\ \overline{F}}$$

Applications are typed by splitting up the linear context: in e.g. the conclusion $\overline{\Gamma} \vdash_{RC} c\ \overline{y} : \mathbb{O}$ below, Γ must be split in as many parts as y, each pair of which must then fulfill $\Gamma \vdash_{RC} y : \beta(c)$. Thus, Γ must contain exactly the owned variables passed to owned parameters, at least the borrowed variables passed to borrowed parameters (leftover borrowed variables can be weakened away), and no reset references (by definition of β in Section 6.4.2). Arguments to partial, variable and constructor applications must be owned because, in general, we cannot statically assert that the resulting value will not escape the current function and thus the scope of borrowed references.

$$\textsc{Ty-Const-App-Full}\quad \frac{\overline{\Gamma \vdash_{RC} y : \beta(c)}}{\overline{\Gamma} \vdash_{RC} c\ \overline{y} : \mathbb{O}} \qquad \textsc{Ty-Const-App-Part}\quad \frac{\beta(c) = \overline{\mathbb{O}}}{\overline{y : \mathbb{O}} \vdash_{RC} \texttt{pap}\ c\ \overline{y} : \mathbb{O}}$$

$$\textsc{Ty-Var-App}\quad \frac{}{x : \mathbb{O}, y : \mathbb{O} \vdash_{RC} x\ y : \mathbb{O}} \qquad \textsc{Ty-Cnstr-App}\quad \frac{}{\overline{y : \mathbb{O}} \vdash_{RC} \texttt{ctor}_i\ \overline{y} : \mathbb{O}}$$

$$\textsc{Ty-Reset}\quad \frac{}{x : \mathbb{O} \vdash_{RC} \texttt{reset}\ x : \mathbb{R}} \qquad \textsc{Ty-Reuse}\quad \frac{}{x : \mathbb{R}, \overline{y : \mathbb{O}} \vdash_{RC} \texttt{reuse}\ x\ \texttt{in}\ \texttt{ctor}_i\ \overline{y} : \mathbb{O}}$$

In order to type (saturated) applications with borrowed parameters, the rule for `let` should support temporarily obtaining a borrowed reference from an owned reference, much like [Wadler, 1990b]'s `let!` construct. The rule makes the owned reference unavailable during the call to ensure that the borrowed reference is valid for that duration. The result type of e ensures that the borrow cannot survive past the call.

$$\textsc{Ty-Let}\quad \frac{\Gamma_1, \overline{x : \mathbb{B}} \vdash_{RC} e : \tau \qquad \tau \in \{\mathbb{O}, \mathbb{R}\} \qquad \Gamma_2, \overline{x : \mathbb{O}}, z : \tau \vdash_{RC} F}{\Gamma_1, \Gamma_2, \overline{x : \mathbb{O}} \vdash_{RC} \texttt{let}\ z = e;\ F}$$

Projections are handled specially. It is sound to treat the projection of a borrowed reference as borrowed because the full object graph reachable from the borrowed references is assumed to be valid for the entire

function call. On the other hand, when projecting an owned reference, we conservatively treat the result as owned as well by requiring that it is incremented immediately; a more flexible model would need a more sophisticated "borrow checker" that makes sure that the projection cannot outlive the projected reference.

$$\frac{\Gamma, x : \mathbb{B}, y : \mathbb{B} \vdash_{RC} F}{\Gamma, x : \mathbb{B} \vdash_{RC} \texttt{let}\ y = \texttt{proj}_i\ x;\ F}\ \textsc{Ty-Proj-Bor} \qquad \frac{\Gamma, x : \mathbb{O}, y : \mathbb{O} \vdash_{RC} F}{\Gamma, x : \mathbb{O} \vdash_{RC} \texttt{let}\ y = \texttt{proj}_i\ x;\ \texttt{inc}\ y;\ F}\ \textsc{Ty-Proj-Own}$$

Definition 6. *The function* valof : $Loc \times Heap \rightharpoonup Value_{pure}$ *projecting a reference into a heap back to a pure value is defined as follows:*

$$\begin{aligned} \mathrm{valof}(l, \sigma) &= \texttt{ctor}_i\ \overline{\mathrm{valof}(l', \sigma)} && \text{if } \sigma(l) = \texttt{ctor}_i\ \overline{l'} \\ \mathrm{valof}(l, \sigma) &= \texttt{pap}\ c\ \overline{\mathrm{valof}(l', \sigma)} && \text{if } \sigma(l) = \texttt{pap}\ c\ \overline{l'} \end{aligned}$$

The central semantics preservation theorem holds for all closed, well-typed function bodies F, which for simplicity we will assume to be part of the program map δ.

Theorem 5 (semantics preservation)**.** *Suppose the program is well-typed,* $\vdash_{RC} \delta$*, and c is a parameter-less constant,* $\delta(c) = F$.

1. *If* $\vdash F \Downarrow v$*, then* $\vdash \langle F, \emptyset \rangle \Downarrow \langle l, \sigma \rangle$ *and* $\mathrm{valof}(l, \sigma) = v$.
2. *If* $\vdash \langle F, \emptyset \rangle \Downarrow \langle l, \sigma \rangle$*, then* $\vdash F \Downarrow \mathrm{valof}(l, \sigma)$.

Proof. Below. The proof directly follows [Chirimar et al., 1996]'s proof of this theorem for a similar linear type system (I direct interested readers to this paper for proofs of further properties such as freedom of memory leaks). The fundamental idea of inducing a *memory graph* from the heap and local variables and proving that the in-degrees of its nodes is equal to the values' reference counts is directly applicable to our owned references. I will quickly discuss parts of Lean's RC system not present in theirs that needed to be fitted into the proofs:

- *Borrowed references* do not change the reference count and thus the definition of the memory graph does not need to be adjusted. However, the proof needed to be extended with an additional hypothesis that every borrowed reference is reachable from an owned *root* variable in a parent stack frame, which implies that the borrowed reference is valid for the duration of the current function call.
- *Reset references* are restricted by the semantics and type system to be used only in `reuse` and `dec`, but otherwise behave linearly like owned references. Because we replace their former contents with $\bot$ instead of leaving them as dangling pointers, treating reset references like owned references in the memory graph results in the correct behavior without further changes. An additional assumption makes sure that every reset references does in fact have this shape.

□

C.6 Proof of Semantics Preservation

A *memory graph* $\mathcal{G}$ is a tuple $(V, \overline{E}, \overline{l})$ where $(V, \overline{E})$ is a directed multigraph of a set of vertices $V \subseteq Loc$ and a multiset of edges $\overline{E} \subseteq V \times V$, and $\overline{l^r} \subseteq V$ is a *root (multi)set*[1]. The *reference count* of a vertex $v \in V$ is the sum of inner and outer references

$$\text{in-degree}(v) + |\{i \mid v = l^r_i\}|$$

A *state* is a pair $(\overline{l^r}, \sigma)$ of a root set $\overline{l^r}$ pointing into a heap σ.

Definition 7. *For a state $S = (\overline{l^r}, \sigma)$, the* memory graph $\mathcal{G}(S)$ induced by S *is a memory graph with* $\text{dom}(\sigma)$ *as its vertices and one edge from $l \in \text{dom}(\sigma)$ to every l' contained in $\sigma(l)$.*

Definition 8. *A state $S = (\overline{l^r}, \sigma)$ is* count-correct *if, for each $\sigma(l) = (v, i)$, the reference count of l in $\mathcal{G}(S)$ is i.*

Definition 9. *A state $S = (\overline{l^r}, \sigma)$ is called* regular, *written $\mathfrak{R}(S)$, provided the following conditions hold:*

[1] We will operate on such lists up to permutations without further mention

$\Re$1 *S is count-correct.*

$\Re$2 $\mathrm{dom}(\sigma)$ *is finite.*

$\Re$3 *The reference count is non-zero for every* $l \in \mathrm{dom}(\sigma)$.

Definition 10 (Reference reachability)**.** *A reference l' is reachable from l in the heap σ,* $\mathrm{reach}_\sigma(l, l')$*, if there is a path from l to l' in* $\mathcal{G}(([], \sigma))^2$.

Theorem 6 (Memory Graph Laws)**.**

B $\Re(\bar{l}, \sigma)$ *iff* $\Re(\bar{l}\ \blacksquare, \sigma)$.

D *If* $\Re(\bar{l}\ l', \sigma)$*, then* $\Re(\bar{l}, \mathrm{dec}(l', \sigma))$.

I *If* $\Re(\bar{l}, \sigma)$ *and* $l' \in \mathrm{dom}(\sigma)$*, then* $\Re(\bar{l}\ l', \mathrm{inc}(l', \sigma))$.

Proof.

B As $\blacksquare$ is not a memory graph vertex ($\blacksquare \notin \mathrm{dom}(\sigma)$ by definition of σ), it does not influence reference counts.

D By induction over the total sum of reference counts (which is finite by regularity).

I Trivial.

∎

Theorem 7 (Preservation of regularity)**.** *Suppose*

$$\begin{aligned}
&\vdash_{RC} \delta, \\
&\overline{y_{\mathbb{O}} : \mathbb{O}}, \overline{y_{\mathbb{B}} : \mathbb{B}}, \overline{y_{\mathbb{R}} : \mathbb{R}} \vdash_{RC} F, \\
&\mathrm{dom}(\rho) = \overline{y_{\mathbb{O}}}\ \overline{y_{\mathbb{B}}}\ \overline{y_{\mathbb{R}}}, && \textit{(static and dynamic contexts coincide)} \\
&\Re(\bar{l}\ \rho(\overline{y_{\mathbb{O}}}\ \overline{y_{\mathbb{R}}}), \sigma), \textit{and} && \textit{(non-borrowed variables are roots)} \\
&\forall y \in \overline{y_{\mathbb{B}}}.\ \exists l \in \bar{l}.\ \mathrm{reach}_\sigma(l, \rho(y)). && \textit{(borrowed variables are reachable)}
\end{aligned}$$

[2] The root set is irrelevant for this definition.

If $\rho \vdash \langle F, \sigma \rangle \Downarrow \langle l', \sigma' \rangle$*, then* $\Re(\bar{l}\ l', \sigma')$*, i.e. regularity is preserved with parameter roots replaced by the return value. Moreover, for any* $\overline{l_1} \subseteq \bar{l}$ *and any* $l \in \mathrm{dom}(\sigma)$ *such that* l *is not reachable from* $\overline{l_1}\ \mathrm{cod}(\rho)$ *in* $\mathcal{G}(\bar{l}\ \mathrm{cod}(\rho), \sigma)$*, we have* $\sigma'(l) = \sigma(l)$ *and* l *is not reachable from* $\overline{l_1}\ l'$ *in* $\mathcal{G}(\bar{l}\ l', \sigma')$ *(the* reachability property*).*

Here $\bar{l}$ are roots outside of the current context, i.e. from a parent stack frame, which guarantee that borrowed references are alive. The reachability property says that if a location is not reachable from from some set of locations including the parameters in the pre-state, then its value will be unchanged in the post-state and it will be unreachable from that same set of locations with parameters replaced by the return value.

Proof. By induction on $\rho \vdash \langle F, \sigma \rangle \Downarrow \langle l', \sigma' \rangle$.

Case Let + Ctor-App

By case inversions of the typing assumption, we have $\Gamma_1 = \overline{y : \mathbb{O}}$. Thus there exists $\overline{y'_{\mathbb{O}}}$ such that $\bar{y}\ \overline{y'_{\mathbb{O}}} = \overline{y_{\mathbb{O}}}$. For the IH we need to show $\Re(\bar{l}\ \rho(\overline{y'_{\mathbb{O}}}\ \overline{y_{\mathbb{R}}})\ l', \sigma[l' \mapsto (\mathtt{ctor}_i\ \rho(\bar{y}), 1)])$ and the reachability property. l' is fresh, so its in-degree is indeed 0. All $\rho(\bar{y})$ have been moved from roots into l', so they are count-correct as well. The reachability property holds because the heap is only extended, not modified, and all locations reachable from l' have already been reachable from $\rho(\overline{y_{\mathbb{O}}})$.

Case Let + Const-App-Part/Var-App-Part

Analogously.

Case Let + Const-App-Full

We have $F = \mathtt{let}\ y = c\ \overline{y'};\ F'$, $\delta(c) = \lambda\ \overline{y_c}.\ F_c$. We first apply the IH to F_c: by the typing assumption, it is well-typed and the types of arguments correspond to their respective parameter types, so owned arguments are rooted and borrowed variables are reachable (either from $\bar{l}$ by the reachability assumption, or from some $y_{\mathbb{O}}$ not passed to c but temporarily borrowed in Ty-Let). We obtain that the new state is regular and fulfills the reachability property after removing all owned variables passed to c from and adding l' to the root set. Thus we can apply the IH to F'.

To show the reachability property, assume $l \in \text{dom}(\sigma)$ is not reachable from $\overline{l_1}$ cod(ρ) where $\overline{l_1} \subseteq \bar{l}$. Then by the first IH, it is unreachable from $\overline{l_1}$ l' in the new state and $\sigma'(l) = \sigma(l)$. Thus we can conclude by the second IH. Like [Chirimar et al., 1996], I will omit further similar proofs of the reachability property.

Case Let + Var-App-Full
Similarly, but we need additional steps to argue for the regularity of the state passed to F_c: $\rho(x)$ is a root by the inductive assumptions, but is not passed to either of F_c or F'. *dec* thus correctly removes it from the root set by law **D**. Conversely, the $\overline{l'}$ in $\sigma(\rho(x))$ are passed as owned arguments not taken from existing roots (but reachable from the root $\rho(x)$, i.e. in dom(σ)), so *inc* correctly adds them to the root set by **I**.

Case Let + Proj
We have $F = \texttt{let}\ y = \texttt{proj}_i\ x;\ F'$. We continue by case analysis of the typing assumption.

Case Ty-Proj-Bor
We have $x, y : \mathbb{B}$, so $x \in \overline{y_{\mathbb{B}}}$. Thus x is reachable by assumption, and so is y as it is contained in $\sigma(\rho(x))$. Therefore we can apply the IH.

Case Ty-Proj-Own
We have $F' = \texttt{inc}\ y;\ F''$ and $x, y : \mathbb{O}$, so $x \in \overline{y_{\mathbb{O}}}$. Because y is both registered as a new root and incremented, we can apply the IH to F''.

Case Return
By the typing assumption, x is the only owned variable left. Thus we can directly apply the regularity assumption.

Case Case
No changes to the context or heap, so the IH applies immediately.

Case Inc
By the typing assumption, we have $\tau \in \{\mathbb{O}, \mathbb{B}\}$. In either case, $\rho(x)$ is equal to or reachable from a root and thus $\rho(x) \in \text{dom}(\sigma)$, so we are done by law **I** and the IH.

Case Dec
: x is a root by the typing assumption, so we are done by law **D** and the IH.

Case Let + Reset-Uniq
: It is easy to see that the new heap is count-correct. While x changes its type, it remains a root, so the state is regular and we can apply the IH.

Case Let + Reset-Shared
: By laws **B** and **D** and the IH.

Case Let + Reuse-Uniq
: Similarly to Ctor-App.

Case Let + Reuse-Shared
: By the IH.

□

Lemma 1. *Suppose (as above)*

$$\begin{aligned}&\vdash_{RC} \delta,\ \overline{y_{\mathbb{O}} : \mathbb{O}}\ \overline{y_{\mathbb{B}} : \mathbb{B}}\ \overline{y_{\mathbb{R}} : \mathbb{R}} \vdash_{RC} F,\\ &\mathrm{dom}(\rho) = \overline{y_{\mathbb{O}}}\ \overline{y_{\mathbb{B}}}\ \overline{y_{\mathbb{R}}},\ \ \Re(\bar{l}\ \rho(\overline{y_{\mathbb{O}}}\ \overline{y_{\mathbb{R}}}), \sigma),\\ &\forall y \in \overline{y_{\mathbb{B}}}.\ \exists l \in \bar{l}.\ \mathrm{reach}_{\sigma}(l, \rho(y)),\end{aligned}$$

and

$$\forall y \in \overline{y_{\mathbb{R}}}.\ \rho(y) = \blacksquare \vee \exists i, n, r.\ \sigma(\rho(y)) = (\mathtt{ctor}_i\ \blacksquare^n, r)$$

(reset variables are reset).

1. *If* $\mathrm{valof}(\rho(\overline{y_{\mathbb{O}}}\ \overline{y_{\mathbb{B}}}), \sigma) \vdash F \Downarrow v$*, then* $\rho \vdash \langle F, \sigma \rangle \Downarrow \langle l, \sigma' \rangle$ *and* $v = \mathrm{valof}(l, \sigma')$.
2. *If If* $\rho \vdash \langle F, \sigma \rangle \Downarrow \langle l, \sigma' \rangle$*, then* $\mathrm{valof}(\rho(\overline{y_{\mathbb{O}}}\ \overline{y_{\mathbb{B}}}), \sigma) \vdash F \Downarrow \mathrm{valof}(l, \sigma')$.

Proof. The first part is proved by induction over $\mathrm{valof}(\rho(\overline{y_{\mathbb{O}}}\ \overline{y_{\mathbb{B}}}), \sigma) \vdash F \Downarrow v$.

Case Let + Proj, Ty-Proj-Own
: We have $F' = \mathtt{inc}\ y;\ F''$. Note that F' and F'' are equivalent in the pure semantics, so we can apply the IH to F'' as well, after noticing that $\mathrm{inc}(\cdot,\cdot)$ does not affect $\mathrm{valof}(\cdot,\cdot)$.

Case LET + PROJ, TY-PROJ-BOR
By the IH.

Case LET + RESET-UNIQ/RESET-SHARED
In either case, the assumption about reset variables is fulfilled and we can apply the IH. Note that the pure context is unchanged.

Case LET + REUSE-UNIQ/REUSE-SHARED
By the assumption about reset references and the IH.

Case LET + CONST-APP-FULL
We have $\delta(c) = \lambda\ \overline{y_c}.\ F_c$. In order to apply the IH on F', we need to show that the value of all remaining context variables has not been changed by the call, which follows from Theorem 7's reachability property.

Case LET + CONST-APP-PART
There exists an $l' \notin \mathrm{dom}(\sigma)$ by $\mathfrak{R}2$. Apply the IH.

Case LET + CTOR-APP/VAR-APP-FULL/VAR-APP-PART
Analogously.

Case RETURN
Trivial.

Case CASE
Directly by the IH.

Case INC
We have $\mathrm{valof}(\rho(\overline{y_{\mathbb{O}}}\ \overline{y_{\mathbb{B}}}\ x), \mathrm{inc}(\rho(x), \sigma)) = \mathrm{valof}(\rho(\overline{y_{\mathbb{O}}}\ \overline{y_{\mathbb{B}}}\ x), \sigma)$ by the definition of valof, so we can apply the IH.

Case DEC
Let $\Gamma, x : \tau$ be the context. By $\mathfrak{R}1$, we have that all values reachable from Γ still have reference count ≥ 1 in $\sigma' := \mathrm{dec}(\rho(x), \sigma)$, so $\mathrm{valof}(\Gamma, \sigma') = \mathrm{valof}(\Gamma, \sigma)$ by the definition of dec, and we can apply the IH.

The reverse direction is proved by induction over $\rho \vdash \langle F, \sigma \rangle \Downarrow \langle l, \sigma' \rangle$ with similar but simpler cases. □

Theorem 5 (semantics preservation)**.** *Suppose the program is well-typed, $\vdash_{RC} \delta$, and c is a parameter-less constant, $\delta(c) = F$.*

1. *If* $\vdash F \Downarrow v$*, then* $\vdash \langle F, \emptyset \rangle \Downarrow \langle l, \sigma \rangle$ *and* $\mathrm{valof}(l, \sigma) = v$.
2. *If* $\vdash \langle F, \emptyset \rangle \Downarrow \langle l, \sigma \rangle$*, then* $\vdash F \Downarrow \mathrm{valof}(l, \sigma)$.

Proof. A direct corollary of Lemma 1. □

C.7 Proof of Compilation Well-Typedness

Theorem 8. *For the specific δ_{RC} given in the paper, we have $\vdash_{RC} \delta_{RC}$.*

Proof. We start with a helper lemma about C.

Lemma 2. *C does not introduce new variables,* $\mathrm{FV}(C(F)) = \mathrm{FV}(F)$.

Proof. By induction over F. □

By the definition of δ_{RC}, we need to show for each $c \in \mathrm{dom}(\delta_\beta)$ that

$$\overline{y : \beta(c)} \vdash_{RC} \mathbb{O}^-(\overline{y}, C(F)) \text{ with } \beta_l := [\overline{y} \mapsto \beta(c), \ldots \mapsto \mathbb{O}]$$

where $\delta_\beta(c) =: \lambda\ \overline{y}.\ F$.

Aside: In this proof, I will style β_l as an implicit variable *to reduce verbosity.*

By the wellformedness of δ_β (Theorem 4), we have $\vdash_\beta F$ and $\overline{y}; \cdot \vdash_{reuse} F$.

Note that all $y_i : \mathbb{O}$ are alive in $F' := \mathbb{O}^-(\overline{y}, C(F))$: If they do not occur in $C(F)$, they will occur in a dec instruction from $\mathbb{O}^-$ instead. Thus we can generalize the goal to

$$\frac{\overline{y_\mathbb{O}}\ \overline{y_\mathbb{R}} \subseteq FV(F') \qquad \overline{y_\mathbb{O}}\ \overline{y_\mathbb{B}}; \overline{y_\mathbb{R}} \vdash_{reuse} F \qquad \vdash_\beta F \qquad \overline{y_\mathbb{O}}\ \overline{y_\mathbb{B}}\ \overline{y_\mathbb{R}} \text{ distinct}}{\overline{y_\mathbb{O} : \mathbb{O}}, \overline{y_\mathbb{B} : \mathbb{B}}, \overline{y_\mathbb{R} : \mathbb{R}} \vdash_{RC} F' \text{ with } \beta_l := [\overline{y_\mathbb{B}} \mapsto \overline{\mathbb{B}}, \ldots \mapsto \mathbb{O}]}$$

where $\overline{y_\mathbb{R}} := []$ and $\overline{y_\mathbb{O}}\ \overline{y_\mathbb{B}} := \overline{y}$.

Unfolding $\mathbb{O}^-$ and applying the dec typing rule repeatedly, we remove all $y_\mathbb{O} \notin \mathrm{FV}(C(F))$ and, reusing the name $\overline{y_\mathbb{O}}$ for the reduced variable list and applying Lemma 2, are left with

$$\frac{\overline{y_\mathbb{O}}\ \overline{y_\mathbb{R}} \subseteq FV(F) \qquad \overline{y_\mathbb{O}}\ \overline{y_\mathbb{B}}; \overline{y_\mathbb{R}} \vdash_{reuse} F \qquad \vdash_\beta F \qquad \overline{y_\mathbb{O}}\ \overline{y_\mathbb{B}}\ \overline{y_\mathbb{R}} \text{ distinct}}{\overline{y_\mathbb{O} : \mathbb{O}}, \overline{y_\mathbb{B} : \mathbb{B}}, \overline{y_\mathbb{R} : \mathbb{R}} \vdash_{RC} C(F) \text{ with } \beta_l := [\overline{y_\mathbb{B}} \mapsto \overline{\mathbb{B}}, \ldots \mapsto \mathbb{O}]}$$

We proceed by induction over F, but not before noting a peculiarity about this induction hypothesis: not only does the type context contain only owned variables that are alive in the remaining body (as one may expect), it also contains each of them no more than once. It turns out that duplicating a reference is only necessary just before applications, which will happen in between the induction steps of the proof. In this sense, we see that the compiler keeps all reference counts as low as possible.

Case $F = \texttt{ret}\ x$

We need to show

$$\overline{y_{\mathbb{O}} : \mathbb{O}}, \overline{y_{\mathbb{B}} : \mathbb{B}}, \overline{y_{\mathbb{R}} : \mathbb{R}} \vdash_{RC} \mathbb{O}^+_x(\texttt{ret}\ x)$$

By the first two inductive assumptions, we have $\overline{y_{\mathbb{O}}}\ \overline{y_{\mathbb{R}}} \subseteq \{x\}$ and $x \in \overline{y_{\mathbb{O}}}\ \overline{y_{\mathbb{B}}}$, respectively. Together with the fourth assumption, x may appear at most once in either $\overline{y_{\mathbb{O}}}$ or $\overline{y_{\mathbb{B}}}$.

If $\beta_l(x) = \mathbb{B}$, it remains to be shown that

$$x : \mathbb{B}, \overline{y : \mathbb{B}} \vdash_{RC} \texttt{inc}\ x;\ \texttt{ret}\ x$$

Applying the inc rule, we get to the same goal as in the case $\beta_l(x) = \mathbb{O}$:

$$x : \mathbb{O}, \overline{y' : \mathbb{B}} \vdash_{RC} \texttt{ret}\ x$$

which is closed by the ret rule plus weakening.

Case $F = \texttt{case}\ x\ \texttt{of}\ \overline{F'}$

We need to show

$$\overline{y_{\mathbb{O}} : \mathbb{O}}, \overline{y_{\mathbb{B}} : \mathbb{B}}, \overline{y_{\mathbb{R}} : \mathbb{R}} \vdash_{RC} \texttt{case}\ x\ \texttt{of}\ \overline{\mathbb{O}^-(\overline{y'}, C(F'))}$$

where $\{\overline{y'}\} = \mathrm{FV}(\texttt{case}\ x\ \texttt{of}\ \overline{F'})$. By the first two inductive assumptions, we have

$$\overline{y_{\mathbb{O}}}\ \overline{y_{\mathbb{R}}} \subseteq \overline{y'}$$
$$x \in \overline{y_{\mathbb{O}}}\ \overline{y_{\mathbb{B}}},\ \ \overline{\overline{y_{\mathbb{O}}}\ \overline{y_{\mathbb{B}}}; \overline{y_{\mathbb{R}}} \vdash_{reuse} F'}$$

Using $x \in \overline{y_{\mathbb{O}}}\ \overline{y_{\mathbb{B}}}$, we can apply the case typing rule, leaving us with, for each F'_i,

$$\overline{y_{\mathbb{O}} : \mathbb{O}}, \overline{y_{\mathbb{B}} : \mathbb{B}}, \overline{y_{\mathbb{R}} : \mathbb{R}} \vdash_{RC} \mathbb{O}^-(\overline{y'}, C(F'_i))$$

By Theorem 3, we have $\overline{y'} \subseteq \overline{y_{\mathbb{O}}}\ \overline{y_{\mathbb{B}}}\ \overline{y_{\mathbb{R}}}$, and can repeatedly apply the `dec` rule for any $y'_j \notin \mathrm{FV}(C(F'_i)), \beta_l(y'_j) \neq \mathbb{B}$. We are left with

$$\overline{y'_{\mathbb{O}} : \mathbb{O}}, \overline{y_{\mathbb{B}} : \mathbb{B}}, \overline{y'_{\mathbb{R}} : \mathbb{R}} \vdash_{RC} C(F'_i)$$

where $\overline{y'_{\mathbb{O}}} = [y \in \overline{y_{\mathbb{O}}} \mid y \in \mathrm{FV}(C(F'_i))]$ (and analogously for $\overline{y'_{\mathbb{R}}}$) , thus $\overline{y'_{\mathbb{O}}}\ \overline{y'_{\mathbb{R}}} \subseteq \mathrm{FV}(C(F'_i))$, which allows us to close the goal by the inductive hypothesis.

Case $F = \mathtt{let}\ y = \mathtt{proj}_i\ x;\ F'$ **if** $\beta_l(x) = \mathbb{O}$
We need to show

$$\overline{y_{\mathbb{O}} : \mathbb{O}}, \overline{y_{\mathbb{B}} : \mathbb{B}}, \overline{y_{\mathbb{R}} : \mathbb{R}} \vdash_{RC} \mathtt{let}\ y = \mathtt{proj}_i\ x;\ \mathtt{inc}\ y;\ \mathbb{O}^-_x(C(F'))$$

From $\beta_l(x) = \mathbb{O}$, we have $x \notin \overline{y_{\mathbb{B}}}$, so together with $\overline{y_{\mathbb{O}}}\ \overline{y_{\mathbb{B}}}; \overline{y_{\mathbb{R}}} \vdash_{reuse}$ `let` $y =$ `proj`$_i\ x;\ F'$, we have $x \in \overline{y_{\mathbb{O}}}$. Applying Ty-Proj-Own, we are left with

$$\overline{y_{\mathbb{O}} : \mathbb{O}}, y : \mathbb{O}, \overline{y_{\mathbb{B}} : \mathbb{B}}, \overline{y_{\mathbb{R}} : \mathbb{R}} \vdash_{RC} \mathbb{O}^-_x(C(F'))$$

If $x \notin \mathrm{FV}(C(F'))$, we apply Ty-Dec. In either case, we need to show

$$\overline{y_{\mathbb{O}}' : \mathbb{O}}, y : \mathbb{O}, \overline{y_{\mathbb{B}} : \mathbb{B}}, \overline{y_{\mathbb{R}} : \mathbb{R}} \vdash_{RC} C(F')$$

for some $\overline{y_{\mathbb{O}}'} \subseteq \mathrm{FV}(C(F'))$. We also have $y \in \mathrm{FV}(C(F')) = \mathrm{FV}(F')$ from δ_{reuse}, as well as $\overline{y_{\mathbb{R}}} \cap FV(C(F')) = \overline{y_{\mathbb{R}}} \cap FV(C(F))$, so we can apply the induction hypothesis.

Case $F = \mathtt{let}\ y = \mathtt{proj}_i\ x;\ F'$ **if** $\beta_l(x) = \mathbb{B}$
By Ty-Proj-Bor and the induction hypothesis.

Case $F = \mathtt{let}\ y = \mathtt{reset}\ x;\ F'$
By Ty-Let, Ty-Reset, and the induction hypothesis.

Case $F = \mathtt{let}\ z = c\ \overline{y};\ F'$
We need to show

$$\overline{y_{\mathbb{O}} : \mathbb{O}}, \overline{y_{\mathbb{B}} : \mathbb{B}}, \overline{y_{\mathbb{R}} : \mathbb{R}} \vdash_{RC} C_{app}(\overline{y}, \beta(c), \mathtt{let}\ z = c\ \overline{y};\ C(F'))$$

We generalize the goal (using $y^l := b^l := []$) to

$$\frac{\overline{y} = \overline{y^l}\ \overline{y^r} \qquad \overline{\beta(c)} = \overline{b^l}\ \overline{b^r} \qquad |\ \overline{y^l}\ |=|\ \overline{b^l}\ | \qquad \overline{y_{\mathbb{O}}'} = O(\overline{y^l})}{\begin{array}{c}\overline{y_{\mathbb{O}} : \mathbb{O}}, \overline{y_{\mathbb{O}}' : \mathbb{O}}, \overline{y_{\mathbb{B}} : \mathbb{B}}, \overline{y_{\mathbb{R}} : \mathbb{R}} \vdash_{RC} \\ C_{app}(\overline{y^r}, \overline{b^r}, \mathtt{let}\ z = c\ \overline{y};\ \mathbb{O}^-(B(\overline{y^l}), C(F')))\end{array}}$$

where

$$
\begin{aligned}
O(\overline{y^l}) &= O_{\mathbb{B}}(\overline{y^l})\ O_{\mathbb{O}}(\overline{y^l}) \\
O_{\mathbb{B}}(\overline{y^l}) &= [y^l_i \in \overline{y^l} \mid \beta(c)_i = \mathbb{O} \wedge \beta_l(y^l_i) = \mathbb{B}] \\
O_{\mathbb{O}}(\overline{y^l}) &= [y^l_i \in \overline{y^l} \mid \beta(c)_i = \mathbb{O} \wedge \beta_l(y^l_i) = \mathbb{O} \wedge \\
&\qquad\qquad (y^l_i \in \mathrm{FV}(F') \vee \exists j.\ y^l_i = y_j \wedge [j > i \vee \beta(c)_j = \mathbb{B}])] \\
B(\overline{y^l}) &= [y^l_i \in \overline{y^l} \mid \beta(c)_i = \mathbb{B} \wedge \beta_l(y^l_i) = \mathbb{O} \wedge \\
&\qquad\qquad \nexists j < i.\ y^l_i = y_j \wedge \beta(c)_j = \mathbb{B}]
\end{aligned}
$$

O and B accurately describe what arguments to increment/decrement in an explicit form: We increment borrowed references that are passed to an owned parameter, as well as owned references passed to an owned parameter that

- are still used in F' or
- are also passed to a borrowed parameter or
- are also passed to another owned parameter later.

We decrement owned references passed to a borrowed parameter and dead in F' (as enforced by $\mathbb{O}^-$), but at most once per variable.

We proceed by parallel induction over y_r and b_r, which, by $\vdash_{pure}$, have the same length:

Case $y^r = y'\ y''^r, b^r = \mathbb{O}\ b'^r$
We need to show

$$
\cdots \vdash_{RC} \mathbb{O}^+_{y'}(\overline{y''^r} \cup \mathrm{FV}(\mathbb{O}^-(B(\overline{y^l}), C(F'))), \\
C_{app}(\overline{y''^r}, \overline{b'^r}, \texttt{let}\ z = c\ \overline{y};\ \mathbb{O}^-(B(\overline{y^l}), C(F')))
$$

We see that $B(\overline{y^l}\ y') = B(\overline{y^l})$, and that $O(\overline{y^l}\ y')$ is $O(\overline{y^l})$ with y' appended iff

$$
\beta_l(y') = \mathbb{B} \vee y' \in \mathrm{FV}(F') \vee y' \in y''^r \vee y' \in B(\overline{y^l})
$$

This is exactly the condition for which an `inc` is inserted by $\mathbb{O}^+_{y'}$, so after conditionally applying Ty-Inc, we can apply the inner induction hypothesis.

Case $y^r = y'\ y'^r, b^r = \mathbb{B}\ b'^r$
We need to show

$$\cdots \vdash_{RC} C_{app}(\overline{y'^r}, \overline{b'^r}, \texttt{let}\ z = c\ \overline{y};\ \mathbb{O}^-_{y'}(\mathbb{O}^-(B(\overline{y^l}), C(F'))))$$

We see that $O(\overline{y^l}\ y') = O(\overline{y^l})$, and that $B(\overline{y^l}\ y')$ is $B(\overline{y^l})$ with y' appended iff $y' \notin B(\overline{y^l}) \wedge \beta_l(y') = \mathbb{O}$. In either case, we have

$$\mathbb{O}^-_{y'}(\mathbb{O}^-(B(\overline{y^l}), C(F'))) = \mathbb{O}^-(B(\overline{y^l}\ y'), C(F'))$$

and can apply the inner induction hypothesis.

Case $y^r = [], b^r = []$
We are left to show

$$\overline{y_{\mathbb{O}} : \mathbb{O}}, \overline{O(\overline{y}) : \mathbb{O}}, \overline{y_{\mathbb{B}} : \mathbb{B}}, \overline{y_{\mathbb{R}} : \mathbb{R}} \vdash_{RC} \\ \texttt{let}\ z = c\ \overline{y};\ \mathbb{O}^-(B(\overline{y}), C(F'))))$$

We have $\overline{y} \subseteq \overline{y_{\mathbb{O}}}\ \overline{y_{\mathbb{B}}}$ by $\vdash_{pure}$. We thus notice that $B(\overline{y})$ is a submultiset of $\overline{y_{\mathbb{O}}}$ and call the difference list D. We further split D into

$$D_1 = [x \in D \mid x \notin \mathrm{FV}(F')]$$
$$D_2 = [x \in D \mid x \in \mathrm{FV}(F')]$$

We apply Ty-Let, moving D_1 and $O(\overline{y})$ into the first goal, temporarily borrowing $B(\overline{y})$, and copying $\overline{y_{\mathbb{B}}}$ into both goals via contraction, leaving us with

$$\overline{D_1 : \mathbb{O}}, \overline{y_{\mathbb{B}} : \mathbb{B}}, \overline{O(\overline{y}) : \mathbb{O}}, \overline{B(\overline{y}) : \mathbb{B}} \vdash_{RC} c\ \overline{y} : \mathbb{O}$$

$$\overline{D_2 : \mathbb{O}}, \overline{y_{\mathbb{B}} : \mathbb{B}}, \overline{y_{\mathbb{R}} : \mathbb{R}}, \overline{B(\overline{y}) : \mathbb{O}}, z : \mathbb{O} \vdash_{RC} \mathbb{O}^-(B(\overline{y}), C(F'))$$

For the first goal, we notice that every argument used as a borrowed parameter is in $\overline{y_{\mathbb{B}}}$ or $B(\overline{y})$ by $\vdash_{pure}$, so by weakening we can fulfill them. For arguments used as owned parameters, we have to pay closer attention to the exact number of hypotheses: We notice that because $\overline{y_{\mathbb{O}}}$ is distinct, so is D_1, so we have covered the first occurrence of every variable dead in F' and not

in $B(\overline{y})$. The missing arguments are by definition exactly $O(\overline{y})$, so we are done.

For the second goal, we iteratively apply Ty-Dec and then apply the outer induction hypothesis: we have $D_2 \subseteq \mathrm{FV}(F')$ by definition, $z \in \mathrm{FV}(F')$ and $z \notin D_2 \cup B(\overline{y})$ by $\vdash_{pure}$, and $D_2 \cap B(\overline{y}) = \emptyset$ by definition of the split.

All other application cases are mostly analogous to the constant case (in particular, without any borrowed parameters). For `pap`, $\vdash_\beta$ proves the assumption $\beta(c) = \overline{\mathbb{O}}$. For `reuse`, the hypothesis $x : \mathbb{R}$, whose existence is guaranteed by $\vdash_{reuse}$, additionally has to be moved into the first goal.

□

Theorem 9. *δ_{RC} refines δ_β in the pure semantics.*

Proof. Trivial, given that the only change is insertion of `inc`/`dec` instructions. □

Corollary 1 (δ_{RC} preserves semantics)**.** *Suppose c is a parameter-less constant, $\delta(c) = F$.*

1. *If $\vdash \delta(c) \Downarrow v$, then $\vdash \langle \delta_{RC}(c), \emptyset \rangle \Downarrow \langle l, \sigma \rangle$ and* $\mathrm{valof}(l, \sigma) = v$.
2. *If $\vdash \langle \delta_{RC}(c), \emptyset \rangle \Downarrow \langle l, \sigma \rangle$, then $\vdash \delta(c) \Downarrow$* $\mathrm{valof}(l, \sigma)$.

Proof. By the pure refinement steps (Theorem 2, Theorem 4, Theorem 9), the welltypedness of δ_{RC} (Theorem 8), and the semantics preservation proof of welltyped programs (Theorem 5). □

Bibliography

[Adams, 2015] Adams, M. D. (2015). Towards the essence of hygiene. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 457–469.

[Allen et al., 2005] Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.-W., Ryu, S., Steele Jr, G. L., Tobin-Hochstadt, S., Dias, J., Eastlund, C., et al. (2005). The Fortress language specification. *Sun Microsystems*, 139(140):116.

[Altenkirch, 2010] Altenkirch, T. (2010). Agda mailing list: Heterogeneous vectors. `https://web.archive.org/web/20221110164046/https://lists.chalmers.se/pipermail/agda/2010/001826.html`. Accessed: 2022-11-10.

[Bachrach et al., 1999] Bachrach, J., Playford, K., and Street, C. (1999). D-expressions: Lisp power, Dylan style. *Style DeKalb IL*.

[Baker, 1994] Baker, H. G. (1994). Minimizing reference count updating with deferred and anchored pointers for functional data structures. *SIGPLAN Not.*, 29(9):38–43.

[Barendregt, 1991] Barendregt, H. (1991). Introduction to generalized type systems. *Journal of functional programming*, 1(2):125–154.

[Barrett et al., 2022] Barrett, L., Christiansen, D. T., and Gélineau, S. (2022). Predictable macros for Hindley-Milner (extended abstract). *Workshop on Type-Driven Development*.

[Barth, 1977] Barth, J. M. (1977). Shifting garbage collection overhead to compile time. *Commun. ACM*, 20(7):513–518.

[Barzilay et al., 2011] Barzilay, E., Culpepper, R., and Flatt, M. (2011). Keeping it clean with syntax parameters. *Proc. Wksp. Scheme and Functional Programming*.

[Bechberger, 2016] Bechberger, J. (2016). Besser benchmarken. Bachelor's thesis, Karlsruher Institut für Technologie (KIT). `https://pp.ipd.kit.edu/publication.php?id=bechberger16bachelorarbeit`.

[Benton et al., 1993] Benton, P. N., Bierman, G. M., Paiva, V. d., and Hyland, M. (1993). A term calculus for intuitionistic linear logic. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, TLCA '93, pages 75–90, London, UK, UK. Springer-Verlag.

[Bloom, 2021] Bloom, T. F. (2021). On a density conjecture about unit fractions. `https://arxiv.org/abs/2112.03726`.

[Bloom and Mehta, 2022] Bloom, T. F. and Mehta, B. (2022). Unit fractions. `https://web.archive.org/web/20220705040548/https://b-mehta.github.io/unit-fractions/`. Accessed: 2023-01-13.

[Brady, 2013] Brady, E. (2013). Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593.

[Brady, 2014] Brady, E. (2014). Resource-dependent algebraic effects. In *International Symposium on Trends in Functional Programming*, pages 18–33. Springer.

[Bravenboer et al., 2006] Bravenboer, M., Tanter, É., and Visser, E. (2006). Declarative, formal, and extensible syntax definition for AspectJ. *ACM SIGPLAN Notices*, 41(10):209–228.

[Bülow, 2022] Bülow, N. (2022). Proof visualization for the Lean 4 theorem prover. Bachelor's thesis, Karlsruher Institut für Technologie (KIT). `https://pp.ipd.kit.edu/publication.php?id=b%C3%BClow22bachelorarbeit`.

[Carneiro, 2019] Carneiro, M. (2019). The type theory of Lean. Master's thesis, Carnegie Mellon University. `https://github.com/digama0/lean-type-theory/releases/download/v1.0/main.pdf`.

[Chang et al., 2019] Chang, S., Ballantyne, M., Turner, M., and Bowman, W. J. (2019). Dependent type systems as macros. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–29.

[Chirimar et al., 1996] Chirimar, J., Gunter, C. A., and Riecke, J. G. (1996). Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming*, 6(2):195–244.

[Choi et al., 2018] Choi, J., Shull, T., and Torrellas, J. (2018). Biased reference counting: Minimizing atomic operations in garbage collection. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, pages 35:1–35:12, New York, NY, USA. ACM.

[Christiansen, 2022] Christiansen, D. T. (2022). Functional programming in Lean. `https://leanprover.github.io/functional_programming_in_lean/`.

[Christiansen and Brady, 2016] Christiansen, D. T. and Brady, E. (2016). Elaborator reflection: Extending Idris in Idris. In Garrigue, J., Keller, G., and Sumii, E., editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016*, pages 284–297. ACM.

[Clinger and Rees, 1991] Clinger, W. and Rees, J. (1991). Macros that work. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 155–162.

[Cohen et al., 2015] Cohen, C., Coquand, T., Huber, S., and Mörtberg, A. (2015). Cubical Type Theory: a constructive interpretation of the univalence axiom. In *21st International Conference on Types for Proofs and Programs*, page 262. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Coquand and Huet, 1988] Coquand, T. and Huet, G. (1988). The calculus of constructions. *Inform. and Comput.*, 76(2-3):95–120.

[de Bruijn, 1970] de Bruijn, N. G. (1970). The mathematical language AUTOMATH, its usage, and some of its extensions. In *Symposium on automatic demonstration*, pages 29–61. Springer.

[de Moura et al., 2015a] de Moura, L., Avigad, J., Kong, S., and Roux, C. (2015a). Elaboration in dependent type theory. `https://arxiv.org/abs/1505.04324`.

[de Moura and Bjørner, 2008] de Moura, L. and Bjørner, N. (2008). Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer.

[de Moura et al., 2015b] de Moura, L., Kong, S., Avigad, J., van Doorn, F., and von Raumer, J. (2015b). The Lean theorem prover (system description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, 2015, Proceedings*, pages 378–388.

[de Moura and Passmore, 2013] de Moura, L. and Passmore, G. O. (2013). The strategy challenge in SMT solving. In *Automated Reasoning and Mathematics*, pages 15–44. Springer.

[de Moura and Ullrich, 2021] de Moura, L. and Ullrich, S. (2021). The Lean 4 theorem prover and programming language. In *International Conference on Automated Deduction*, pages 625–635. Springer.

[Delahaye, 2000] Delahaye, D. (2000). A tactic language for the system Coq. In *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Proceedings*, pages 85–95.

[Delahaye, 2002] Delahaye, D. (2002). A proof dedicated meta-language. *Electr. Notes Theor. Comput. Sci.*, 70(2):96–109.

[Dybjer, 1994] Dybjer, P. (1994). Inductive families. *Formal aspects of computing*, 6(4):440–465.

[Dybvig et al., 1986] Dybvig, R. K., Friedman, D. P., and Haynes, C. T. (1986). Expansion-passing style: Beyond conventional macros. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 143–150.

[Dybvig et al., 1993] Dybvig, R. K., Hieb, R., and Bruggeman, C. (1993). Syntactic abstraction in Scheme. *Lisp and symbolic computation*, 5(4):295–326.

[Earley, 1970] Earley, J. (1970). An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102.

[Eastlund and Felleisen, 2010] Eastlund, C. and Felleisen, M. (2010). Hygienic macros for ACL2. In *International Symposium on Trends in Functional Programming*, pages 84–101. Springer.

[Ebner et al., 2017] Ebner, G., Ullrich, S., Roesch, J., Avigad, J., and de Moura, L. (2017). A metaprogramming framework for formal verification. *Proc. ACM Program. Lang.*, 1(ICFP).

[Erkök and Launchbury, 2002] Erkök, L. and Launchbury, J. (2002). A recursive do for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 29–37.

[Ershov, 1958] Ershov, A. P. (1958). On programming of arithmetic operations. *Communications of the ACM*, 1(8):3–6.

[Feldman, 2021] Feldman, R. (2021). Outperforming imperative with pure functional languages. Recorded presentation. `https://youtu.be/vzfy4EKwG_Y`.

[Feller, 1950] Feller, W. (1950). *An introduction to probability theory and its applications, Volume 1*. John Wiley & Sons.

[Férey and Shankar, 2016] Férey, G. and Shankar, N. (2016). Code generation using a formal model of reference counting. In *Proceedings of the 8th International Symposium on NASA Formal Methods - Volume 9690*, NFM 2016, pages 150–165, New York, NY, USA. Springer-Verlag New York, Inc.

[Flanagan et al., 1993] Flanagan, C., Sabry, A., Duba, B. F., and Felleisen, M. (1993). The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 237–247, New York, NY, USA. ACM.

[Flatt, 2002] Flatt, M. (2002). Composable and compilable macros: you want it when? *ACM SIGPLAN Notices*, 37(9):72–83.

[Flatt, 2016] Flatt, M. (2016). Binding as sets of scopes. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 705–717, New York, NY, USA. ACM.

[Ford, 2002] Ford, B. (2002). Packrat parsing: Simple, powerful, lazy, linear time (Functional Pearl). *SIGPLAN Not.*, 37(9):36–47.

[Foster et al., 2007] Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., and Schmitt, A. (2007). Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(3):17–es.

[GHC, 2020] GHC (2020). List of tools needed to build GHC. `https://web.archive.org/web/20210116000343/https://gitlab.haskell.org/ghc/ghc/-/wikis/building/preparation/tools`. Accessed: 2022-03-10.

[Gibbons and dos Santos Oliveira, 2009] Gibbons, J. and dos Santos Oliveira, B. C. (2009). The essence of the iterator pattern. *Journal of functional programming*, 19(3 and 4).

[Girard, 1972] Girard, J.-Y. (1972). *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, University of Paris VII.

[Gordon et al., 1978] Gordon, M., Milner, R., Morris, L., Newey, M., and Wadsworth, C. (1978). A metalanguage for interactive proof in LCF. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 119–130.

[Gratzer et al., 2022] Gratzer, D., Sterling, J., Angiuli, C., Coquand, T., and Birkedal, L. (2022). Controlling unfolding in type theory. `https://arxiv.org/abs/2210.05420`.

[Grelck and Trojahner, 2004] Grelck, C. and Trojahner, K. (2004). Implicit memory management for SAC. In *Implementation and Application of Functional Languages, 16th International Workshop, IFL*, volume 4, pages 335–348.

[Hallenberg et al., 2002] Hallenberg, N., Elsman, M., and Tofte, M. (2002). Combining region inference and garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*. ACM Press. Berlin, Germany.

[Ho and Protzenko, 2022] Ho, S. and Protzenko, J. (2022). Aeneas: Rust verification by functional translation. *Proceedings of the ACM on Programming Languages*, 6(ICFP):711–741.

[Howard, 1980] Howard, W. A. (1980). The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490.

[Hudak and Bloss, 1985] Hudak, P. and Bloss, A. (1985). The aggregate update problem in functional programming systems. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '85, pages 300–314, New York, NY, USA. ACM.

[Huisinga, 2019] Huisinga, M. (2019). Formally verified insertion of reference counting instructions. Bachelor's thesis, Karlsruher Institut für Technologie (KIT). `https://pp.ipd.kit.edu/publication.php?id=huisinga19bachelorarbeit`.

[Inria, CNRS and contributors, 2021] Inria, CNRS and contributors (2021). Record types - Coq 8.16.1 documentation. `https://web.archive.org/web/20221115002532/https://coq.inria.fr/refman/language/core/records.html#primitive-projections`. Accessed: 2022-12-01.

[Jim et al., 2002] Jim, T., Morrisett, J. G., Grossman, D., Hicks, M. W., Cheney, J., and Wang, Y. (2002). Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference, General Track*, pages 275–288.

[Kaiser et al., 2018] Kaiser, J.-O., Ziliani, B., Krebbers, R., Régis-Gianas, Y., and Dreyer, D. (2018). Mtac2: typed tactics for backward reasoning in Coq. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–31.

[Kaposi and von Raumer, 2020] Kaposi, A. and von Raumer, J. (2020). A syntax for mutual inductive families. In Ariola, Z. M., editor, *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, volume 167 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:21, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[Kelley, 2020] Kelley, A. (2020). 'zig cc': a powerful drop-in replacement for GCC/Clang. `http://web.archive.org/web/20220115054443/https://andrewkelley.me/post/zig-cc-powerful-drop-in-replacement-gcc-clang.html`. Accessed: 2022-01-28.

[Kelsey, 1995] Kelsey, R. A. (1995). A correspondence between continuation passing style and static single assignment form. *ACM SIGPLAN Notices*, 30(3):13–22.

[Kohlbecker et al., 1986] Kohlbecker, E., Friedman, D. P., Felleisen, M., and Duba, B. (1986). Hygienic macro expansion. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 151–161.

[Lample et al., 2022] Lample, G., Lachaux, M.-A., Lavril, T., Martinet, X., Hayat, A., Ebner, G., Rodriguez, A., and Lacroix, T. (2022). Hypertree proof search for neural theorem proving. `https://arxiv.org/abs/2205.11491`.

[Launchbury and Peyton Jones, 1995] Launchbury, J. and Peyton Jones, S. L. (1995). State in Haskell. *Lisp and symbolic computation*, 8(4):293–341.

[Leijen, 2014] Leijen, D. (2014). Koka: Programming with row polymorphic effect types. *Electronic Proceedings in Theoretical Computer Science*, 153:100–126.

[Limperg and From, 2023] Limperg, J. and From, A. H. (2023). Aesop: White-box best-first proof search for Lean. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 253–266.

[Lorenzen and Leijen, 2022] Lorenzen, A. and Leijen, D. (2022). Reference counting with frame limited reuse. *Proceedings of the ACM on Programming Languages*, 6(ICFP):357–380.

[MacQueen, 1984] MacQueen, D. (1984). Modules for Standard ML. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, pages 198–207.

[Mahboubi and Tassi, 2021] Mahboubi, A. and Tassi, E. (2021). *Mathematical Components*. Zenodo. `https://doi.org/10.5281/zenodo.4457887`.

[Marlow et al., 2016] Marlow, S., Peyton Jones, S., Kmett, E., and Mokhov, A. (2016). Desugaring Haskell's do-notation into applicative operations. *ACM SIGPLAN Notices*, 51(12):92–104.

[Mathlib, 2022] Mathlib (2022). Mathlib documentation: Mathlib tactics. https://web.archive.org/web/20220308182313/https://leanprover-community.github.io/mathlib_docs/tactics.html. Accessed: 2022-03-22.

[Matichuk et al., 2016] Matichuk, D., Murray, T., and Wenzel, M. (2016). Eisbach: A proof method language for Isabelle. *Journal of Automated Reasoning*, 56(3):261–282.

[Matsakis and Klock, 2014] Matsakis, N. D. and Klock, II, F. S. (2014). The Rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 103–104, New York, NY, USA. ACM.

[Maurer et al., 2017] Maurer, L., Downen, P., Ariola, Z. M., and Peyton Jones, S. (2017). Compiling without continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 482–494, New York, NY, USA. ACM.

[McBride, 2005] McBride, C. (2005). Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, pages 130–170. Springer.

[McBride and McKinna, 2004] McBride, C. and McKinna, J. (2004). Functional Pearl: I am not a number — I am a free variable. In Nilsson, H., editor, *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2004*, pages 1–9. ACM.

[McGraw et al., 1983] McGraw, J., Skedzielewski, S., Allan, S., Grit, D., Oldehoeft, R., Glauert, J., Dobes, I., and Hohensee, P. (1983). SISAL: streams and iteration in a single-assignment language. Language reference manual, version 1. Technical report, Lawrence Livermore National Lab., CA (USA).

[Mennicken, 2022] Mennicken, J. (2022). Locating and presenting lexical references in a theorem prover. Bachelor's thesis, Karlsruher Institut für Technologie (KIT). https://pp.ipd.kit.edu/publication.php?id=mennicken22bachelorarbeit.

[Milner, 1979] Milner, R. (1979). LCF: A way of doing proofs with a machine. In *International Symposium on Mathematical Foundations of Computer Science*, pages 146–159. Springer.

[Moggi, 1991] Moggi, E. (1991). Notions of computation and monads. *Information and computation*, 93(1):55–92.

[Nipkow, 1998] Nipkow, T. (1998). Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186.

[Norell, 2009] Norell, U. (2009). Dependently typed programming in Agda. In *Advanced Functional Programming*, pages 230–266. Springer.

[Okasaki, 1999] Okasaki, C. (1999). Red-black trees in a functional setting. *Journal of functional programming*, 9(4):471–477.

[Paterson, 2001] Paterson, R. (2001). A new notation for arrows. *ACM SIGPLAN Notices*, 36(10):229–240.

[Paulin-Mohring, 2015] Paulin-Mohring, C. (2015). *Introduction to the calculus of inductive constructions*. College Publications.

[Peyton Jones, 2003] Peyton Jones, S. (2003). *Haskell 98 language and libraries: the revised report*. Cambridge University Press.

[Peyton Jones, 1996] Peyton Jones, S. L. (1996). Compiling Haskell by program transformation: a report from the trenches. In *Proc. European Symp. on Programming*, pages 18–44. Springer-Verlag.

[Pit-Claudel, 2020] Pit-Claudel, C. (2020). Untangling mechanized proofs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2020, page 155–174, New York, NY, USA. Association for Computing Machinery.

[Polu et al., 2022] Polu, S., Han, J. M., Zheng, K., Baksys, M., Babuschkin, I., and Sutskever, I. (2022). Formal mathematics statement curriculum learning. `https://arxiv.org/abs/2202.01344`.

[Pombrio et al., 2017] Pombrio, J., Krishnamurthi, S., and Wand, M. (2017). Inferring scope through syntactic sugar. *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–28.

[Pratt, 1973] Pratt, V. R. (1973). Top down operator precedence. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 41–51.

[Rafkind and Flatt, 2012] Rafkind, J. and Flatt, M. (2012). Honu: syntactic extension for algebraic notation through enforestation. In *ACM SIGPLAN Notices*, volume 48, pages 122–131. ACM.

[Reinking et al., 2021] Reinking, A., Xie, N., de Moura, L., and Leijen, D. (2021). Perceus: Garbage free reference counting with reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 96–111.

[Richter and Nasarre, 2008] Richter, J. and Nasarre, C. (2008). *Windows via C/C++, fifth edition*. Microsoft Press.

[Rosen et al., 1988] Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1988). Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27.

[Rust, 2021] Rust (2021). Guide to rustc development: Bootstrapping the compiler. `https://web.archive.org/web/20211123003848/https://rustc-dev-guide.rust-lang.org/building/bootstrapping.html`. Accessed: 2022-03-10.

[Scholz, 1994] Scholz, S.-B. (1994). Single assignment C — functional programming using imperative style. In *In John Glauert (Ed.): Proceedings of the 6th International Workshop on the Implementation of Functional Languages. University of East Anglia*.

[Schulte, 1994] Schulte, W. (1994). Deriving residual reference count garbage collectors. In *Proceedings of the 6th International Symposium on Programming Language Implementation and Logic Programming*, PLILP '94, pages 102–116, London, UK. Springer-Verlag.

[Sheard and Peyton Jones, 2002] Sheard, T. and Peyton Jones, S. (2002). Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM.

[Smetsers et al., 1994] Smetsers, S., Barendsen, E., van Eekelen, M., and Plasmeijer, R. (1994). Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. In *Graph Transformations in Computer Science: International Workshop Dagstuhl Castle, Germany, January 4–8, 1993 Proceedings*, pages 358–379. Springer.

[Sozeau et al., 2020] Sozeau, M., Anand, A., Boulier, S., Cohen, C., Forster, Y., Kunze, F., Malecha, G., Tabareau, N., and Winterhalter, T. (2020). The MetaCoq project. *Journal of Automated Reasoning*, 64(5):947–999.

[Sozeau et al., 2019] Sozeau, M., Boulier, S., Forster, Y., Tabareau, N., and Winterhalter, T. (2019). Coq Coq Correct! Verification of type checking and erasure for Coq, in Coq. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–28.

[Sterling and Harper, 2021] Sterling, J. and Harper, R. (2021). A metalanguage for multi-phase modularity. *ML Family workshop*.

[Swierstra and Duponcheel, 1996] Swierstra, S. D. and Duponcheel, L. (1996). Deterministic, error-correcting combinator parsers. In *International School on Advanced Functional Programming*, pages 184–207. Springer.

[Taha and Sheard, 2000] Taha, W. and Sheard, T. (2000). MetaML and multi-stage programming with explicit annotations. *Theoretical computer science*, 248(1-2):211–242.

[The Coq Team, 2017] The Coq Team (2017). The Coq proof assistant, version 8.7.0. `https://doi.org/10.5281/zenodo.1028037`.

[The mathlib Community, 2020] The mathlib Community (2020). The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 367–381, New York, NY, USA.

[The mathlib Community, 2022a] The mathlib Community (2022a). Completion of the Liquid Tensor Experiment. `https://web.archive.org/web/20221129050433/https://leanprover-community.github.io/blog/posts/lte-final/`. Accessed: 2023-01-13.

[The mathlib Community, 2022b] The mathlib Community (2022b). Mathlib statistics. `https://web.archive.org/web/20221203000358/https://leanprover-community.github.io/mathlib_stats.html`. Accessed: 2022-12-13.

[The Univalent Foundations Program, 2013] The Univalent Foundations Program (2013). *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book, Institute for Advanced Study.

[Ullrich, 2016] Ullrich, S. (2016). Simple verification of Rust programs via functional purification. Master's thesis, Karlsruher Institut für Technologie (KIT). https://pp.ipd.kit.edu/publication.php?id=ullrich16masterarbeit.

[Ullrich and de Moura, 2019a] Ullrich, S. and de Moura, L. (2019a). Counting immutable beans: Reference counting optimized for purely functional programming. In *31st Symposium on Implementation and Application of Functional Languages*.

[Ullrich and de Moura, 2019b] Ullrich, S. and de Moura, L. (2019b). Counting immutable beans – appendix. https://leanprover.github.io/papers/beans_appendix.pdf.

[Ullrich and de Moura, 2020] Ullrich, S. and de Moura, L. (2020). Beyond notations: Hygienic macro expansion for theorem proving languages. In *International Joint Conference on Automated Reasoning*, pages 167–182. Springer.

[Ullrich and de Moura, 2022a] Ullrich, S. and de Moura, L. (2022a). Beyond notations: Hygienic macro expansion for theorem proving languages. *Logical Methods in Computer Science*, Volume 18, Issue 2.

[Ullrich and de Moura, 2022b] Ullrich, S. and de Moura, L. (2022b). 'do' unchained: Embracing local imperativity in a purely functional language (Functional Pearl). *Proc. ACM Program. Lang.*, 6(ICFP):512–539.

[Ullrich and de Moura, 2023] Ullrich, S. and de Moura, L. (2023). Supplement for A Macro System for Theorem Provers.

[Ungar et al., 2017] Ungar, D., Grove, D., and Franke, H. (2017). Dynamic atomicity: Optimizing Swift memory management. In *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages*, DLS 2017, pages 15–26, New York, NY, USA. ACM.

[van der Walt and Swierstra, 2012] van der Walt, P. and Swierstra, W. (2012). Engineering proof by reflection in Agda. In Hinze, R., editor, *Implementation and Application of Functional Languages - 24th International Symposium, IFL 2012*, volume 8241 of *Lecture Notes in Computer Science*, pages 157–173. Springer.

[van Doorn et al., 2023] van Doorn, F., Massot, P., and Nash, O. (2023). Formalising the h-principle and sphere eversion. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 121–134.

[Visser, 1997] Visser, E. (1997). *Scannerless generalized LR parsing*. Universiteit van Amsterdam. Programming Research Group.

[Wadler, 1990a] Wadler, P. (1990a). Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78.

[Wadler, 1990b] Wadler, P. (1990b). Linear types can change the world! In Broy, M. and Jones, C., editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, IFIP TC 2, pages 347–359. North Holland.

[Wadler and Blott, 1989] Wadler, P. and Blott, S. (1989). How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76. ACM.

[Weeks, 2006] Weeks, S. (2006). Whole-program compilation in MLton. In *Proceedings of the 2006 Workshop on ML*, ML '06, pages 1–1, New York, NY, USA. ACM.

[Whitehead and Russell, 1910] Whitehead, A. N. and Russell, B. (1910). *Principia mathematica*, volume 1. Cambridge University Press.

[Yang et al., 2015] Yang, E. Z., Campagna, G., Ağacan, Ö. S., El-Hassany, A., Kulkarni, A., and Newton, R. R. (2015). Efficient communication and collection with compact normal forms. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, pages 362–374.

[Ziliani et al., 2015] Ziliani, B., Dreyer, D., Krishnaswami, N. R., Nanevski, A., and Vafeiadis, V. (2015). Mtac: A monad for typed tactic programming in Coq. *J. Funct. Program.*, 25.

Index

. (prefix dot notation), 154
. (projection notation), 4, 53

antiquotation, 64
antiquotation splice, 76, 81
antiquotation, token, 79

bootstrapping, 56, 75
`by`, 8

Calculus of Constructions, 6
Calculus of Inductive Constructions, 6
`class`, 4
code generator, **54**, 144
core language, 34, 95
Cubical Type Theory, 11
Curry-Howard correspondence, 6

`def`, 4
dependent function type, 6
dependent type, 5
dependent type theory, 9, 13, 35, 46
`do`, 27, 74, **93**

elaborator, 25, **52**, 82, 118
`elab_rules`, 84
environment, 17
η-conversion (structure), 38
`example`, 7
extended dot notation, 109

field (inductive type), 36
frontend, 2, **34**
`fun`, 4

global context, *see* environment, 68

Homotopy Type Theory, 11

import, 34, 58, 156
impredicative, 7
index (inductive type), 6, 36
`inductive`, 5
inductive type, 5
info tree, 59
interactive theorem proving, 1
intermediate representation, 54, **135**

kernel, 15, **35**

language server, 46, **57**
Language Server Protocol, 57, 128
large elimination, 37, 41
Lean, **4**
Lean 0.1, 10
Lean 2, **11**, 40, 53
Lean 3, 11, **13**, 35, 48, 49, 53, 57, 84
Lean 4, **33**
Lean 5, 12
local context, 18, 68
locally nameless, 18

macro, 34, **61**
`macro`, **64**, 85, 171
macro expander, 68
macro hygiene, **67**, 99
macro, anaphoric, 70
`macro_rules`, **64**, 171
mathlib, 2, 11, 21, 29, 45, 48, 167
memory mapping, 48
metaprogramming, 4, 13, 46, 54
metavariable, 19, 53
metavariable context, 19
module system, 45

`opaque`, 16, 35

parameter (inductive type), 6
parser, **49**, 64, 77
`partial`, 16
precompilation, 55
pretty printer, 34
proof irrelevance, 8
`Prop`, 7

quasiquotation, 52, 64
quasiquotation, prechecked, 86

recursor, 9, 15, 37, **42**

`structure`, 4
structure projection, primitive, 36
subsingleton elimination, 42
surface language, 26, 34
symbol, 69
syntactic category, 26, **52**, 66, 115
`Syntax` (type), 51, 72
`syntax`, 64

tactic, 8, 13, 53, 84
tactic interpreter, 85
telescope, 36
`this`, 70
Trusted Code Base, 15, 34, 36
`Type`, 7
type family, 5
type system, 35
typeclass, 4, 45, 53, 83, 118

universe, 7
universe variable, 7

white-box automation, 9

List of Publications

[1] Sebastian Buchwald, Denis Lohner, and Sebastian Ullrich. Verified construction of static single assignment form. In Manuel Hermenegildo, editor, *25th International Conference on Compiler Construction*, CC 2016, pages 67–76. ACM, 2016.

[2] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. *Proc. ACM Program. Lang.*, 1(ICFP), 2017.

[3] Sebastian Ullrich and Leonardo de Moura. Counting immutable beans: Reference counting optimized for purely functional programming. In *31st Symposium on Implementation and Application of Functional Languages*, 2019.

[4] Sebastian Ullrich and Leonardo de Moura. Beyond notations: Hygienic macro expansion for theorem proving languages. In *International Joint Conference on Automated Reasoning*, pages 167–182. Springer, 2020.

[5] Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *International Conference on Automated Deduction*, pages 625–635. Springer, 2021.

[6] Sebastian Ullrich and Leonardo de Moura. Beyond notations: Hygienic macro expansion for theorem proving languages. *Logical Methods in Computer Science*, Volume 18, Issue 2, 2022.

[7] Sebastian Ullrich and Leonardo de Moura. ‘do’ unchained: Embracing local imperativity in a purely functional language (Functional Pearl). *Proc. ACM Program. Lang.*, 6(ICFP):512–539, 2022.