EPFL

Thèse n° 8030

Scaling Language Features for Program Verification

Présentée le 22 juillet 2022

Faculté informatique et communications
Laboratoire d'analyse et de raisonnement automatisés
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Georg Stefan SCHMID

Acceptée sur proposition du jury

Prof. R. Guerraoui, président du jury
Prof. V. Kuncak, directeur de thèse
Prof. Ph. Rümmer, rapporteur
Dr D. Vytiniotis, rapporteur
Prof. C. Koch, rapporteur

École
polytechnique
fédérale
de Lausanne

2022

To my parents

Acknowledgements

Doing a PhD has been a humbling experience. When I started my studies I was full of naive enthusiasm for building the *next great thing* in static checking and getting it "out there" for lots of people to play with. My advisor, Viktor, indulged me in my adventures and I got to explore a wide range of topics over the following years, leading up to the thesis that you're about to read (or, more likely, skim). I thank Viktor for the freedom, guidance and wisdom he offered throughout my studies. Above all, though, I thank him for being the kind and thoughtful person he is — I think he doesn't get enough credit for that.

I also thank the members of my thesis jury, Rachid Guerraoui, Christoph Koch, Philipp Rümmer and Dimitrios Vytiniotis for their insightful comments and an enjoyable discussion during the oral exam. I am especially grateful towards Dimitrios, who took a chance on me after we had run into each other at a workshop. Working with him and the team at DeepMind (Dominik, Michael, Norman, Tamara and collaborators like Adam) exposed me to an exciting new domain and I learned a lot in little time. Most importantly, it inspired me to pursue more such collaborations in the future!

Getting from the beginning of the PhD to its conclusion means following a long and winding path, and I am indebted to many companions that guided me along. In compiling this list I am bound to miss some names that ought to be in it, but let me try nonetheless.

Firstly, I thank my colleagues in LARA: Etienne, Ravi, Regis, Mikaël, Andreas, Sarah, Manos, Marco, Stevan, Jad, Nicolas, Romain, Romain, and (the new generation!) Dragana, Rodrigo, Simon and Mario. I learned much from you, and my work on Stainless would have been impossible without the tremendous research and engineering done by lab mates past and present. I should also note that much credit is due to the brilliant undergraduate and Master's students that I was fortunate to have as collaborators over one or several semesters. In that regard I am especially thankful to Maxime, Antoine and Yann. And, of course, Fabien, Sylvie and Natascha also come to mind — they are the unsung heroes keeping LAMP & LARA afloat technically and logistically!

Early on in my PhD I found a second home of sorts in our sister-lab, LAMP. I am grateful to have had the chance to work alongside Martin and take an active part in the lab life, both academically and socially. The LAMPions are an incredible bunch, and I will fondly remember the many adventures I had in, around and outside of EPFL with Vojin, Dmitry, Eugene, Mano, Nada, Sandro, Séb, Felix, Allan, Denys, Paolo, Jonathan, Fengyun, Nico, Guillaume, Olivier, Aggelos, Anatolii, and Alex. The same goes for the Scala center crew, especially Julien, Guillaume, Darja, Martin, Jamie, Jorge and Ólafur.

I also had the great fortitude of stumbling upon amazing teams in industry during my studies, making it worthwhile to escape not once, but twice from EPFL's campus. I already mentioned my lasting impressions with Dimitrios' team. Two years earlier I got to spend an amazing summer in Munich, tinkering on compilers with Tobias, Jaro, Benedikt and many others in the V8 team. Not only are they terrific engineers, they also share many of my hobbies and interests, and I always look forward to visiting Munich thanks to that.
Of course the list doesn't end there. EPFL is a large place, and (sometimes!) I managed to break out of my programming-language bubble to hang out with a few of the other amazing folks spread around the campus. Right across the yard there were Jonas, Arseniy, David, Stuart, Marios, Sahand and Adrien, ready to dispense light coffee chat or dish out sage advice (depending on which was needed). Adrien, in particular, has been both a reliable source of comic relief, and, above all, a great friend. And then there are the many people with whom I often hiked, biked, travelled, played Spikeball or simply had a good time by the lake and at Sat: Patrik, Georg, Beril, Cey, Fredrik, Helena, Hermina, Cesare, Kostas, Lauriane, ...
A few special thanks are in order for the peers that were pivotal to my thesis: Nicolas, who has proven to be an amazing engineer time and time again, and helped me along with Stainless and life in general over coffee and beers. Jad, who is, undoubtedly, the best interactive proof assistant I know and without whom much of the work in Part II would have been untenable. Olivier, in whom I found not only a resourceful collaborator, but also a great friend and teacher of many skills I hadn't even considered I needed (like when I got a three-hour, mandatory, intensive crash course in the game of Skat during a train ride to Milan).
My acknowledgments cannot be complete without speaking of the many long-distance bike tours I enjoyed with Ólaf, Ronja, Olivier and Mia. These trips became a fixture in our summers, a yearly milestone of sorts, that allowed me to disconnect from research for two weeks, recharge my batteries, and refine said card-playing skills.
Last, but not least, I thank my family for the loving and unquestioning support they have provided over all those years. Having my parents, sister, grandparents and aunt to fall back on is a great privilege indeed, and I strive to be as dependable to those around me, as my family is to me. Finally, I thank Mia, who has saved me from many dire-looking situations with her sometimes forceful, often level-headed, but always earnest advice. Above all, I am thankful for her infinite patience, and showing me the bright days that lie ahead.

Lausanne, February 2022 G. S.

Abstract

Formal verification of real-world software systems remains challenging for a number of reasons, including lack of automation, friction in specifying properties, and limited support for the diverse programming paradigms used in industry. In this thesis we make progress towards a better verification experience in general-purpose programming languages by contributing improvements both to the *automated checking* and the *specification* of safety properties in languages combining functional and imperative features. We present our extensions in two parts – reasoning about shared mutable data, and types as specifications – both of which ultimately rely on reductions of expressive surface languages to a functional core. Throughout, we instantiate our techniques for the particular example of Scala, a mixed-paradigm language widely-used in industry.

The first part shows how to extend a verifier for higher-order functions and immutable data to support imperative programs with shared mutable data. We build upon Stainless, a contract-based verification system that relies on SMT solvers to automatically verify a large fragment of Scala. Our technique extends Stainless to check general heap-manipulating programs against modular specifications in the style of dynamic frames. A novelty of our approach is the translation of imperative function contracts that encodes frame conditions using quantifier-free formulas in first-order logic, instead of relying on quantifiers or on dedicated separation logic reasoning. Our quantifier-free encoding enables SMT solvers to both prove safety and to report counterexamples relative to the semantics of procedure contracts.

In the second part we turn to types and type-level programming as an alternative means of specifying correctness properties. While dependent types have been studied extensively for purely-functional languages, we investigate their applications to languages with subtyping and (abstractions of) imperative features. We first study a calculus that provides type-level computation through singleton types and allows abstraction of state and IO through a non-deterministic choice operator. This allows for modelling interactions with existing imprecisely-typed and impure code. Our calculus is formalized and mechanically proven correct using the Coq proof assistant. In addition, we develop a prototypical implementation in the Scala compiler and study typical type-level programming use cases in the Scala ecosystem.

Keywords: functional programming, imperative programming, formal verification, dependent type systems, counterexample finding, shared mutable data, dynamic frames

Résumé

La vérification formelle des systèmes logiciels dans des utilisations réelles reste difficile pour un certain nombre de raisons, notamment le manque d'automatisation, les difficultés dans la spécification des propriétés et le manque de support pour divers paradigmes de programmation utilisés dans l'industrie. Dans cette thèse, nous progressons vers une meilleure expérience de vérification dans les langages de programmation généraux en apportant des améliorations à la *vérification automatisée* ainsi que la *spécification* des propriétés de sécurité dans les langages combinant des caractéristiques fonctionnelles et impératives. Nous présentons nos contributions en deux parties - une première qui traite des données mutables partagées, puis une seconde qui aborde l'utilisation de types en tant que spécifications. Les deux parties sont basées sur la réduction de langages de surface expressifs à un noyau fonctionnel. Tout au long, nous instancions nos techniques pour l'exemple particulier de Scala, un langage à paradigmes mixtes largement utilisé dans l'industrie.

La première partie démontre comment étendre un vérificateur pour des fonctions d'ordre supérieur et des données immuables afin de supporter des programmes impératifs avec des données mutables partagées. Nous faisons usage de Stainless, un système de vérification basé sur des contrats qui s'appuie sur des solveurs SMT pour vérifier automatiquement un large fragment de Scala. L'intégration de notre technique permet à Stainless de vérifier des programmes généraux qui manipulent le tas par rapport à des spécifications modulaires dans le style des cadres dynamiques. Une nouveauté de notre approche est la traduction des contrats qui encodent les conditions de cadre des fonctions impératives en utilisant des formules sans quantificateur en logique du premier ordre, au lieu de s'appuyer sur des quantificateurs ou sur la logique de séparation. Notre encodage sans quantificateur permet aux solveurs SMT de prouver la sécurité et, surtout, de rapporter des contre-exemples aux contrats des procédures.

Dans la deuxième partie, nous nous tournons vers les types et la programmation au niveau du type comme moyen alternatif de spécifier les propriétés de correction. Alors que les types dépendants ont été largement étudiés pour les langages purement fonctionnels, nous abordons ici leurs applications aux langages avec sous-typage et des (abstractions de) traits impératifs. Nous étudions d'abord un système déductif qui fournit un calcul au niveau du type via des types singletons et permet l'abstraction de l'état et de l'IO via un opérateur de choix non déterministe. Cela permet de modéliser les interactions avec le code imprécis et impur. Notre système déductif est formalisé et prouvé mécaniquement correct à l'aide de l'assistant de preuve Coq. De plus, nous développons une implémentation prototypique dans le compila-

teur Scala et étudions des cas d'utilisation typiques de la programmation au niveau du type dans l'écosystème Scala.

Mots-clefs : programmation fonctionnelle, programmation impérative, vérification formelle, types dépendants, génération de contre-exemples, données modifiables partagées, cadres dynamiques

Contents

List of Figures

1 Introduction

Software plays a crucial role in modern life: Automation through computers has thoroughly transformed our economy, most of our relationships are now mediated by machines, and scientific progress routinely comes down to our ability to process ever larger quantities of data. Just among the most valuable publicly-traded companies, the majority are chiefly concerned with the development of software and hardware. But as the footprint of computers across society grows, so does the impact of their failures.

Unfortunately, designing software that does *the right thing*, and implementing it in a reasonable amount of time is hard. Over the years, industry and academia have put forth a range of approaches to prevent faulty software, or at least mitigate its negative impact. Software development processes address the social aspect of discovering a project's set of requirements, building consensus with stakeholders, and coordinating the implementation by a team of software engineers. On the technical side, software engineers have also found value in ideas that stem from more established engineering disciplines. Fault tolerance and redundancy, in particular, are central to the smooth operation of the distributed systems underlying today's cloud infrastructure.

The activity that software engineers spend most of their time on, however, is the design and implementation of individual software components. Software testing is the most widely-deployed approach to discover and address bugs early on in the development process. Ideas such as property-based testing and fuzzing further improve the efficacy of this approach, but the enormous number of possible inputs makes exhaustive testing intractable for all but the simplest software components.

Formal verification, on the other hand, establishes strong guarantees about the correctness of a software artifact by showing that an implementation adheres to some formal specification. While, in principle, this makes for a very attractive means of achieving reliability, very little software so far has been developed in tandem with machine-checkable correctness proofs. Arguably, specifying, implementing and proving programs correct is simply a difficult task, oftentimes not considered worth the additional development effort. Nonetheless, there are

early indications that this is changing, as the cost of software failure increases and the quality of tools improves. For instance, Microsoft has been successfully using formal verification to ensure the correctness of critical software components such as drivers [BCLR04], a hypervisor [LS09], complex distributed systems [HHK+15] and an implementation of the TLS protocol stack [BBDL+17]. Similarly, Amazon has begun investing significantly in the formal correctness of their cloud services [NRZ+15, BJA+21], bringing verification to a wider audience of software developers and the general-purpose languages they use.

In this thesis I try to address some of the issues holding verification back from being adopted more broadly, and propose solutions that are both ergonomic and apply to programming languages used in industry *today*. Below, I give a (very) brief history of software verification, introduce the central questions addressed by my thesis, and outline its concrete contributions.

1.1 State of the Art

Formal verification is one of the longest-standing challenges in software development. From the outset of computer science in the 1930s, pioneers such as David Hilbert, Kurt Gödel, Alonzo Church and Alan Turing were interested in which formal statements could be decided in a fully-automated way [HA28]. This included properties about programs themselves, such as termination, as posed in Turing's famous halting problem, and had to be answered in the negative [Göd31, Chu36, Tur37]. In fact, as Rice showed later, any non-trivial property about programs is undecidable [Ric53].

Over the following decades a number of mechanical and electronic designs realized the universal computing model previously envisioned by Church and Turing. In short time computers proved to be tremendously useful and versatile tools, assuming the right kind of programming, which remained more art than craft, much less a science. Just as today, programmers were largely undeterred by the apparent difficulty of producing correct programs.

Eventually, researchers such as Robert Floyd [Flo67a], Tony Hoare [Hoa69] and Edsger Dijkstra [Dij76] paved the way for a mathematically rigorous treatment of software. The 70s and 80s saw a flurry of advances in programming languages, both in terms of theory and in tooling available to practitioners. Among the most exciting developments besides higher-level compilers was the introduction of automated theorem provers. The first program verifiers, such as the Stanford Pascal Verifier [LGvH+79], lifted the systematic, but manual process of proving correctness in calculi like Hoare's, and provided a glimpse of how programs and proofs might be developed in a single unified process.

While much of the foundational technology to verify software existed by the late 80s, we didn't learn how to apply it to larger-scale projects until recently. Programmers, mostly coming out of the academic community, have since built and proven correct artifacts of impressive size, such as compilers [Ler09, KMNO14], an operating system kernel [KEH+09], and the encryption protocol stack underlying most of the internet [BBDL+17]. Many of these were

initially impressive merely in that they could be achieved at all, covering complex correctness properties and hundreds of thousands of lines of code. In effect, these projects established and improved bounds of what engineering effort is necessary to build trustworthy software.

In practice, successful verified software projects (like those mentioned above) tend to be written a) from scratch, b) with a clear idea of the specification, and c) implemented in verification-oriented languages. The latter roughly fall into two categories. Proof assistants, such as Coq [BC04], Isabelle [NPW02] or HOL4 [SN08], restrict themselves to languages with clear logical semantics, but have high barriers to entry, making it challenging to develop large software systems with them. On the other hand, verifiers for general-purpose programming languages make it easier to integrate with existing software ecosystems and apply standard development practices, but typically limit themselves to "well-behaved" features or tailor the language to be "verification-first" [SHK+16, Lei10, HMWC15].

1.2 Thesis

While much progress has been made over the past decades, the cost of developing verified software is still significant. Furthermore, the software industry and its tools are evolving rapidly, and verification technology has arguably failed to keep pace.

This thesis is my attempt to move verification one step closer to modern software development. I describe verification techniques that apply to high-level programming languages and provide significant automation. The particular context of my work is Scala, a high-level, mixed-paradigm programming language that combines object-oriented and functional features. Scala has seen broad adoption in industry and has influenced many other language designs in recent years, so it is my expectation that the techniques described herein will apply to a range of modern and future languages.

Broadly speaking, I am interested in two questions:

- How can we build program verifiers that support the full generality of imperative programs, but retain the simplicity of functional abstractions whenever possible?
- How can we leverage the type system to concisely specify programs in languages with subtyping and imperative features?

Program Verifiers for High-Level, Imperative Languages

To address the first question, my work expands upon a long history of automated theorem provers and verifiers. In particular, I build on Stainless, a program verifier that already supports a significant chunk of the Scala language, but only a very restricted fragment of imperative programs.

Languages combining functional with imperative programming are a somewhat rarely-explored area in the design space of verification. As noted above, large-scale verified software projects tend to be carried out using specialized toolchains, such as dedicated program verifiers or proof assistants. These tools typically specialize either in reasoning about imperative or functional programs, which puts verifiers such as F* [SHK+16] and Stainless into a rather unique position. Both have functional foundations, rely on SMT solvers like Z3 [dMB08b] to discharge proof obligations in first-order logic, but ultimately target languages with imperative idioms and thus require reasoning about the heap. More to the point, they resemble the kind of languages that programmers in industry actually develop large projects in, but simultaneously provide the functional abstractions that facilitate proofs.

Stainless further differentiates itself by combining some of the features typically found (separately) in verifiers and model checkers: it is not only sound for proofs, but also sound and complete wrt. counterexamples. In other words, the tool will automatically generate failing test cases if and only if such cases exist. Great care has been taken in Stainless' design to maintain this property in the presence of higher-order functions and state. That being said, Stainless has, so far, restricted itself to reasoning about unshared mutable data, ruling out programs that alias objects.

The first part of this thesis is dedicated to lifting this restriction by introducing a new reduction from imperative to functional programs with arrays. Specifying the behavior of effectful functions wrt. the heap typically requires the use of universal quantification. Alas, SMT solvers are generally unable to report models in the presence of such quantifiers, meaning that Stainless would lose its ability to report counterexamples. Our encoding sidesteps the issue by translating to a generalized, but decidable theory of arrays, extending Stainless to support the "full" imperative fragment of Scala without compromising its existing features.

Specification through Types

In the second part I turn my attention to the issue of *specifying* functional correctness properties through type annotations in languages with subtyping and imperative features.

Most program verifiers, including Stainless, employ explicit annotations at the level of functions, which are typically referred to as *contracts* (as popularized in Eiffel [Mey97]), or *pre-* and *postconditions*, denoting the particular conditions that must hold at function entry and exit. But there are good reasons to revisit this widespread choice of specification mechanism. Alternative verification systems such as LiquidHaskell [VSJ+14, VRJ13, VBJ15] have shown that there is great potential in leveraging the type system to express and propagate specifications, because it lowers the annotation burden on programmers, and allows more safety properties to be checked in a highly-automated manner.

Independently of these developments, communities of existing industrial programming languages are increasingly (ab)using their advanced typing features to recover the kind of precise

reasoning usually reserved to verifiers. For instance, Scala and TypeScript possess enough type-level machinery to encode arbitrary computation on types [Die17], which can, with some effort, be used to encode functional contracts. In Scala this has been realized by combining various type system features, such as higher-kinded types, F-bounded polymorphism, and implicit resolution.

The resulting style of type-level programming emulates dependently-typed languages [McB02], but leaves much to be desired in terms of performance and ergonomics. The second part of this thesis tackles this issue and proposes a principled solution to type-level programming in an existing, industrial programming language. In particular, we explore how one might retrofit dependent types in Scala, which notably requires dealing with subtyping and imperative constructs.

In summary, the existing mechanisms to solve both of these problems – *specification* and *checking* of programs mixing functional and imperative features – constitute good starting points with clean functional foundations. Throughout this thesis we leverage the idea that new, more powerful mechanisms can be implemented by reducing to the underlying functional core language.

Thesis Statement

Verification of general-purpose programming languages with both functional and object-oriented features is feasible and practical by reduction to a higher-order, functional language. Reductions can extend functional program verifiers to enable reasoning about shared mutable data by modelling heaps as arrays in a decidable first-order theory. Reductions can also model type-level programming in an imperative language with subtyping by using a translation to a dependently-typed, pure calculus that introduces additional parameters. Such reductions thus increase the expressiveness of existing type systems and scale the language fragment supported by traditional program verifiers.

1.3 Contributions

This thesis is structured in two technicals parts. Part I discusses how to encode imperative programs manipulating shared mutable data, while retaining a) simplicity of reasoning about functional code, and b) the ability to report counterexamples, i.e., concrete program inputs that violate a specification.

In particular, the first part of this thesis makes the following contributions:

- We describe a novel translation of frame conditions into quantifier-free formulas of combinatory array logic [dMB09], yielding a heap encoding that can reliably produce abstract counterexamples modulo function contracts.

- We show how to soundly incorporate into our approach the notion of first-class heaps, affording additional flexibility in proving lemmas about inductive heap predicates, while coming at essentially no additional cost in translation. First-class heaps also increase our system's expressive power in that they enable proofs of hyperproperties.
- We show how this encoding can be integrated with an expressive program verifier such as Stainless. Our implementation supports a mixture of imperative and functional features, including higher-order functions and generics, and provides dynamic frames as a modular specification mechanism.

Part II explores a new foundation for specifying and checking expressive properties through the type system of statically-typed, general-purpose languages such as Scala that combine functional with object-oriented programming.

In summary, the second part of this thesis contains the following contributions:

- We present our calculus $\lambda^{\mathrm{nd}}_{<:\{\}}$, which illustrates the novel elements of our extension to Scala. The type system of $\lambda^{\mathrm{nd}}_{<:\{\}}$ combines dependent types, subtyping and a generalization of singleton types to non-deterministic terms. We demonstrate how the interplay of these features allows us to leverage term-level programs for type-level computation.
- We provide a soundness proof of $\lambda^{\mathrm{nd}}_{<:\{\}}$ by reusing reducibility semantics of System FR [HVK19]. We prove the soundness of our rules. These proofs are mechanized using the Coq proof assistant [BC04].
- We show a concrete use-case of our system by implementing it as an extension of Scala, and using it to develop a strongly-typed wrapper for Apache Spark [ZXW+16]. Thanks to dependent types, we can statically ensure the type safety of database operations such as join and filter. We compare our implementation with an equivalent implicit-based one and show significant compilation time savings.

The work in Part I will appear at VMCAI 2022 under the name "Generalized Arrays for Stainless Frames". The work in Part II was done in close collaboration with

- Olivier Blanvillain, who contributed throughout the entire project, but especially to the prototypical implementation in Dotty (the Scala 3 compiler), and
- Jad Hamza, who took charge of the proof mechanisation in Coq which, in turn, informed subsequent changes in the calculus.

It has previously been published as a technical report [SBHK20].

2 Background: Two Approaches to Static Safety in Scala

In this chapter I will briefly tour some salient features of the Scala programming language, and Stainless, a program verifier for Scala. Our focus will be on two advanced, but complementary mechanisms for achieving safety guarantees statically in Scala: *type-level programming* and *contract-based verification*. I will demonstrate through a few examples what *can* and *cannot yet* be verified, in particular in the presence of mutable data. The identified shortcomings will set the stage for extensions to both approaches that I describe in later chapters of this thesis.

2.1 The Scala Programming Language

Scala is somewhat unique in its fusion of the object-oriented and functional programming paradigm. Its type system is built around subtyping and a notion of multiple inheritance (through classes and traits), yet unlike other object-oriented languages it encourages the use of purely-functional primitives wherever possible. In particular, its standard library is bifurcated along the lines of immutable and mutable data structures; programmers can (and often do) choose either depending on their use case. In addition, much thought in Scala's design has been put into maintaining interoperability with the JVM ecosystem, making it easy to interface with existing Java code.

It is worth noting that despite significant industry adoption, Scala is an actively evolving language. From its inception it served as a testbed for new ideas in language design, and it continues to do so with the release of major version 3 in May 2021. With the promotion of "Dotty" (the working title of Scala 3's new compiler) comes a host of changes, including many new language features (some of which we will touch upon later), and a completely new whitespace-based syntax. In the interest of accessibility and coherence between different chapters, I will present all code examples in the familiar Scala 2 syntax that Dotty continues to support.[1]

[1] Stainless does not yet possess a stable Scala 3 frontend, making Scala 2 syntax the natural choice for examples throughout this thesis.

Consider the following Scala program defining a data type `List[T]` that represents purely-functional linked lists in the classical way:

```
sealed trait List[T]
case class Nil[T]() extends List[T]
case class Cons[T](head: T, tail: List[T]) extends List[T]
```

In this example, `List[T]` corresponds to an algebraic data type (ADT) with exactly two constructors, the empty list `Nil`, and `Cons(x, xs)` where `x` represents the head of the list and `xs` its tail. The **`case`** keyword requests automatically derived implementations for equality, hashing and pretty-printing for `Nil` and `Cons`.

Scala's type system allows for more general type hierarchies than just ADTs. For instance, the `Option[T]` data type (analogous to Haskell's `Maybe`) found in the standard library is defined approximately as follows:

```
sealed trait Option[+T]
case object None extends Option[Nothing]
case class Some[+T](value: T) extends Option[T]
```

Several differences from our definition of `List[T]` stand out: Firstly, the type parameter `T` of `Option[T]` is prefixed by a plus sign, indicating that the data type is *covariant* in this parameter. Secondly, `None` was defined as a **`case object`**, indicating that only a single instance of `None` is desired (it is a so-called *singleton*). On the term level we can refer to its unique value as `None`, and analogously we can refer to the *singleton type* of that particular instance as `None.type`. Finally, `None` does not possess any type parameter of its own, but instead extends the type constructor `Option[_]` instantiated at type `Nothing`, i.e., the bottom of Scala's subtyping hierarchy. Thanks to covariance, we get that `None.type <: Option[Nothing] <: Option[T] <: Option[Any] <: Any` for arbitrary `T`, allowing us to use `None` polymorphically:

```
None: None.type   // None is typeable at its singleton type (its most precise type)
None: Option[Nothing]
None: Option[Boolean]
None: Option[Int]
None: Any         // None is well–typed (typeable at top type Any)
```

Similarly to `None` we could have defined `Nil` to be a **`case object`** extending `List[Nothing]` and made `List[T]` covariant. Beyond these examples Scala also allows for non-**`sealed`** **`trait`**s and **`class`**es, permitting client code to extend those parts of the type hierarchy.

We can define methods on `List[T]` as part of the **`trait`**. For instance, the following method `lastOption` computes the last element of the list, if any:

```
trait List[T] {
  def lastOption: Option[T] =
```

```
    this match {
      case Nil()          => None
      case Cons(x, Nil()) => Some(x)
      case Cons(_, xs)    => xs.lastOption
    } }
```

Much like Java, Scala is based on a type system built around subtyping, but in many ways goes beyond what is available in comparable languages. Among other things, Scala supports multiple inheritance through **`trait`**s, higher-kinded types, and various forms of metaprogramming through a combination of **`implicit`**s, reflection and macros. We will touch upon some of these features later, but, for the time being, let us simply note that Scala often allows multiple ways to solve the same problem. Case in point, the `lastOption` method shown above could have equally been expressed using virtual dispatch (though this would be somewhat wasteful given the closed nature of the `List` data type):

```
sealed trait List[T] {
  def lastOption: Option[T]   // abstract method
}

case class Nil[T]() extends List[T] {
  def lastOption: Option[T] = None
}

case class Cons[T](head: T, tail: List[T]) extends List[T] {
  def lastOption: Option[T] =
    tail match {
      case Nil() => Some(head)
      case _     => tail.lastOption
    }
}
```

The `lastOption` method also illustrates pattern matching, another staple of functional programming. While we know from other object-oriented languages that virtual dispatch or specialization can be used to encode higher-order functions, Scala naturally provides first-class support for the latter. As a simple example, consider this method that produces a new list by applying a function `f` to each element:

```
sealed trait List[T] {
  def map[U](f: T => U): List[U] =
    this match {
      case Nil()       => Nil()
      case Cons(x, xs) => Cons(f(x), xs.map(f))
    } }
```

This concludes our tour of Scala's basic features. In the remainder I will discuss two particular ways of achieving additional safety at compile-time.

2.2 Type-Level Programming

Scala provides a number of advanced type system features such as higher-kinded types, F-bounded polymorphism, generalized algebraic data types (GADTs), path-dependent types, type members, and implicit programming. While useful on their own, put together, these features enable a paradigm often called *type-level programming*. As a result, Scala programmers can enforce a surprisingly large range of properties just by using (and abusing) the type checker.

This approach is popular with DSL and library authors, as it allows them to produce safer public interfaces, and relies on nothing but the standard type checker to detect and report errors. Examples of such libraries include strongly-typed wrappers for database and data processing interfaces (by reflecting table schemas into Scala's types; for example, Frameless [Fra21]), Refined [Tho21], an encoding of refinement types with a pre-defined set of predicates and combinators, and Shapeless [Sab21], which provides various strongly-typed data structures and is intended to serve as a standard library for type-level programming in Scala.

As a simple example of such a data-structure we will consider heterogeneous lists [KLS04]. The idea is to generalize the concept of tuples to lists whose shape and distinct element types are tracked statically, so that type-level functions can operate upon them generically. Heterogeneous lists have long served as a concise benchmark for new typing features, illustrating the ergonomics, performance and safety of type-level programming approaches.[2]

We start by defining the corresponding data type `HList`:

```
sealed trait HList
case object HNil extends HList
case class HCons[H, T <: HList](head: H, tail: T) extends HList
```

For example, the tuple type `(Int, Boolean)` would be represented by an `HList` containing an `Int` followed by a `Boolean`:

```
// Analogous to '(1, true) : (Int, Boolean)'
HCons(1, HCons(true, HNil)) : HCons[Int, HCons[Boolean, HNil.type]]
```

Note that `HList` possesses exactly two constructors: The singleton `HNil`, and `HCons[H, T]`, which can only be instantiated with `T <: HList`. This rules out non-sensical instantiations such as `HList[Int, Int]`. Subtyping thus serves the role typically taken on by GADTs and promoted data types [YWC+12] in Haskell.

[2] See, for instance, the `Data.HList` library of Haskell and its discussion of alternative implementations.

2.2.1 The Scala 2 Idiom: Implicitly-Resolved Traits as Type-Level Functions

To leverage this generic representation of tuples we would now like to define type-level functions that, e.g., project out the n-th element of an `HList`, or concatenate two such lists. In Scala 2 this is typically achieved by means of *implicit resolution* [OBL+18], a form of type-based program synthesis that is deeply integrated with Scala's existing type checker.

The idea of the Scala 2 idiom is to encode type-level functions as **`trait`**s with type parameters for each input and one type-parameter for the output.

```
trait Concat[Xs <: HList, Ys <: HList, Out <: HList] {
  def apply(xs: Xs, ys: Ys): Out
}
```

Implicits can then be used to provide exactly those instances of `Concat[Xs, Ys, Out]` that informally satisfy `Xs ++ Ys == Out`, with `apply` performing the actual concatenation of two `HList`s on the term-level.

```
// Define implicit instances of Concat in the trait's companion object.
object Concat {
  implicit def concatNil[Ys <: HList]: Concat[HNil.type, Ys, Ys] =
    new Concat[HNil.type, Ys, Ys] {
      def apply(xs: HNil.type, ys: Ys) = ys
    }

  implicit def concatCons[X, Xs0 <: HList,
                          Ys <: HList,
                          Out0 <: HList]
    (implicit ev: Concat[Xs0, Ys, Out0]):
      Concat[HCons[X, Xs0], Ys, HCons[X, Out0]] =
        new Concat[HCons[X, Xs0], Ys, HCons[X, Out0]] {
          def apply(xs: HCons[X, Xs0], ys: Ys) =
            HCons(xs.head, ev.apply(xs.tail, ys))
        }
}
```

This allows us to define a function `concat` that takes two `HList`s and returns the concatenated `HList`, while retaining the exact shape and element types of the resulting list.

```
def concat[Xs <: HList, Ys <: HList, Out <: HList]
  (xs: Xs, ys: Ys)(implicit ev: Concat[Xs, Ys, Out]): Out =
    ev.apply(xs, ys)
```

Upon every invocation of `concat` with `xs: Xs` and `ys: Ys` we require an implicit derivation of `Concat[Xs, Ys, Out]`. This effectively coaxes the Scala type checker into performing the

computation for us at compile-time:

```
// Concatenate various hlists and check against the expected result types.
def example[Ys <: HList](x1: Int, x2: String,
                         ys: Ys, zs: HList): Unit = {
  val xs = HCons(x1, HCons(x2, HNil))
  concat(HNil, xs):
    HCons[Int, HCons[String, HNil.type]]
  concat(xs, xs):
    HCons[Int, HCons[String,
      HCons[Int, HCons[String, HNil.type]]]]
  concat(xs, ys):
    HCons[Int, HCons[String, Ys]]
  concat(xs, zs):
    HCons[Int, HCons[String, HList]]
}
```

While this approach to type-level computation has been very popular in the Scala community, it also presents several shortcomings:

- **Programming style:** Instead of expressing `concat` as a simple recursive function, we had to adopt a logic programming style. The actual term-level computation is expressed in terms of method calls to several anonymous classes.
- **Performance:** Implicit resolution incidentally does the right thing, but was never designed to drive general-purpose computation. The corresponding search algorithm in Scala's type checker makes for a rather inefficient interpreter, leading to significantly increased compilation times [Tor17, Can18, KMV19]. In addition, the desire to optimize for such use cases has led to the introduction of various heuristics over the years, making implicit search brittle once one wanders off the beaten path.
- **Accessibility:** Developing new type-level functions using this idiom is somewhat of a dark art among Scala developers. The use of implicits often makes it hard to understand why one's program fails to compile. Furthermore, to deal with functions more complicated than `concat`, additional tricks such as low-priority implicit instances and auxiliary type members may be needed.

2.2.2 Progress in Scala 3: Match Types

To address some of these issues, Scala 3 introduces *match types*, a type-computation mechanism similar to type families in Haskell. The idea is to add a new form of type resembling the term-level match expression, and apply normalization steps during type checking. Consequently, our `concat` example can be expressed as follows:

```
type Concat[Xs <: HList, Ys <: HList] <: HList =
  Xs match {
    case HNil.type => Ys
    case HCons[x, xs0] => HCons[x, Concat[xs0, Ys]]
  }

def concat[Xs <: HList, Ys <: HList](xs: Xs, ys: Ys): Concat[Xs,Ys] =
  xs match {
    case xs: HNil.type => ys
    case xs: HCons[_, _] => HCons(xs.head, concat(xs.tail, ys))
  }
```

The above snippet illustrates how match types enable type-level programming in a functional style, and eliminate much of the verbosity found in the implicit-based solution. The term-level function `concat` is expressed recursively, as is its type-level counterpart `Concat`. By ascribing `Concat[Xs, Ys]` as the result type, the user instructs Scala's type checker to ensure that `concat` in fact always produces an abstract `Concat[Xs, Ys]`. Conversely, the type checker will infer precisely-typed instances of `HList` at call sites of `concat`, meaning that our function `example` from before will pass without further modifications.

Match types were developed concurrently with the solution we present in the second part of this thesis. While they address many of the same painpoints, the match type approach stops short of unifying terms and types, as one would expect in a dependently-typed system. This means that acquiring precisely-typed term-level functions like `concat` still requires explicit duplication on the type level. In fact, given the current implementation of match types, `concat` must syntactically reflect the structure of `Concat`. If one wishes to express the function differently (for instance, in order to gain efficiency), they need to fall back to unsafe casting.

2.3 Contract-Based Verification

The techniques presented so far rely on a deep integration with the type checker, and shine when they are used to prove properties that enjoy a privileged representation in the type system. For instance, Scala's type checker is excellent at ensuring the structural properties of data: it does so efficiently and ergonomically, but can nonetheless provide useful error messages when things go wrong. Through some clever encodings (that we gave a taste of in the previous section) it can also prove more involved functional properties. For instance, we might use `HList`s to ensure that a `zip` function will only ever be applied to two lists of the same length. Unfortunately, proving more complex properties quickly becomes cumbersome without some notion of type-level computation and an automated proof system to go with it.

In this section I will give an overview of *contract-based verification*, an alternative approach to ensuring safety statically. I will introduce the underlying ideas by the example of Stainless, a verification pipeline for Scala. Stainless extracts Scala programs after type checking, and

progressively lowers function-level contracts to verification conditions, that are then discharged by an automated theorem prover. Unlike current type-level programming techniques in Scala, Stainless allows for the verification of deeper properties, such as checking that the implementation of a red-black tree is functionally correct.

2.3.1 Verifying Scala using Stainless

Scala's many features conspire to make it not only a language of great expressivity, but also a rather challenging target for formal verification. A program verifier for a realistic subset of Scala must provide support for its well-behaved functional fragment (including higher-order functions), but also for imperative features such as shared mutable data, virtual dispatch and open type hierarchies. Verifying Scala thus means solving many challenging aspects of program verification (perhaps with the notable exception of memory safety, which is a non-issue on the JVM).

Stainless is a particular verification pipeline [HVK19, KH21] that addresses a significant chunk of these challenges. At its heart, Stainless is an automated verifier for higher-order functional programs that progressively unfolds function invocations and uses SMT solvers such as Z3 [dMB08b] to discharge proof obligations over theories such as uninterpreted functions, algebraic data types and bit-vectors. Unlike many similar tools, Stainless avoids the use of universal quantification and triggers in its encoding to first-order logic. This is remarkable, in that it yields a system that is sound and complete for counter-examples, and yet provides a high degree of automation for proving functional correctness. In other words, Stainless requires relatively few annotations to make proofs go through (as we will see later), but – given enough time – can also produce concrete program inputs that violate a program's specification, if and only if those specifications are in fact violated.

As a simple example of what we mean by *contract-based*, consider the following Scala function computing `n` factorial:[3]

```
def fac(n: BigInt): BigInt = {
  require(n >= 0)
  if (n <= 1) 1 else n * fac(n - 1)
} ensuring (_ > 0)
```

Note that besides the usual recursive implementation, `fac` also features a precondition `require(...)` and a postcondition `ensuring(...)`. In particular, we assume that `fac` may only be called with non-negative `n`, and would like to ensure that it only ever returns positive results. If we invoke Stainless on this example, it will automatically verify that the implementation satisfies the given specification by virtue of an inductive proof. If, on the other hand, we try to prove some property about `fac` that does not hold, Stainless will eventually produce a counterexample for us.

[3] `BigInt` is Scala's built-in type for arbitrary-precision integers.

```
def facIsStrictlyMonotonic(n: BigInt): Unit = { require(n >= 0); () }
  ensuring (_ => fac(n) < fac(n+1))
```

For instance, in the above example we assert that `fac` is strictly monotonic. By unfolding the function definition on both sides the verifier quickly discovers that the property may be violated (for instance, when `n` equals zero).

2.3.2 Example: Encoding Types as Predicates

We will now consider an extended example to provide a tour of Stainless' features and demonstrate its expressive power. Taking inspiration from the kind of type-level computation seen in Section 2.2, we will, for the moment, avoid most of Stainless' support for Scala typing features, and show how one might emulate various type constraints through contracts.

The combination of algebraic data types and higher-order functions alone (which forms the core of Stainless) provides a highly-expressive language into which we can encode many of the advanced type system features. Below we will consider the example of Lisp-like data upon which we impose strongly-typed structures such as homogeneous lists.

We start by defining a simplified "top" type that, as in Lisp, distinguishes between `Nil`, pairs (`Cons`) and `Atom`s:

```
sealed trait Top
case object Nil extends Top
case class Cons(fst: Top, snd: Top) extends Top
case class Atom(value: Int) extends Top
```

Values of type `Top` are Lisp-like in that they describe s-expressions, making no distinction between different data types. For simplicity, we assume `Atom`s merely store some 32-bit integer.

Recursive Types as Inductive Predicates. We can impose structure upon a Lisp-like value using an inductive predicate such as `isList` below:

```
// A predicate describing lists embedded in the Top type hierarchy.
def isList(xs: Top): Boolean =
  xs match {
    case Cons(_, xs0) => isList(xs0)
    case Nil          => true
    case _            => false
  }
```

Stainless allows us to simply define such predicates as pure functions and then refer to them in contracts. For instance, we might require that the input given to a function computing the `length` of a list satisfy this predicate:

```
def length(xs: Top): BigInt = {
  require(isList(xs))
  xs match {
    case Cons(x, xs0) => 1 + length(xs0)
    case Nil          => 0
    // case Atom(_) => *no need*!
  }
}
```

Stainless will generally ensure exhaustiveness of pattern matching, and report a counterexample when a feasible scrutinee is unhandled. We are nonetheless allowed to omit a case for `Atom`s in the above example – Stainless infers by unfolding the precondition `isList(xs)` that `xs` will necessarily be either `Nil` or `Cons`. By the same mechanism it also deduces that `isList(xs0)`, establishing the precondition necessary for the recursive call to `length(xs0)`.

Following this approach of using `isList` to stand in for a dedicated `List` type, we can define some more interesting functions such as list concatenation:

```
// Concat, maintaining list property and proving another property (length)
// concat : List => List => List
def concat(xs: Top, ys: Top): Top = {
  require(isList(xs) && isList(ys))
  xs match {
    case Cons(x, xs0) => Cons(x, concat(xs0, ys))
    case Nil          => ys
  }
} ensuring (res => isList(res) &&
  length(res) == length(xs) + length(ys))
```

Compared to the `length` function, we face two additional challenges: Above all, we would like to establish that the data structure returned from `concat` is again a list, i.e., `isList(res)`. While we are at it, we might also want to provide a stronger guarantee about the resulting list; namely, that its length `length(res)` equals the length of the two input lists. Since both of these properties follow directly from an inductive proof structured like the recursion in `concat`, Stainless can deduce this fully automatically.

Type Constructors as Higher-Order Functions. So far we have been using inductive predicates (i.e., pure, recursive functions) to describe recursive, but monomorphic data structures. We can go one step further and also emulate parametric and refinement types. The analogy so far has been to replace types by predicates. Since predicates are merely functions, they remain first-class citizens in Stainless:

```
type Pred = Top => Boolean  // A predicate
```

If we take the analogy further, then higher-order functions of type `Pred => Pred` are to predicates `Pred` as type constructors `* => *` are to proper types `*`. In other words, we can view a type constructor like `List[_]` as a *predicate transformer* which takes a predicate `p` (representing type argument `T`) and produces another predicate (representing the list type instantiated at `T`):

```
// Like 'isList', but ensuring predicate 'p' for each element:
// isListWith : Pred => Pred
def isListWith(p: Pred)(xs: Top): Boolean =
  xs match {
    case Cons(x, xs0) => p(x) && isListWith(p)(xs0)
    case Nil          => true
    case _            => false
  }
```

This definition allows us to parameterize the list predicate further by restricting each individual element. For example, we could express "lists containing only atoms of positive value" by first defining the auxiliary predicate

```
def isPosAtom(x: Top): Boolean =
  x match {
    case Atom(value) => value > 0
    case _           => false
  }
```

and then using the composite predicate `isListWith(isPosAtom)(xs)`. When accessing an element `x` of such a list, Stainless can now deduce that `x > 0` and use this to prove the safety of other operations (e.g., that dividing by `x` is well-defined). In effect, constraining a function parameter `xs: Top` with the above composite predicate is much like declaring it with a composite type `List[{a: Atom where a.value > 0}]` in a system that supports parametric polymorphism (`List[...]`), subtyping (using the constructor `Atom` as a dedicated type, implying a type test) and refinement types (`{... where a.value > 0}`).

Analogously to type parameters, we must now parameterize functions by the predicate `p`:

```
// Concat, maintaining T–list property
// concat : forall T. List[T] => List[T] => List[T]
def concat(p: Pred)(xs: Top, ys: Top): Top = {
  require(isListWith(p)(xs) && isListWith(p)(ys))
  xs match {
    case Cons(x, xs0) => Cons(x, concat(p)(xs0, ys))
    case Nil          => ys
  }
} ensuring (res => isListWith(p)(res))
```

On the other hand, `length` only requires `xs` to be a list, so if we would like it to remain unchanged we could redefine `isList` in terms of `isListWith`:

```
def tru(x: Top): Boolean =
  true

def isList(xs: Top): Boolean =
  isListWith(tru)(xs)
```

Proving Lemmas. Hanging onto `isList` and leaving `length` unchanged comes at a price. Stainless cannot automatically deduce `isList` from `isListWith(p)`, so we have to provide a lemma whenever both predicates are in play. We run into this issue if we try to prove the original variant of `concat` that comes with additional guarantees about the length of the resulting list:

```
// Essentially, List[T] <: List[Any].
def lemmaListWithIsList(p: Pred)(xs: Top): Unit = {
  require(isListWith(p)(xs))
  xs match {
    case Cons(_, xs0) => lemmaListWithIsList(p)(xs0)
    case _            => ()
  }
} ensuring (_ => isList(xs))

// Concat, maintaining T–list property and proving another property (length)
def concat(p: Pred)(xs: Top, ys: Top): Top = {
  require(isListWith(p)(xs) && isListWith(p)(ys))
  val res = xs match {
    case Cons(x, xs0) => Cons(x, concat(p)(xs0, ys))
    case Nil          => ys
  }
  lemmaListWithIsList(p)(xs)
  lemmaListWithIsList(p)(ys)
  lemmaListWithIsList(p)(res)
  res
} ensuring (res => isListWith(p)(res) &&
  length(res) == length(xs) + length(ys))
```

Note that in the postcondition of `concat` we invoke `length` with `xs`, `ys` and `res`, as before, but `concat`'s precondition now refers to `isListWith(p)` rather than `isList`. An issue arises: Stainless knows that `isListWith(p)(xs)` holds, but not whether `isList(xs)`, i.e., `isListWith(tru)(xs)`. The role of `lemmaListWithIsList` here is to, effectively, prove covariance of `List[_]`, and correctly deduce that `List[T] <: List[Any]`, yielding `isList(xs)` in

the postcondition of the lemma.

Unfortunately, Stainless cannot automatically insert invocations of `lemmaListWithIsList`, so we had to provide hints in the above example. On the other hand, a more uniform encoding would have circumvented this issue altogether: we could have instead added a predicate parameter `p` to `length` and used `isListWith(p)` in its precondition (analogously to typing `length : forall T. List[T] => BigInt`), removing the necessity for a lemma. Conversely, one could also define `length` as a total function on `Top`, returning some dummy value for non-lists and remove the precondition entirely. Stainless provides considerable freedom in structuring one's proofs, and different encodings may be desirable depending on the properties we are trying to show overall.

Nesting via Partial Application. Finally, partially-applied functions allow us to nest predicates, and thereby also encode composite applications of type constructors. For instance, we might want to define `flatten` which concatenates a list of lists into one "flat" list:

```
// flatten : forall T. List[List[T]] => List[T]
def flatten(p: Pred)(xss: Top): Top = {
  require(isListWith(isListWith(p))(xss))
  xss match {
    case Cons(xs, xss0) => concat(p)(xs, flatten(p)(xss0))
    case Nil            => Nil
  }
} ensuring (res => isListWith(p)(res))
```

Note that we encoded `List[List[T]]` as `isListWith(isListWith(p))(xss)`, i.e., we partially applied `isListWith` to provide the resulting predicate to the outer `isListWith`.

Discussion. Stainless verifies the entirety of examples seen so far in a matter of seconds, providing static checks that are both efficient and highly expressive compared to the alternatives available in Scala's existing type checker. For this subsection we purposefully chose the encoding of typing features to give a verification benchmark whose desired properties are intuitively clear, but that also illustrates some of the trade-offs involved (i.e., a more specialized verifier like a type checker will do a better job at propagating constraints than Stainless).

One significant drawback of our encoding of generics is that it turns type parameters into term parameters which are then additionally passed to functions at runtime. Given our usage of `p: Pred` above this is unnecessary, however – after all, only the contracts, but not the result of these functions depend on `p`. Stainless thus provides some facilities for erasing computationally irrelevant parts. Firstly, we can mark certain parameters and code blocks as **`@ghost`**, which prompts Stainless to check that they indeed do not influence the result. Secondly, contracts (the **`require`** and **`ensuring`** clauses) and **`@ghost`** code are removed from

generated code through an integration with Scala's build tool.

Finally, taking type information into account is extremely useful when verifying programs, so Stainless also supports many of Scala's typing features out of the box. That is, we could have equally defined `List[T]` as in Section 2.1 and made functions such as `length` and `concat` parametric in `T`, and Stainless would have nonetheless been able to infer the exhaustivity of pattern matches et cetera. Internally, Stainless uses a series of lowerings to transform programs from an expressive type system approaching Scala's to one of possibly-parametric algebraic data types and refinement types. By the time Stainless emits verification conditions to an SMT solver, the resulting encoding bears some similarity to the "manual" approach seen in this section.

2.4 Dealing with State

One particular area for which neither Scala's type system nor Stainless provide comprehensive reasoning capabilities is imperative code – in particular in the presence of shared mutable data structures.

Scala's type checker generally treats mutable data by over-approximation: Unlike immutable bindings (`val x: T`) mutable fields and variables (`var x: T`) are not considered *stable* prefixes under Scala's notion of dependent object types [RA16]. This means that one cannot refer to `x` in a singleton type and inference will never yield such precise types either, but instead widen to the underlying type `T`. Ideas like type state [SY86] and session types [HVK98] have been implemented successfully in Scala [CP16, SY16], but only offer coarse-grain guarantees.

On the other hand, Stainless has historically provided precise reasoning about imperative programs assuming the absence of aliasing. In particular, Blanc [Bla17] introduced initial support for local mutation and an effectively linear fragment of imperative, heap-manipulating programs in Leon, which was expanded upon in its successor, Stainless.

In the remainder I will take our well-worn linked list example to illustrate the benefits and issues with shared mutable data. Recall the definition of the polymorphic `List[T]` datatype we saw in Section 2.1:

```
sealed trait List[T]
case class Nil[T]() extends List[T]
case class Cons[T](head: T, tail: List[T]) extends List[T]
```

We can efficiently access the head and the tail of a given list, or prepend an element to it.

```
val xs: List[T] = ???
xs.head           // O(1)
xs.tail           // O(1)
Cons(???: T, xs)  // O(1)
```

Note that `???[S]` is a Scala idiom to indicate an unimplemented computation of type `S`. At runtime, this expression will simply throw a `NotImplementedError`. In the above code snippet the two occurrences of `???` rely on Scala's local type inference to figure out the required type instantiations, namely, at `List[T]` and `T`.

Certain other list operations, such as accessing the last element, require a traversal of the entire data structure, but can be memoized once computed, and thus retain referential transparency. For convenience, Scala provides **`lazy val`** fields whose value is computed once upon the first access to the field, and stored within the object for later accesses.

```
sealed trait List[T] {
  // The last element of the list, if any.
  // Amortized runtime in O(1)
  lazy val lastOption: Option[T] =
    this match {
      case Nil()          => None
      case Cons(x, Nil()) => Some(x)
      case Cons(_, xs)    => xs.lastOption
    }
}
```

Unfortunately, other operations such as appending elements and concatenation require rebuilding the entire list, and a naive solution thus ends up taking linear time. For instance, consider the canonical implementation of appending to a list:

```
trait List[T] {
  def append(y: T): List[T] =
    this match {
      case Nil()       => Cons(y, Nil())
      case Cons(x, xs) => Cons(x, xs.append(y))
    }
}
```

To append an element to a list, we traverse the entire struture, create a singleton list containing only the new element `y`, and finally prepend each of the existing elements in reverse order. Not only is the asymptotic runtime of `append` in `O(n)` (where `n` is the length of the list), but it also requires keeping `O(n)` additional stack space when we reach the base case.[4]

On modern architectures this performance issue is seriously compounded by a phenomenon known as *pointer chasing*: Since most architectures nowadays employ multi-level cache hierarchies, the cost of accessing heap memory is non-uniform, and programs that operate on

[4] This can, in principle, be alleviated by implementing `append` in terms of a tail-recursive list reversal, followed by a prepend and another list reversal. Obviously, the reduction in memory consumption is paid for by a higher constant factor in the linear runtime.

a small working set of data may execute significantly faster. Expert programmers go out of their way to design programs that achieve a high level of such *data locality*, often yielding orders of magnitude speed-ups. Even worse than the pointer chasing, the above implementation of `append` also induces a linear number of additional allocations (proportional to the number of existing list elements).

More often than not the list we are extending is simply discarded after the append operation. Creating an entirely new, almost identical list is rather wasteful in those cases. While linear type systems [Wad90b] and related optimization techniques [UdM19] might try to identify such opportunities automatically, it is still more typical that programmers take manual control, and implement a version of `append` that modifies a mutable data structure *in-place*.

Consider the following definition of a `Node[T]` data type that is iso-morphic to the purely-functional type `List[T]` shown before:

```
case class Node[T](value: T, next: Option[Node[T]])
```

We can represent every `List[T]` as an `Option[Node[T]]` and vice-versa. That is, we map `Nil()` to `None`, `Cons(x, Nil())` to `Some(Node(x, None))` and so forth. `Node[T]` is closer to the definition of a linked-list node typically seen in imperative programming languages such as the `C` family. To achieve in-place updates of the list structure the `next` field ought to become mutable. In Scala we can explicitly opt into mutability by declaring the `next` field as **`var`**:

```
case class Node[T](value: T, var next: Option[Node[T]])
```

We additionally define a wrapper type that encapsulates the initial node and can meaningfully represent empty lists:

```
case class LinkedList[T](var root: Option[Node[T]])
```

Already, this new data type allows us to define `append` as an operation that mutates a `LinkedList[T]` in-place. But we can do better: if we include a shortcut to the last element of the list, that allows us to sidestep the linear traversal in `append`.

```
case class LinkedList[T](var root: Option[Node[T]],
                         var last: Option[Node[T]])
```

At this point we can efficiently implement `append` for `LinkedList[T]`:

```
def append[T](ll: LinkedList[T], newValue: T): Unit = {
  val newNode = Some(Node(newValue, None))
  if (ll.last == None)
    ll.root = newNode
  else
    ll.last.get.next = newNode
  ll.last = newNode
```

```
}
```

Unfortunately, we have now introduced two complications in the verification of `append`. Firstly, keeping track of modifications to a `ll: LinkedList[T]` has become considerably harder, since the last node will be *aliased*, i.e., reachable both through `ll.last` and through a series of references starting from `ll.root` and following repeated `.next` fields. Modifications to one will thus affect our interpretation of the other. Secondly, to make sense of how `ll.last` relates to `ll.next` and prove anything of interest about `append`, we need to maintain an invariant that characterizes `ll.last`.

Stainless in its existing form cannot help us reason about `LinkedList[T]`, because of the first issue, shared mutable data. In Part I of this thesis we address this shortcoming without sacrificing any of Stainless' other strengths like reasoning about higher-order functions and counterexample-completeness. In fact, we leverage its support for functional abstractions to describe inductive heap predicates like those needed to characterize `ll.last` (our second issue), and ultimately prove correctness of procedures like `append`.

Part I

Decidable and Expressive Reasoning about Heaps in Stainless

In this first part we will see how a verifier such as Stainless can be extended to support reasoning about programs with shared mutable state, while retaining the ability to produce counterexamples.

Formal verification of programs with shared mutable data structures is a long-standing problem. Among the most promising techniques used in today's verification tools are separation logic and dynamic frames. Separation logic [ORY01] with bi-abduction [CDOY11] has proved practical; its variant is implemented in the Infer tool [DFLO19] used by Facebook. It is also a common framework for foundational semantic-based approaches for reasoning about state inside the Coq proof assistant [JKJ+18]. On the other hand, we are attracted to dynamic frames [Kas06] because they are both semantically straightforward and expressive. Tools that embrace them, such as Dafny [Lei10], were used to verify complex software systems at Microsoft [HHK+15]. Separation logic and dynamic frames are closely related and one can view separation logic as a logical framework that infers sets that represent dynamic frames in certain circumstances. This was first illustrated by the VeriCool verifier [SJP12], rigorously analyzed in subsequent research [PS12], and used to relate subsets of VeriFast [JSP+11] to Chalice [JS13, LMS09].

In the following we introduce an alternative approach for reasoning about mutable programs and present its realization in the Stainless verifier [HVK19] for a subset of the Scala programming language [OSV19]. Like the dynamic frames approach, we use constrained sets of objects to specify frame conditions. Like Dafny, our tool uses SMT solvers to establish properties instead of dedicated symbolic execution for heap-manipulating programs as in several other approaches [BCO05, DPJ08, JSP+11, MSS16]. We also model the heap as a function from storage locations to values.

However, our encoding of frame conditions is different from the one in Dafny. Whereas Dafny makes use of *universal quantifiers with triggers* to encode frame conditions (expressing that *all* non-modified locations remain the same), we avoid quantifiers and instead use the *generalized theory of arrays* [dMB09] of Z3. Notably, this expressive array theory retains completeness guarantees for satisfiability checking of quantifier-free formulas even in the presence of model-based theory combination [dMB08a] with other decidable theories. Thanks to our new encoding and the decision procedures of Z3, our verification tool can report meaningful counterexamples for invalid properties, even in those cases where the bodies of methods are abstracted by their modifies clauses. In contrast, SMT solvers either refuse to report counterexamples to satisfiability for formulas with universal quantifiers, or permit extraction of assignments that may or may not be witnesses to satisfiability. Unlike Dafny, which reduces programs to a guarded-command language Boogie [BCD+05], our approach reduces imperative code to recursive functional programs that manipulate data types supported by the Z3 SMT solver [dMB08b], building on the existing Stainless infrastructure. While Stainless could already deal with imperative constructs [Bla17], the supported fragment did not permit any aliasing. In contrast, the new encoding we describe enables Stainless to verify shared mutable data structures.

Our approach reduces verification conditions to functional programs but need not encode immutable algebraic data types using the heap. Read-only functions do not return a heap in our encoding, whereas functions that do not read mutable references do not even take a heap argument. The result is a better verification experience on a mix of purely functional and mutable code, compared to a more uniform encoding. This feature enables users to leverage the expressive power of recursive functional programming in implementation and specification, and encourages the use of executable specifications. Following this paradigm, we further allow users to define inductive heap predicates as `Boolean`-typed recursive functions. Lemmas about such predicates typically require inductive proofs and the ability to explicitly relate to states at different program points. We propose first-class heaps as a solution which provides the necessary fine-grained control and is readily expressible using our approach.

Outline

In the remainder we rely on running examples to demonstrate how our technique enables the specification and verification of Scala programs containing mutable data (Chapter 3). We then sketch our encoding into recursive functions via the extended array theory [dMB09] (Chapter 4), and discuss our experience with the tool on verifying shared mutable data structures (Chapter 5). Our implementation is part of Stainless[5] and can be tested on examples in `frontends/benchmarks/full-imperative/` via the `--full-imperative` flag.

[5] An archived artifact and the current version of Stainless are available at https://zenodo.org/record/5683321 and https://github.com/epfl-lara/stainless, respectively.

3 Verifying Mutable Data in Scala

3.1 First Example: Stack

As a simplest example to illustrate a mix of functional and imperative programming, Figure 3.1 presents a mutable stack implementation using the textbook singly-linked list. The code is valid Scala accepted by the Scala 2.12/2.13 compilation pipeline given appropriate library imports. The data structure is simple to specify: a minimal specification would only include reads and modifies clauses, with bodies of functions themselves serving as specifications. While in general functions might read and modify arbitrary computable `Set`s of objects, here we only access **`this`**, i.e. the instance of `Stack` itself.

Figure 3.1 extends such basic specification by introducing the abstraction function `list` and calling it in postconditions (**`ensuring`**) to re-state the precise effect of the function. For instance, the postcondition of `push` states that `list == a :: old(list)`, meaning that the result of invoking parameter-less abstraction function `list` in the post-state is structurally equivalent (`==`) to element `a` cons-ed (`::`) with `list` evaluated in the pre-state (`old(list)`). The proofs of all these conditions in `push` and `pop` are trivial and our system performs them in a fraction of a second. The clients can reason about the behavior of stack by referring to the immutable `list`, which is suited for inductive proofs, much like such list data types in proof assistants Coq [BC04] and Isabelle [NPW02]. Users can create shared references to such mutable stacks, which goes beyond what was possible with the previous, unique-mutable-reference model of Stainless, inherited from Leon [Bla17, Ch. 3].

3.2 Extended Example: Map on a Tree

Moving to a slightly more complex example, Figure 3.2 shows a binary tree data whose interior nodes are immutable but whose leaves are mutable and store values of generic type `T`. We support a fragment of Scala with functional features (such as pure first-class functions) as well as imperative features (mutable fields) and object-oriented features (traits and dynamic dispatch). For any class, users explicitly opt into mutability and heap reasoning by inheriting

```
1  case class Stack[T](private var data: List[T]) extends AnyHeapRef
2  {
3    // An abstraction function describing the mutable stack as an immutable list.
4    def list: List[T] = {
5      reads(Set(this))
6      data
7    }
8
9    def push(a: T): Unit = {
10     reads(Set(this))
11     modifies(Set(this))
12
13     data = a :: data    // executable code
14   } ensuring(_ ⇒ list == a :: old(list))
15
16   def pop: T = {
17     reads(Set(this))
18     modifies(Set(this))
19     require(!list.isEmpty)
20
21     val n = data.head   // executable code
22     data = data.tail    // executable code
23     n                   // executable code
24   } ensuring (res ⇒ res == old(list).head && list == old(list).tail)
25 }
```

Figure 3.1 – A mutable stack.

```
1  case class Cell[T](var value: T) extends AnyHeapRef
2
3  case class Leaf[T](data: Cell[T]) extends Tree[T]
4  case class Branch[T](left: Tree[T], right: Tree[T]) extends Tree[T]
5
6  sealed abstract class Tree[T]
7  {
8    // The set containing all cells in the tree.
9    @ghost def repr: Set[AnyHeapRef] = this match {
10     case Leaf(data) ⇒ Set[AnyHeapRef](data)
11     case Branch(left, right) ⇒ left.repr ++ right.repr
12   }
13
14   // A minimally−specified map function over the tree.
15   def tmap(f: T ⇒ T): Unit = {
16     reads(repr)
17     modifies(repr)
18
19     this match {
20       case Leaf(data) ⇒ data.value = f(data.value)
21       case Branch(left, right) ⇒ left.tmap(f); right.tmap(f)
22     }
23   }
24 }
```

Figure 3.2 – A tree with mutable leaves and a parallelizable in-place map, including read and write frame conditions. The ++ symbol denotes union of sets, as in Scala.

from `AnyHeapRef`. For instance, in our example the class `Tree` inherits from `AnyHeapRef`. It is also marked as **`sealed`**, indicating that all of `Tree`'s subclasses are defined locally (as opposed to Scala's default behavior of keeping type hierarchies *open*). In effect, `Tree` constitutes an algebraic data type with constructors `Leaf` and `Branch`.

Our focus is the method **`def`** `tmap(f: T ⇒ T)` on the `Tree` class, which applies an in-place transformation `f` to all leaf cells. For example, given a `tree: Tree[BigInt]`, invoking `tree.tmap(n ⇒ n + 1)` increments the values in all the leaves of `tree` by one. The method recursively traverses the tree and updates all cells upon reaching the leaves.

Verifying Effects. Figure 3.2 is also a minimally-specified program accepted by our tool, which automatically verifies the conformance of `tmap` to its declared effects. The **`reads`** clause indicates that the only mutable references that `tmap` reads are given by the value returned from auxiliary function `repr`, which computes the set of mutable cells in a given tree. Similarly, **`modifies`** indicates that these are the only sets the method is allowed to modify, which means that all other mutable objects remain the same after a call to `tmap`. The **`@ghost`** annotation ensures that the `repr` function is not accidentally executed, but can only be used in specifications that are erased at run time.

If we try to omit a reads or modifies clause, or incorrectly define `repr` to not descend into subtrees, the tool reports a counterexample showing that the specification **`reads`** or **`modifies`** is violated, with a message such as

```
tmap  body assertion: reads of Tree.tmap   invalid
```

pointing to an undeclared effect, e.g. on line 20 of Figure 3.2.

Counterexamples. Our approach enables the generation of counterexamples on the basis of function contracts alone. Consider the following `test` method:

```
def test[T](t: Tree[T], c: Cell[T], y: T) = {
  reads(t.repr ++ Set[AnyHeapRef](c))
  modifies(t.repr)

  t.tmap(x ⇒ y)
} ensuring(_ ⇒ c.value == old(c.value))
```

If we mark `tmap` using the **`@opaque`** annotation to prevent it from being unfolded and try to verify `test`, the system reports a counterexample, such as this one:

```
Found counter−example:
  t: Tree[T]          → Leaf[Object](HeapRef(12))
  c: HeapRef          → HeapRef(12)
```

```
y: T                                → SignedBitvector32(1)
heap0: Map[HeapRef, Object] →
  {HeapRef(12) → Cell(Cell[Object](SignedBitvector32(0))), * →
 SignedBitvector32(2)}
```

indicating that, when `tmap` is approximated with its effects, the **`ensuring`** clause can be violated when tree `t` contains precisely the reference `c`.

Tools such as Dafny have difficulties in discovering such counterexamples, as they rely on an encoding of frame conditions that involves quantifiers. Aiming for soundness of counterexamples, the underlying SMT solvers may refuse to produce any output or, in some cases, may produce an assignment that is not guaranteed to be a model. This limitation is due to the fact that certifying that a model exists in the presence of general quantifiers is a very difficult problem. Generalized arrays [dMB09] avoid it by "building in" restricted forms of quantifiers into the semantics of pointwise (map) operators, improving the predictability.

Verifying Functional Correctness. To illustrate specification of stronger correctness properties, we show that `tmap` behaves like `map` on purely functional lists. This stronger specification of `tmap` is in the **`ensuring`** (postcondition) clause of the version of `tmap` in Figure 3.3 (line 18). The property is interesting because it gives us assurance of correctness while being able to write code that reuses memory locations and permits parallelization. The property is expressed by defining an abstraction function [AL91] `toList` that maps the tree into the sequence of elements stored in its leaf cells. (The purely functional `List` data type and the `map` function on lists are defined in the standard library of Stainless.) To prove the **`ensuring`** clause, it is necessary to introduce a precondition for `tmap`, expressed using the construct **`require`**`(valid)`. The `valid` method returns true when all subtrees store disjoint cells. The `tmap` method may then only be called when this predicate holds. The assertion on line 14 follows directly from `valid` and expresses disjointness of the side effects of calls on line 15.

In many cases our tool can automatically prove properties of interest thanks to SMT solvers and the unfolding algorithm of Stainless. For instance, the `valid` method (which we use to establish separation of subtrees) does not depend on the content of mutable cells, but only on the identity of references. Our tool checks this independence thanks to the absence of **`reads`** and **`modifies`** clauses in the signature of `valid`. Because it does not depend on mutable state, `valid` trivially continues to hold after each invocation of `tmap` on line 15.

On the other hand, showing complex properties such as functional correctness may require more elaborate reasoning. The first challenge in our example is to establish on line 16, *after* the modifications have taken place, the correctness property we desire for each subtree, i.e., `left.toList == oldList1.map(f)` and `right.toList == oldList2.map(f)`. This requires using the heap separation between `left` and `right` (witnessed by `valid`) to deduce that the two recursive calls are in fact entirely independent of another. This, in turn, requires taking into account `tmap`'s **`modifies`** clause, which states that only objects in `repr` are modified.

```
1  def tmap(f: T ⇒ T): Unit = { // strong specification
2    reads(repr)
3    modifies(repr)
4    require(valid)
5    @ghost val oldList = toList
6
7    this match {
8      case Leaf(data) ⇒
9        data.value = f(data.value)
10       ghost { check(toList == oldList.map(f)) }
11
12     case Branch(left, right) ⇒
13       @ghost val (oldList1, oldList2) = (left.toList, right.toList)
14       assert(left.repr ∩ right.repr == ∅)
15       left.tmap(f); right.tmap(f)
16       ghost { lemmaMapConcat(oldList1, oldList2, f) }; ()
17   }
18 } ensuring (_ ⇒ toList == old(toList.map(f)))   // main property
19
20 def valid: Boolean =   // tree invariant: subtrees store disjoint cells
21   this match {
22     case Leaf(data) ⇒ true
23     case Branch(left, right) ⇒
24       left.repr ∩ right.repr == ∅ &&
25       left.valid && right.valid
26   }
27
28 def toList: List[T] = { // abstraction function
29   reads(repr)
30   this match {
31     case Leaf(data) ⇒ List(data.value)
32     case Branch(left, right) ⇒ left.toList ++ right.toList
33   }
34 }
35
36 def lemmaMapConcat[T,R](xs: List[T], ys: List[T], f: T⇒R): Unit = {
37   xs match {
38     case Nil() ⇒ ()
39     case Cons(_, xs) ⇒ lemmaMapConcat(xs, ys, f)
40   }
41 } ensuring (_ ⇒ xs.map(f) ++ ys.map(f) == (xs ++ ys).map(f))
```

Figure 3.3 – Functional correctness of the `tmap` method including the abstraction function, the invariant, and a proven lemma about purely functional lists. We use ∩ to display intersection of sets, and use ∅ for the empty set of heap references `Set[AnyHeapRef]()`. The `++` symbol denotes concatenation of functional lists and union of sets, as in Scala.

In previous works such a clause is encoded in one of two ways. Systems such as Dafny encode frame axioms as quantified first-order formulas and rely on *triggers* to automate their instantiation. In contrast, separation logic verifiers explicitly control the choice of frame, and thus move the burden of instantiations out of the SMT solver. We propose a third solution, which is to encode the frame conditions as quantifier-free assumptions in array theory, injected at each function call site. Our approach avoids the need for quantifiers, but retains the automation of SMT solvers.

Despite that automation and the decidability of the generalized array theory, the size and complexity of SMT formulas may overwhelm the solver. In such cases the user can add auxiliary assertions, e.g., expressed through **`assert`** and **`check`** statements in Figure 3.3. Furthermore, certain properties may require explicit guidance on inductive proofs when reasoning does not follow the pattern of functions that are iteratively unfolded. In such cases, we need to introduce lemmas and prove them using recursion to express inductive arguments, as with `lemmaMapConcat` defined in lines 36-41 and instantiated on line 16. This lemmas is independent of any state reasoning and would naturally fit in a standard list library. With these specifications and hints in place, our tool successfully verifies the functional correctness of `tmap`.

3.3 First-Class Heaps

For some proofs it is useful to directly refer to and manipulate the heap states at different points in the program. In our system's surface language we expose heaps as first-class values of abstract type `Heap`, and our standard library contains several primitives to manipulate such values: a function `Heap.get` which returns the current implicit heap, a primitive `h.eval(e)` which evaluates expression `e` in the context of heap `h`, and the function `Heap.unchanged(s, h0, h1)` which evaluates to `true` iff there exists no object `o` in the set `s: Set[AnyHeapRef]` such that heaps `h0` and `h1` interpret `o` differently (in the shallow sense).

For instance, we might want to re-establish an inductive heap predicate after having modified a node-based data structure:

```
case class Node(var next: Option[Node]) extends AnyHeapRef

def sll(nodes: List[Node]): Boolean = {
  reads(nodes.content.asRefs)
  nodes match {
    case Cons(node1, rest @ Cons(node2, _)) ⇒
      node1.next == Some(node2) && sll(rest)
    case _ ⇒ true
  }
}
```

In the above example we have a heap type of `Node`s with pointers to `next` nodes and an inductive heap predicate, `sll`, witnessing that a given sequence of `nodes` forms a singly-linked list. Note that `nodes: List[Node]` itself is a purely functional data structure and only present for specification purposes; one would typically store it as a **`@ghost`** variable.

Say we would like to prove that removing the last element of a non-empty singly-linked list `nodes` maintains the `sll` property. This is easy to specify using our functional abstraction `nodes`: assuming `sll(nodes)` holds in the pre-state, we would like to show that `sll(nodes.init)` holds in the post-state, where `.init` is a method in the standard library that drops the last element of a `List[T]`. When `nodes` consists of a single element, the property follows immediately, since `sll(nodes.init)` reduces to `sll(Nil)` which holds by definition of `sll`. On the other hand, if `nodes` contains at least two elements, we need to modify the `next` field of the second-to-last node, i.e., set `nodes(nodes.size - 2).next = None()`. In the latter case we effectively want to establish the Hoare triple

```
{sll(nodes) ∧ F} nodes(nodes.size - 2).next = None() {sll(nodes.init)}
```

where `F` is some additional precondition ensuring that the list has at least two elements, and that all nodes up to the last two are separate from the rest.

```
// A lemma proving that popping from a SLL maintains singly-linked-ness.
def sllPopLemma(h0: Heap, h1: Heap, nodes: List[Node]): Unit = {
  require(
    nodes.nonEmpty &&
    h0.eval { sll(nodes) } &&
    (nodes.size == 1 || (
      Heap.unchanged(nodes.init.init.content.asRefs, h0, h1) &&
      h1.eval { nodes(nodes.size - 2).next == None() }
    )) )
  if (nodes.size > 1) sllPopLemma(h0, h1, nodes.tail)
} ensuring (_ ⇒ h1.eval { sll(nodes.init) })
```

Above, `sllPopLemma` establishes the desired property by explicitly referring to the pre-state as `h0` and the post-state as `h1`. Its proof proceeds by induction on `nodes`, and is mostly automatic; we merely have to invoke the right induction hypothesis when `nodes.size > 1`. An implementation of `pop` would likely resort to a stronger invariant like distinctness of all objects in `nodes`, and then invoke the lemma after the modification as follows

```
val h0 = Heap.get   // Get the pre-state
if (nodes.size > 1)
  nodes(nodes.size - 2).next = None()   // Unlink the last element
sllPopLemma(h0, Heap.get, nodes)
```

along with some hints that deduce `F` from the stronger invariant (not shown). In addition,

for `nodes` to be marked **`@ghost`**, we would need to maintain `nodes(nodes.size - 2)` in a separate non-**`@ghost`** variable. Our benchmark suite includes similar, but more elaborate examples `Queue` and `NodeCycle`.

While our current system does not provide as much automation as separation logic for tree-like data, our approach is not limited to such structures and retains full flexibility in treating heaps as first-class values. Interestingly, this also enables us to prove hyperproperties, i.e., properties such as determinism, which involve multiple heap states. For example, consider the following lemma stating that a memoized function `f : Int ⇒ Int` evaluates to the same result in every heap:

```
def lemmaHeapIsIrrelevant(h0: Heap, h1: Heap, x: Int) = { () }
  ensuring (_ ⇒ h0.eval { f(x) } == h1.eval { f(x) })
```

In many cases such lemmas can be proven automatically by our system, as demonstrated, for instance, by the `FibCache` benchmark.

4 Heap Encoding

In the following, we introduce our heap encoding and how it achieves framing without quantification. Our approach builds upon the existing counterexample-complete unfolding procedure of the Stainless verifier and exploits the additional expressive power afforded by combinatory array logic [dMB09], an extended array theory available in Z3. This use of array combinators for framing is, to the best of our knowledge, novel. Notably, our encoding allows for a high degree of proof automation without giving up counterexamples.

Our tool models stateful operations by explicitly reading from and updating a locally-mutable map that relates each object to its state. In a later transformation step such programs with local mutations are reduced to functional ones. Each stateful function gains an explicit heap parameter and returns a new, potentially updated heap along with its regular output. In terms of Scala's type system, the heap can be thought of as a map `heap` of type `HeapMap = Map[HeapRef, Any]` where `Any` is the top type and `HeapRef` is a data type representing an object's identity. Conceptually, our approach employs a monadic translation [Mog91, Wad90a] that we partially-evaluate [AHM+17], replacing stateful operations such as reads and writes by pure operations on a map.

4.1 Encoding tmap

We first give an informal explanation of our encoding by the example of the minimally-specified version of `tmap` on `Tree` (the version without postconditions, shown in Figure 3.2). In Figure 4.1 we show the data types after transformation.

We treat *heap types*, i.e., descendants of `AnyHeapRef`, like `Cell`, differently from immutable types such as `Tree`. The latter are translated into algebraic data types in the obvious way (lines 5-7). References to heap types, on the other hand, are erased to the internal ADT `HeapRef` that represents locations on the heap (line 1). For instance, the field `data: Cell[T]` of `Leaf` becomes $\texttt{data}_{\texttt{ref}}$`: HeapRef` (line 6). Additionally, each heap class like `Cell` is translated to a single-constructor ADT that encapsulates an object's state at a given time, e.g., $\texttt{Cell}_{\texttt{Data}}$ (line 3).

```
case class HeapRef(id: BigInt)

case class Cell_Data[T](value: T)

sealed abstract class Tree[T]
case class Leaf[T](data_ref: HeapRef) extends Tree[T]
case class Branch[T](left: Tree[T], right: Tree[T]) extends Tree[T]
```

Figure 4.1 – The data types of the `tmap` example in Figure 3.2 after our encoding.

```
def tmap[T](h0: HeapMap, t: Tree[T], f: T ⇒ T): (Unit, HeapMap) = {
  val (rs, ms) = (repr(t), repr(t))
  t match {
    case Leaf(data_ref) ⇒
      assert(data_ref ∈ rs, "‘data‘ must be in reads set")
      assert(data_ref ∈ ms, "‘data‘ must be in modifies set")
      val data: Cell_Data = {
        assume(h0(data_ref).isInstanceOf[Cell_Data[T]])
        h0(data_ref).asInstanceOf[Cell_Data[T]]
      }
      val data': Cell_Data = Cell_Data(f(data.value))
      ((), h0.updated(data_ref, data'))

    case Branch(left, right) ⇒
      val (_, h1) = tmap_shim(h0, rs, ms, left, f)
      tmap_shim(h1, rs, ms, right, f)
  }
}

def tmap_shim[T](h0: HeapMap, rd: RSet, md: RSet, t: Tree[T], f: T ⇒
  T): (Unit, HeapMap) = {
  val (rs, ms) = (repr(t), repr(t))
  assert(rs ⊆ rd, "reads set of Tree.tmap")
  assert(ms ⊆ md, "modifies set of Tree.tmap")
  val res  = tmap(h0, t, f)
  val resR = tmap(rs.mapMerge(h0, dummyHeap), t, f)
  assume(res._1 == resR._1)
  assume(res._2 == ms.mapMerge(resR._2, res._2))
  assume(res._2 == ms.mapMerge(res._2, h0))
  res
}
```

Figure 4.2 – The result of encoding the minimally-specified `tmap` method of Figure 3.2. We use ⊆ to typeset `subsetOf`, ∈ for `contains`, and abbreviate `Set[AnyHeapRef]` by `RSet`.

Note that the language after encoding still supports `Any` and subtyping, so $\texttt{Cell}_{\texttt{Data}}$ `<: Any`.

In Figure 4.2 we show the encoding of `tmap` itself. The method is reduced to a type-parametric function that takes its original argument `f`, the method receiver `t` and a heap parameter `h0`. The imperative operations in `tmap` are translated to functional operations on `HeapMap` as mentioned above, and the modified heap is returned along with the original return value. In particular, if the current tree `t` is a leaf, then we extract its reference to a cell $\texttt{data}_{\texttt{ref}}$ (line 4) and index the initial heap `h0` at $\texttt{data}_{\texttt{ref}}$ (line 9). Note that since the heap map stores values of type `Any` we have to perform a downcast (lines 8-9). This is safe, since we will only verify well-typed Scala programs, so any such cast will be correct by construction. In a later type-encoding phase [Voi19] Stainless translates type tests such as line 8 to conditions in the theory of inductive data types. On line 11 we apply the function `f` to the old `value` of `data` and construct a $\texttt{Cell}_{\texttt{Data}}$ value reflecting the new state of `data`. We then return the updated heap on line 12. In case the tree `t` is a `Branch` we simply perform two recursive calls (lines 15-16), albeit through the newly-introduced wrapper function $\texttt{tmap}_{\texttt{shim}}$ which we discuss below.

Our encoding achieves modular verification of heap contracts (**`reads`** and **`modifies`**) by injecting some additional assertions and assumptions. We bind the **`reads`** and **`modifies`** sets (`rs` and `ms`) at the top of the function (line 2). For each object that is read or modified we check that the object is in the respective set (lines 5-6). For function calls we check that the callee's **`reads`**, resp. **`modifies`**, set is subsumed by the caller's. We achieve this by invoking a wrapper function $\texttt{tmap}_{\texttt{shim}}$, that additionally takes as parameters the *domains* on which the passed heap is defined for reads and modifications (`rd` and `md`). Within the wrapper we bind the original function's **`reads`** and **`modifies`** sets (line 21), check subsumption wrt. the domains (lines 22-23) and call the original function `tmap` (line 24).

Finally, we assume the modular guarantees about `tmap` wrt. the pre- and post-state, i.e., its *frame conditions*. Our approach is to relate the "actual" function call at heap state `h0` (line 24) to a "hypothetical" call (line 25) that operates on an alternative heap state which interprets all objects in **`reads`** as `h0` does, and assigns globally fixed dummy values for all objects outside of **`reads`**. The core idea is that both of these calls ought to produce equivalent results modulo the untouched parts of the heap.

In particular, lines 26-27 state that the result of `tmap` only depends on the **`reads`** subset of the heap, whereas line 28 states that the heap resulting from `tmap` may only have changed on objects in **`modifies`**. For the **`reads`**-related frame conditions we depend on the hypothetical application of `f` to the projected heap `rs.mapMerge(h0, dummyHeap)`, which contains the state of `h0` for all objects in `rs` and that of `dummyHeap` elsewhere. The first assumption thus states that the result computed by `f` is the same no matter whether we apply it to `h0` or to some other arbitrary (but well-typed) heap that is only known to agree on the valuations of objects in `rs`. The second assumption states the analogous property about the locations that might have been modified by `f`. Finally, the third assumption expresses that the pre-state equals the post-state in all locations but those in the **`modifies`** clause, i.e., the set `ms`.

Variables … x, y, h, ρ, μ

Surface Language

Types … S, T	:=	C | D | **Set**[T] | **AnyHeapRef**
Terms … t	:=	x | f($\bar{t}$) | **let** x = t **in** t | t.f | t.f := t
Functions … f	:=	**def** f($\overline{x : T}$) : S = {**reads**(t); **modifies**(t); t}

Lowered Language

Types … S, T	:=	D | **Set**[T] | **Map**[T, T] | **Any** | **HeapRef**
Terms … t	:=	x | f($\bar{t}$) | **let** x = t **in** t | t.f | t.f := t |
		let var x = t **in** t | x := t |
		t[t] | t.update(t, t) | t.mapMerge(t, t) |
		t.isInstOf[T] | t.asInstOf[T] |
		assume(t); t | **assert**(t); t
Functions … f	:=	**def** f($\overline{x : T}$) : S = {t}

Figure 4.3 – Selected terms and types of the languages before and after heap encoding.

The crucial component of our encoding here is the `mapMerge` primitive, which can be seen as a ternary operator of type `∀ K V. Set[K] ⇒ Map[K,V] ⇒ Map[K,V] ⇒ Map[K,V]`. Specifically, `mapMerge` takes a set `s` along with two maps `m1`, `m2` and produces a map `m' = s.mapMerge(m1, m2)` such that `∀ k:K. (k ∈ s ⟶ m'[k] = m1[k]) ∧ (k ∉ s ⟶ m'[k] = m2[k])`. We will discuss how `mapMerge` is translated to Z3's extended array theory in Section 4.3.

4.2 Translation Rules

We now describe the general translation rules as applied in our system. We will consider only a subset of the language supported, focussing on constructs of particular interest in the translation (shown in Figure 4.3).

We distinguish the terms t and types T of the surface language from those of the language after encoding. The surface language comprises of both (immutable) algebraic data types D and (mutable) heap types C, along with terms for field reads t.f and updates t.f := t, which are interpreted as either functional or imperative operations, depending on whether the receiver is an ADT or a heap type. In the lowered language the latter are always interpreted functionally, and the only imperative feature available are locally-mutable variables **let var** x = t **in** t and assignments thereof, x := t. Though not discussed here, it is straightforward to convert programs with local mutation into purely functional ones [Bla17, Fil03]. Our simplified language also omits first-class functions. In practice we require them to be pure, while effectful ones can

be encoded using abstract classes with heap contracts (see `Task` in Figure 5.2 for an example).

At its heart, our translation turns imperative operations on heap types $C_1, C_2, \ldots$ into functional operations on a map representing the entire heap. What should be the key and value types of the heap map? For keys, i.e., the references in our heap model, we choose an abstract type **HeapRef** isomorphic to the natural numbers, but with equality as its only operation. For values, i.e., the state of individual objects, we pick the top type **Any** as the trivial solution which subsumes the representations of all heap types. While SMT solvers do not directly support subtyping, this is convenient in Stainless, as we can leverage its existing support for subtyping and **Any** [Voi19]. Our design differs from that supported by the Boogie verifier, whose type system provides higher-rank map types [LR10, Lei08] in which the heap map may be typed as `∀T. Map[Ref[T], T]`, avoiding the need for (correct-by-construction) downcasts and an additional type encoding phase to deal with the Any type.

Due to our choice of heap representation, the lowered language includes maps and type-tests to express various assumptions about the heap that are correct by construction. For maps, we use $t[t_k]$ to denote indexing and $t.\mathsf{update}(t_k, t_v)$ to denote the (functional) result of updating a map t at key t_k. To recover information from **Any**-typed values, we provide $t.\mathsf{isInstOf}[T]$ to express type tests and $t.\mathsf{asInstOf}[T]$ for the corresponding downcasts. Furthermore, $\textbf{assume}(t); t$ and $\textbf{assert}(t); t$ mark assumptions and assertions to be used during VC generation. Combining these constructs, we can express a downcast of t to T that is assumed correct as $\textbf{let}\ x = t_1\ \textbf{in}\ \textbf{assume}(x.\mathsf{isInstOf}[T]); t_2\{x \mapsto x.\mathsf{asInstOf}[T]\}$, which we abbreviate by $\textbf{let}\ x = t_1\ \textbf{as}\ T\ \textbf{in}\ t_2$. As in the example in Section 4.1, we take **HeapMap** and **RSet** to be shorthands for $\textbf{Map}[\textbf{HeapRef}, \textbf{Any}]$ and $\textbf{Set}[\textbf{HeapRef}]$, respectively.

We define two translation relations that take types T, resp. well-typed terms t, and produce their lowered counterparts. The translation relation for types, $T \triangleright T'$, witnesses the erasure of type T to T'; for instance, if Cell is a heap type, then $\textbf{Set}[\mathsf{Cell}] \triangleright \textbf{Set}[\textbf{HeapRef}]$. The translation relation for terms is notated as $h, \rho, \mu; \Gamma \vdash t \triangleright t'$ and depends on a locally-mutable heap variable h, its reads and modifies domains, ρ and μ, and the typing environment Γ. When implicitly clear or the same in all occurrences, we omit h, ρ, μ and Γ and simply write $t \triangleright t'$. We assume the existence of a typing relation $\Gamma \vdash t : T$ and also omit Γ when it is clear from the context.

The encoding proceeds by translating each definition of an ADT D, heap type C, or function f in the surface program to a corresponding lowered definition. The data type definitions of the encoded program are obtained by taking all of the ADT definitions D with argument types erased by $T \triangleright T'$, and additionally introducing one single-constructor ADT for each heap type C (also with its field types erased). We refer to the resulting lowered ADTs as D_D and D_C. For each function definition $\textbf{def}\ f(\overline{x : T}) : S = \{\textbf{reads}(t_\rho); \textbf{modifies}(t_\mu); t\}$ in the original program we introduce two functions f and f_{shim} in the encoded program. The encoded function f takes the pre-state as an additional argument, and returns the resulting post-state along with its result value, yielding

$$\textbf{def}\ f(h_0 : \textbf{HeapMap}, \overline{x : T'}) : (S', \textbf{HeapMap}) = \{\textbf{let}\ \rho = t'_\rho\ \textbf{in let}\ \mu = t'_\mu\ \textbf{in}\ t'\}$$

where $h_0, \rho, \mu; \Gamma_0 \vdash t \triangleright t'$, as well as $h_0, \rho, \emptyset; \Gamma_0 \vdash t_s \triangleright t'_s$ for $s \in \{\rho, \mu\}$, $\Gamma_0 = \overline{x : T}$, $\overline{T} \triangleright \overline{T'}$ and $S \triangleright S'$. Note that the translation of the reads set t'_ρ bound to ρ may refer to ρ itself – after all the reads clause may, in fact, depend on and read objects on the heap, and we ought to check that those objects are part of the reads set. While this may seem alarming at first, ρ is nonetheless well-defined, since t'_ρ will only refer to ρ from assertions injected during our translation. This circularity is analogous to the self-framing of assertions in implicit dynamic frames [SJP12]. In practice, we sidestep the issue by translating t_ρ twice: once without injecting assertions to bind it to ρ, and once more with the checks.

For each function f we also define a companion f_{shim} which encapsulates both the assumption of frame conditions and the checks of associated heap contracts expected at each call site of f:

```
def f_shim(h_0 : HeapMap, ρ_dom : RSet, μ_dom : RSet, x̄ : T̄) : (S', HeapMap) = {
    let ρ = t'_ρ in let μ = t'_μ in
    assert(ρ ⊆ ρ_dom); assert(μ ⊆ μ_dom);
    let y_res = f(h_0, x̄) in
    let y_resR = f(ρ.mapMerge(h_0, dummyHeap), x̄) in
    assume(y_res._1 = y_resR._1);
    assume(y_res._2 = μ.mapMerge(y_resR._2, y_res._2));
    assume(y_res._2 = μ.mapMerge(y_res._2, h_0));
    y_res
}
```

As an optimization, we omit the parts of the encoding that relate to the post-state when the **modifies** clause is empty. When the **reads** clause is empty as well, we avoid changing the function's signature altogether, so that pure functions remain pure.

The crucial rules of $t \triangleright t'$ are listed in Figure 4.4. Both FIELDREADI and FIELDUPDATEI deal with field accesses of immutable data types and do not require interaction with the heap. In general, pure constructs are left untouched and their translation rules merely map over subexpressions. Imperative constructs, on the other hand, read or modify the locally-mutable heap h and refer to ρ and μ to enforce the heap contracts. For instance, FIELDREADM handles field reads from a heap type C. It translates a read t.f to an assertion that the receiver object is in the reads set ($t' \in \rho$), after which the object state is read from the heap ($h[t']$) and downcast to the corresponding lowered data type D_C, from which the actual value is then projected (x.f). For readability we duplicate some encoded terms such as t' in FIELDREADM, whereas in practice we introduce additional let-bindings to avoid exponential blowup of encoded programs.

The rule for function calls, CALL, merely rewrites invocations of f to invocations of f_{shim}, passing in the current heap h and the domains on which the callee is permitted to read and modify the heap. We always inline these shim functions, so the assertions in f_{shim} are

$$\frac{t : D \quad t \rhd t'}{t.f \rhd t'.f} \qquad \text{(FieldReadI)}$$

$$\frac{t_1 : D \quad t_1 \rhd t_1' \quad t_2 \rhd t_2'}{t_1.f := t_2 \rhd t_1'.f := t_2'} \qquad \text{(FieldUpdateI)}$$

$$\frac{t : C \quad t \rhd t' \quad x \text{ is fresh}}{t.f \rhd \textbf{assert}(t' \in \rho); \textbf{let } x = h\left[t'\right] \textbf{ as } D_C \textbf{ in } x.f} \qquad \text{(FieldReadM)}$$

$$\frac{t_1 : C \quad t_1 \rhd t_1' \quad t_2 \rhd t_2' \quad x \text{ is fresh}}{t_1.f := t_2 \rhd \begin{array}{c}\textbf{assert}(t_1' \in \rho \cap \mu); \textbf{let } x = h\left[t_1'\right] \textbf{ as } D_C \textbf{ in} \\ h := h.\mathsf{update}(t_1', (x.f := t_2'))\end{array}} \qquad \text{(FieldUpdateM)}$$

$$\frac{\overline{t} \rhd \overline{t'} \quad x \text{ is fresh}}{f(\overline{t}) \rhd \textbf{let } x = f_{\mathsf{shim}}(h, \rho, \mu, \overline{t'}) \textbf{ in } h := x.\_2; x.\_1} \qquad \text{(Call)}$$

Figure 4.4 – Basic rules of the term translation relation $h, \rho, \mu; \Gamma \vdash t \rhd t'$. We abbreviate the relation as $t \rhd t'$, since the omitted arguments are merely passed through by the above rules. The form $\textbf{let } x = t_1 \textbf{ as } T \textbf{ in } t_2$ is syntactic sugar for downcasts (see Section 4.2).

effectively lifted to each call site of f and ensure that the reads and modifies clauses of the callee is subsumed by the caller's.

4.3 Quantifier-Free Frame Conditions

In the previous subsection we assumed a language construct called mapMerge that made it straightforward to express the necessary frame conditions. The crucial question that remains is how to lower mapMerge and its arguments to an efficiently decidable theory supported by an SMT solver. Our solution is to target the theory of (infinite, extensional) arrays in Z3, leveraging the fact that Stainless translates both sets and maps to such arrays. This means that reads and modifies expressions of type **Set**[**HeapRef**] become arrays typed HeapRef ⇒ Boolean, while heap maps of type **Map**[**HeapRef**, **Any**] are translated to HeapRef ⇒ Any. We can then use the array combinator $\mathrm{map}_f(a_1, \ldots, a_n)$ to express mapMerge efficiently. This array combinator is part of Z3's extended array theory [dMB09] and axiomatized as $\forall i. \mathrm{map}_f(a_1, \ldots, a_n)[i] = f(a_1[i], \ldots, a_n[i])$. While the combinator can in practice only be applied to built-in functions, this is sufficient for our purposes. Suppose $\langle t \rangle$ represents the translation of a term t in our lowered language to a term in first-order logic. Given Stainless' encoding of sets and maps, one can use the if-then-else function ite of Z3, and translate $s.\mathsf{mapMerge}(m_1, m_2)$ as $\mathrm{map}_{\mathrm{ite}}(\langle s \rangle, \langle m_1 \rangle, \langle m_2 \rangle)$.

$$\begin{array}{lll} \text{Types } \ldots\, S, T & := & \ldots \mid \textbf{Heap} \\ \text{Terms } \ldots\, t & := & \ldots \mid \mathsf{Heap.get} \mid \mathsf{t.eval(t)} \mid \mathsf{Heap.unchanged(t,t,t)} \end{array}$$

$$\frac{}{\mathsf{Heap.get} \triangleright \rho.\mathsf{mapMerge}(\mathsf{h}, \mathsf{dummyHeap})} \quad (\textsc{HeapGet})$$

$$\frac{t_h : \textbf{Heap} \quad t_h \triangleright t'_h \quad h' \text{ is fresh} \quad h', U, U; \Gamma \vdash t_e \triangleright t'_e}{t_h.\mathsf{eval}(t_e) \triangleright \textbf{let var } h' = t'_h \textbf{ in } t'_e} \quad (\textsc{HeapEval})$$

$$\frac{t_{h1} : \textbf{Heap} \quad t_{h2} : \textbf{Heap} \quad t_s \triangleright t'_s \quad t_{h1} \triangleright t'_{h1} \quad t_{h2} \triangleright t'_{h2}}{\mathsf{Heap.unchanged}(t_s, t_{h1}, t_{h2}) \triangleright t'_{h1} = t'_s.\mathsf{mapMerge}(t'_{h2}, t'_{h1})} \quad (\textsc{HeapUnchanged})$$

Figure 4.5 – Syntax of the surface language with first-class heaps and related term translation rules. The symbol U denotes the universal set of all **HeapRef**s.

4.4 First-Class Heaps

A benefit of our encoding is that it naturally extends to explicit reasoning about alternative heap states within the program logic. Since our heaps are merely **Map**s, we can consider contexts with multiple heaps and express hyperproperties like determinism. Compare this to verifiers based on imperative languages, where relational verification requires constructions such as self-composition and product programs, limiting the applicability of existing toolchains [BCK11, EMH19].

The syntax extensions related to first-class heaps are shown in Figure 4.5 alongside the additional translation rules. The type translation simply erases **Heap** $\triangleright$ **HeapMap**. All of the new constructs are straightforward to encode in our scheme. Heap.get exposes the currently readable heap (HeapGet). We reduce t_h.eval(t_e) to translating t_e in the context of a fresh heap variable initialized to t_h (HeapEval). Notably, during this translation we do not inject any further checks of **reads** and **modifies** by setting ρ and μ to the sentinel value U (denoting the universal set). While the lack of checks allows for reads outside a heap's original domain, they are well-defined (i.e., they equal the dummyHeap on those locations). Finally, Heap.unchanged(t_s, t_{h1}, t_{h2}) translates to an equality that holds iff for all objects in t_s the heaps t_{h1} and t_{h2} agree. The corresponding lowering rule HeapUnchanged leverages mapMerge in a way similar to our encoding of frame conditions. Namely, we take t'_s.mapMerge(t'_{h2}, t'_{h1}) (the heap which interprets all objects as in t'_{h1}, except those in t'_s, which it interprets as in t'_{h2}), and require that it equals t'_{h1} itself.

4.5 Allocations

The translation we have seen so far is sufficient to reduce the core of an imperative language to a functional one with arrays and array combinators. Using these basic primitives we can already model various high-level constructs. For instance, we might model finite arrays (as opposed to the struct-like objects seen thus far) as functional lists of mutable `Cell`s. Similarly, object allocations can already be handled by adding a user-defined model of an allocator and rewriting allocations in the surface language to invocations of the allocator. We include corresponding verified examples, `CellArraySimple` and `AllocatorMono`, in our evaluation (Chapter 5). Nonetheless, we would like allocation to be integrated more tightly in the future in order to provide as much automation as possible. For this reason we now sketch an extension of the language and encoding of the previous sections to (first-class) heaps that allow reasoning about the set of allocated objects.

So far we have refrained from explicitly representing the allocation status of objects. Our model of the heap was a total map from object references to immutable data types representing the dynamic type and state of each object. Outside its reads-accessible regions the heap would take on some unspecified values.

Instead, we now want to interpret the heap as a partial map to explicitly distinguish *allocated* from *free* heap locations. One way to achieve this is to add a fresh, designated data type **Free** that is isomorphic to unit. That is, $\mathbf{free} : \mathbf{Free}$ does not possess any fields and is distinct from all other heap classes. We can then discuss the set of allocated objects $\mathsf{t_h.alloc} : \mathbf{Set}[\mathbf{HeapRef}]$ of a heap $\mathsf{t_h}$ in our surface language. Intuitively, its interpretation is the set of all objects $\mathsf{x} : \mathbf{AnyHeapRef}$ such that $\mathsf{t_h.eval(x)} \neq \mathbf{free}$.

Suppose we had an appropriate primitive in the lowered language $\mathsf{m.proj_{\neg D}}$ that projects from a map $\mathsf{m} : \mathbf{Map}[\mathsf{S}, \mathsf{T}]$ the set of all those keys whose corresponding value is not of dynamic type (i.e., constructor) D. Then we could translate $\mathsf{t_h.alloc}$ as follows:

$$\frac{\mathsf{t_h} : \mathbf{Heap} \quad \mathsf{t_h} \rhd \mathsf{t'_h}}{\mathsf{t_h.alloc} \rhd \mathsf{t'_h.proj_{\neg free}}} \qquad \text{(HeapAlloc)}$$

Recall that **Heap** in the surface language is desugared to $\mathbf{HeapMap} = \mathbf{Map}[\mathbf{HeapRef}, \mathbf{Any}]$, so $\mathsf{t'_h.proj_{\neg free}}$ on a heap map $\mathsf{t'_h}$ effectively projects all those heap locations that do not store **free**.

Luckily, an SMT solver such as Z3, that supports both the theory of algebraic data types and combinatory array logic, can also express $\mathsf{t_h.alloc}$ without quantifiers. Ultimately, Stainless represents **Any** by an ADT containing a constructor for each data type D, including one for our new designated data type **Free**. For each constructor of an (SMT-LIB) ADT and thus each data type (of our calculus) D we get a built-in tester function is-$\langle \mathsf{D} \rangle$. In particular, is-$\langle \mathbf{Free} \rangle(\cdot)$ will be a unary function in first-order logic indicating whether the argument value corresponds to $\mathbf{free} : \mathbf{Free}$. The $\mathsf{m.proj_{\neg D}}$ primitive can then be translated as $\mathsf{map_{\neg}}\left(\mathsf{map}_{\text{is-}\langle \mathbf{Free} \rangle}\left(\langle \mathsf{m} \rangle\right)\right)$,

yielding a decidable encoding of t_h.alloc.

This relatively minor extension is surprisingly versatile: not only does it allow us to model allocations more directly, but we can also use it to express separation between heaps. A newly-allocated object x : **AnyHeapRef** is one such that, prior to allocation, $x \notin$ Heap.get.alloc. To translate an object allocation in the surface language we can choose any such "free" heap location and update the heap with the new object's initial state. To pick object x we can leverage Inox's `choose[T]((x:T) => p(x))` primitive, which expresses an opaque, but determinstic choice among all `x:T` satisfying predicate `p(x)`.[1]

Separation between two heaps t_{h1} and t_{h2} then simply becomes a derived notion:

$$\mathsf{Heap.separate}(t_{h1}, t_{h2}) \quad := \quad t_{h1}.\mathsf{alloc} \cap t_{h2}.\mathsf{alloc} = \emptyset$$

In Appendix A we show a minimal self-contained example of this encoding in SMT-LIB syntax, as supported by the Z3 solver.

The additional expressiveness afforded by $\mathsf{Heap.separate}(t_{h1}, t_{h2})$ is not terribly useful, though, unless we also change our encoding of function calls. So far we would simply pass the *entire* heap into a stateful function, and add appropriate frame conditions at each call site asserting that the resulting heap had remained unchanged on all objects other than the **`modifies`** clause et cetera. Instead, we can treat calls analogously to the frame rule of separation logic. The idea may be summarized as follows: upon every function call, one first uses the mapMerge primitive to decompose the input heap into two heaplets corresponding to the frame and the function's footprint. One would then feed only the footprint heaplet to the function, and merge the frame with the output heaplet upon the function's return.

With this departure from our current encoding we would seem to arrive at an interesting alternative design point: an instantiation of separation logic that retains the full flexibility of dynamic frames and first-class heaps. That being said, we are still investigating whether such an encoding would in fact be preferable in terms of performance and implementation complexity. At least one complication arises when we model function calls to only operate on the footprint heaplet. Namely, if we were to naively choose a free location from within the function (which operates only on the footprint heaplet), we might end up allocating an object in a location that is already part of the frame. To ensure that the frame remains untouched by allocations one would thus have to additionally constrain the choice of new object locations. Analogously to our `AllocatorMono` implementation, one solution would be to share a mutable "free list" across all allocating function calls and constrain its elements to be disjoint from all allocated objects before each function call.

[1] The choose primitive can be roughly thought of as an uninterpreted function invocation parameterized by the surrounding function's arguments and an additional argument identifying the call site of choose.

5 Evaluation

We used our system to verify a number of benchmarks ranging in size and complexity. Among the examples we developed are both shallowly and deeply mutable data structures, a model of an object allocator, and a parallelization primitive for the fork-join model. In Figure 5.1 we summarize these benchmarks quantitatively in terms of total lines of code, and the time our system takes to verify the example. In particular, we report *T*, the total wall time elapsed when running an individual benchmark, which includes the time it takes the Scala compiler to process both our standard library and the benchmark, our extraction pipeline to lower from imperative Scala code to the functional fragment, and the time spent on generating and checking verification conditions. The latter component is reported separately as *C*, and the time thereof spent on checking heap contracts as *HC*. The reported numbers were obtained on a machine with an AMD Ryzen 3700X 8-core CPU @ 3.6GHz and 32GB of RAM running Ubuntu 20.04, and using Z3 version 4.8.12. We explicitly list an *empty* benchmark that entails no verification conditions, but provides a baseline for the time spent on JVM startup, and, more importantly, extraction through the traditional Scala compilation pipeline plus various lowerings in Stainless before the actual generation and solving of VCs is performed.

We next discuss our experience using the tool and elaborate on some of the benchmarks listed.

5.1 Shallowly-Mutable Data Structures

We first consider "shallowly-mutable" data structures such as `Cell[T]` seen in Section 3.2 whose mutable data is stored directly in its fields, i.e., without any indirection. They provide a simple baseline for our system and play an important role as building blocks for larger data structures such as trees and arrays with fine-grained separation properties. However, shallowly-mutable data structures are useful in their own right: For instance, we implemented `UpCounter` which tracks a monotonically increasing variable and maintains an invariant relative to the counter's initial value. We also implemented a simple array (`ArraySimple`) and stack (`StackSimple`) which essentially act as wrappers around functional data structures in that they only store the reference to the head of an immutable list. For instance, `ArraySimple[T]`

Benchmark	#LoC	#VCs	T	C	HC
Empty	10	0	6.3	0.0	0.0
AllocatorMono	73	80	12.8	4.1	0.8
ArraySimple	38	16	7.6	0.6	0.2
CellArraySimple	21	9	7.2	0.4	0.1
FibCache	38	32	11.1	2.8	0.3
MutList	81	148	46.2	35.4	2.2
MutListSetsOnly	45	54	30.3	22.4	1.4
NodeCycle	72	69	12.0	4.0	0.2
Queue	190	290	36.6	20.5	3.6
Stack	66	62	10.5	2.6	0.7
StackSimple (Fig. 3.1)	27	26	8.3	1.1	0.1
TaskParallel	46	38	8.3	1.1	0.2
TaskParallelBasic	58	51	8.6	1.2	0.2
TraitsReadsWrites	39	33	7.8	0.8	0.2
TreeImmutMapGeneric (Fig. 3.3)	55	33	17.1	8.3	0.2
UpCounter	48	32	8.0	0.9	0.2

Figure 5.1 – Evaluation results. For each benchmark we list the # of verification conditions discharged, the # lines of Scala code (including annotations), the total runtime *T*, the time spent checking VCs *C*, and the particular amount of time spent on VCs of heap contracts *HC*. Timings are given in seconds.

consists of a single mutable field `var list: List[T]`. In our examples we show safety wrt. bounds checks and non-emptiness when popping an element off the stack. We found that our system easily deals with this kind of mutability, requiring no additional proof hints whatsoever, in particular since the associated operations typically require no recursion through stateful functions, making them straightforward to verify and invalidate with counter-examples.

5.2 Mutable Linked Lists and Queues

As an example of a more complex data structure we implemented multiple variations of a mutable, acyclic, singly-linked list. We focussed on an `append` operation, which takes two valid linked lists `l1` and `l2` with disjoint representations and concatenates them, leaving `l1` in a valid state. This is challenging in a system without a built-in notion of lists or trees, since establishing the well-formedness of lists (e.g., the absence of cycles) requires knowledge of heap separation and an inductive proof that maintains the property for intermediate nodes.

We considered several options to track a node's representation `repr`. One could express `repr` as a recursive function as in Section 3.2, or, instead, as a mutable `@ghost` field on each node. In our benchmarks we present two variants of the latter approach: `MutList` encodes the ghost field `repr` as `List[AnyHeapRef]`, which has the added benefit of allowing predicates like `valid` to recurse on the representation, and can be converted to a `Set[AnyHeapRef]` as

required by our **reads** and **modifies** clauses. `MutListSetsOnly` instead implements `repr` as `Set[AnyHeapRef]`, whose encoded form requires no further conversion to interact with the `mapMerge` primitive we use for framing.

We used a similar approach to implement `Queue`, which provides constant-time enqueue and dequeue methods using references to the first and last nodes. Given a `valid` queue we prove that enqueue and dequeue maintain `valid`ity and are functionally correct with respect to a serialized representation similar to `toList` in Section 3.2. The example demonstrates how safety properties can be established even in the presence of sharing and arbitrarily deep data structures.

The `NodeCycle` example, which we reproduce in Appendix B, illustrates how to define the inductive heap predicate for a cyclic list. We also establish that the prepend operation on such a list maintains cyclicity. Both this and the aforementioned example leverage first-class heaps to carry out the inductive proofs showing that the corresponding heap predicates continue to hold after modifications to the data structure.

5.3 Slices, Monolithic and Cell-Based Arrays

Arrays are some of the most common data structures found in imperative code and thus a worthwhile target for verification. When specifying algorithms involving arrays it often pays to introduce slices, i.e., subarrays, as a means of abstraction. By extending the `ArraySimple` example we arrived at `ArraySlice` which provides safe indexing, update and re-slicing operations wrt. an underlying array. In the absence of sharing, this solution of encapsulating all array state in a single "monolithic" mutable heap object (the underlying array) is the natural and practical choice.

To analyze divide-and-conquer algorithms on arrays, on the other hand, we require some more fine-grained control, since we would like our dynamic frames to reflect the fact that slices of an array may only access a subset of heap locations. A more complex representation based on lists of `Cell[T]`s allows us to achieve such fine-grained framing of arrays and slices. In example `CellArraySimple` we illustrate this approach and verify safety of accesses.

5.4 Fork-Join Parallelism

Since dynamic frames in our system are simply given by read-only expressions, users may define their own imperative abstractions. For instance, in `TaskParallel` we demonstrate how one can specify a primitive modelling fork-join parallelism. Figure 5.2 shows an excerpt introducing the `Task` interface that encapsulates an asynchronous computation and declares the set of heap objects that may be read and modified in the process. Further below we define the `parallel(t1, t2)` construct [KP18] itself, imposing a number of restrictions: Firstly, callers of `parallel` have to establish accessibility to both `t1` and `t2`'s frames (lines 14-15).

```
1  abstract class Task {
2    @ghost def readSet: Set[AnyHeapRef]
3    @ghost def writeSet: Set[AnyHeapRef] = { ??? }
4      ensuring (_ ⊆ readSet)
5
6    def run(): Unit = {
7      reads(readSet)
8      modifies(writeSet)
9      ??? : Unit
10   }
11 }
12
13 def parallel(task1: Task, task2: Task): Unit = {
14   reads(task1.readSet ++ task2.readSet)
15   modifies(task1.writeSet ++ task2.writeSet)
16   require((task1.writeSet ∩ task2.readSet == ∅) &&
17           (task2.writeSet ∩ task1.readSet == ∅))
18   task1.run(); task2.run()
19   // task1 and task2 complete before this function returns
20 }
```

Figure 5.2 – An interface for asynchronous computations and a sequential specification for fork-join parallelism. The `???` denotes unimplemented code in abstract classes.

Secondly, we require that the write set of `t1` is disjoint from `t2`'s read set and vice-versa (lines 16-17). This separation property justifies replacing our sequential model of `parallel` by a more efficient runtime implementation executing the two tasks concurrently.

Users can define new asynchronous tasks by implementing `Task`. Operations such as those on cell-based data structures discussed above are straightforward to parallelize in this way. Our introductory example of Section 3.2 could be parallelized by defining a new class `TMapTask[T](t: Tree[T], f: T ⇒ T)` whose `run` method calls `tmap`, and replacing the recursive calls in `tmap` by `parallel(TMapTask(left, f), TMapTask(right, f))`.

Note that `Task` is also an example of how to compensate for the lack of effectful first-class functions in our system. Namely, a function value of type `S ⇒ T` is assumed to neither read nor modify the heap and thus remains untouched by our transformation. Instantiations of `Task`, on the other hand, can be used to emulate effectful function values, and closures, in particular, by defining anonymous classes implementing the `Task` interface.

Part II
Type-Level Programming in a Language with Subtyping

In this second part of the thesis we shift our focus from traditional program verifiers like Stainless to more lightweight static checks facilitated by a type checker, like the one in the Scala compiler. Compared to what we have presented so far, our approach in this part differs in two significant ways. We integrate with an existing type system and will now leverage dependent types, rather than contracts, to specify and propagate the desired safety properties. In particular, we lift a functional fragment of the term-level language to improve popular idioms for type-level computation in Scala. Unlike before, our goal is not to model the *entire* language precisely, but only its pure fragment. Our approach here is to face the abundance of weakly-typed and impure code head-on, and provide new facilities for sound approximation of such program fragments. As we will see below, one of our main ideas is the introduction of non-determinism into dependent types, allowing programmers to specify their programs using a mix of precise (functional) type-level operators and non-deterministic choice from base types.

Dependent types have been met with considerable interest from the research community in recent years. Their primary application so far has been in proof assistants such as Agda [Nor07], Coq [BC04] and Idris [Bra13], where they provide a sound and expressive foundation for theorem proving. However, dependent types are still largely absent from general-purpose programming languages, despite a long history of lightweight approaches [XP98]. In the context of Haskell, much research has gone into extending the language to support computations on types, for instance in the form of functional dependencies [Jon00], type families [KJS10] and promoted datatypes [YWC+12]. These techniques have seen adoption by Haskell programmers, showing that there is a real demand for such mechanisms. Furthermore, recent research has explored how dependent types could be added to the language for the same purpose [Eis16, WVdAE17]. In a largely orthogonal direction, inference for dependent refinement types is reaching significant maturity [VTVH18, VTC+17, VBJ15].

Dependently-typed languages often rely on a unified syntax to describe both terms and types. The simplicity of this approach is unfortunately at odds with the design of most programming languages, where types and terms are expressed using separate syntactic categories. Singleton types provide a simple solution to this problem by allowing every term to be represented as a type. The singleton type of a term therefore gives us the most precise specification for that term.

We describe a concrete proposal of how to combine an industrial mixed-paradigm language, Scala, with dependent types. We offer both a formalization of our type system and a discussion of the challenges faced in a practical implementation. Unlike proof assistants, we do not aim to use types as a general-purpose logic, which would favor designs ensuring totality of functions through termination checks. Instead, our focus is on improving type safety of software by increasing the expressive power of the type system.

We present $\lambda^{\text{nd}}_{<:\{\}}$, a dependently-typed calculus with subtyping and singleton types. The main novelty of our calculus is a new approach to expressing type-level computation that, at first,

seems diametrically opposed to the purity other systems favor. A new term is added for non-deterministic choice from a base type, similar to Floyd's choice operator [Flo67b]. Designing a sound system in the presence of non-determinism is challenging. Our solution provides systematic translation of non-determinism using additional parameters that are existentially quantified at a syntactically well-defined point. Consequently, a term in $\lambda^{\text{nd}}_{<:\{\}}$ may reduce to different values. Our system generalizes the traditional notion of singleton type: when the lifted term t contains a non-deterministic choice, the resulting type $\{t\}$ denotes the set of values that t could possibly reduce to. As a result, our type system is capable of type computations by manipulating types which are based on terms, but can nonetheless contain more than a single value. In combination with subtyping, this allows us to seamlessly integrate with impure, or imprecisely-typed programs.

Outline

We begin with a series of examples to demonstrate our extension of the Scala language and to build an intuition for the power added by allowing non-deterministic terms in singleton types (Chapter 6). To capture the essence of our extension we introduce $\lambda^{\text{nd}}_{<:\{\}}$, a calculus which combines dependent types, subtyping and the generalized notion of singleton types (Chapter 7). We then describe its denotational semantics by reduction to System FR [HVK19] and a mechanized soundness proof in Coq (Chapter 8). Finally, we discuss details of our prototypical implementation in the Scala compiler (Chapter 9), and evaluate our extension by exploring the use case of a strongly-typed wrapper for Apache Spark (Chapter 10).

6 First-Class Type-Level Programming for Scala

We begin by motivating why dependent types are desirable in general purpose programming, and how one might use them to improve type safety. In our first example, we design an API that keeps track of database tables' schemas in the type. We demonstrate how dependently-typed list operations can be used to compute schemas resulting from join operations at the type level. Our second example shows how to build a safer version of the zip operation on lists that only accepts equally-sized arguments. The examples in the following sections are written in our dependently-typed extension of Scala described in Chapter 9.

6.1 Example: Safe Join

As a first step, we show how our system supports type-level programming in the style of term-level programs. Consider the following definition of the list datatype, which is standard Scala except for the new **dependent** keyword:

```
sealed trait Lst { . . . }
dependent case class Cons(head: Any, tail: Lst) extends Lst
dependent case class Nil() extends Lst
```

We can define list concatenation in the usual functional style of Scala, that is, using pattern matching and recursion:

```
sealed trait Lst {
  dependent def concat(that: Lst) <: Lst =
    this match {
      case Cons(x, xs) => Cons(x, concat(xs, that))
      case Nil()       => that
    }
}
```

By annotating a method as **dependent**, the user instructs our system that the result type of

`concat` should be as precise as its implementation. Effectively, this means that the body of `concat` is lifted to the type level, and will be partially evaluated at every call site to compute a precise result type which *depends* on the given inputs. For recursive **`dependent`** methods such as `concat`, we infer types that include calls to `concat` itself. The `<:` annotation lets us provide an upper bound on `concat`'s result type, which will be used while type checking the method's definition. Finally, by qualifying the definition of `Cons` and `Nil` as **`dependent`** we also allow their constructors and extractors to be lifted to the type level. Using these definitions, we can now request the precise type whenever we manipulate lists by annotating the new **`val`** binding with **`dependent`**:

```
dependent val l1 = Cons("A", Nil())
dependent val l2 = Cons("B", Nil())
dependent val l3 = l1.concat(l2)
l3.size: { 2 }
l3: { Cons("A", Cons("B", Nil())) }
```

Enclosing a pure term in braces (`{ ... }`) denotes the singleton type of that term. In the last two lines of this example we are therefore asking our system to prove that `l3` has size 2 (given a **`dependent`** method `size`) and that `l3` is equivalent to `Cons("A", Cons("B", Nil()))`.

In Scala we often deal with impure or imprecisely-typed code, however. To integrate with such terms, we provide the **`choose`**`[T]` construct. Operationally, we interpret **`choose`**`[T]` as a non-deterministic choice from `T`, which can be modelled faithfully on the type level as an existentially quantified inhabitant of `T` in a singleton type. Thus, we equate `{ `**`choose`**`[T] }` to `T`, and when typing an impure term such as `Cons(readString(), Nil())` we can assign the type `{ Cons(`**`choose`**`[String], Nil()) }`. Returning to the previous example, this means that even in the presence of impurity, we can perform useful type-level computation and checking:

```
dependent val l2 = Cons(readString(), Nil())
dependent val l3 = l1.concat(l2)
l3: { Cons("A", Cons(choose[String], Nil())) }
```

In a style similar to `concat`, we can define remove on `Lst`:

```
sealed trait Lst {
  dependent def remove(e: String) <: Lst =
    this match {
      case Cons(head, tail) =>
        if (e == head) tail
        else Cons(head, tail.remove(e))
      case _ => throw new Error("element not found")
    }
}
```

Removing `"B"` yields the expected result, while trying to remove `"C"` from `l3` leads to a *compilation error*, since the given program will provably fail at runtime.

```
l3.remove("B"): { Cons("A", Nil()) }
l3.remove("C") // Error: element not found
```

The lists we defined so far can be used to implement a type-safe interface for database tables.

```
dependent case class Table(schema: Lst, data: spark.DataFrame) {
  dependent def join(right: Table, col: String) <: Table = {
    val s1 = this.schema.remove(col)
    val s2 = right.schema.remove(col)
    val newSchema = Cons(col, s1.concat(s2))
    val newData = this.data.join(right.data, col)
    new Table(newSchema, newData)
  }
}
```

In this example, we wrap a weakly-typed implementation of Spark's `DataFrame` in the **`dependent`** class `Table`. The first argument of this class represents the schema of the table as a precisely-typed list. The second argument is the underlying `DataFrame`. In the implementation of `join`, we execute the join operation on the underlying tables (`newData`) and compute the resulting schema corresponding to that join (`newSchema`). By annotating the `join` method as **`dependent`**, the resulting schema is reflected in the type:

```
dependent val schema1 = Cons("age", Cons("name", Nil()))
dependent val schema2 = Cons("name", Cons("unit", Nil()))
dependent val table1  = Table(schema1, ...)
dependent val table2  = Table(schema2, ...)
dependent val joined  = table1.join(table2, "name")
joined: { Table(Cons("name", Cons("age", Cons("unit", Nil())))),
  choose[DataFrame]) }
```

Reflecting table schemas in types increases type safety over the existing weakly-typed interface. For instance, it becomes possible to raise compile-time errors when a user tries to use non-existent columns. This is an improvement over the underlying Spark implementation that would instead fail at runtime.

6.2 Example: Safe Zip

Our first example demonstrated how dependent methods allow inference of precise types. Conversely, we can also use singleton types to constrain method parameters further. In this example, our goal is to write a safer wrapper for functions like `zip` that should only be applicable to lists of the same length. To accomplish this, we can constrain the second

parameter of `zip` as follows:

```
def safeZip(xs: Lst, ys: { sizedLike(xs) }) = unsafeZip(xs, ys)
```

Here we would like `{ sizedLike(xs) }` to be inhabited by all lists of equal length as `xs`, regardless of their elements' values. How can this be achieved, given that `sizedLike(xs)` is a term? By exploiting the non-deterministic interpretation of **`choose`**`[T]`, we can provide a succinct definition for `sizedLike`:

```
dependent def sizedLike(xs: Lst) <: Lst =
  xs match {
    case Nil() => Nil()
    case Cons(x, ys) => Cons(choose[Any], sizedLike(ys))
  }
```

Consider, for instance, the meaning of `{ sizedLike(xs) }` for `xs = Cons(1, Cons(2, Nil()))`. After reduction, we obtain `{ Cons(choose[Any], Cons(choose[Any], Nil())) }`, which is a type that represents all lists of size 2. Thus `safeZip` requires every caller to prove that `xs` and `ys` are of the same length, which ensures that the underlying implementation in `unsafeZip` will never fail or truncate elements from one of the lists.

Unlike `concat` and `remove` that can be used both on the term and the type level, `sizedLike` is intended to be used as a type function, but not at runtime.

6.3 Discussion: From Choices to Existentials

Note that `{ sizedLike(xs) }` cannot be readily expressed using existential types and singletons alone. The given list `xs` might be of an arbitrary size, so the number of existentials needed for all the occurrences of **`choose`**`[Any]` is abstract at this point. More specifically, depending on the size of `xs`, `{ sizedLike(xs) }` corresponds to one of the following existential types:

$$\{\text{nil}\} \quad \exists x_1 : \text{Top}.\,\{\text{cons } x_1 \text{ nil}\} \quad \exists x_1 : \text{Top}.\,\exists x_2 : \text{Top}.\,\{\text{cons } x_1\ (\text{cons } x_2 \text{ nil})\} \quad \ldots$$

An important contribution of our type system is that it allows users to express such existential quantifications conditional on the program unfolding. Our calculus (described in Chapter 7) achieves this by encoding all non-deterministic choices using a single existential per-type annotation. In particular, we represent `{ sizedLike(xs) }` by

$$\exists z : \text{Trail}.\,\{\text{sizedLike' } z \text{ xs}\}$$

where (sizedLike' z xs) is defined by

$$\text{xs match nil}; x, y \Rightarrow \text{cons } (\text{unpack } z.1)\ (\text{sizedLike' } z.2.3\ y)$$

Conceptually, z:Trail corresponds to a map of input values passed to a deterministic version of the program, i.e., sizedLike'. Programs resulting from our encoding are pure and deterministic, so we can perform equational reasoning and apply well-understood techniques for designing sound type systems. At the same time, our encoding is adequate with respect to non-determinism (which, in turn, can approximate other language features). In our example, (unpack z.1) extracts the value at index .1 from the input z. Note that using the argument of the recursive call, z.2.3, we ensure that each invocation of `choose[T]` in the original program is translated with a different index (Section 7.2). This is necessary for sizedLike' to faithfully model the original (non-deterministic) sizedLike, in the sense that each invocation of `choose[T]` can be mapped to a different value. For instance, when xs is a concrete list of two elements, we end up with a type encoded as

$$\exists z\text{:Trail}.\,\{\text{cons (unpack } z.1)\ (\text{cons (unpack } z.2.3.1)\ \text{nil})\}$$

which, given our interpretation of the Trail type, selections like z.1, and the unpack operation, is equivalent to all the lists of two elements.

During type checking, we explicitly eliminate the references to unpack and replace them by fresh existentials:

$$\exists x_1\text{:Top}.\,\exists x_2\text{:Top}.\,\{\text{cons } x_1\ (\text{cons } x_2\ \text{nil})\}$$

That is, we "untangle" individual existentials that had previously been tied up together (Section 7.4). As part of our overall soundness proof (Chapter 8) we show that untangling produces equivalent types, which allows us to match different occurrences of types containing non-deterministic choices when they denote the same sets of values.

Our type system rules are designed to support type checking with such existentials and subtyping. We find that it achieves an appealing combination of expressive power and simplicity: the developers can denote types using functions that generate sets of values, instead of manipulating syntactic representations of types. Studying the essence of such a type system in combination with recursive functions (as we do in the following chapter) is particularly interesting: Not only do recursive functions enable inductive proofs in the usual way where return types form induction hypotheses, but lifting fixpoints to the type level also allows us to encode recursive types. Finally, we note that even if our current set of type-checking rules does not cover as many type equivalences as we may wish to have, our soundness approach based on reducibility semantics and System FR [HVK19] allows us to modularly introduce and prove additional rules in the future.

7 A Calculus for Type-Level Computation

We present a calculus and a type system that capture the core mechanisms required for type-level computation in a dependently-typed language with subtyping. While an implementation on top of Scala must operate in the presence of a much more general subtyping relation, our formalism does not cover all the features of Scala's type system. In the following section, we introduce a functional language with primitives for operating on Lisp-like lists, which gives similar power as the closed type hierarchies that our Scala-implementation can reason about. An extension to other algebraic data types should be straightforward. Our calculus also supports non-deterministic choice from base types and Top. This choice operator allows us, on the one hand, to model imprecisely-typed functions and, on the other hand, to emulate type-level computation.

7.1 Syntax and Semantics

The terms and types of our calculus, $\lambda^{\text{nd}}_{<:\{\}}$, are defined in Figure 7.1. We consider terms and types equivalent up to alpha-renaming. As usual, variables are named x, y or z. We denote the set of free variables of a term t by $fv(t)$. Our language contains first-class functions and constructors for lists, along with pattern matching, and a fixpoint combinator. Programs in our language always terminate because our fixpoint combinator is bounded to a maximum recursion depth and returns a default value otherwise. In $\text{fix}_n(x{:}\,T \Rightarrow t_1,\ t_2)$, n corresponds to the maximum recursion depth, t_1 is the body of fix and t_2 is the default value. We expect that our approach extends to more general solutions, for example, requiring proofs of termination as in most dependently-typed languages [Nor07, BC04], or controlling reduction on the type level using iso-types [YBO16].

The small-step operational semantics given in Figure 7.2 is mostly standard, save for two aspects. First, term evaluation does not get stuck on variables (we include them among the values v) and behaves non-deterministically on the term choose$[B]$, which evaluates to an arbitrary value of type B (i.e., base types or Top). Unlike many other dependently-typed

Terms and Types of $\lambda^{\text{nd}}_{<:\{\}}$:

$$
\begin{array}{rcl}
p, t & := & x \mid \lambda x{:}T.\, t \mid t\ t \mid \text{nil} \mid \text{cons}\ t\ t \mid t\ \text{match}\ t; x, y \Rightarrow t \mid \\
 & & \text{fix}_n(x{:}T \Rightarrow t,\ t) \mid \text{choose}[B] \\
S, T, U, V & := & B \mid \{t\}_T \mid \Pi x{:}T.\, T \\
B & := & \text{Top} \mid \text{List}
\end{array}
$$

Values:

$$
\begin{array}{rcl}
v, v^{\text{Top}} & := & x \mid \lambda x{:}T.\, t \mid v^{\text{List}} \\
v^{\text{List}} & := & \text{nil} \mid \text{cons}\ v\ v^{\text{List}}
\end{array}
$$

Figure 7.1 – The terms and types for $\lambda^{\text{nd}}_{<:\{\}}$.

Evaluation contexts:

$$
\mathcal{E} \quad := \quad [] \mid \mathcal{E}\ t \mid v\ \mathcal{E} \mid \text{cons}\ \mathcal{E}\ t \mid \text{cons}\ v\ \mathcal{E} \mid \mathcal{E}\ \text{match}\ t; x, y \Rightarrow t
$$

Term evaluation:

$$\frac{t \to_\beta t'}{\mathcal{E}[t] \to_\beta \mathcal{E}[t']}\ (\text{BCTX}) \qquad \frac{}{(\lambda x{:}A.\, t)\ v \to_\beta t[x \mapsto v]}\ (\text{BAPP})$$

$$\frac{}{(\text{nil match}\ t_1; x, y \Rightarrow t_2) \to_\beta t_1}\ (\text{BMATCHNIL})$$

$$\frac{}{((\text{cons}\ v_1\ v_2^{\text{List}})\ \text{match}\ t_1; x, y \Rightarrow t_2) \to_\beta t_2[x \mapsto v_1][y \mapsto v_2^{\text{List}}]}\ (\text{BMATCHCONS})$$

$$\frac{n = n' + 1}{\text{fix}_n(x{:}A \Rightarrow t_1,\ t_2) \to_\beta t_1[x \mapsto \text{fix}_{n'}(x{:}A \Rightarrow t_1,\ t_2)]}\ (\text{BFIXREC})$$

$$\frac{n = 0}{\text{fix}_n(x{:}A \Rightarrow t_1,\ t_2) \to_\beta t_2}\ (\text{BFIXDEFAULT})$$

$$\frac{fv(v) = \emptyset}{\text{choose}[\text{Top}] \to_\beta v}\ (\text{BCHOOSETOP}) \qquad \frac{fv(v^{\text{List}}) = \emptyset}{\text{choose}[\text{List}] \to_\beta v^{\text{List}}}\ (\text{BCHOOSELIST})$$

Figure 7.2 – The term evaluation rules and evaluation contexts.

systems, this allows us to express more than just purely-functional programs, as choose[B] conservatively models a term lacking referential transparency. Second, choose[B] allows us to model the situation in which parts of our program may be pure, but are typed in a less precise

manner.

Besides the dependent products usually found in dependently-typed languages, we also include singleton types [Hay91], denoted $\{t\}_U$, which are inhabited only by terms observationally equivalent to t. The *underlying* type U provides an upper bound for the singleton type and is used to guide type inference. For instance, nil can be typed precisely as $\{\text{nil}\}_{\text{List}}$. The identity function on Top can be typed as:

$$\{\lambda x{:}\text{Top}.\, x\}_{\Pi x:\text{Top}.\, \{x\}_{\text{Top}}}$$

For better legibility we often write singletons without their underlying types: $\{\lambda x{:}\text{Top}.\, x\}$.

When used in a type annotation, choose[B] existentially quantifies over an arbitrary value in B. As a result, the base type List is equivalent to {choose[List]} which in turn allows us to express the type of non-empty lists as follows:

$$\{\text{cons (choose[Top]) (choose[List])}\}$$

During type checking, our system rewrites choose[B] to explicit existential quantifications, that are not available in the surface syntax. Internally, we end up with the following type for the above example:

$$\exists x_1{:}\text{Top}.\, \exists x_2{:}\text{List}.\, \{\text{cons } x_1\ x_2\}$$

Semantically, this type corresponds to the infinite union over all elements x_1:Top, x_2:List of $\{\text{cons } x_1\ x_2\}$. As a first step towards representing the impure choose[B] construct, we translate programs in $\lambda^{\text{nd}}_{<:\{\}}$ to a deterministic language, as described below.

7.2 Lowering to a Deterministic Language

In this section, we detail how we eliminate the non-deterministic choose[B] construct. The essence of our translation is to collect all the choices that a non-deterministic execution might need and turn them into an input argument of a deterministic version of the program. Our translation is therefore analogous to a translation from a non-deterministic Turing machine to a deterministic machine that acts as the corresponding verifier [Sip13, Theorem 7.20 on p. 294].

The encoding is performed before type checking, and as a consequence choose[B] is absent from subsequent typing rules. Depending on the context where choose[B] occurs, it takes on different meanings. In the context of terms, choose[B] refers to a specific value in B, picked non-deterministically during program execution. When invoked from inside a singleton type, such as in $\{\text{choose}[B]\}$, our translation will give it the meaning of all values in B. This result arises due to existential quantification over choices, which the translation introduces independently for each type annotation in the program.

Terms and Types of $\lambda^{\text{det}}_{<:\{\}}$:

$$
\begin{array}{rcl}
p, t & := & x \mid \lambda x{:}T.\, t \mid t\ t \mid \text{nil} \mid \text{cons}\ t\ t \mid t\ \text{match}\ t; x, y \Rightarrow t \mid \\
 & & \text{fix}_n(x{:}T \Rightarrow t,\ t) \mid \ t.1 \mid t.2 \mid t.3 \\
S, T, U, V & := & B \mid \{t\}_T \mid \Pi x{:}T.\, T \mid \\
 & & \ \text{Cons}\ T_1\ T_2 \mid t\ \text{Match}\ T; x, y \Rightarrow T \mid \exists x{:}T.\, T \mid \text{Trail}
\end{array}
$$

Values:

$$
v, v^{\text{Top}} \quad := \quad x \mid \lambda x{:}T.\, t \mid v^{\text{List}} \mid \ v.1 \mid v.2 \mid v.3
$$

Figure 7.3 – The terms and types in $\lambda^{\text{det}}_{<:\{\}}$. Constructs not present in $\lambda^{\text{nd}}_{<:\{\}}$ are marked in gray.

We define a lowering from $\lambda^{\text{nd}}_{<:\{\}}$, the surface language, to $\lambda^{\text{det}}_{<:\{\}}$, which we then use in subsequent type checking. In Figure 7.3 we give the terms and types of $\lambda^{\text{det}}_{<:\{\}}$ with the differences to $\lambda^{\text{nd}}_{<:\{\}}$ highlighted in gray. First, note the absence of choose[B], which is eliminated by the lowering. We include types for list constructors (Cons T_1 T_2) and matches. The type for matches, t Match T_2; $x, y \Rightarrow T_3$, represents either T_2 or the substituted form of T_3, depending on the value of t. These types are used later, during type inference, and guide subtyping. The other additions, i.e., existential types, the base type Trail, and selections on trails, $t.n$, are discussed below in the lowering step.

Encoding choose[T]

Lowering produces a deterministic program that, thanks to an extra parameter, captures all of the potential behaviors of the original (non-deterministic) program. We express the lowered program as a function of *trails*. Intuitively, a trail τ contains all the information necessary to recover the non-deterministic choices made in a concrete execution of the original program.

Given a program t in $\lambda^{\text{nd}}_{<:\{\}}$, the lowering yields $t_f = \lambda z_\alpha : \text{Trail}.\, \langle\langle t \rangle\rangle^{z_\alpha}$ in $\lambda^{\text{det}}_{<:\{\}}$, which encodes the behavior of t as a pure function. That is, for any given (potentially non-deterministic) reduction resulting in v, there exists a trail τ such that $(t_f\ \tau) \to^*_\beta \langle\langle v \rangle\rangle$[1].

In Figure 7.4 we describe the transformation of terms, $\langle\langle t \rangle\rangle^p$, and the transformation of types, $\langle\langle T \rangle\rangle$. At its core, $\langle\langle t \rangle\rangle^p$ replaces each invocation of choose[B] by an application of a function unpack$_B$ to a trail. Given the original program, one of its non-deterministic executions can be characterized by a mapping from every invocation of choose[B] to the resulting value in B. With respect to our evaluation relation $\to_\beta$, such a mapping can be obtained by recording the sequence of non-deterministic choices in BChooseTop and BChooseList. The initial trail

[1] We omit the trail argument in $\langle\langle v \rangle\rangle$, as it is irrelevant when translating values, which can only contain choose[B] underneath lambdas.

$$
\begin{aligned}
\langle\langle T\rangle\rangle &: \text{Type} \to \text{Type} \\
\langle\langle B\rangle\rangle &:= B \\
\langle\langle \{t\}_T\rangle\rangle &:= \exists z{:}\text{Trail}.\, \{\langle\langle t\rangle\rangle^z\}_{\langle\langle T\rangle\rangle} \quad \text{where } z \text{ is fresh} \\
\langle\langle \Pi x{:}S.\, T\rangle\rangle &:= \Pi z{:}\text{Trail}.\, \Pi x{:}\langle\langle S\rangle\rangle.\, \langle\langle T\rangle\rangle \quad \text{where } z \text{ is fresh} \\
\langle\langle \text{Cons } T_1\ T_2\rangle\rangle &:= \text{Cons } \langle\langle T_1\rangle\rangle\ \langle\langle T_2\rangle\rangle \\
\langle\langle t \text{ Match } T_2; x, y \Rightarrow T_3\rangle\rangle &:= \exists z{:}\text{Trail}.\, \langle\langle t\rangle\rangle^z \text{ Match } \langle\langle T_2\rangle\rangle; x, y \Rightarrow \langle\langle T_3\rangle\rangle \\
&\qquad \text{where } z \text{ is fresh}
\end{aligned}
$$

$$
\begin{aligned}
\langle\langle t\rangle\rangle^p &: \text{Term} \to \text{Term} \to \text{Term} \\
\langle\langle \text{choose}[B]\rangle\rangle^p &:= \mathit{unpack}_B\ p \\
\langle\langle \lambda x{:}T.\, t\rangle\rangle^p &:= \lambda z{:}\text{Trail}.\, \lambda x{:}\langle\langle T\rangle\rangle.\, \langle\langle t\rangle\rangle^z \quad \text{where } z \text{ is fresh} \\
\langle\langle t_1\ t_2\rangle\rangle^p &:= t_1'\ p.3\ t_2' \quad \text{where } t_1' = \langle\langle t_1\rangle\rangle^{p.1} \text{ and } t_2' = \langle\langle t_2\rangle\rangle^{p.2} \\
\langle\langle x\rangle\rangle^p &:= x \\
\langle\langle \text{nil}\rangle\rangle^p &:= \text{nil} \\
\langle\langle \text{cons } t_1\ t_2\rangle\rangle^p &:= \text{cons } \langle\langle t_1\rangle\rangle^{p.1}\ \langle\langle t_2\rangle\rangle^{p.2} \\
\langle\langle t_1 \text{ match } t_2; x, y \Rightarrow t_3\rangle\rangle^p &:= \langle\langle t_1\rangle\rangle^{p.1} \text{ match } \langle\langle t_2\rangle\rangle^{p.2}; x, y \Rightarrow \langle\langle t_3\rangle\rangle^{p.3} \\
\langle\langle \text{fix}_n(x{:}T \Rightarrow t_1,\ t_2)\rangle\rangle^p &:= \text{fix}_n(x{:}\langle\langle T\rangle\rangle \Rightarrow \langle\langle t_1\rangle\rangle^{p.1},\ \langle\langle t_2\rangle\rangle^{p.2})
\end{aligned}
$$

Figure 7.4 – The rules for lowering programs in $\lambda^{\text{nd}}_{<:\{\}}$ to $\lambda^{\text{det}}_{<:\{\}}$, yielding a deterministic program without the non-deterministic choose[B] construct.

z_α used to evaluate the lowered program corresponds to a complete mapping for some non-deterministic execution. Throughout the lowered program we build up *selections* on the initial trail using $t.n$, which correspond to subtrails. Calls to unpack_B then use the given subtrail to return a value. In our translation we take care never to apply unpack_B to the same trail twice: Doing so would incorrectly constrain the outcome of the corresponding invocations to be coupled together.

In the translation of abstractions, we create a fresh trail parameter z, which is then used to translate the function's body. This is essential, as it ensures that in each function invocation we allow for different non-deterministic choices. Note that it does not seem feasible to enumerate all the possible invocations of choose[B] statically: For one, the outcome of a choose[B] might influence the control flow of the original program, and, in general, the length of the execution may be unbounded. To translate an application, we select on the current trail and pass it as the additional argument. Extending the selection is crucial to ensure that recursive calls can be distinguished in their non-deterministic choices. Consequently, we also adapt types that occur in the annotations of abstractions and fixpoints using $\langle\langle T\rangle\rangle$. In particular, Π-types are rewritten to account for the newly-introduced Trail parameter.

Note that in $\langle\langle T \rangle\rangle$ we do not propagate and extend an existing trail as we do with p in $\langle\langle t \rangle\rangle^p$. When translating a singleton type $\{t\}_U$ we instead wrap the resulting type in a fresh existential type $\exists z : \text{Trail}.$, which is used in the translation of t. This is what gives choose[B] its dual meaning at the type level: Rather than referring to one particular choice, it encompasses all of them.

Our lowering is related to monadic encodings in the style of Wadler [Wad90a]. The resulting encoding is simpler than a typical State monad because we only care about the distinctness of trails, rather than encoding the evaluation order and threading the resulting state from one subterm to another.

Trails, More Carefully

We will now give a more concrete definition of what properties a trail, and the operations that act upon it, must satisfy. We organize the sequence of values of a trail τ as a ternary tree. Leaves of this tree contain a value, and a tag that encodes the type of the value. Consider $t..p$ as notation for $(\ldots(t.n_1)\ldots).n_k$, i.e., applying a series of selections $p = .n_1 \ldots .n_k$ where $n_1, \ldots, n_k \in \{1,2,3\}$ to t. Given a trail τ and selections p, $\tau..p$ represents the subtree of τ when selecting the n_i-th child at the i-th level of τ. For trees τ, τ' and a selection p, (update $\tau\ p\ \tau'$) replaces the subtree selected by p in τ by τ', so that (update $\tau\ p\ \tau'$)$..p = \tau'$. The unpack$_B$: (Πx: Trail. B) function returns the value at the root of the given tree, if the type-tag of the value there encodes B, and nil otherwise.

7.3 The Type System

We introduce our type system for $\lambda^{\text{det}}_{<:\{\}}$ which consists of several inter-dependent relations:

- type inference and checking ($\Uparrow$ and $\Downarrow$ in Figure 7.5),
- subtyping ($<:$ in Figure 7.6), and
- type normalization ($\rightarrow_N$ in Figure 7.7).

To improve legibility of the rules, we omit well-formedness conditions, and presume that types are well-formed in the given context. For singleton types, in particular, we maintain the assumption that for any $\{t\}_U$ we encounter, t inhabits U. Similarly, for every list match type (t Match T_2; $x, y \Rightarrow T_3$) we assume that t inhabits List.

7.3.1 Type Inference and Underlying Types

Figure 7.5 presents rules that infer the most precise type for a given term t. In particular, type inference will yield a singleton type $\{t\}_U$, if t is well-typed. For each construct we attach an

$$\frac{\Gamma(x) = T}{\Gamma \vdash x \Uparrow \{x\}_T} \text{ (TVAR)} \qquad \frac{\Gamma, x : S \vdash t \Uparrow T}{\Gamma \vdash \lambda x{:}S.\, t \Uparrow \{\lambda x{:}S.\, t\}_{\Pi x:S.\, T}} \text{ (TABS)}$$

$$\frac{\Gamma \vdash t_1 \Uparrow V \quad \lceil V \rceil = \Pi x{:}S.\, T \quad \Gamma \vdash t_2 \Downarrow S}{\Gamma \vdash t_1\ t_2 \Uparrow T[x \mapsto t_2]} \text{ (TAPP)}$$

$$\frac{\Gamma, x : T \vdash t_1 \Downarrow T \quad \Gamma \vdash t_2 \Downarrow T}{\Gamma \vdash \mathrm{fix}_n(x{:}T \Rightarrow t_1,\ t_2) \Uparrow \{\mathrm{fix}_n(x{:}T \Rightarrow t_1,\ t_2)\}_T} \text{ (TFIX)}$$

$$\frac{}{\Gamma \vdash \mathrm{nil} \Uparrow \{\mathrm{nil}\}_{\mathrm{List}}} \text{ (TNIL)}$$

$$\frac{\Gamma \vdash t_1 \Uparrow T_1 \quad \Gamma \vdash t_2 \Uparrow T_2 \quad \Gamma \vdash T_2 <: \mathrm{List}}{\Gamma \vdash \mathrm{cons}\ t_1\ t_2 \Uparrow \{\mathrm{cons}\ t_1\ t_2\}_{\mathrm{Cons}\ T_1\ T_2}} \text{ (TCONS)}$$

$$\frac{\Gamma \vdash t_1 \Downarrow \mathrm{List} \quad \Gamma \vdash t_2 \Uparrow T_2 \quad \Gamma, x : \mathrm{Top}, y : \mathrm{List} \vdash t_3 \Uparrow T_3}{\Gamma \vdash t_1\ \mathrm{match}\ t_2;\ x, y \to t_3 \Uparrow \{t_1\ \mathrm{match}\ t_2;\ x, y \to t_3\}_{t_1\ \mathrm{Match}\ T_2; x, y \Rightarrow T_3}} \text{ (TMATCH)}$$

$$\frac{\Gamma \vdash t \Downarrow \mathrm{Trail}}{\Gamma \vdash t.k \Uparrow \{t.k\}_{\mathrm{Trail}}} \text{ (TDOT)} \qquad \frac{\Gamma \vdash t \Uparrow T' \quad \Gamma \vdash T' <: T}{\Gamma \vdash t \Downarrow T} \text{ (TCHECK)}$$

$$\lceil \{t\}_U \rceil := \begin{cases} \Pi x{:}S.\, \{t\ x\}_T & \text{if } U = \Pi x{:}S.\, T \text{ for some types } S, T \\ \lceil U \rceil & \text{otherwise} \end{cases}$$

$$\lceil T \rceil := T \quad \text{if there exist no } t \text{ and } U \text{ such that } T = \{t\}_U$$

Figure 7.5 – The inference and checking rules.

upper bound as the singleton's underlying type U. In TABS, for instance, we "tag" the singleton type inferred for a function with the corresponding Π-type, and in TCONS we attach a special type Cons T_1 T_2 only present during type checking. The underlying type is used to guide checks in TAPP and various subtyping rules.

In TAPP, in particular, we expose and match against the underlying function type of t_1 using the auxiliary $\lceil \cdot \rceil$ function. Our goal here is to check that applying t_1 to t_2 is safe, as usual, while also maintaining a precise version of the underlying type. Assuming we inferred $V = \{t_1\}_{\Pi x:S.\, T}$ for t_1, $\lceil V \rceil$ will yield a Π-type equivalent to t_1 for all x in S, i.e., $\Pi x{:}S.\, \{t_1\ x\}_T$. We then substitute the argument term in the result type, yielding $\{t_1\ t_2\}_{T[x \mapsto t_2]}$. In TAPP, TCONS and TFIX we also refer to a type-checking relation ($t \Downarrow T$) which is defined as a shorthand (see TCHECK) for inferring the type of t and checking against the expected type T using the subtyping relation.

$$\frac{}{\Gamma \vdash T <: T} \text{ (SubRefl)} \qquad \frac{}{\Gamma \vdash T <: \text{Top}} \text{ (SubTop)}$$

$$\frac{\Gamma \vdash T_1 <: T_2}{\Gamma \vdash \{t\}_{T_1} <: T_2} \text{ (SubSing)} \qquad \frac{\Gamma, x : S \vdash T <: U}{\Gamma \vdash \exists x{:}S.\, T <: U} \text{ (SubExistsLeft)}$$

$$\frac{\{t\}_U = \text{solve}_x(T_1, S, T_2) \quad \Gamma \vdash \{t\}_U <: S \quad \Gamma \vdash T_1 <: T_2[x \mapsto t]}{\Gamma \vdash T_1 <: \exists x{:}S.\, T_2} \text{ (SubExistsRight)}$$

$$\frac{}{\Gamma \vdash \text{Cons } S\ T <: \text{List}} \text{ (SubCons1)} \qquad \frac{\Gamma \vdash S_1 <: S_2 \quad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash \text{Cons } S_1\ T_1 <: \text{Cons } S_2\ T_2} \text{ (SubCons2)}$$

$$\frac{\Gamma \vdash S_2 <: T \quad \Gamma, x : \text{Top}, y : \text{List} \vdash S_3 <: T}{\Gamma \vdash t_1 \,\text{Match}\, S_2;\, x, y \Rightarrow S_3 <: T} \text{ (SubMatch)}$$

$$\frac{\Gamma \vdash S_2 <: S_1 \quad \Gamma, x : S_2 \vdash T_1 <: T_2}{\Gamma \vdash \Pi x{:}S_1.\, T_1 <: \Pi x{:}S_2.\, T_2} \text{ (SubPi)}$$

$$\frac{\Gamma \vdash T_1 \to_N T_1' \quad \Gamma \vdash T_2 \to_N T_2' \quad \Gamma \vdash \mathscr{U}(T_1') <: \mathscr{U}(T_2')}{\Gamma \vdash T_1 <: T_2} \text{ (SubNorm)}$$

Figure 7.6 – The subtyping rules.

7.3.2 Subtyping and Type Normalization

The subtyping relation is given in Figure 7.6. Rules for reflexivity (SubRefl), Π-types (SubPi), Top and the List base type (SubTop, SubCons1) are standard. The Cons type introduced during inference can be subtyped covariantly (SubCons2); the type (x Match T_2; $x, y \Rightarrow T_3$) assigned to matches behaves like a union of T_2 and T_3, while allowing T_3 to retain variables bound in the pattern (SubMatch). Using SubSing we can approximate a singleton type $\{t\}_{T_1}$ occuring on the left-hand side by its upper bound T_1.

Our system allows for computation on types to take place during subtyping. Subtyping rule SubNorm bundles two kinds of normalizing behavior: We first reduce both sides T_1 and T_2 using type normalization. We then attempt to replace any newly-exposed occurrences of unpack_B by fresh existentials of type B via the untangle function $\mathscr{U}(\cdot)$.

The rules for type normalization are detailed in Figure 7.7. We merely distribute over Π-types, existentials, and Cons-types (NPi, NExists1, NExists2, NCons). Since S is assumed to be inhabited in existential types $\exists x{:}S.\, T$, we eliminate such quantifications whenever the result type T does not contain x free (NExists1).

Singleton types $\{t\}_U$ may be normalized using NSing, in which we first reduce t using beta-delta reduction (Figure 7.8). Beta-delta reduction is defined as a context-aware extension

$$\frac{T \in \{\text{Top}, \text{List}\}}{\Gamma \vdash T \to_N T}\ (\text{NBASE}) \qquad \frac{\Gamma \vdash t \to^*_{\beta\delta} t' \quad \Gamma \vdash t' \Uparrow \{t''\}_V}{\Gamma \vdash \{t\}_U \to_N \{t''\}_V}\ (\text{NSING})$$

$$\frac{\Gamma \vdash S \to_N S' \quad \Gamma, x{:}S' \vdash T \to_N T'}{\Gamma \vdash \Pi x{:}S.\, T \to_N \Pi x{:}S'.\, T'}\ (\text{NPI})$$

$$\frac{\Gamma \vdash S \to_N S' \quad \Gamma, x : S' \vdash T \to_N T' \quad x \notin fv(T)}{\Gamma \vdash \exists x{:}S.\, T \to_N T'}\ (\text{NEXISTS1})$$

$$\frac{\Gamma \vdash S \to_N S' \quad \Gamma, x : S' \vdash T \to_N T' \quad x \in fv(T)}{\Gamma \vdash \exists x{:}S.\, T \to_N \exists x{:}S'.\, T'}\ (\text{NEXISTS2})$$

$$\frac{\Gamma \vdash T_1 \to_N T_1' \quad \Gamma \vdash T_2 \to_N T_2'}{\Gamma \vdash \text{Cons}\ T_1\ T_2 \to_N \text{Cons}\ T_1'\ T_2'}\ (\text{NCONS})$$

$$\frac{\Gamma \vdash t \to^*_{\beta\delta} \text{nil} \quad \Gamma \vdash T_2 \to_N T_2'}{\Gamma \vdash t\,\text{Match}\, T_2; x, y \Rightarrow T_3 \to_N T_2'}\ (\text{NMATCH1})$$

$$\frac{\Gamma \vdash t \to^*_{\beta\delta} \text{cons}\ t_1\ t_2 \quad \Gamma, x{:}\{t_1\}_{\text{Top}}, y{:}\{t_2\}_{\text{List}} \vdash T_3 \to_N T_3'}{\Gamma \vdash t\,\text{Match}\, T_2; x, y \Rightarrow T_3 \to_N T_3'}\ (\text{NMATCH2})$$

$$\frac{\Gamma \vdash t \to^*_{\beta\delta} t' \quad \text{if neither of the above rules apply}}{\Gamma \vdash t\,\text{Match}\, T_2; x, y \Rightarrow T_3 \to_N t'\,\text{Match}\, T_2; x, y \Rightarrow T_3}\ (\text{NMATCH3})$$

Figure 7.7 – The type normalization rules.

of beta reduction (seen previously in Figure 7.2) with a new rule BDDELTA, which allows the elimination of variables whose precise definition is known from the context (a similar evaluation relation is found in [Cou03]). Delta reduction steps may lead the underlying type U to go out-of-sync with the newly computed term t'. For instance, given the context $\Gamma = x : \{\text{nil}\}_{\text{List}}$ and the type $\{x\}_{\text{Top}}$, if we were to normalize using the beta-delta reduction $x \to^*_{\beta\delta} \text{nil}$ alone, we would arrive at $\{\text{nil}\}_{\text{Top}}$. We can improve upon this — and in fact might rely on it in later subtyping queries — by redoing type inference on t', yielding a singleton type with a better bound (in our example, $\{\text{nil}\}_{\text{List}}$).

$$\frac{\Gamma \vdash t \to_{\beta\delta} t'}{\Gamma \vdash \mathscr{E}[t] \to_{\beta\delta} \mathscr{E}[t']}\ (\text{BDCTX}) \qquad \frac{\Gamma(x) = \{t\}_U}{\Gamma \vdash x \to_{\beta\delta} t}\ (\text{BDDELTA}) \qquad \frac{t \to_\beta t'}{\Gamma \vdash t \to_{\beta\delta} t'}\ (\text{BDBETA})$$

Figure 7.8 – The rules of beta-delta reduction.

The rules for match allow reduction of (t Match T_2; $x, y \Rightarrow T_3$) depending on the beta-delta reduction of t. That is, we normalize to T_2 when t = nil (NMATCH1), and to T_3 when t is a cons (NMATCH2). In the latter case, we add precisely-typed bindings that allow for x and y to be δ-reduced during the normalization of T_3. If t does not fit either case, we instead normalize to a type that incorporates the reduced t.

7.3.3 Subtyping Existential Types

Existentials only enter the program when lowering type annotations in $\lambda^{\text{nd}}_{<:\{\}}$ to $\lambda^{\text{det}}_{<:\{\}}$, and in SUBNORM via $\mathscr{U}(\cdot)$. When encountered on the left-hand side, existential types are eliminated by adding $x{:}S$ to the context (SUBEXISTSLEFT). When an existential occurs on the right-hand side, we try to guess a valid instantiation t for x (SUBEXISTSRIGHT). The subroutine that guesses t is modelled abstractly by $\text{solve}_x(T_1, S, T_2)$, which is expected to return a singleton $\{t\}_U$. We make no assumptions on the implementation of solve_x, but verify that the outcome is a valid solution by checking that it conforms to S and makes the instantiated right-hand side a super-type of the left-hand side. In Section 7.5 we discuss one possible concrete implementation of solve_x.

7.4 Untangling Trails

In Section 7.2 we explained how to translate the non-deterministic choose[B] construct into an application of unpack_B to a trail. Therefore, during type checking, we often face subtyping queries involving applications of unpack_B on the right-hand side. For instance, when checking the program

$$(\lambda x{:}\{\text{cons choose[Top] choose[List]}\}.\, x)\ (\text{cons nil nil})$$

we will encounter the following subtyping query:

$$\{\text{cons nil nil}\} <: \exists z{:}\text{Trail}.\, \{\text{cons } (\text{unpack}_{\text{Top}}\ z.1)\ (\text{unpack}_{\text{List}}\ z.2)\}$$

Though the right-hand side is an existential type, this query cannot be solved by SUBEXISTSRIGHT directly, unless the solve subroutine possesses some deep knowledge about unpack_B. That is, a priori it is not evident that there exists a trail z such that both $(\text{unpack}_{\text{Top}}\ z.1)$ and $(\text{unpack}_{\text{List}}\ z.2)$ reduce to nil.

Using the properties on trails and unpack_B introduced in Section 7.2 we can prove that the type on the right-hand side is, in fact, equivalent to an explicitly quantified version, i.e.,

$$\exists z{:}\text{Trail}.\, \{\text{cons } (\text{unpack}_{\text{Top}}\ z.1)\ (\text{unpack}_{\text{List}}\ z.2)\} \;=\; \exists x_1{:}\text{Top}.\, \exists x_2{:}\text{List}.\, \{\text{cons } x_1\ x_2\}$$

To see why the inclusion from left to right holds, consider any trail τ with values v_1^{Top} and v_2^{List} stored at indices .1 and .2. We can thus instantiate x_1 and x_2 to v_1^{Top} and v_2^{List}, to obtain the same term on both sides. From right to left we can construct a tree containing the values of x_1

$$\mathcal{U} : \text{Type} \to \text{Type} \qquad \textit{“Untangle all Trail existentials”}$$

$$\mathcal{U}(\exists x{:}\text{Trail}.\, T) := \mathcal{W}_x(\text{ps}, \mathcal{U}(T)) \quad \text{where ps} = \text{trailsOf}_x(\mathcal{U}(T))$$

$$\mathcal{U}(\exists x{:}S.\, T) := \exists x{:}\mathcal{U}(S).\, \mathcal{U}(T) \quad \text{if } S \neq \text{Trail}$$

$$\mathcal{U}(\Pi x{:}S.\, T) := \Pi x{:}\mathcal{U}(S).\, \mathcal{U}(T)$$

$$\mathcal{U}(\{t\}_U) := \{t\}_{\mathcal{U}(U)} \qquad \mathcal{U}(\text{Cons } T_1\ T_2) := \text{Cons } \mathcal{U}(T_1)\ \mathcal{U}(T_2)$$

$$\mathcal{U}(B) := B \qquad \mathcal{U}(t \text{ Match } T_2;\, x, y \Rightarrow T_3) := t \text{ Match } \mathcal{U}(T_2);\, x, y \Rightarrow \mathcal{U}(T_3)$$

$$\mathcal{W}_x : \text{Id} \to 2^{\text{Term}} \to \text{Type} \to \text{Type} \qquad \textit{“Untangle one Trail existential”}$$

$$\mathcal{W}_x(\{\}, T) := T$$

$$\mathcal{W}_x(\{p\} \uplus \text{ps}, T) := \begin{cases} \mathcal{W}_x(\text{ps}, \exists y{:}B.\, T') & \text{if } p \notin \text{trailsOf}_x(T') \\ \mathcal{W}_x(\text{ps}, \exists y{:}\text{Trail}.\, T'') & \text{otherwise} \end{cases}$$

where T' is T with all occurrences of $(\text{unpack}_B\ x..p)$ replaced by y,
T'' is T with all occurrences of $x..p$ replaced by y, and y is fresh.

$\text{trailsOf}_x(T) \subseteq \text{Term}$ *“Collect all trails rooted in x”*

$\text{trailsOf}_x(T)$ is a set of maximal selections p where $x..p$ appears in T

Figure 7.9 – The untangle function $\mathcal{U}$ and additional auxiliary functions.

and x_2 at indices .1 and .2. This same reasoning can be applied to all functions of Trail that we encounter after our lowering to $\lambda^{\text{det}}_{<:\{\}}$. Furthermore, it generalizes to an arbitrary number of selections on z, as long as the selections are not prefixes of one another, which is ensured by our lowering step.

To exploit this property, we define the untangle function $\mathcal{U}(\cdot)$, which transforms the left-hand side of the equality above to the right-hand side. We use $\mathcal{U}(\cdot)$ during normalization in SUBNORM. In our example, this leads to a simpler subtyping query:

$$\{\text{cons nil nil}\} <: \exists x_1{:}\text{Top}.\, \exists x_2{:}\text{List}.\, \{\text{cons } x_1\ x_2\}$$

At this point we can apply SUBEXISTSRIGHT twice, which could find valid assignments for both x_1 and x_2 (i.e., nil) using straightforward unification.

The definition of $\mathcal{U}(T)$ is given in Figure 7.9. Given the conditions on trails mentioned above, we prove untangling always yields equivalent types (see Chapter 8).

7.5 From Rules to Algorithms

The type system we presented above comes close to being algorithmic. All of the rules for type inference and most of the rules for subtyping and type normalization are already syntax-directed. To derive the normalization of an existential type one has to choose between NEXISTS1 and NEXISTS2, but only one of them will ever succeed due to the condition on x being free in T'. It is therefore straightforward to formulate these cases as a single, effective rule. In the remainder, we note two more substantial adjustments that are needed for an effective formulation of our type system.

The first adjustment is to only apply SUBNORM (type normalization) at the very beginning of subtyping queries (for example, in TCHECK), and before any subderivation that adds a binder to the context (for example, in the second premise of SUBPI). Applying SUBNORM must remain optional, however, since forcing normalization can lead to subderivations that grow *ad infinitum*, for instance by normalizing under matches and re-entering type inference in NSING. Beyond making SUBNORM optional, in practice it is useful to allow for a fast path in subtyping. Given a subtyping query $T_1 <: T_2$, one can first try to prove a stronger subtyping relation, where the left-hand side T_1 is approximated by $\lceil T_1 \rceil$. We found that this greatly reduces the need for complex subtyping derivations, e.g., when checking against the List base type in TCONS and TMATCH.

The second adjustment lies in SUBEXISTSRIGHT, where we require a procedure for solve_x. In principle, $\text{solve}_x(T_1, S, T_2)$ could do arbitrarily deep reasoning about the involved types, but our experience shows that a unification-like procedure is sufficient for the use cases presented here. We experimented with a particularly simple variant: solve_x performs a separate subtyping query which constrains x in a greedy manner using a modified version of SUBREFL. In this modified rule, the computation of (syntactic) type equality looks for any appearances of x. If x appears on either side of the comparison, the corresponding term on the other side is picked as a greedy solution of solve_x. This naive syntactic approach can result in an instantiation that is not well-formed in the original context Γ, in which case we simply fail. One could, of course, try to incrementally improve this approach by trying to rewrite t to an equivalent term well-formed in Γ. It would be interesting to explore a more general constraint-solving approach.

To more easily experiment with the algorithmic rules we implemented a standalone type checker for $\lambda^{\text{nd}}_{<:\{\}}$.[2] Its implementation provides an extended surface syntax for the calculus and integrates the heuristics outlined above. We have used it to debug typing derivations and establish a suite of basic use cases such as heterogeneous lists and manipulations thereof. By further adjusting the heuristic for solve_x in SUBEXISTSRIGHT we managed to check encodings of various existing typing features such as recursive types and genericity. A number of such benchmarks can be found in the `examples/sdep/` subfolder of our implementation's repository.

[2] Our type checker for $\lambda^{\text{nd}}_{<:\{\}}$ is available at https://github.com/epfl-lara/StainlessFit/tree/scaladep-2.

8 Soundness by Reduction to System FR

In this chapter we discuss how soundness of $\lambda^{\text{det}}_{<:\{\}}$ follows by reduction to another dependently-typed calculus. Our formalization uses the language of System FR [HVK19], a calculus that was recently presented as a foundation for the Stainless program verifier [LAR21] and possesses a denotational semantics. We build upon the existing mechanized soundness proof of System FR in Coq.[1] Below we describe the embedding $\langle\cdot\rangle$ of $\lambda^{\text{det}}_{<:\{\}}$ terms and types into System FR (see Figure 8.1) and state the main result.

8.1 Embedding Terms

Functions and applications are represented trivially using System FR's lambda abstractions and applications, which behave identically to ours. Lists are encoded in the typical way as a sum of unit for nil and a pair of a head and a tail for cons.

Our embedding of fix ensures trivial termination, following the bounded recursion behavior of $\lambda^{\text{nd}}_{<:\{\}}$. While we believe that the addition of general recursion to $\lambda^{\text{nd}}_{<:\{\}}$would not present a problem (as discussed before in Section 7.1), System FR is normalizing, and thus prevents us from lifting this restriction in our current formalization.

8.2 Embedding Types

Π-types along with existential types are represented trivially. The type of lists is expressed in the usual way through a recursive type. A singleton type $\{t\}_T$ is encoded using the type $\{\{\ v : T \mid v \equiv t\ \}\}$ of System FR, which represents all values in T that are observationally equivalent to t. Observational equivalence is supported as a type in the current formalization of System FR, even though this type was not supported in the paper [HVK19]. The (Cons T_1 T_2) type of $\lambda^{\text{det}}_{<:\{\}}$ is translated by existentially quantifying over any combination of values in T_1 and T_2. For the type of matches, $(t$ Match $T_2; x, y \Rightarrow T_3)$, we take the union of each of T_2's and T_3's

[1] The proof development is available at https://github.com/epfl-lara/SystemFR.

Translation of terms to System FR:

$$
\begin{aligned}
\langle x \rangle &:= x \\
\langle \lambda x{:}T.\, t \rangle &:= \lambda x.\, \langle t \rangle \\
\langle t_1\ t_2 \rangle &:= \langle t_1 \rangle\ \langle t_2 \rangle \\
\langle \text{nil} \rangle &:= \text{left}\ (\text{Unit} + (\text{Top}, \langle \text{List} \rangle))(()) \\
\langle \text{cons}\ t_1\ t_2 \rangle &:= \text{right}\ (\text{Unit} + (\text{Top}, \langle \text{List} \rangle))((\langle t_1 \rangle, \langle t_2 \rangle)) \\
\langle t_1\ \text{match}\ t_2;\, x, y \Rightarrow t_3 \rangle &:= \text{either\_match}(\langle t_1 \rangle, z \Rightarrow \langle t_2 \rangle, z \Rightarrow \langle t_3 \rangle[x \mapsto \pi_1\, z][y \mapsto \pi_2\, z]) \\
\langle \text{fix}_n(x{:}X \Rightarrow t_1,\ t_2) \rangle &:= \text{fix}(x \Rightarrow \lambda y{:}\text{Nat}.\ \text{match}(y,\ \langle t_2 \rangle,\ y' \Rightarrow \langle t_1 \rangle[x \mapsto x\, y'])\ n
\end{aligned}
$$

Translation of types to System FR:

$$
\begin{aligned}
\langle \text{Top} \rangle &:= \text{Top} \\
\langle \text{List} \rangle &:= \forall n.\, \text{Rec}(n)(X \Rightarrow \text{Unit} + (\text{Top}, X)) \\
\langle \{t\}_T \rangle &:= \{\{\ v : \langle T \rangle \mid v \equiv \langle t \rangle\ \}\} \\
\langle \Pi x{:}S.\, T \rangle &:= \Pi x{:}\langle S \rangle.\, \langle T \rangle \\
\langle \text{Cons}\ T_1\ T_2 \rangle &:= \exists\, x_1{:}\langle T_1 \rangle.\, \exists\, x_2{:}\langle T_2 \rangle.\, \{\langle \text{cons}\ x_1\ x_2 \rangle\}_{\text{List}} \\
\langle t\ \text{Match}\ T_2;\, x, y \Rightarrow T_3 \rangle &:= \{\{\ v : \langle T_2 \rangle \mid t \equiv \text{left}\ ()\ \}\} \cup \\
&\qquad \exists\, y_1{:}\text{Top}.\, \exists\, y_2{:}\langle \text{List} \rangle. \\
&\qquad\quad \{\{\ v : \langle T_3 \rangle[x_1 \mapsto y_1][x_2 \mapsto y_2] \mid t \equiv \text{right}\ (y_1, y_2)\ \}\} \\
\langle \exists x{:}S.\, T \rangle &:= \exists x{:}\langle S \rangle.\, \langle T \rangle
\end{aligned}
$$

Figure 8.1 – The embedding of $\lambda^{\text{det}}_{<:\{\}}$ terms and types into System FR.

interpretation, conditional on whether the scrutinee t reduces to nil or a cons.

Given that System FR assigns a reducibility semantics to its types, our embedding also affords us with denotations for all the types of $\lambda^{\text{det}}_{<:\{\}}$. That is, given the set of reducible values $[\![T]\!]_v$ of type T in System FR, the meaning of a type T' in $\lambda^{\text{det}}_{<:\{\}}$ is given by $[\![\langle T' \rangle]\!]_v$. For instance, $[\![\langle \text{List} \rangle]\!]_v = \{\text{cons}\ v_1\ (\ldots(\text{cons}\ v_n\ \text{nil})\ldots) \mid n \geq 0, \forall i.\, v_i \in [\![\text{Top}]\!]_v\}$. Existential types $[\![\exists x{:}S.\, T]\!]_v$ are the union of all $[\![T[x \mapsto s]]\!]_v$ for all $s \in [\![S]\!]_v$.

8.3 Formalized Soundness Statement

Using the above embedding, we have proved that all of the rules for type inference, subtyping and type normalization presented in Section 7.3 are admissible with respect to the reducibility semantics of types. We built our mechanization on top of System FR's existing Coq formalization. The respective lemmas are proven under the additional assumptions given to us via the well-formedness rules mentioned in Section 7.3. Namely, the following are assumed to hold:

- In rules for type inference, subtyping, and type normalization, we require well-formedness in the current context and inhabitedness for singleton and list match types.
- During delta-beta reduction, we require terms to be normalizing in the current context.
- Trails and their operations are kept abstract and specified using axioms in file `Trail.v`.

The entirety of our definitions and proofs consists of ~7k lines of Coq in addition to the previous development of System FR soundness, which consisted of ~20k lines.

We can thus state soundness for $\lambda^{\text{det}}_{<:\{\}}$ programs in terms of the reducibility judgment $\Gamma \vDash t : T$ of System FR. The latter holds when, for all substitutions γ, such that for all $(x, S) \in \Gamma$ we have $\gamma(x) \in [[\gamma(S)]]_v$, then $\gamma(t) \in [[\gamma(T)]]_v$. Let $\langle\Gamma\rangle$ be the context with all $\lambda^{\text{det}}_{<:\{\}}$ types embedded into System FR types.

Theorem 1 (Soundness) *Given a context Γ and a $\lambda^{\text{det}}_{<:\{\}}$ term t, if type inference yields a type T, then t is reducible at that type. That is, if $\Gamma \vdash t \Uparrow T$ holds, then $\langle\Gamma\rangle \vDash \langle t\rangle : \langle T\rangle$.*

Note that the traditional notion of type safety for t follows, i.e., well-typedness of t implies the existence of value v such that $t \to^*_\beta v$, since $\langle t\rangle$ is normalizing exactly when t is. Similarly, using the correspondence of $\lambda^{\text{det}}_{<:\{\}}$ programs to non-deterministic $\lambda^{\text{nd}}_{<:\{\}}$ programs after lowering (see Section 7.2), we get type safety for $\lambda^{\text{nd}}_{<:\{\}}$.

9 A Prototypical Implementation in Dotty

In this section we give an overview of how we extended Scala with dependent types. This development was an experiment to explore the feasibility of retrofitting dependent types into an existing language and its compiler. We implemented our prototype as an extension of Dotty, the reference compiler for future versions of the Scala language. Our presentation focuses on facets of the implementation that are not reflected in the formalism presented in Chapter 7.

On a syntactic level, our Scala extension consists of three additions:

- the singleton types syntax `{ t }`,
- the **`dependent`** modifier for methods, values and classes,
- the **`choose`**`[T]` construct.

The newly-introduced singleton type syntax enables a subset of Scala expressions to be used in types. This subset approximately corresponds to the core functional subset of Scala, plus the **`choose`**`[T]` construct, as illustrated in $\lambda^{\text{nd}}_{<:\{\}}$. Within this subset, the main differences between our formalism and implementation lie in the handling of pattern matching.

9.1 Pattern Matching

Pattern matching in Scala supports a wide range of matching techniques [EOW07]. For example, *extractor patterns* rely on user-defined methods to extract values from objects. As a result, these custom extractors can contain arbitrary side effects. Our implementation limits the kind of patterns available in types to the two simplest forms: decomposition of case classes and the type-tests/type-casts patterns.

During type normalization, our system evaluates pattern matching expressions according to Scala's runtime semantics, that is, patterns are checked top-to-bottom, and type-tests are evaluated using runtime type information available after type erasure.

For example, consider the following pattern matching expression:

```
s match { case _: T1 => v1 case _: T2 => v2 }
```

When used in a type, this expression reduces to `v1` if the scrutinee's type is a subtype of `T1`. In order to reduce to `v2`, type normalization must make sure `T1` and the scrutinee's type are disjoint, namely that the dynamic type of `s` cannot possibly be smaller than `T1`. Disjointness proofs are built using static knowledge about the class hierarchy and make use of the guarantees implied by the **`sealed`** and **`final`** qualifiers, which are Scala's way of declaring closed-type hierarchies.

9.2 Two Modes of Type Inference

In order to retain backwards-compatibility, our system supports two modes of type inference: the precise inference mode which infers singleton types, and the default inference mode that corresponds to Scala's current type-inference algorithm. Concretely, users opt into our new inference mode using the **`dependent`** qualifier on methods, values, and classes.

When inferring the result type of a **`dependent`** method, our system lifts the method's body into a type. This lifting will be precise for the subset of expressions that is representable in types, and approximative for the rest. When we encounter an unsupported construct, we compute its type using the default mode, yielding a type `T` which we then integrate in the lifted body as **`choose`**`[T]`.

For example, given the following definition:

```
dependent def getName(personalized: Boolean) =
  if (personalized) readString() else "Joe"
```

our system infers the following result type:

```
{ if (personalized) choose[String] else "Joe" }
```

Scala requires recursive methods to have an explicit result type, and this restriction also applies to **`dependent`** methods. However, in the case of a **`dependent`** method, an explicit result type is only used as an upper bound for the actual precise result type and will only be used to type-check the method's body. At other call sites, the (precise) inferred result type is used. Bounds of dependent methods are written using a special syntax (`<: T`), which emphasizes the difference from normal result types (`: T`).

9.3 Approximating Side Effects

State

Scala's type system permits uncontrolled side effects in programs. Given the absence of an effect system, result types of methods do not convey any information about the potential use of side effects in the method body. The situation is analogous for **dependent** methods. Thanks to `choose[T]` we can still formulate precise result types when terms depend on the result of side-effectful operations. Since we uniformly approximate all side effects, we avoid the situation where a type refers to a value that may be modified during the program execution. For instance, if `z` is a mutable integer variable, we will never introduce `z` in a singleton type, but we can still assign a better type than `Lst` to an expression like `Cons(z, Nil())`, that is, `{ Cons(choose[Int], Nil()) }`.

Exceptions

Similarly to how we model other side effects, exceptions are approximated in types. Our type-inference algorithm uses a new error type, `Error(e)`, which we infer when raising an exception with **throw** `e`. Exception handlers are typed imprecisely using the default mode of type-inference. Exceptions thrown in statement positions are not reflected in singleton types, since the type of `{e1; e2}` is simply `{ e2 }`. However, exceptions thrown in tail positions (such as in remove from Chapter 6) can lead to types normalizing to `Error(e)`. In these cases, our type system can prove that the program execution will encounter exceptional behavior, and reports a compilation error. This approach is conservative in that it might reject programs that recover from exceptions. Also note that this is a sanity check, rather than a guarantee of no exceptions occurring at runtime. That is, depending on which rules are used during subtyping, it is possible to succeed without entering type normalization, resulting in such errors going undetected. Despite these shortcomings, our treatment of exceptions results in a practical way to raise compile-time errors. It would be interesting to explore the addition of an effect system to our Scala extension and formalization.

9.4 Virtual Dispatch

Our extension does not model virtual dispatch explicitly in singleton types. Instead, the result type of a method call `t.m(...)` is always the result type of `m` in `t`'s static type. Consequently, **dependent** methods effectively become **final**, given that only a provably-equivalent implementation could be used to override it.

Special care must be taken when an imprecisely-typed method is overridden with a **dependent** one. In this situation, the result type of a method invocation can lose precision depending on type of the receiver. Calls to the `equals` methods are a common example of this: `equals` is defined at the top of Scala's type hierarchy as referential equality and can be overridden

arbitrarily. Given a class Foo with a **dependent** overrides of `equals`, calls to `Foo.equals(Any)` and `Any.equals(Foo)` are not equivalent; the former precisely reflects the equality defined in `Foo` whereas the latter merely returns a `Boolean`.

9.5 Termination

We distinguish two important aspects of termination.

The first question is whether type-checked programs are guaranteed to terminate. For simplicity, our work side-steps this question, requiring bounds for recursion. A more general solution would be to compute or infer such bounds using measure functions, as done in System FR [HVK19]. Another approach would be to extend our translation of non-determinism to permit non-termination, but we believe this question is largely orthogonal to the choice of mechanism for type-level computation. Our work targets general-purpose programming language whose type safety is defined with regards to its runtime semantics and that may include non-terminating interactive computations.

The second question is termination of our type checker. Non-termination of type checking implies that the type checker can give three possible answers, “type correct”, “type incorrect” or “do not know” (or timeout). Treating “do not know” as “type incorrect” makes the non-termination unproblematic from a soundness perspective. A similar argument is made for other dependently-typed languages with unbounded recursion, such as Dependent Haskell [Eis16] or Cayenne [Aug98]. In practice, our system deals with infinite loops using a fuel mechanism. Every evaluation step consumes a unit of fuel, and an error is reported when the compiler runs out of fuel. The default fuel limit can be increased via a compiler flag to enable arbitrarily long compilation times.

10 Use Case

In this chapter, we extend the motivating example presented in Chapter 6 by building a type-safe interface for Spark datasets. We use dependent types to implement a simple domain-specific type checker for the SQL-like expressions used in Spark. We then compare the compilation time of our dependently-typed interface against an equivalent encoding based on implicits.

10.1 A Type-Safe Database Interface

The type-safe interface presented in this section illustrates the expressive power of our system and is implemented purely as a library. For brevity, our presentation only covers a small part of Spark's dataset interface, but the approach can be scaled to cover that interface in its entirety. The type safety of database queries is a canonical example and has been studied in many different settings [LM99, KGV+19, MBB06, Chl10].

The example built in Chapter 6 uses lists of column names to represent schemas. A straightforward improvement is to also track the type of columns as part of the schema. Instead of using column names directly, we introduce the following `Column` class with a phantom type parameter `T` for the column type, and a field `name` for the column name:

```
dependent case class Column[T](name: String) { ... }
```

Table schemas become lists of `Column`-s and thereby gain precision. The definition of `join` given in Chapter 6 can be adapted to this new schema encoding to prevent joining two tables that have columns with matching names but different types.

A large proportion of the weakly-typed Spark interface is dedicated to building expressions on table columns. Such expressions can currently be built from strings, in a subset of SQL, or using a Scala DSL which is essentially untyped.

The lack of type safety for column expressions can be particularly dangerous when mixing

columns of different types. The pitfall is caused by Spark's inconsistency: depending on types of columns and operations involved, programs will either crash at runtime, or, more dangerously, data will be silently converted from one type to another.

By keeping track of column types it becomes possible to enforce the well-typedness of column expressions. As an example, consider the following Spark program:

```
table.filter(table.col("a") + table.col("b") === table.col("c"))
```

We would like our interface to enforce the following safety properties:

- Columns *a*, *b* and *c* are part of the schema of `table`.
- Addition is well-defined on columns *a* and *b*.
- The result of adding columns *a* and *b* can be compared with column *c*.
- The overall column expression yields a `Boolean`, which conforms to filter's argument type.

Automatic conversions during equality checks can be prevented by restricting column equality to expressions of the same type `T`:

```
dependent case class Column[T](k: String) {
  def ===(that: Column[T]): Column[Boolean] =
    Column(s"(${this.k} === ${that.k})")
}
```

Addition in Spark is defined between numeric types and characters. The result type of an addition depends on the operand types. For numeric types, Spark will pick the larger of the operand types according to the following ordering: `Double > Long > Int > Byte`. The situation is quite surprising with characters as any addition involving a `Char` will result in a `Double`.

Dependent types can be used to precisely model these conversions. We define a type function to compute the result type of additions:

```
def addRes(a: Any, b: Any) =
  (a, b) match {
    case (_: Char, _: Char | Byte | Int  | Long | Double) =>
      choose[Double]
    case (_: Byte, _: Byte | Int  | Long | Double)        => b
    case (_: Int,  _: Int  | Long | Double)               => b
    case (_: Long, _: Long | Double)                      => b
    case (_: Double, _: Double)                           =>
```

```
      choose[Double]
    case (_: Byte | Int | Long | Double, _)              =>
      addRes(b, a)
    case _ => throw new Error("incompatible types in addition")
  }

type AddRes[A, B] = { addRes(choose[A], choose[B]) }
```

Also note the use of recursion in the second-to-last case, to avoid duplicating symmetric cases. The `AddRes` type can be used to define a `Column` addition that accurately models Spark's runtime:

```
dependent case class Column[T] private (k: String) {
  dependent def +[U](that: Column[U]) <: Column[_] =
    Column[AddRes[T, U]](s"(${this.k} + ${that.k})")
}
```

Allowing programmers to construct `Column`-s from string literals would defeat the purpose of a type-safe interface. Instead, programmers should extract columns from a `Table`'s schema. For that purpose, we implement the `col` method on `Table` and annotate the `Column` constructor as private.

```
dependent case class Table(schema: Lst, data: spark.DataFrame) {
  dependent def col(name: String) <: Column[_] = {
    dependent def find(key: String, list: Lst) <: Any =
      list match {
        case Cons(head: Column[_], tail) =>
          if (head.k == key) head else find(key, tail)
        case _ => throw new Error("column not found in schema")
      }
    find(name, schema)
  }

  dependent def filter(predicate: Column[Boolean]) <: Table =
    new Table(this.schema, this.data.filter(predicate.k))
}
```

The `col` method is implemented using a nested dependent method to find the column corresponding to the given name. Thanks to the dependent annotation, the type-checker is able to statically evaluate calls to `col`. Assuming the table's schema contains a column `a` of type `Int` and columns `b` and `c` of type `Long`, the compiler will be able to infer types as follows:

```
val p =    table.col("a")    +    table.col("b")     ===    table.col("c")
//Inf'd { Column[Int]("a") }   { Column[Long]("b") }      { Column[Long]("c") }
```

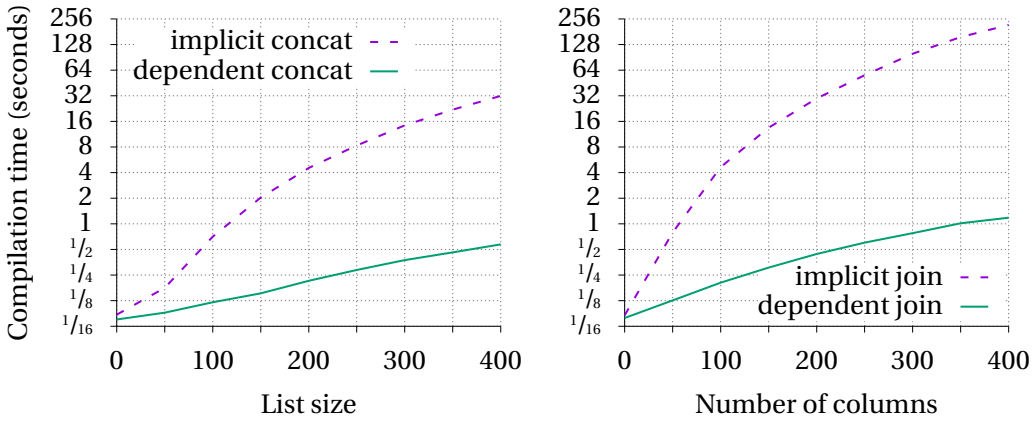

Figure 10.1 – Comparing the compilation times of two implementations of list concatenation (left) and join (right) on a logarithmic scale.

Given our definitions of column addition and equality, the expression `p` is typed as `Column[Boolean]`. All the safety properties stated above are therefore enforced by the dependently-typed interface presented in this section.

10.2 Comparison to an Existing Technique

Programmers have managed to find clever encodings that circumvent the lack of first-class support for type-level programming in many languages. These encodings can be very cumbersome, as they often entail poor error reporting and a negative impact on compilation times [McB02], [KLS04]. In Scala, implicits are the primary mechanism by which programmers implement type-level programming [OBL+18].

Frameless [Fra21] is a Scala library that implements a type-safe interface for Spark by making heavy use of implicits. Most type-level computations in this library are performed on the heterogeneous lists provided by Shapeless [Sab21].

We compared the dependently-typed Spark interface presented in this section against the implicit-based implementation of Frameless. To do so, we isolated the implicit-based implementation of the `join` operation on table schemas, and compared its compilation time against the dependently-typed version presented in this section. To evaluate the scalability of both approaches we generated test cases with varying schema sizes and compiled each test case in isolation. A similar comparison is done for list concatenation, which constitutes a building block of `join`.

Figure 10.1 shows that, in both benchmarks, the dependently-typed implementation compiles faster than the version with implicits, and compilation time scales better with the size of the input. In the join benchmark, we see that the implicit-based implementation exceeds 30 seconds of compilation time around the 200 columns mark, and continues to grow quadrat-

ically. This can be explained by the nature of implicit resolution, which might backtrack during its search. The compilation time of the dependently-typed implementation grows close to linearly and stays below one second until the 350 columns mark. We were able to observe similar trends in the concatenation benchmark. These measurements were obtained by averaging 120 compilations using a warmed-up compiler instance running on Oracle JVM 1.8.0 and Linux with an Intel i7-7700K processor.

11 Related Work

In the following chapter we summarize and compare to related work ranging from contract-based verification to type-level programming techniques. We begin by putting our contributions in context to earlier work on the Stainless verifier and Scala's type system. Subsequently, we survey dependent types, including prior attempts to integrate with non-functional features such as object-orientation and effectful operations. From there we continue to the other end of the spectrum of static safety, and touch upon a few program verifiers whose design influenced or overlaps with Stainless. Finally, we look at some alternative solutions to reason about shared mutable data.

11.1 Leon and Stainless

The Stainless verifier has its roots in an earlier system called Leon which introduced the underlying semi-decision procedure for a first-order language with recursive functions, algebraic data types and unbounded integers [SDK10, SKK11]. Later work by Blanc et al. added support for imperative constructs such as loops and local mutable state [BKKS13], and a sound treatment of modular arithmetic [BK15] leveraging the dedicated bit-vector theories found in modern SMT solvers. Voirol et al. extended this fragment to encompass parametric types and higher-order functions [VKK15] while maintaining completeness and soundness of counterexamples. Subsequent work by Voirol [Voi19] evolved Leon into two separate components: Inox, the core solver for polymorphic ADTs and higher-order functions, and Stainless, which progressively lowers object-oriented, imperative and type system features of Scala to the functional language of Inox. More recently, Hamza et al. [HVK19] proposed System FR as a formal foundation for Inox and Stainless' verification condition generator. A soundness proof extending to lowering phases (including the ones presented here) remains future work.

In this thesis we specifically focussed on the way Stainless encodes and reasons about *heaps*. Support for mutable data beyond local variables was initially implemented by Blanc and described in his thesis [Bla17], which presents an imperative fragment permitting *unshared* mutable data on the heap. His approach expands upon the earlier imperative phase, and

introduces a largely-automated effect analysis to ensure that all references to mutable data are unique. Our heap encoding presented in Chapter 4 shares some central ideas (effectful procedures are reduced to functions that transform immutable representations of state) and infrastructure (such as the elimination of locally-mutable variables). Functional encodings of destructive updates can easily lead to an exponential blowup in the size of verification conditions [FS01], which Blanc's technique avoids by introducing conditionals in lowered imperative programs. This pushes the underlying complexity into the SMT solver and provides a chance to explore branches lazily or prune them entirely. We initially investigated a relaxation of Blanc's fragment to allow *some* aliasing, but ultimately opted for a fully-general solution with dynamic frames as the specification mechanism. As with Blanc's solution, our encoded programs grow only linearly in size. We also leave the complexity of search to the SMT solver, in particular through Z3's generalized theory of arrays [dMB09].

The general design philosophy behind Stainless is one of verification by reduction to functional programs. This approach maintains counterexamples and allows us to provide a combination of guarantees usually found separately in proof assistants, program verifiers and bounded model checkers. Proofs are sound, but the progressive unfolding of functions also provides completeness wrt. counterexamples, and the executable nature of our specification language ensures that counterexamples correspond to failing executions of the program. Our extensions maintain the same principles by providing a precise encoding of the heap and avoiding the introduction of any quantifiers (which would adversely affect the ability of SMT solvers to generate models).

11.2 Static Safety through Metaprogramming in Scala

Scala possesses a number of type-level and metaprogramming mechanisms that can be used to gain additional static safety. We already discussed some idiomatic patterns for type functions in Section 2.2. *Implicit resolution* [OBL+18], which typically drives type-level computation in Scala 2, was introduced in early versions of the language and initially conceived as an improved mechanism for dependency injection and automatic conversion between types. Their versatility subsequently made implicits the standard mechanism to encode type classes in Scala [OMO10, WB89], and eventually led to their use in emulating type functions. The latter typically requires the collaboration of multiple of Scala's typing features, such as F-bounded polymorphism [CCH+89], type members [AR17] and higher-kinded types [MPO08b, MPO08a, OMP16].

Starting from Scala 3, *match types* [BBKO22] offer a more direct and ergonomic way to encode certain type functions. Match types and the approach we present in this thesis were developed concurrently and have much in common. Firstly, the primary intended use case of match types is essentially the same as ours, and the associated reduction rules are analogous to those governing evaluation of pattern matches in our prototype (see Section 9.1). Match types can be thought of as a special case of our extension, though the representation differs (in our

prototype we model pattern matches explicitly in their lowered form of if-expressions and type tests). Furthermore, match types stand firmly in the realm of types, but benefit from particular inference rules that allow some term-level pattern matches to be typed precisely, i.e., at the corresponding match type. On the other hand, our approach lifts a larger selection of (pure) language constructs onto the type level, and blurs the distinction between types and terms. It should also be noted that, by now, match types benefit from a mature implementation in Dotty, and have become available to a broad audience with the release of Scala 3. That being said, additional evaluation by Blanvillain [Bla21] showed that checking match types in their current form, while more performant than implicit resolution, still exhibits super-linear growth in compile times for the benchmark in Section 10.2. We conjecture that this is due to match type checking being embedded in Dotty's subtype checks, whereas our prototype performs normalization separately and frugally.

Macros and inlining provide yet another mechanism to compute types. Scala has long provided advanced metaprogramming facilities in the form of macros [Bur13], and, more recently, multi-stage programming [SBO21]. So-called white-box macros are expanded during type checking, which allows programmers to effectively run their own computations at compile-time. For instance, a white-box macro can be used to dynamically produce a witness term possessing a type member that represents the result of the computation, which will then be propagated by the type checker outside the macro. Scala 3 features a new system that makes typed metaprogramming seamless and safe using a technique similar to binding time analysis. When combined with the new `inline` keyword (which forces inlining during type checking, analogously to white-box macros), it would also enable type-level computation.

11.3 Dependent Types for General-Purpose Programming

Haskell. As of today, Haskell is perhaps closest to becoming dependently-typed among the general-purpose programming languages used in industry. Haskell's type families [KJS10] provide a direct way to express type-level computations. Other language extensions such as functional dependencies [Jon00] and promoted datatypes [YWC+12] are also moving Haskell towards dependent types. Nevertheless, programming in Haskell remains significantly different from using full-spectrum dependently-typed languages. Above all, it imposes a strict separation between terms and types. As a result, writing dependently-typed programs in Haskell may involve code duplication between types and terms. These redundancies can be somewhat avoided using the singletons package [EW12], which uses metaprogramming to automatically generate types from datatypes and function definitions (not unlike the application of metaprogramming in Scala described above).

In the context of Haskell, Eisenberg's work on Dependent Haskell [Eis16] is closest to ours, in that it adds first-class support for dependent types to an established language, in a backwards-compatible way. Dependent Haskell also supports general recursion without termination checks. While we share similar goals, our work is differentiated by the contrasting paradigms

of Scala and Haskell. Like many object-oriented languages, Scala is primarily built around subtyping and does not restrict the use of side effects. Furthermore, Eisenberg's system provides control over the relevance of values and type parameters. In contrast, our system does not support any erasure annotations and simply follows Scala's canonical erasure strategy: types are systematically erased to JVM types, and terms are left untouched. Weirich established a fully mechanized type safety proof for the core of Dependent Haskell using the Coq proof assistant [WVdAE17].

Cayenne is a Haskell-like language with dependent types introduced by Augustsson [Aug98]. Like Dependent Haskell, it resembles our system in its treatment of termination, and differs by being a purely functional programming language. Cayenne's treatment of erasure is similar to Scala's: types are systematically erased. Augustsson proves that Cayenne's erasure is semantics-preserving, but does not provide any other metatheoretical results.

Type-Level Computation for Object-Oriented Languages. Adding dependent types to object-oriented languages is a remarkably under-explored area of research. A notable exception is the recent work of Kazerounian et al. on adding dependent types to Ruby [KGV+19]. Their goals are very much aligned with ours: using type-level programming to increase safety in a general-purpose programming language. Given the dynamic nature of Ruby, it is unsurprising that their solution greatly differs from ours. In their work, type checking happens entirely at runtime and has to be performed at every function invocation to account for possible changes in function definitions. Safety is obtained by inserting dynamic checks, similarly to gradual typing.

The work of Campos and Vasconcelos on DOL (Dependent Object-oriented Language) [CV18] shares similar goals but is limited to inequality constraints on integer parameters (in the style of [XP98]).

Dependent Types and Subtyping. Dependently-typed lambda calculi with subtyping were described at least as far back as 1988 by Cardelli [Car88]. The latter type system is much more expressive than ours and allows bounded quantification over both types and terms using the notion of a `Type` type and power types. Unlike our system, which is designed with the concrete evaluation of types in mind, Cardelli does not provide semantics for their system and leaves the equivalence relation among types unspecified.

In [Asp94] Aspinall introduces $\lambda_{\leq}$, a dependently-typed system with subtyping *and* singleton types that resembles ours in its type language. Their equivalence relation on types is more powerful and is not syntax-directed, unlike our type evaluation relation. Furthermore, singleton types in this work are indexed by the type through which equality is "viewed", thereby enabling a form of polymorphism beyond ours. Aspinall's system also has primitive types and allows for atomic subtyping among them, but no congruence rules, hence partially-widened forms like {cons choose[Top] nil} cannot be represented.

System $\lambda P_{\leq}$ [AC96] combines subtyping and dependent types in the Edinburgh Logical Frame-

work. In this work, Aspinall et al. propose a type-checking algorithm for $\lambda P_{\leq}$ which they show to be complete and terminating. Their system uses a kinding relation to ensure well-formedness of type applications. A kind system is not required in $\lambda^{\text{nd}}_{<:\{\}}$ as we emulate type applications inside singleton types.

In [SH00], Stone and Harper describe a dependently-typed calculus with singleton kinds and subkinding. Their type-and-kind system is similar to Aspinall's $\lambda_{\leq}$ term-and-type system, but operates one level up the hierarchy.

More recently, Courant [Cou03] developed a variant of Aspinall's $\lambda_{\leq}$ with a type-inference algorithm that is proven sound and complete. The main takeaway from Courant's work is the inclusion of a coercion rule in delta reduction. These coercions are used to "tag" variables with their declared type, which prevents these types from being lost during substitution. Our formalism resembles Courant's system, it shares the SUBSING subtyping rule (SUB/SINGL in Courant's work), and $\beta\delta$-reduction.

Pure Type Systems [Bar91] provide a unified presentation of systems of dependently-typed λ-calculus by using a single syntactic category for both terms and types. In [Zwa99], Zwanenburg defines an extension of pure type systems that include both subtyping and bounded quantification. A central design decision of their system is that subtyping rules do not depend on typing rules. The absence of circularity simplifies both the theory and the metatheory, at the cost of having to define subtyping on pseudoterms rather than only well-typed terms. Another limitation of Zwanenburg's theory is that it cannot be extended with a Top-type.

Pure Subtype Systems [Hut10] is another framework with unified syntax; it differs from traditional approaches in that it uses a single relation, subtyping, that subsumes typing, subtyping, and type evaluation as found in our system. Their system allows for partially-widened types similar to ours and also enables the computation with different levels of precision. For instance, it provides rules for Int + 5 to be approximated as Int. The paper presents a partial investigation of the metatheory, but the proof of soundness remains incomplete. Nevertheless, Hutchins reports that they have not been able to construct a counter-example, even with the addition of fixpoints.

In [YO17], Yang and Oliveira propose a dependently-typed generalization of System $F_{\leq}$ with unified syntax and a single relation that subsumes typing and subtyping. In their system, type computations are driven by cast operators: each reduction or expansion step requires an annotation to explicitly instruct the type checker to take a step. Explicit casts make it possible to allow general recursion without compromising decidability of type checking. It would be interesting to study variants of $\lambda^{\text{nd}}_{<:\{\}}$ based on explicit casts instead of our finitized fix.

Dependent-object types [AR17] model the core of Scala's type system and include type members and path-dependent types, which are not represented in our formalism. Even though they introduce a form of dependency, path-dependent types were not designed for type-level computation, rendering their original goals largely orthogonal to ours.

11.4 Proof Assistants and Verification-Oriented Languages

A great variety of *proof assistants* has been developed to aid in the construction of machine-checkable proofs. Tools such as Coq [BC04], Isabelle [NPW02] and HOL4 [SN08] have attracted sizeable communities and are increasingly being adopted to construct verified software. They typically provide a combination of an expressive logic and machinery such as tactics to automate proofs. Several popular proof assistants are based on dependent type theories inspired by Martin-Löf's constructive type theory [MLS84]. While Coq [BC04] and Agda [Nor07] allow the extraction of executable programs, they arguably prioritize ease of proofs and are not geared towards programming in the large. Idris [Bra13] and Lean [dMKA+15] are two systems with similar foundations, but an explicit goal of becoming general-purpose programming languages in their own right and integrating with real-world systems. Nonetheless, neither of them can draw on an existing ecosystem like Scala's, and both primarily target the functional paradigm.

Researchers have also proposed a number of *verification-oriented languages* that straddle the continuum between proof assistants and general-purpose programming. F* [SHK+16] is an ML-like language that has been used to verify a realistic implementation of the TLS protocol stack [BBDL+17]. It combines dependent types and computation types to capture the possible effects of different program fragments, but uses a weakest precondition calculus rather than dependent type theory as its foundation. Like Stainless, most tools in this category primarily rely on contracts to specify programs. ACL2 [KM96] is an industry-strength verifier for a subset of Common Lisp. Since Lisp is untyped, ACL2 uses inductive predicates to describe recursive data structures, yielding an approach similar to what we described in Section 2.3.2. Spec# [BLS04] is a mature verifier for a subset of C#, and pioneered many aspects of verification for object-oriented languages. Dafny [Lei10] provides its own object-oriented surface language with additional facilities for specification (such as predicates and ghost variables), and is in many ways a continuation of the work on Spec#.

Similarly to how Stainless relies on translating programs to Inox, a functional, polymorphically-typed core language, other verifiers target reusable *intermediate representations for verification*. Both Spec# and Dafny rely on Boogie [Lei08], an intermediate representation for verification of imperative programs, which ultimately also acts as a verification condition generator for SMT solvers. Unlike Inox, which constitutes an ML-like kernel, Dafny is styled after Dijkstra's guarded command language [Dij75], and takes a significantly different approach by generating VCs from the program's control-flow graph [BL05] and encouraging the use of quantifiers to axiomatizing user-defined theories. Why3 [FP13] also targets an imperative, first-order language, but dispatches solving to a range of automated and interactive theorem provers. Inox is somewhat narrower in scope, since it has only been designed as a backend for Stainless, but provides support for higher-order functions and completeness wrt. counterexample finding [Voi19].

11.5 Verification of Heap-Manipulating Programs

Verifying stateful code poses problems both from a specification and an automation standpoint. To compensate for the additional complexity of destructive assignment various new formalisms were developed over the years, most notably Hoare calculus [Hoa69] and weakest-precondition inference [Dij76]. Boogie [Lei08] and F* [SHK+16], for instance, effectively compute weakest preconditions and then use SMT solvers to check for entailment by the function's precondition.

In Stainless we eliminate state early on (both local and on the heap), and instead translate to a functional language whose meaning can be expressed directly in a fragment of first-order logic. Compared to other approaches the translation approach maintains relative proximity between our input language, e.g., a function with contracts, and the generated VC, i.e, an encoding of the function's result value and a check whether the function's precondition entails its postcondition. This correspondence between programs and SMT queries allows us to extract counterexamples from models and aids debugging.[1]

Neither of these approaches directly addresses the complications arising from heaps with shared references to mutable data, however. One option is to restrict programs to an effectively linear fragment (as in Stainless prior to our contributions in Part I [Bla17]) or statically track all possible aliases, as in Why3 [FP13]. Unfortunately this results in a rather restrictive programming style, so researchers have proposed various alternative disciplines to enable reasoning in the presence of sharing. We touch upon two such approaches, separation logic and dynamic frames.

Separation Logic [ORY01, Rey02] and its variants, permission logics, build upon Hoare calculus [Hoa69] by distinguishing disjoint regions of the heap through explicit *separating conjunctions*. Combined with assumptions about objects' dynamic types this yields a powerful reasoning principle to prove that object references of interest are distinct, and that shared mutable data assumes shapes such as lists or trees.

Our heap encoding replaces the frame rule of separation logic by frame conditions asserting equality of heaps on objects in the frame. As mentioned in Section 4.5, one might recast our encoding in a way closer to separation logic by explicitly referring to the allocation status of locations on the heap, and explicitly separating the heap into a footprint and a frame heaplet.

Many verifiers have been built upon the foundation of separation logic. VeriFast [JSP+11] is notable for its mature support of C and Java programs. It also possesses an extensive annotation language to describe inductive heap predicates and lemmas, though that language is first-order and separate from the program surface language. In Stainless programs, lemmas and contracts are all expressed in Scala. Specifications, in particular, may leverage the full flexibility of higher-order functions, while VeriFast relies on more ad-hoc constructs such

[1] That being said, the unfolding procedure in Inox requires an arbitrarily large number of queries in general, and emits many auxiliary blocking literals, so readability quickly deteriorates in all but simple examples.

as predicate families. F* [SHK+16] enjoys similar flexibility. Thanks to its dependent type system F* can express a framing mechanism analogous to separation logic in library code. By arranging objects in a tree structure called a *hyper heap* one can derive separation between objects in distinct subtrees.

Viper [MSS16] is an intermediate representation much like Boogie [Lei08], but based upon a permission logic that provides fine-grained control over individual fields, and, through so-called fractional permissions [Boy03], can recover unique ownership after temporarily sharing an object. Our approach could be adapted to grant or deny access rights more finely by modelling the heap as a map from references and field names to field state, rather than from references to object states. Fractional permissions, on the other hand, are already expressible through explicit reads and modifies sets in our approach.

It is worth noting that Stainless ultimately acts as a verification condition generator, whereas most separation logic verifiers rely on symbolic execution [BCO05]. Viper [MSS16] is a notable exception in that it can also emit a verification condition to be verified through Boogie.

Infer [CD11] is another particularly notable example of a verifier based on separation logic, in part because it has been deployed at an impressive scale across Facebook to detect null pointer accesses and resource leaks. The compositional inference principle behind Infer is bi-abduction [CDOY11], a shape analysis that uses a fragment of separation logic as its abstract domain and modularly infers both the frame and footprint of functions.

Shape Analysis deserves additional mention in this context. Static analyses of this kind are based on abstract interpretation of the object graph, and were originally devised to automatically optimize data layouts and elide garbage collections of Lisp-like structures [JM79, JM82]. Over the years shape analysis proved useful in many domains such as alias analysis [LH88], automatic parallelization [HN90] and program verification. The particular application relevant here is that it can be used to infer the kinds of inductive heap predicates we saw in Section 3.3. For instance, by inspecting the code in the `pop` procedure of our example a shape analysis might infer the singly-linked list structure described by `sll`. A notably general framework for the inference of such "shape invariants" supporting destructive updates is based on three-valued logic [SRW02] and was implemented in TVLA [LAS00].

Decidable Logics for Heap Reasoning. While shape analysis is based on abstract interpretation, analogous problems have been recast into logics that allow explicit reasoning about object graphs. For instance, PALE [MS01] deals with programs annotated with pointer logic assertions and graph types that essentially capture the shapes of data structures. PALE generates verification conditions in a decidable fragment of monadic second-order logic, as supported by the MONA solver [KM01]. The TREX logic [WMK11] later provided an efficient decision procedure for tree-like structures. Eventually this led to the development of decision procedures that specifically target decidable fragments of separation logic and allow their combination with other SMT theories, e.g., GRASS [PWZ13], GRASShopper [PWZ14b] and GRIT [PWZ14a]. It would be interesting to explore how integration with such solvers could provide automation

around inductive heap predicates in Stainless.

Dynamic Frames. Both Dafny [Lei10] and our heap encoding are heavily inspired by dynamic frames [Kas06]. Unlike separation logic, which syntactically tracks access to objects and fields, dynamic frames merely define the accessible portion of the heap as a set that may be constrained by other program expressions. This provides flexibility, e.g., to define the representation of a data structure by means of recursive functions or mutable ghost fields, as shown in our examples. It also allows us to refer to heap-dependent expressions in assertions, which is problematic for standard separation logic verifiers.

On the flip side, keeping track of a function's and data structure's footprint explicitly can be tedious and reduces opportunities for automation. Implicit dynamic frames [SJP12] address this issue by automatically deriving the set of accessible objects from contracts, including the special `acc(o)` predicate which denotes whether `o` is accessible. Our system currently requires users to explicitly define reads and modifies sets, but it seems desirable to infer these in an analogous way.

Another aspect present in many related systems (including [SJP12]) is a stricter notion of encapsulation. In order to ensure modular reasoning these systems separate the use of *external* and *internal* axioms. That is, when calling a method from outside a class, only axioms corresponding to the frame condition and the post-condition are available, whereas during verification of the class itself the implementations of its methods are revealed. By default Stainless does not respect any modular discipline and will unfold method calls successively, making it difficult to hide information and control verification overhead for large type hierarchies. In the future we would like to extend Stainless with more explicit control for the folding and unfolding of function definitions. This would allows us to leverage encapsulation to automatically treat as pure those classes that only maintain state internally (such as the `FibCache` example).

12 Conclusion

In this thesis we explored and developed two aspects of formal verification in modern general-purpose programming languages. The first part was dedicated to automated checking of contracts in languages with both functional and imperative features. We showed how a verifier for the functional fragment can be extended to reason about heaps, including shared mutable data, and still leverage its functional features for the purpose of specifications. Our translation represents heaps as maps from locations to object state, and transforms stateful procedures into functions that take a heap (map) and produce the modified heap. Our main contribution was a novel encoding of heap maps and frame conditions in first-order logic. We leverage the generalized array theory of Z3 to cast frame conditions as applications of the map combinator on arrays. This retains decidability of verification conditions, and allows the verifier to produce counterexamples.

We found our heap encoding to be comparatively simple to implement, while retaining the flexibility to reason about heaps as first-class values, and leaving the complexity of search to the SMT solver. Applying our technique to Stainless, in particular, allowed us to build a verifier that can deal with the full generality of heap-manipulating Scala programs.

The second part investigated type-level programming as a lightweight form of specification and checking. Our starting point was to improve the ergonomics and performance of popular type-level programming use cases in the Scala ecosystem, where mechanisms like implicit resolution are used to propagate more precise type information and increase safety of user libraries such as database wrappers (which are traditionally weakly-typed). In an effort to find a more principled solution for type-level programming we then posed the question of what dependent types might look like for a language like Scala, which notably combines functional, imperative and object-oriented features.

We developed our extension as a prototype on top of Dotty, the Scala 3 compiler, which allowed us to explore questions of representation, integration with type inference and general maintainability. Our prototype served to validate several benchmarks in our extended language, suggesting a more concise and performant type-level programming experience is possible.

To capture some of the particular challenges with integrating full-spectrum dependent types into a language like Scala we proposed $\lambda^{\text{nd}}_{<:\{\}}$ as a formal foundation. We provided a soundness proof by reduction to a functional calculus, System FR, and found that experimenting with a dedicated implementation of $\lambda^{\text{nd}}_{<:\{\}}$'s type checker allowed us to study interesting use cases in isolation and even recover some traditional type system features.

In both parts of this thesis our goal was to preserve precise reasoning for a functional core language, and reduce other language features to functional ones. We found this approach attractive for both its generality and conceptual simplicity. Above all, it allowed us to reuse mature formal foundations (System FR) and decision procedures (combinatory array logic).

Future Work

Precise Reasoning for an Expressive Type System. One avenue of future work would be to combine both parts of this thesis in a single system by using Stainless as part of a type checker. The typing rules for $\lambda^{\text{det}}_{<:\{\}}$ we presented are primarily designed to result in predictable checking and a simple implementation. Instead, one could use Stainless' existing infrastructure to automatically discharge certain subtyping obligations arising in $\lambda^{\text{det}}_{<:\{\}}$ by generating semantically-equivalent verification conditions in Inox.

We have some experience with this approach from earlier work on integrating refinement types into the Scala type checker [SK16]. That system, LiquidHaskell [VSJ+14] and System FR [HVK19] all have explicit "semantic" typing rules that allow the type checker to invoke a solver like Inox or Z3.

The relation with refinement types, in particular, runs deep: our generalized singleton types can be seen as a complement to refinement types, much like generators and predicates are two faces of the same coin. It would be interesting to explore a system where generators, $\{t\}_U$, are explicitly paired with their corresponding predicates, $\{x : U \mid p(x)\}$ such that

$$\forall\, t\, v.\ t \to^*_\beta v \iff p(v) \to^*_\beta \textbf{true}\ .$$

The resulting type system would allow switching between the two views, which might be helpful to prove complex properties or drive test case generation.

Decision Procedures for Heap Reasoning. When extending Stainless our strategy to report counterexamples was to avoid quantifiers. This is by no means the only possibility, as witnessed by the success of other approaches (e.g., verifiers based on symbolic execution) that use quantifiers effectively. Yet we believe that targeting decision procedures in the long term results in a more predictable verification experience than direct encodings via general quantifiers. Our experiments suggest that the approach holds promise, even though the performance of map combinators indicates that they nonetheless require non-trivial reasoning in the Z3 solver.

An integration of insights from verifiers and proof frameworks based on separation logic is a promising direction to potentially improve usability of our approach. SMT-LIB notations and competitions for separation logic [SPR+19] are likely to be a useful resource for this task, even if these benchmarks typically do not focus on reasoning about as detailed functional correctness properties as our examples. Another direction for improving automation is inductive reasoning, both for separation logic predicates themselves [TLKC19] and for pure recursive functions [RK15].

Our work made the initial case for an approach that is semantically simple and promises to be predictable. We hope that it will motivate both the SMT solver builders and verification tool builders to collaborate to improve the performance, the predictability, and the ability to report counterexamples for verification, with array theories being among the most promising future directions [DHK16, dMB09, SBDL01, BMS06].

Combining Static Alias Tracking and Dynamic Frames. It would be interesting to investigate a combination of our general heap encoding and Stainless' existing imperative phase, which has the benefit of producing simpler functional programs. Our heap encoding breaks the fundamental assumption of the existing imperative phase, i.e., references may now be aliased. One could address this issue in various ways. Languages like Rust have had great success in popularizing ideas from linear type systems. The idea would be to adopt a typing discipline that treats mutable types as unique by default and requires explicitly annotating any sharable references. Stainless' existing imperative phase would only transform operations on unique references.

It would also be interesting to explore an approach based on encapsulation [TSKJB17]. One would allow aliasing of fields that are annotated as implementation-internal (e.g., `private`), and prevent aliased references from leaking through the public interface of the class. In Section 3.3 we noted how first-class heaps let us prove determinism of a cache, justifying that the cached function's result is the same no matter the cache state. Now, if we were to achieve encapsulation of the cache field (preventing it from leaking elsewhere in the program), we could go one step further and treat the entire component as referentially transparent, avoiding the overhead of even modelling it on the heap of client code.

A related approach for alias control has been successfully deployed in the Pony language for concurrent programming. Pony differentiates between `iso`lated and normal `ref`erence capabilities [CDBM15] (among several others). The crucial idea is that while `ref`s may alias, they cannot escape the actor owning the object, which in our setting would correspond to some abstraction boundary (a class or module).

Applying Stainless to Systems Programming. Another opportunity would be to use the expressiveness added in Part I of this thesis to expand Stainless to new applications in systems programming. Stainless already has support for compiling a limited fragment to C [Ant17],

which could now be expanded.

Furthermore, Rust constitutes an attractive target for verification, because, like Scala, it features memory-safety by default as well as high-level language abstractions such as ADTs, higher-order functions and type classes. Perhaps most importantly it also includes typing features that ensure heap separation in many situations, meaning Rust programmers are already familiar with the mindset required to provide reads and modifies sets. We undertook some experiments to build an alternative frontend for Stainless that consumes typed Rust programs [SBRY21]. This work was expanded upon in the context of a master thesis [Bol21] and now supports some promising examples such as verifying correctness of in-place insertion into a red-black tree. It would be interesting to extend this work to make use of our new heap encoding and integrate more information from Rust's type system.

Parting Words

Like many before us, we posed the question of how programming languages ought to evolve to make verification an integral part of modern software development. In this thesis we proposed solutions for two particular aspects: how to reason about heap-manipulating programs without imposing a particular aliasing discipline, and how to gradually introduce specifications through an expressive type system. Richer type systems, in particular, appear to be one of the most promising approaches to lower the barrier to entry and make verification not only expressive but also ergonomic. We hope that automated program verifiers and general-purpose language design will continue to converge, and that in the not so far future verification will become a part of software development much like testing and interface design are nowadays.

A An SMT-LIB Encoding of Heaps

Below we show a minimalistic encoding of heaps in SMT-LIB which explicitly distinguishes `free` from `allocated` locations. Both indices and state of allocated objects are simply modelled as integers. The example also illustrates how to project the set of `alloc`ated objects from a heap and how to express separation between two heaps. Note that the encoding relies on both the theory of algebraic data types and the generalized theory of arrays supported by Z3.

```
(set-option :produce-models true)

; A simple ADT representing object state on the heap.
(declare-datatype State ( (free) (allocated (data Int)) ))

; Object references are integers. Objects are sets of integers. Heaps map objects to state.
(define-sort Object () Int)
(define-sort Objects () (Set Object))
(define-sort Heap () (Array Object State))

; The empty set of objects.
(define-fun empty-alloc-set () Objects ((as const Objects) false))
; The heap without any objects allocated.
(define-fun empty-heap () Heap ((as const Heap) free))

; The set of allocated objects in a given heap.
(define-fun alloc ((heap Heap)) Objects
  ((_ map (is-allocated (State) Bool)) heap) )

; Whether two heaps are orthogonal (i.e. separate).
(define-fun orthogonal ((h1 Heap) (h2 Heap)) Bool
  (= (intersection (alloc h1) (alloc h2)) empty-alloc-set) )
```

```
; Two heaplets containing two distinct objects.
(declare-fun h1 () Heap)
(declare-fun h2 () Heap)
(declare-fun o1 () Object)
(declare-fun o2 () Object)
(assert (is-allocated (select h1 o1)))
(assert (is-allocated (select h2 o2)))
(assert (not (= o1 o2)))

; We bind the alloc sets of each heap so they will show up in the model below.
(declare-fun alloc-set1 () Objects)
(declare-fun alloc-set2 () Objects)
(assert (= alloc-set1 (alloc h1)))
(assert (= alloc-set2 (alloc h2)))

; Are h1 and h2 necessarily separate?
(assert (not (orthogonal h1 h2)))

(check-sat)
; => No!
;  sat
(get-model)
; => They might have a third object in common, for instance:
;  (model
;   (define-fun o1 () Int 0)
;   (define-fun o2 () Int 1)
;   (define-fun alloc-set1 () (Set Int)
;    (lambda ((x!1 Int)) (or (= x!1 0) (= x!1 4))))
;   (define-fun alloc-set2 () (Set Int)
;    (lambda ((x!1 Int)) (or (= x!1 1) (= x!1 4))))
;   (define-fun h1 () (Array Int State)
;    (store (store ((as const (Array Int State)) free)
;      0 (allocated 2))
;      4 (allocated 6)))
;   (define-fun h2 () (Array Int State)
;    (store (store ((as const (Array Int State)) free)
;      1 (allocated 3))
;      4 (allocated 5)))
;  )
```

B Verifying an Inductive Heap Property

The code listing below contains the `NodeCycle` example of Chapter 5. In it we define a simplified `Node` data structure that only contains a `next` pointer to another node, along with an inductive heap predicate `cyclic` that holds if and only if the given list of `nodes` forms a cyclic, singly-linked list. The `prepend` function underneath implements insertion of a new `node` at the beginning of an existing `cyclic` list consisting of `nodes`. We prove its correctness by establishing that after `prepend` has been called `node :: nodes` again forms a cyclic list. Note that in a real implementation `nodes` would merely be used as a specification variable and could thus be marked **`@ghost`**. To access the first and last nodes at runtime (to update the `next` pointers in `prepend`), one would separately maintain references to these two particular nodes.

```
import stainless.lang._
import stainless.annotation._
import stainless.collection._
import stainless.proof._
import ListOps.noDuplicate

object NodeCycleExample {
  /* Auxiliary definitions and lemmas */

  object ListLemmas {
    def lastByIndex[T](xs: List[T]): Unit = {
      require(xs.nonEmpty)
      xs.tail match {
        case Nil() => ()
        case xs0 => lastByIndex(xs0)
      }
    } ensuring (_ => xs(xs.size - 1) == xs.last)
```

```
  def initByIndex[T](xs: List[T], i: BigInt): Unit = {
    require(xs.nonEmpty && 0 <= i && i < xs.size - 1)
    if (i > 0) initByIndex(xs.tail, i - 1)
  } ensuring (_ => xs(i) == xs.init(i))

  def applyContent[T](xs: List[T], i: BigInt): Unit = {
    require(0 <= i && i < xs.size)
    xs match {
      case Cons(_, xs0) => if (i > 0) applyContent[T](xs0, i - 1)
    }
  } ensuring (_ => xs.content.contains(xs.apply(i)))

  def noDuplicateLast[T](xs: List[T]): Unit = {
    require(xs.nonEmpty && noDuplicate(xs))
    if (xs.size > 1) noDuplicateLast(xs.tail)
    ()
  } ensuring (_ => !xs.init.content.contains(xs.last))
}

/* Node data structure and cyclicity property */

case class Node(var next: Option[Node]) extends AnyHeapRef

def cyclic(nodes: List[Node], i: BigInt = 0): Boolean = {
  reads(nodes.content.asRefs)
  require(0 <= i && i < nodes.size)
  ListLemmas.applyContent(nodes, i)
  if (i == nodes.size - 1)
    nodes(i).next == Some(nodes(0))
  else
    nodes(i).next == Some(nodes(i + 1)) && cyclic(nodes, i + 1)
}

/* Lemma: Prepending maintains cyclicity */

def cyclicPrependLemma(h0: Heap, h1: Heap, nodes: List[Node],
                       node: Node, i: BigInt = 0): Unit = {
  require(
    0 <= i && i < nodes.size &&
    h0.eval { cyclic(nodes, i) } &&
    Heap.unchanged(nodes.init.content.asRefs, h0, h1) &&
    h1.eval { nodes.last.next == Some(node) }
```

```
    )
    if (i == nodes.size - 1) {
      ListLemmas.lastByIndex(nodes)   // nodes(nodes.size – 1) == nodes.last
    } else {
      ListLemmas.initByIndex(nodes, i)         // nodes(i) == nodes.init(i)
      ListLemmas.applyContent(nodes.init, i)   // nodes.init.content
                                               //  .contains(nodes.init(i))
      assert(h1.eval { nodes(i).next == Some(nodes(i + 1)) })
      cyclicPrependLemma(h0, h1, nodes, node, i + 1)
    }
  } ensuring (_ => h1.eval { cyclic(node :: nodes, i + 1) })

  def prepend(nodes: List[Node], node: Node): List[Node] = {
    reads(nodes.content.asRefs ++ Set(node))
    modifies(nodes.content.asRefs ++ Set(node))
    require(nodes.nonEmpty && cyclic(nodes) && noDuplicate(nodes) &&
            !nodes.content.contains(node) )
    val h0 = Heap.get

    node.next = Some(nodes.head)
    nodes.last.next = Some(node)

    val h1 = Heap.get
    ListLemmas.noDuplicateLast(nodes)   // Heap.unchanged(
                                        //  nodes.init.content.asRefs, h0, h1)
    cyclicPrependLemma(h0, h1, nodes, node)
    node :: nodes
  } ensuring (newNodes => newNodes == node :: nodes &&
                          cyclic(newNodes) && noDuplicate(newNodes) )
}
```

Bibliography

[AC96] David Aspinall and Adriana Compagnoni. Subtyping dependent types. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 1996.

[AHM+17] Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra monads for free. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 515–529, 2017.

[AL91] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[Ant17] Marco Antognini. Extending safe C support in Leon. Master's thesis, EPFL, Lausanne, 2017.

[AR17] Nada Amin and Tiark Rompf. Type soundness proofs with definitional interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL'17. ACM, 2017.

[Asp94] David Aspinall. Subtyping with singleton types. In *International Workshop on Computer Science Logic*. Springer, 1994.

[Aug98] Lennart Augustsson. Cayenne — a language with dependent types. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP'98. ACM, 1998.

[Bar91] Henk Barendregt. Introduction to generalized type systems. *Journal of functional programming*, 1(2), 1991.

[BBDL+17] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay Lorch, Kenji Maillard, Jianyang Pang, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella-Béguelin, and Jean-Karim Zinzindohoué. Everest: Towards a verified, drop-in replacement of HTTPS. In *2nd Summit on Advances in Programming Languages*, May 2017.

[BBKO22] Olivier Blanvillain, Jonathan Brachthäuser, Maxime Kjaer, and Martin Odersky. Type-level programming with match types. In *Symposium on Principles of Programming Languages (POPL, to appear)*, 2022.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[BCD+05] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.

[BCK11] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In *International Symposium on Formal Methods*, pages 200–214. Springer, 2011.

[BCLR04] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *International Conference on Integrated Formal Methods*, pages 1–20. Springer, 2004.

[BCO05] Josh Berdine, Cristiano Calcagno, and Peter W O'Hearn. Symbolic execution with separation logic. In *Asian Symposium on Programming Languages and Systems*, pages 52–68. Springer, 2005.

[BJA+21] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, et al. Using lightweight formal methods to validate a key-value storage node in amazon s3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, pages 836–850, 2021.

[BK15] Régis Blanc and Viktor Kuncak. Sound Reasoning about Integral Data Types with a Reusable SMT Solver Interface. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala*, SCALA 2015, page 35–40, New York, NY, USA, 2015. Association for Computing Machinery.

[BKKS13] Régis Blanc, Viktor Kuncak, Etienne Kneuss, and Philippe Suter. An Overview of the Leon Verification System: Verification by Translation to Recursive Functions. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, New York, NY, USA, 2013. Association for Computing Machinery.

[BL05] Mike Barnett and K Rustan M Leino. Weakest-precondition of unstructured programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 82–87, 2005.

[Bla17] Régis William Blanc. *Verification by Reduction to Functional Programs*. PhD thesis, EPFL, Lausanne, 2017.

[Bla21] Olivier Blanvillain. Private communication, 2021.

[BLS04] Mike Barnett, K Rustan M Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69. Springer, 2004.

[BMS06] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's Decidable About Arrays? In E. Allen Emerson and Kedar S. Namjoshi, editors, *Verification, Model Checking, and Abstract Interpretation*, Lecture Notes in Computer Science, pages 427–442, Berlin, Heidelberg, 2006. Springer.

[Bol21] Yann Bolliger. Formal Verification of Rust with Stainless. Master's thesis, EPFL, Lausanne, 2021.

[Boy03] John Boyland. Checking interference with fractional permissions. In *International Static Analysis Symposium*, pages 55–72. Springer, 2003.

[Bra13] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(05), 2013.

[Bur13] Eugene Burmako. Scala Macros: Let Our Powers Combine! On How Rich Syntax and Static Types Work with Metaprogramming. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, New York, NY, USA, 2013. Association for Computing Machinery.

[Can18] Jorge Vicente Cantero. Speeding Up Compilation Time with scalac-profiling. https://www.scala-lang.org/blog/2018/06/04/scalac-profiling.html, 2018.

[Car88] Luca Cardelli. Structural subtyping and the notion of power type. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'88. ACM, 1988.

[CCH+89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of the fourth international conference on functional programming languages and computer architecture*, pages 273–280, 1989.

[CD11] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of C programs. In *NASA Formal Methods Symposium*, pages 459–465. Springer, 2011.

[CDBM15] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, pages 1–12, 2015.

[CDOY11] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26:1–26:66, 2011.

[Chl10] Adam Chlipala. Ur: Statically-typed metaprogramming with type-level record computation. In *PLDI'10: Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation*, June 2010.

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[Cou03] Judicaël Courant. Strong normalization with singleton types. *Electronic Notes in Theoretical Computer Science*, 70(1), 2003.

[CP16] Silvia Crafa and Luca Padovani. On the chemistry of typestate-oriented actors. *CoRR*, abs/1607.02927, 2016.

[CV18] Joana Campos and Vasco T Vasconcelos. Dependent types for class-based mutable objects. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[DFLO19] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Scaling static analyses at Facebook. *Commun. ACM*, 62(8):62–70, 2019.

[DHK16] Przemyslaw Daca, Thomas A. Henzinger, and Andrey Kupriyanov. Array folds logic. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 230–248. Springer, 2016.

[Die17] Henning Dieterichs. TypeScript Issue #14833: TypeScripts Type System is Turing Complete. https://github.com/Microsoft/TypeScript/issues/14833, 2017.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[Dij76] Edsger W. Dijkstra. *A discipline of programming*. Prentice-Hall, 1976.

[dMB08a] Leonardo Mendonça de Moura and Nikolaj Bjørner. Model-based theory combination. *Electron. Notes Theor. Comput. Sci.*, 198(2):37–49, 2008.

[dMB08b] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[dMB09] Leonardo Mendonça de Moura and Nikolaj Bjørner. Generalized, efficient array decision procedures. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*, pages 45–52. IEEE, 2009.

[dMKA+15] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

[DPJ08] Dino Distefano and Matthew J Parkinson J. jStar: Towards practical verification for Java. *ACM Sigplan Notices*, 43(10):213–226, 2008.

[Eis16] Richard A Eisenberg. *Dependent types in Haskell: Theory and practice*. PhD thesis, University of Pennsylvania, 2016.

[EMH19] Marco Eilers, Peter Müller, and Samuel Hitz. Modular product programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 42(1):1–37, 2019.

[EOW07] Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, ECOOP'07, Berlin, Heidelberg, 2007. Springer-Verlag.

[EW12] Richard A Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. In *Haskell Symposium 2012*, volume 47. ACM New York, NY, USA, 2012.

[Fil03] Jean-Christophe Filliâtre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4):709–745, 2003.

[Flo67a] Robert W Floyd. Assigning meanings to programs. In *Proceedings of Symposium in Applied Mathematics*, volume 19, pages 19–32. American Mathematical Society, 1967.

[Flo67b] Robert W. Floyd. Nondeterministic algorithms. *J. ACM*, 14(4), October 1967.

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 – where programs meet provers. In *European Symposium on Programming (ESOP)*, pages 125–128. Springer, 2013.

[Fra21] Frameless Contributors. Frameless. https://github.com/typelevel/frameless, 2016–2021.

[FS01] Cormac Flanagan and James B Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, pages 193–205, 2001.

[Göd31] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38(1):173–198, 1931.

[HA28] David Hilbert and Wilhelm Ackermann. Grundzüge der theoretischen Logik, volume XXVII of. *Die Grundlehren der Mathematischen Wissenschaften*, 1928.

[Hay91] Susumu Hayashi. Singleton, union and intersection types for program extraction. In *International Symposium on Theoretical Aspects of Computer Software*, TACS'91. Springer, 1991.

[HHK+15] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. IronFleet: Proving Practical Distributed Systems Correct. In Ethan L. Miller and Steven Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 1–17. ACM, 2015.

[HMWC15] Duc Hoang, Yannick Moy, Angela Wallenburg, and Roderick Chapman. SPARK 2014 and GNATprove. *International Journal on Software Tools for Technology Transfer*, 17(6):695–707, 2015.

[HN90] Laurie J. Hendren and Alexandru Nicolau. Parallelizing programs with recursive data structures. *IEEE Trans. Parallel Distributed Syst.*, 1(1):35–47, 1990.

[Hoa69] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hut10] DeLesley S. Hutchins. Pure subtype systems. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'10. ACM, 2010.

[HVK98] Kohei Honda, Vasco T Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *European Symposium on Programming (ESOP)*, pages 122–138. Springer, 1998.

[HVK19] Jad Hamza, Nicolas Voirol, and Viktor Kunčak. System FR: Formalized Foundations for the Stainless Verifier. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[JKJ+18] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018.

[JM79] Neil D Jones and Steven S Muchnick. Flow analysis and optimization of LISP-like structures. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL)*, pages 244–256, 1979.

[JM82] Neil D Jones and Steven S Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 66–74, 1982.

[Jon00] Mark P. Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, ESOP '00, London, UK, UK, 2000. Springer-Verlag.

[JS13] Daniel Jost and Alexander J Summers. An automatic encoding from VeriFast predicates into implicit dynamic frames. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 202–221. Springer, 2013.

[JSP+11] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods Symposium*, pages 41–55. Springer, 2011.

[Kas06] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2006.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 207–220, 2009.

[KGV+19] Milod Kazerounian, Sankha Narayan Guria, Niki Vazou, Jeffrey S Foster, and David Van Horn. Type-level computations for Ruby libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019.

[KH21] Viktor Kuncak and Jad Hamza. Stainless verification system tutorial. In *13th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*, 2021.

[KJS10] Oleg Kiselyov, Simon Peyton Jones, and Chung-chieh Shan. Fun with type functions. In *Reflections on the Work of CAR Hoare*. Springer, 2010.

[KLS04] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*. ACM, 2004.

[KM96] Matt Kaufmann and J Strother Moore. ACL2: An industrial strength version of Nqthm. In *Proceedings of 11th Annual Conference on Computer Assurance. (COMPASS)*, pages 23–34. IEEE, 1996.

[KM01] Nils Klarlund and Anders Møller. *MONA version 1.4: User manual*. BRICS, Department of Computer Science, University of Aarhus Denmark, 2001.

[KMNO14] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *Principles of Programming Languages (POPL)*, pages 179–191. ACM Press, January 2014.

[KMV19] Filip Křikava, Heather Miller, and Jan Vitek. Scala implicits are everywhere: a large-scale study of the use of Scala implicits in the wild. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–28, 2019.

[KP18] Viktor Kuncak and Aleksandar Prokopec. Parallel programming (Lecture 1.4: Running computations in parallel). EPFL Courseware, February 2018. https://courseware.epfl.ch/courses/course-v1:EPFL+parprog1+2018_T1/about and https://www.youtube.com/watch?v=DbVt8C0-Oe0.

[LAR21] EPFL IC LARA. Stainless: Formal verification for Scala. https://stainless.epfl.ch/, 2021.

[LAS00] Tal Lev-Ami and Mooly Sagiv. TVLA: A system for implementing static analyses. In *International Static Analysis Symposium (SAS)*, pages 280–301. Springer, 2000.

[Lei08] K. Rustan M. Leino. This is Boogie 2. *Manuscript KRML*, 178(131):9, 2008.

[Lei10] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[LGvH+79] David C Luckham, Steven M German, Friedrich W von Henke, Richard A Karp, PW Milne, Derek C Oppen, Wolfgang Polak, and William L Scherlis. Stanford Pascal Verifier user manual, 1979.

[LH88] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In Richard L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, pages 21–34. ACM, 1988.

[LM99] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Proceedings of the Second Conference on Domain-Specific Languages*. ACM, 1999.

[LMS09] K Rustan M Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with Chalice. In *Foundations of Security Analysis and Design V*, pages 195–222. Springer, 2009.

[LR10] K Rustan M Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 312–327. Springer, 2010.

[LS09] Dirk Leinenbach and Thomas Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In *International Symposium on Formal Methods*, pages 806–809. Springer, 2009.

[MBB06] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 2006.

[McB02] Conor McBride. Faking it: Simulating dependent types in Haskell. *Journal of functional programming*, 12(4-5), 2002.

[Mey97] Bertrand Meyer. *Object-oriented software construction*, volume 2. Prentice Hall Englewood Cliffs, 1997.

[MLS84] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*, volume 9. Bibliopolis Naples, 1984.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, 1991.

[MPO08a] Adriaan Moors, Frank Piessens, and Martin Odersky. Generics of a higher kind. volume 43, page 423–438, New York, NY, USA, Oct 2008. Association for Computing Machinery.

[MPO08b] Adriaan Moors, Frank Piessens, and Martin Odersky. Safe type-level abstraction in Scala. In *Proceedings of the International Workshop on Foundations of Object-Oriented Languages (FOOL)*, pages 1–13, 2008.

[MS01] Anders Møller and Michael I Schwartzbach. The Pointer Assertion Logic Engine. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation (PLDI)*, pages 221–231, 2001.

[MSS16] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Automatic verification of iterated separating conjunctions using symbolic execution. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 405–425. Springer International Publishing, 2016.

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[NRZ+15] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon web services uses formal methods. *Communications of the ACM*, 58(4):66–73, 2015.

[OBL+18] Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. Simplicitly: Foundations and applications of implicit function types. In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL'18. ACM, 2018.

[OMO10] Bruno CdS Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA)*, pages 341–360, 2010.

[OMP16] Martin Odersky, Guillaume Martres, and Dmitry Petrashko. Implementing higher-kinded types in Dotty. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*, pages 51–60, 2016.

[ORY01] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.

[OSV19] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala, Fourth Edition (A comprehensive step-by-step guide)*. Artima, 2019.

[PS12] Matthew J. Parkinson and Alexander J. Summers. The relationship between separation logic and implicit dynamic frames. *Log. Methods Comput. Sci.*, 8(3), 2012.

[PWZ13] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating separation logic using SMT. In *International Conference on Computer Aided Verification (CAV)*, pages 773–789. Springer, 2013.

[PWZ14a] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating separation logic with trees and data. In *International Conference on Computer Aided Verification (CAV)*, pages 711–728. Springer, 2014.

[PWZ14b] Ruzica Piskac, Thomas Wies, and Damien Zufferey. GRASShopper: Complete heap verification with mixed specifications. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 124–139. Springer, 2014.

[RA16] Tiark Rompf and Nada Amin. Type soundness for dependent object types (DOT). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 624–641, 2016.

[Rey02] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.

[Ric53] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[RK15] Andrew Reynolds and Viktor Kuncak. Induction for SMT solvers. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2015.

[Sab21] Miles Sabin. Shapeless. https://github.com/milessabin/shapeless, 2011–2021.

[SBDL01] A. Stump, C.W. Barrett, D.L. Dill, and J. Levitt. A decision procedure for an extensional theory of arrays. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, pages 29–37, Boston, MA, USA, 2001. IEEE Comput. Soc.

[SBHK20] Georg Stefan Schmid, Olivier Blanvillain, Jad Hamza, and Viktor Kuncak. Coming to Terms with Your Choices: An Existential Take on Dependent Types. *CoRR*, abs/2011.07653, 2020.

[SBO21] Nicolas Stucki, Jonathan Immanuel Brachthäuser, and Martin Odersky. Multi-stage programming with generative and analytical macros. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2021, page 110–122, New York, NY, USA, 2021. Association for Computing Machinery.

[SBRY21] Georg Stefan Schmid, Yann Bolliger, Romain Ruetschi, and Ivan Yurov. rust-stainless. https://github.com/epfl-lara/rust-stainless, 2020–2021.

[SDK10] Philippe Suter, Mirco Dotta, and Viktor Kuncak. Decision procedures for algebraic data types with abstractions. In *Symposium on Principles of Programming Languages (POPL)*, page 199–210, 2010.

[SH00] Christopher A Stone and Robert Harper. Deciding type equivalence in a language with singleton kinds. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2000.

[SHK+16] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin.

Dependent types and multi-monadic effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270. ACM, January 2016.

[Sip13] Michael Sipser. *Introduction to the Theory of Computation (3rd ed.)*. Cengage Learning, 2013. ISBN-13: 978-1-133-18779-0.

[SJP12] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1):2:1–2:58, 2012.

[SK16] Georg Stefan Schmid and Viktor Kuncak. SMT-based Checking of Predicate-qualified Types for Scala. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala (SCALA)*, pages 31–40, 2016.

[SKK11] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability modulo recursive programs. In *International Static Analysis Symposium (SAS)*, pages 298–315. Springer, 2011.

[SN08] Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 28–32, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[SPR+19] Mihaela Sighireanu, Juan Antonio Navarro Pérez, Andrey Rybalchenko, Nikos Gorogiannis, Radu Iosif, Andrew Reynolds, Cristina Serban, Jens Katelaan, Christoph Matheja, Thomas Noll, Florian Zuleger, Wei-Ngan Chin, Quang Loc Le, Quang-Trung Ta, Ton-Chanh Le, Thanh-Toan Nguyen, Siau-Cheng Khoo, Michal Cyprian, Adam Rogalewicz, Tomás Vojnar, Constantin Enea, Ondrej Lengál, Chong Gao, and Zhilin Wu. SL-COMP: competition of solvers for separation logic. In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*, volume 11429 of *Lecture Notes in Computer Science*, pages 116–132. Springer, 2019.

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.

[SY86] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, SE-12(1):157–171, 1986.

[SY16] Alceste Scalas and Nobuko Yoshida. Lightweight Session Programming in Scala. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:28, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Tho21] Frank Thomas. refined: Simple refinement types for Scala. https://github.com/fthomas/refined, 2015–2021.

[TLKC19] Quang-Trung Ta, Ton Chanh Le, Siau-Cheng Khoo, and Wei-Ngan Chin. Automated mutual induction proof in separation logic. *Formal Aspects Comput.*, 31(2):207–230, 2019.

[Tor17] Eric Torreborre. Achieving 3.2x Faster Scala Compile Time. https://engineering.zalando.com/posts/2017/04/achieving-3.2x-faster-scala-compile-time.html, 2017.

[TSKJB17] Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. A Logical Relation for Monadic Encapsulation of State: Proving Contextual Equivalences in the Presence of RunST. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017.

[Tur37] Alan Mathison Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.

[UdM19] 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.

[VBJ15] Niki Vazou, Alexander Bakst, and Ranjit Jhala. Bounded refinement types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. ACM, 2015.

[VKK15] Nicolas Voirol, Etienne Kneuss, and Viktor Kuncak. Counter-example complete verification for higher-order functions. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala*, SCALA 2015, page 18–29, New York, NY, USA, 2015. Association for Computing Machinery.

[Voi19] Nicolas Charles Yves Voirol. *Verified Functional Programming*. PhD thesis, EPFL, Lausanne, 2019.

[VRJ13] Niki Vazou, Patrick M Rondon, and Ranjit Jhala. Abstract refinement types. In *European Symposium on Programming*, pages 209–228. Springer, 2013.

[VSJ+14] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for haskell. *SIGPLAN Not.*, 49(9):269–282, aug 2014.

[VTC+17] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G Scott, Ryan R Newton, Philip Wadler, and Ranjit Jhala. Refinement reflection: complete verification with smt. *Proceedings of the ACM on Programming Languages*, 2(POPL), 2017.

[VTVH18] Niki Vazou, Éric Tanter, and David Van Horn. Gradual liquid type inference. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), 2018.

[Wad90a] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 61–78, 1990.

[Wad90b] Philip Wadler. Linear types can change the world! In *Programming concepts and methods*, volume 3, page 5. Citeseer, 1990.

[WB89] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, page 60–76, New York, NY, USA, 1989. Association for Computing Machinery.

[WMK11] Thomas Wies, Marco Muñiz, and Viktor Kuncak. An efficient decision procedure for imperative tree data structures. In *International Conference on Automated Deduction (CAV)*, pages 476–491. Springer, 2011.

[WVdAE17] Stephanie Weirich, Antoine Voizard, Pedro Henrique Azevedo de Amorim, and Richard A Eisenberg. A specification for dependent types in Haskell. In *Proceedings of the ACM on Programming Languages*, volume 1 of *ICFP'17*. ACM, 2017.

[XP98] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'98, Montreal, June 1998.

[YBO16] Yanpeng Yang, Xuan Bi, and Bruno CDS Oliveira. Unified syntax with iso-types. In *Asian Symposium on Programming Languages and Systems*. Springer, 2016.

[YO17] Yanpeng Yang and Bruno CDS Oliveira. Unifying typing and subtyping. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA), 2017.

[YWC+12] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI'12. ACM, 2012.

[Zwa99] Jan Zwanenburg. Pure type systems with subtyping. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 1999.

[ZXW+16] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache Spark: A unified engine for big data processing. *Commun. ACM*, 59(11), 2016.

Georg Stefan Schmid

EPFL-IC-LARA, BC355, St. 14
1015 Lausanne, Switzerland

Research Interests

I believe programming languages should be versatile tools that allow programmers to perform common tasks concisely and safely. Most of my PhD work was dedicated to taking the Scala ecosystem one step in this direction through type-level programming and automated verification. More generally, I am excited by problems at the intersection of formal methods and systems programming, and compilers seem to be a never-ending source of such problems!

Work Experience

Feb – Jun 2021 **Tech Infrastructure Research Intern at DeepMind**

I contributed to the design, implementation and evaluation of PartIR, an automated partitioner for ML models. PartIR leverages data and model parallelism to enable the training of large-scale models on accelerators such as Google's TPUs. I explored new techniques to harness the vast search space, significantly simplified baselines, and generalized PartIR to apply to a larger class of models.

May – Aug 2019 Software Engineering Intern at Google Munich

I worked on V8, the JavaScript engine embedded in Google Chrome. Among other things, I extended its JavaScript *load elimination* phase to exploit the constancy of fields at runtime, speeding up the DeltaBlue benchmark by almost 5%. I also implemented a completely new load elimination phase for the lower-level IR. Finally, I contributed to the design and implementation of Torque, a DSL for builtins, adding *generic structs* and improving array support through the addition of Go-style *slices*.

Jun – Aug 2015 Software Engineering Intern at Google Mountain View

I designed and implemented a distributed system that samples activity in YouTube's video transcoding infrastructure and generates representative workloads on the fly. My system helped improve the accuracy of regression testing and is now part of the transcoding team's release process.

Education

Sep 2016 – **École polytechnique fédérale de Lausanne (EPFL)**

PhD thesis in Computer Science: "Scaling Language Features for Program Verification"
Advisor: Prof. Viktor Kunčak

Sep 2014 – Jul 2016 École polytechnique fédérale de Lausanne

Master's degree program in Computer Science, GPA: 5.75/6.00
Master thesis on "SMT-Based Checking of Predicate-Qualified Types in Scala" supervised by Prof. Viktor Kunčak

Oct 2010 – Jul 2014 Vienna University of Technology (VUT)

Bachelor's degree in Computer Science (Software and Information Engineering) with distinction ("mit ausgezeichnetem Erfolg")

PUBLICATIONS

Under submission — Memory-efficient array redistribution through portable collective communication

Norman A. Rink, Adam Paszke, Dimitrios Vytiniotis and **Georg Stefan Schmid**
On arXiv: https://arxiv.org/abs/2112.01075

Jan 2022 — Generalized Arrays for Stainless Frames

Georg Stefan Schmid and Viktor Kunčak, to appear at VMCAI 2022: 23rd International Conference on Verification, Model Checking, and Abstract Interpretation

Dec 2021 — Automap: Towards Ergonomic Automated Parallelism for ML Models

Michael Schaarschmidt, Dominik Grewe, Dimitrios Vytiniotis, Adam Paszke, **Georg Stefan Schmid**, Tamara Norman, James Molloy, Jonathan Godwin, Norman A. Rink, Vinod Nair and Dan Belov
On arXiv: https://arxiv.org/abs/2112.02958

Nov 2020 — Coming to Terms with Your Choices: An Existential Take on Dependent Types

Georg Stefan Schmid, Olivier Blanvillain, Jad Hamza and Viktor Kunčak
On arXiv: https://arxiv.org/abs/2011.07653

Oct 2016 — SMT-Based Checking of Predicate-Qualified Types for Scala

Georg Stefan Schmid and Viktor Kunčak, at Scala'16: 7th ACM SIGPLAN Symposium on Scala

HONORS

International Olympiad in Informatics (IOI)

I participated in the IOI as part of the Austrian team for three consecutive years:

2008 in Cairo, Egypt – came in *second* in national qualifications
2007 in Zagreb, Croatia – came in *third* in national qualifications
2006 in Mérida, Mexico – came in *fourth* in national qualifications

Merit-based scholarships at Vienna University of Technology

Throughout my studies at VUT I received merit-based scholarships. These are typically awarded to less than 20 students in each year's class (out of approximately 800 students starting every year).

Research scholarship at École polytechnique fédérale de Lausanne

During my Master's studies I worked as a research assistant in Prof. George Candea's Dependable Systems Lab. In addition to granting me early experiences in research, EPFL provided me a monthly stipend.