Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

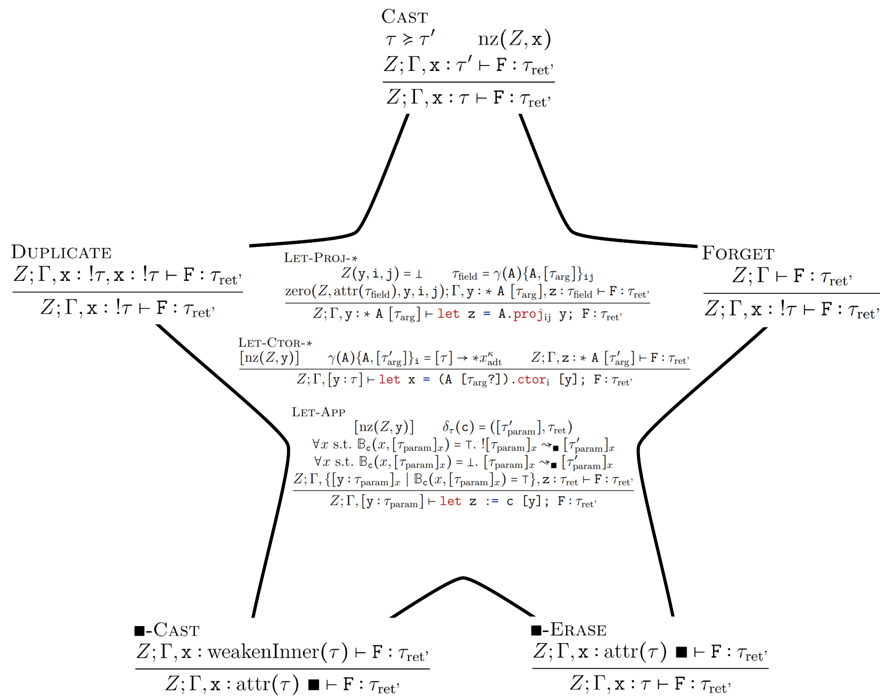
Lehrstuhl Prof. Dr.-Ing. Snelting

Static Uniqueness Analysis for the Lean 4 Theorem Prover

Masterarbeit von

Marc Huisinga

an der Fakultät für Informatik



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

Abgabedatum: 5. April 2023

Abstract

Der Lean 4 Theorembeweiser, welcher auch als vollwertige Programmiersprache verwendbar ist, implementiert eine auf Referenzzählung basierende Optimierung, die das destruktive Mutieren von pur-funktionalen Werten ermöglicht: Wenn der Referenzzähler eines Werts gleich 1 ist, dann kann dieser sicher an Ort und Stelle mutiert werden. Das Greifen dieser Optimierung ist insbesondere für Arrays unabdingbar, da dort die einzige Alternative zu einer Mutation das vollständige Kopieren des Arrays ist. Um das Greifen dieser Optimierung zu garantieren, mustern wir den Entwurfsraum der substrukturellen Typtheorien und implementieren eine eigene Uniqueness-Typtheorie für ein Modell einer Zwischenrepräsentation von Lean 4. Unsere Typtheorie unterstützt Uniqueness-Typen, algebraische Datentypen, gelöschte Typen, externe Deklarationen, Subtyping von Uniqueness-Attributen und einen Borrowing-Mechanismus, welcher mittels einer Escape-Datenflussanalyse implementiert ist.

The Lean 4 theorem prover and programming language implements an optimization based on reference counting that allows for destructively updating purely functional values: If the reference count of a value is equal to 1, it can be safely updated in-place. Especially for arrays, where the only alternative to an in-place update is a full copy of the array, it is essential that this optimization always applies. To ensure this, we survey the design space of substructural type theory and implement a uniqueness type theory of our own. Our type theory targets a model of an intermediate representation for Lean 4. It supports uniqueness types, algebraic data types, erased types, external declarations, subtyping for uniqueness attributes and a borrowing mechanism that is implemented using an escape data-flow analysis.

Contents

1 Introduction **7**

2 Background **9**
2.1 Lean 4 Theorem Prover 9
2.2 Dependent Type Theory 11
2.3 Intermediate Representations for Lean 4 13
2.4 Destructive Updates 16
2.5 Linear Type Theory 18
2.6 Quantitative Type Theory 24
2.7 Uniqueness Type Theory 28
2.8 Borrowing 32

3 Design Space Exploration **37**
3.1 Substructural Framework 37
3.2 Uniqueness Type Systems 39
3.2.1 Higher-Order Functions 39
3.2.2 Implicit Coercion 42
3.2.3 Uniqueness Propagation 42
3.2.4 Borrowing 43

4 Formal Specification **45**
4.1 Types 45
4.1.1 Syntax 45
4.1.2 Propagation 47
4.1.3 Definitional Nuances 47
4.1.4 Utilities 48
4.1.5 Subtyping 49
4.2 Intermediate Representation 50
4.2.1 Syntax 50
4.3 Escape Analysis 51
4.3.1 Syntax 51
4.3.2 Computing Escapees 52
4.4 Borrowing 55
4.4.1 Syntax 56
4.4.2 Unique Fields 56
4.4.3 Borrowed Parameters 58

4.5 Type-Checking 59
4.5.1 Syntax 59
4.5.2 Type Theory 59

5 Implementation **67**
5.1 Deviations From Specification 67
5.2 Constructor Type Inference 68
5.3 Examples 71

6 Future Work **77**
6.1 Type Inference 77
6.2 Integration With Lean 4 78
6.3 Borrowing 78
6.4 Higher-Order Functions 79
6.5 Polymorphism 79
6.6 Proof of Soundness 80

7 Related Work **83**

8 Conclusion **87**

1 Introduction

The Lean 4 theorem prover and programming language [de Moura and Ullrich, 2021] features an intermediate representation (IR) language that allows for efficiently mutating values in a purely functional programming language [Ullrich and de Moura, 2020]. Garbage collection is implemented via reference counting (RC) and values can be safely and efficiently updated in-place at runtime when the value is uniquely referenced, i.e. the reference count is equal to 1, a technique known as "destructive update". This feature yields fewer memory allocations, faster updates and enables the use of arrays in purely functional programming languages [Ullrich and de Moura, 2020].

However, for programmers it can be difficult to judge whether a value is always uniquely referenced during program execution, leading to possibly huge disparities in runtime when referential uniqueness is violated by accident. This issue is especially significant when using arrays or array-derived types, where the fall-back when not being able to update the array in-place is to copy the array in its entirety, bumping the runtime for that specific function call from $\Theta(1)$ to $\Theta(n)$.

The obvious remedy for this issue is to use a type system which ensures that values are uniquely referenced and issues an error when referential uniqueness is violated. In addition to aiding debuggability, the information of successfully type-checking a program can also be used for optimization purposes: If, at compile time, we know that a value is always uniquely referenced, we can eliminate the instructions that check the reference count at runtime and always directly perform the in-place update.

To guarantee referential uniqueness in this manner, we survey the design space of substructural type theory and design and implement a uniqueness type system [Marshall et al., 2022], a kind of type system pioneered by the Clean programming language [Smetsers et al., 1994]. We implement our type checker in the Lean 4 programming language, targeting a model of Lean 4's IR language.

In chapter 2, we revisit Lean 4, its IR language and the details related to using reference counting to perform in-place updates, as well as explain and evaluate different techniques for statically ensuring referential uniqueness. In chapter 3, we explore the different facets and challenges of designing a uniqueness type system. In chapter 4, we provide a formal description of our type theory, and in chapter 5, we detail challenges and particularities affecting our implementation of a type checker for our type system. Finally, in chapter 6, we describe possible future work, and in chapter 7, we compare our approach to other similar type systems.

2 Background

2.1 Lean 4 Theorem Prover

Lean 4 is a proof assistant and programming language developed primarily by Leonardo de Moura at Microsoft Research and Sebastian Ullrich at KIT with numerous open source contributions by many other authors.

Up to and including version 3, Lean served mostly only as a proof assistant, i.e. an interactive tool where users can input proofs that are then checked by the proof assistant. Many proof assistants also implement proof-generating automation, so-called “tactic languages”, to make the task of writing a perfectly formal proof by hand less tedious. Already in Lean 3, the previous version of Lean, the same term language was used for proofs, theorem statements, definitions, programs, type declarations, specifications, and implementing automation. We will go into some detail on how Lean uses a single unified language for all of these things in Section 2.2. For automation, the term language was also evaluated by a separate interpreter for more efficient execution.

Unfortunately, evaluating the term language using a separate interpreter would still yield inadequate performance, both for implementing more demanding automation and for implementing real world programs, which meant that such demanding programs were written in C++ instead and then made available to Lean using a foreign-function interface [Ullrich and de Moura, 2020].

To improve on this, Lean 4 now implements its own self-hosted compiler toolchain for both a C backend and a work-in-progress LLVM [Lattner and Adve, 2004] backend, including its own IR, optimization pipeline and custom garbage collection algorithm. Being almost entirely self-hosted, Lean 4 is now also well capable of being used as a general purpose programming language.

Examples

Let us now consider some basic examples of Lean 4 code to get a feeling for the language. It should be noted that all of the following can be expressed more succinctly, but that we have chosen not to do so in order to make these snippets easier to grasp for readers not familiar with Lean.

```
def List.map (f : α → β) : List α → List β
  | []      => []
  | x :: xs => f x :: map f xs
```

`List.map` is implemented by recursion on the second argument. The empty list

again yields the empty list. If the list is a cons cell, we map the head of the cons cell using `f`, recurse on the tail and then build a new cons cell from the results of both.

```
def List.get? : List α → Nat → Option α
  | [],       _      => Option.none
  | x :: _,   0      => Option.some x
  | _ :: xs, n + 1 => List.get? xs n
```

`List.get?` is implemented by recursion both on the provided list and the index provided in the second argument. It yields an `Option` α, i.e. either `Option.some` α if the index is in the list, or `Option.none` otherwise.

```
def Array.groupBy (p : α → α → Ordering) (xs : Array α)
  : RBMap α (Array α) p := Id.run do
  let mut result : RBMap α (Array α) p := RBMap.empty
  for x in xs do
    let group := Option.getD (RBMap.find? p result x) #[]
    result := RBMap.insert p result x (Array.push group x)
  return result
```

`Array.groupBy` is implemented using an imperative domain-specific language (DSL) based on do-notation [Ullrich and de Moura, 2022]. It takes a relation `p` that yields an `Ordering`, i.e. whether the first argument is greater than, smaller than or equal to the second argument, as well as the array to group the elements of. It returns a red-black tree based map ordered by `p`, where the keys are arbitrary representatives of the group and the values denote groups with type `Array` α of `p`-equivalent values.

In order to use do-notation, we need to run the code in a monad. Since we do not intend to accumulate any effects and only use do-notation for its imperative domain-specific language, we use the `Id` monad, entering it using `Id.run`. First, we initialize a mutable but empty red-black tree based map, denoting our result. Then, we iterate over every element `x` in the provided array and look for a group in the current result that is `p`-equivalent to `x`. If we find such a group, we store it in `group`. Otherwise, we allocate a new empty group `#[]` for elements `p`-equivalent to `x` using the call to `Option.getD`, which returns the first element if it was equal to `some x` and otherwise returns the second element if it was equal to `none`. Then, we add `x` itself to the group and re-insert it into our mutable `result` map. At the end, we return the accumulated `result` map.

Imperative DSL

This piece of imperative code is implemented as a DSL, i.e. the imperative code is translated to a functional equivalent.

One may reasonably wonder why one would ever use a purely functional language to then write imperative code, translate it back to purely functional code, only to then compile the result to imperative machine code. However, since Lean 4 is also

an interactive theorem prover, the answer is simple: Imperative programming can be convenient, but it is easier to use a purely functional programming language for all the domains of proof, specification and writing programs at once, since all the computational effects are already neatly packed away.

In fact, if we were to write a proof about `Array.groupBy`, the first thing we would likely do is fold away the syntactical imperative layer in order to uncover the purely functional term representing the program, without ever having to think about loop invariants or state.

One may also wonder whether code written in this imperative manner is about as efficient as real imperative code. Because of the mechanism that we will describe in Section 2.4, this is indeed the case if the code in question is written in such a way that every value is uniquely referenced, which is usually the case for the kind of code that one would also write in an imperative language. Importantly, functional code will benefit from this as well, which means that we can write compositional and functional code that also updates values in-place instead of making new allocations in every single combinator.

It must however be noted that our implementation of `Array.groupBy` is actually an example of an imperative implementation that is unexpectedly inefficient. We will resolve this inefficiency in Section 2.4 when it is instrumental to do so.

Lean as a functional language

For most of this thesis, we will not treat Lean 4 as a theorem prover, but instead as a purely functional programming language that implements dependent type theory. For more details on Lean 4 as a programming language, we refer to the book Functional Programming in Lean by Christiansen [2023].

2.2 Dependent Type Theory

As described in Section 2.1, Lean uses a single language for programming and proving. It accomplishes this by implementing dependent type theory (DTT), a type theory powerful enough to declare mathematical objects, implement programs and write specifications and proofs for both. What follows is only a very brief introduction to some of the important details of Lean's type theory. We recommend the book Theorem Proving in Lean 4 by Avigad et al. [2022] for a proper introduction.

Core mechanisms

The central idea of DTT is that types are allowed to depend on terms. In most type theories, terms and types are entirely different constructs, and while terms have types, terms cannot be used in types. Removing this restriction blurs the line between programs and their static specification.

There are several mechanisms required to make this work:

1. In a quantified type $\forall x.\ \tau(x)$, the variable x is allowed to range over terms of a type (e.g. $x \triangleq n : \mathbb{N}$), not just types themselves as is the case in languages that support polymorphic functions $\forall \alpha.\ \tau(\alpha)$.

2. Terms in types can be reduced with the usual reduction rules of lambda calculus, e.g. $(\forall x.\ \tau((\lambda y.\ y)\ x)) \rightsquigarrow (\forall x.\ \tau(x))$.

3. Support for inductive type families, which are essentially algebraic data types where each constructor creates a term in a type that can be parametrized by other terms. For example, we might declare a type $\lambda \alpha : \text{Type}.\ \lambda n : \mathbb{N}.\ \text{Vec}\ \alpha\ n$ for lists over a type α of size n with constructors $\text{nil} : \text{Vec}\ \alpha\ 0$ and $\text{cons} : \forall n : \mathbb{N}.\ \alpha \to \text{Vec}\ \alpha\ n \to \text{Vec}\ \alpha\ (n+1)$. Each inductive type family also yields a recursion principle that allows for pattern matching and using recursion on the value of a type to compute an accumulate value.

In addition to $\forall x : \tau_1.\ \tau_2(x)$, dependent type theories also support dependent sigma types $(x : \tau_1) \times \tau_2(x)$ and sum types $\tau_1 + \tau_2$.

Propositions

For convenience, Lean's type theory also supports a separate type universe of propositions $\mathbb{P}$, the terms of which are types $p : \mathbb{P}$ with proof terms $h : p$ witnessing the truth of the proposition p. For example, if $\text{refl} : \forall x.\ x = x$, we have $(\text{refl}\ n) : (n = n) : \mathbb{P}$ and $(\text{refl}\ n) : (n + 1 - 1 = n) : \mathbb{P}$ for $n : \mathbb{N}$, as $n + 1 - 1 \rightsquigarrow n$ by reduction. Meanwhile, there is no term $h : (n = n + 1) : \mathbb{P}$. For less trivial propositions, we use the recursion principles of inductive type families, which become induction principles if the accumulated value is a proposition $p : \mathbb{P}$.

What distinguishes propositions $p : \mathbb{P}$ from other types is that $\mathbb{P}$ is impredicative and proof-irrelevant, i.e. whenever we quantify over a proposition $p : \mathbb{P}$ s.t. $\forall x.\ p$, the resulting type is again a proposition, and for proofs $h_1, h_2 : p$, we have $h_1 = h_2$. In other words, propositions are contained to $\mathbb{P}$ and all proofs of a proposition are considered equal, i.e. only their existence is relevant, not the concrete content of the proof. Hence, proofs are inherently non-computational; since the content of a proof is irrelevant, it can be erased. Lean also introduces additional non-computational classical axioms into its universe of propositions $\mathbb{P}$, most prominently the law of the excluded middle $p \vee \neg p$.

Combined power

Putting all of these mechanisms together yields a type theory powerful enough to declare types like $\text{Vec}\ \alpha\ n$ and all the usual objects that are used in mathematics, as well as logical operators like $\cdot \wedge \cdot$, $\exists x.\ p(x)$, $\cdot = \cdot$ and even well-founded recursion. Additionally, the term language is strong enough to write arbitrary programs, as well as classical proofs within $\mathbb{P}$.

For a detailed formal description of Lean's type theory, we refer to Carneiro [2019].

2.3 Intermediate Representations for Lean 4

As outlined in Section 2.1, Lean 4 ultimately compiles to either C code or LLVM IR code [Lattner and Adve, 2004], with the latter being work-in-progress. Once compiled, much of the structure of the original Lean 4 program is lost, including types and the global guarantee of purity. Hence, introducing additional intermediate representations between Lean 4 and the resulting compiled program can help with leveraging the additional structure in the original Lean 4 code for a first pass of Lean-specific optimizations.

Pure IR

Since very early on in the development of Lean 4, the Lean compiler has compiled to the intermediate representations described by Ullrich and de Moura [2020], with optimizations like common subexpression elimination [Cocke, 1970], verified rewrite rules [Jones et al., 2001], lambda lifting [Johnsson, 1985], erasure of computationally irrelevant terms [Tejiščák, 2019], specialization [Augustsson, 1993] and inlining [Jones and Marlow, 2002] being performed on Lean 4 expressions before compiling to the IR. Instead, the IR is used to implement reference counting. Programs compile to the untyped IR defined below, commonly called “the pure IR”:

$$
\begin{aligned}
\mathtt{x}, \mathtt{y} &\in \mathrm{Var}_\lambda \\
\mathtt{i} &\in \mathrm{Ctor}_\lambda \\
\mathtt{j} &\in \mathrm{Proj}_\lambda \\
\mathtt{c} &\in \mathrm{Const}_\lambda \\
\mathtt{e} &\in \mathrm{Expr}_\lambda &&::= \mathtt{c}\ \overline{\mathtt{y}} \mid \mathtt{pap}\ \mathtt{c}\ \overline{\mathtt{y}} \mid \mathtt{x}\ \mathtt{y} \mid \mathtt{ctor}_\mathtt{i}\ \overline{\mathtt{y}} \mid \mathtt{proj}_\mathtt{j}\ \mathtt{y} \\
\mathtt{F} &\in \mathrm{FnBody}_\lambda &&::= \mathtt{ret}\ \mathtt{x} \mid \mathtt{let}\ \mathtt{x} := \mathtt{e};\ \mathtt{F} \mid \mathtt{case}\ \mathtt{x}\ \mathtt{of}\ \overline{\mathtt{F}} \\
f &\in \mathrm{Fn}_\lambda &&::= \lambda\ \overline{\mathtt{y}}.\ \mathtt{F} \\
\delta &\in \mathrm{Program}_\lambda &&= \mathrm{Const}_\lambda \rightharpoonup \mathrm{Fn}_\lambda
\end{aligned}
$$

$\overline{\mathtt{y}}$ is a vector of variables $\mathtt{y}$. Expressions are full applications $\mathtt{c}\ \overline{\mathtt{y}}$, partial applications $\mathtt{pap}\ \mathtt{c}\ \overline{\mathtt{y}}$, variable applications $\mathtt{x}\ \mathtt{y}$ of a variable $\mathtt{y}$ to a higher-order function $\mathtt{x}$ created by $\mathtt{pap}$, constructor applications $\mathtt{ctor}_\mathtt{i}\ \overline{\mathtt{y}}$ for a constructor $\mathtt{i}$ or projections $\mathtt{proj}_\mathtt{j}\ \mathtt{y}$ for a field $\mathtt{j}$ in $\mathtt{y}$. Function bodies consist of $\mathtt{ret}\ \mathtt{x}$, $\mathtt{let}\ \mathtt{x} := \mathtt{e};\ \mathtt{F}$ and a case-instruction $\mathtt{case}\ \mathtt{x}\ \mathtt{of}\ \overline{\mathtt{F}}$ that makes $\mathtt{x}$ denote the $\mathtt{i}$-th constructor in $\overline{\mathtt{F}}_\mathtt{i}$. All functions are lifted out to a global map $\delta \in \mathrm{Program}_\lambda$.

Reference-counted IR

The Lean 4 compiler then inserts the following additional impure instructions into code in pure IR form:

$$
\begin{aligned}
\mathtt{e} \in \text{Expr} &::= \ldots \mid \mathtt{reset}\ \mathtt{x} \mid \mathtt{reuse}\ \mathtt{x}\ \mathtt{in}\ \mathtt{ctor}_i\ \overline{\mathtt{y}} \\
\mathtt{F} \in \text{FnBody} &::= \ldots \mid \mathtt{inc}\ \mathtt{x};\ \mathtt{F} \mid \mathtt{dec}\ \mathtt{x};\ \mathtt{F}
\end{aligned}
$$

The resulting IR is also commonly called "the reference-counted IR". We will cover `reset x` and `reuse x in` $\mathtt{ctor}_i\ \overline{\mathtt{y}}$ in detail in Section 2.4. `inc x; F` and `dec x; F` increment and decrement the reference count of the value `x` respectively. When the reference count reaches 0, the value in question is not referenced anymore and can be freed safely. Since Lean 4 is a strict purely functional programming language, there can be no reference cycles.

When compiling the pure IR to the reference-counted IR, the Lean compiler performs additional optimizations: Parameters are sometimes inferred as "borrowed", which means that the caller is keeping these parameters alive, and so they need not be reference-counted in the callee. We will encounter a similar but slightly different notion of "borrowing" in Section 2.8, albeit both have in common that the caller is managing a resource for a callee.

In addition to the instructions described by Ullrich and de Moura [2020], Lean's implementation of this IR also supports operations for boxing and unboxing of scalar values [Henglein and Jørgensen, 1994], as well as join point declarations and corresponding jump instructions that allow the re-joining of control flow branches [Maurer et al., 2017]. In other words, Lean's IR is implemented in A-normal form as described by Maurer et al. [2017], where functions can only take variables as arguments, nested expressions are factored out into let-expressions and join point declarations are generated where possible.

We will henceforth ignore boxing as an implementation detail and briefly discuss join points in Section 4.2 and Section 6.1.

LCNF

While the IRs defined above are well suited for implementing garbage collection and optimizations associated with reference counting, they lack types and hence much of the structure of the original Lean 4 program, which is why the current Lean 4 compiler performs other optimizations directly on Lean expressions.

In the current rewrite of the Lean 4 compiler toolchain that is still work-in-progress, another IR, the Lean Compiler Normal Form (LCNF), is added to mitigate this issue.

It is also stated in A-normal form and defined by the following grammar:

$$
\begin{aligned}
\tau &\in \text{LeanType} \\
\mathtt{D} &\in \text{DeclName}_{\text{LCNF}} \\
\mathtt{x}, \mathtt{y} &\in \text{Var}_{\text{LCNF}} \\
\mathtt{i} &\in \text{Ctor}_{\text{LCNF}} \\
\mathtt{j} &\in \text{Proj}_{\text{LCNF}} \\
\mathtt{a} \in \text{Arg}_{\text{LCNF}} &::= \blacksquare \mid \mathtt{y} \mid \tau \\
\mathtt{v} \in \text{LetValue}_{\text{LCNF}} &::= \blacksquare \mid \mathtt{D}\ \overline{\mathtt{a}} \mid \mathtt{y}\ \overline{\mathtt{a}} \mid \mathtt{D.proj_j}\ \mathtt{y} \\
\mathtt{c} \in \text{Case}_{\text{LCNF}} &::= \mathtt{ctor_i}\ \overline{(\mathtt{y} : \tau_\mathtt{y})} \Rightarrow \mathtt{K} \mid \mathtt{default} \Rightarrow \mathtt{K} \\
\mathtt{K} \in \text{Code}_{\text{LCNF}} &::= \mathtt{let}\ (\mathtt{x} : \tau)\ \mathtt{:=}\ \mathtt{v};\ \mathtt{K} \\
&\mid ((\mathtt{def}\ \mathtt{x}\ \overline{(\mathtt{y} : \tau_\mathtt{y})}\ \mathtt{:=}\ \mathtt{K_x}) : \tau);\ \mathtt{K} \\
&\mid ((\mathtt{jpdef}\ \mathtt{x}\ \overline{(\mathtt{y} : \tau_\mathtt{y})}\ \mathtt{:=}\ \mathtt{K_x}) : \tau);\ \mathtt{K} \\
&\mid \mathtt{jmp}\ \mathtt{x}\ \overline{\mathtt{a}} \\
&\mid ((\mathtt{D.case}\ \mathtt{x}\ \mathtt{of}\ \overline{\mathtt{c}}) : \tau) \\
&\mid \mathtt{ret}\ \mathtt{x} \\
\mathtt{d} \in \text{Decl}_{\text{LCNF}} &::= (\lambda \overline{(\mathtt{y} : \tau_\mathtt{y})}.\ \mathtt{K} : \tau) \\
\delta \in \text{Program}_{\text{LCNF}} &= \text{DeclName}_{\text{LCNF}} \rightharpoonup \text{Decl}_{\text{LCNF}}
\end{aligned}
$$

We have omitted some technical details from Lean's implementation of this grammar that are not relevant to this thesis.

LeanType denotes arbitrary dependent types in the form of generic Lean expressions. Declarations, variables, constructors and projections are all identified by names in their respective syntactical category. Arguments to declarations can either be erased (■), variables or types. Let values can either be erased, an application of a constant declaration $\mathtt{D}\ \overline{\mathtt{a}}$, an application $\mathtt{y}\ \overline{\mathtt{a}}$ of arguments $\overline{\mathtt{a}}$ to a variable `y` that may contain a declaration or a function, or a projection $\mathtt{D.proj_j}\ \mathtt{y}$ of a variable `y` that stores a type with only a single constructor. Cases are either destructuring patterns $\mathtt{ctor_i}\ \overline{(\mathtt{y} : \tau_\mathtt{y})} \Rightarrow \mathtt{K}$ that match against a constructor `i` and denote its fields using variables $\overline{\mathtt{y}}$ with types $\overline{\tau_\mathtt{y}}$, or a `default` case.

Code instructions can either be `let` instructions, function definitions `def` of type τ, join point declarations `jpdef` of type τ, `jmp` instructions that jump to a given join point `x` with a given vector of arguments $\overline{\mathtt{a}}$, `case` or `ret` instructions.

After compiling Lean code to LCNF, the resulting LCNF code is simplified by many of the same optimizations that are applied to Lean code in the current compiler that works directly on Lean expressions. Near the end of the optimization phase, type dependencies, computationally irrelevant terms and a few other types are heuristically erased and the resulting LCNF code can be converted to the pure IR. In the future, both the pure and the reference-counted IR will be replaced by a closely related LCNF-equivalent as well.

2.4 Destructive Updates

In Section 2.3, we saw that the Lean 4 compiler inserts instructions `let x := reset y` and `let z := reuse x in ctor`$_i$ $\overline{y}$ when compiling the pure IR to the reference-counted IR. These instructions are used to perform so-called destructive updates: In a purely functional language, if a value is only referenced by a single function, then that function can safely update the value in-place without the mutation being observed by other functions, thus retaining both purity and efficiency. It is obvious that if we are using reference counting for garbage collection, then we can use the reference count for performing destructive updates by checking that the reference count of the value is equal to 1.

Reset and reuse

One important question remains: In a purely functional language, what constitutes an update? After all, there is no dedicated instruction that performs updates. To answer this question, Ullrich and de Moura [2020] introduce what they call "the resurrection hypothesis": many values are used for the last time just before a value of the same kind is created. This should be intuitive to most functional programmers, as functions that would otherwise mutate a value in an imperative language will instead create a new instance of the old value with the change applied to it in purely functional languages.

As per Ullrich and de Moura [2020], the `let x := reset y` instruction is inserted as early as possible when `y` is not used anymore for the remainder of the function and a corresponding `let z := ctor`$_i$ $\overline{y}$ call of the same kind as `y` exists later on in the same function. `let z := reuse x in ctor`$_i$ $\overline{y}$ is used to replace the latter `let z := ctor`$_i$ $\overline{y}$ call. Here, "of the same kind" means that both constructors occupy the same amount of memory.

Semantically, `let x := reset y` checks the reference count of `y` and denotes in `x` both `y` itself and whether `y` had a reference count equal to 1. Additionally, if the reference count is equal to 1, `reset y` detaches the components of the soon-to-be-freed `y` pre-emptively by decrementing the reference counts of the components of `y`. `reuse x in ctor`$_i$ $\overline{y}$ then checks `x` and either updates the referenced memory in-place if the reference count was equal to 1 at the call to `reset` or allocates new memory according to `ctor`$_i$ $\overline{y}$.

The fact that there are two separate instructions for checking the reference count, detaching the components and performing the in-place update is crucial, as no destructive updates can be performed on the components of a value as long as the value itself is still alive. This is evidenced by the following example taken from Ullrich and de Moura [2020]:

```
map f xs = case xs of
  (ret xs)
  (let x = proj₁ xs;
   inc x;
   let s = proj₂ xs;
   inc s;
   let w = reset xs;
   let y = f x;
   let ys = map f s;
   let r = reuse w in ctor₂ y ys;
   ret r)
```

If `reuse` was to check the reference count of `xs`, then `s` would have a reference count of 2: one reference from `s` itself, and one from `xs`, thus preventing further destructive updates in the recursive call to `map`.

Updating values within values

The optimization above usually works if the functional code in question is written in a manner that is similar to clean imperative code, where old values are not used after updating them and references are not duplicated in such a manner that a mutation is observable in an entirely different part of the program. In fact, since this optimization also applies to code written with functional combinators like `map` or `filter`, even code that would otherwise be inefficient in imperative languages because explicit copies are created to maintain local purity can be optimized to be as fast as if it was written in an imperative manner. There are however some common exceptions to this rule, one of which we have already seen in detail in Section 2.1:

```
def Array.groupBy (p : α → α → Ordering) (xs : Array α)
  : RBMap α (Array α) p := Id.run do
  let mut result : RBMap α (Array α) p := RBMap.empty
  for x in xs do
    let group := Option.getD (RBMap.find? p result x) #[]
    result := RBMap.insert p result x (Array.push group x)
  return result
```

Here, after executing the line containing `RBMap.find? p result x`, there can be two references to the array in `group`: one from `group` itself, and one within the red-black tree based map `result`. This precludes Lean from performing a destructive update in `Array.push group x`, resulting in an expensive $\Theta(n)$-complexity copy of the whole array. Since this issue occurs every single loop iteration, our `Array.groupBy` implementation is accidentally quadratic. Our imperative intuition lead us astray: If `Array.push group x` was to update `group` in-place, then this mutation would be observable in `result`, but since we never observe it and immediately replace the reference to `group` in `result` using `RBMap.insert p result x ...`, we consider the mutation safe.

The common solution to this issue is to introduce a function similar to the following for any type that can act as a container:

```
def RBMap.update (p : α → α → Ordering) (map : RBMap α β p)
  (a : α) (f : β → β) : RBMap α β p :=
  let x := RBMap.find? p map a
  match x with
  | Option.none   => map
  | Option.some b =>
    let map := RBMap.erase p map a
    RBMap.insert p map a (f b)
```

`match x with` uses pattern matching on `x`. After calling `RBMap.find? p map a`, there are two references to the value in `b`. Using `RBMap.erase p map a`, the second reference is erased and `b` can now be updated destructively in `f`. Afterwards, the result is re-inserted. Using this function, we can now fix the performance bug in `Array.groupBy` with the following implementation:

```
def Array.groupBy (p : α → α → Ordering) (xs : Array α)
  : RBMap α (Array α) p := Id.run do
  let mut result : RBMap α (Array α) p := RBMap.empty
  for x in xs do
    if ! RBMap.contains p result x then
      result := RBMap.insert p result x #[]
    else
      result := RBMap.update p result x
        (fun group => Array.push group x)
  return result
```

`(fun x => ...)` denotes a lambda expression. As we are using `RBMap.update`, the `group` corresponding to `x` is updated in-place, and now our implementation has the time complexity that we would expect it to have.

Situations akin to this one can occur for other types as well. We will discuss similar challenges related to containers that occur when statically ensuring referential uniqueness in Section 2.8.

2.5 Linear Type Theory

Substructural rules

Type systems typically guarantee certain functional or extensional properties for a given program e using a typing relation $\Gamma \vdash e : \tau$, where Γ is a set of type judgements $x : \tau'$. The following Exchange, Weaken and Contract rules are usually

assumed implicitly:

$$\textsc{Exchange}\ \frac{\Gamma_1, y : \tau_2, x : \tau_1, \Gamma_2 \vdash e : \tau}{\Gamma_1, x : \tau_1, y : \tau_2, \Gamma_2 \vdash e : \tau} \qquad \textsc{Weaken}\ \frac{\Gamma \vdash e : \tau}{\Gamma, x : \tau' \vdash e : \tau} \qquad \textsc{Contract}\ \frac{\Gamma, x : \tau', x : \tau' \vdash e : \tau}{\Gamma, x : \tau' \vdash e : \tau}$$

In other words, judgements in the context can be reordered, discarded and duplicated freely.

In the formal descriptions of dependent type theories, like the one described in Section 2.2, Exchange is inhibited, because τ_2 may depend on x, which induces an order of declaration on variables in the context. However, if we want to guarantee only extensional properties for e, Weaken and Contract can always be freely assumed, as there is nothing to gain from retaining the exact count of every judgement in the context.

However, if we wish to guarantee non-functional or intensional properties for a given program, then discarding the Weaken and Contract rules can be useful. And indeed doing just that constitutes the core idea of substructural type theories: By retaining the exact amount of each judgement in the context, we can use typing rules to count objects in our program to ensure various kinds of intensional properties.

Beginnings of linear type theory

In most substructural type theories, the extra detail in the context is used to count variable uses. Girard [1987] was the first to notice that not assuming Weaken and Contract allows one to define so-called linear logics with dualities that allow reasoning about resource usage, and Wadler [1990] transports this idea to form a linear type theory with a set of entirely separate linear and non-linear types, even on the term level. Both Girard [1987] and Wadler [1990] use the $\multimap$ operator to denote the type of linear functions or implications that take exactly one instance of an argument and produce exactly one instance of a return value. See Figure 2.1 for Wadler's original linear type system.

Properly linear or invariably unique

Wadler [1991] then goes on to define the Dereliction and Promotion rules listed below that allow coercing a non-linear type to a linear type and promoting a linear type to a nonlinear one if it only depends on nonlinear types, as well as a "steadfast" type system where there is no coercion between linear and non-linear types, but both use the same term language. Especially in the former, non-linear types can be understood as having an unlimited quantity of something, whereas linear types can be understood as having exactly one of something.

$$\textsc{Dereliction}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma, x : !\tau_1 \vdash e : \tau_2} \qquad \textsc{Promotion}\ \frac{!\Gamma \vdash e : \tau}{!\Gamma \vdash e : !\tau}$$

Here, $!\tau$ is a nonlinear type and $!\Gamma$ refers to every type in the context Γ being non-linear. The motivation for including Dereliction is that ! can be understood as having an unlimited quantity of something, and if one has an unlimited quantity of something, one also has one of something. Promotion is justified because any single resource that can be created from resources of unlimited quantity can also be created in unlimited quantity by repeating the process.

With Dereliction and Promotion, linear types make a guarantee for the future: As it could have been coerced from non-linear type, we do not know whether this variable has always been linear, but now that it is, we guarantee that it will be used exactly once. Meanwhile, in Wadler's "steadfast" version of the type system, as there is no coercion from non-linear to linear types, a linear variable has and will always be used exactly once.

In a type system with Dereliction, linearity is unsuited to guarantee the referential uniqueness of a variable, as it may have been duplicated in the past, an issue that is acknowledged by Wadler [1991]. For linear type systems where linearity is guaranteed from construction onwards, Chirimar et al. [1996] prove that the linearity of a variable implies the uniqueness of the associated reference.

Future linear type systems [Odersky, 1992] [Atkey, 2018] [Bernardy et al., 2018] [Brady, 2021] [Choudhury et al., 2021] [Li et al., 2022] [Spiwack et al., 2022] always adopt one of these two approaches and either allow for a non-linear to linear coercion, or no coercion at all, thus cementing the idea of "linearity" referring to either "always uniquely referenced" or "used linearly from this point onwards". The latter can be made to act like the former by guaranteeing referential uniqueness through other means, e.g. by making every constructor of a certain type return a value of linear type, effectively turning referential uniqueness into a library design decision.

To make the distinction between the two kinds of linear type systems clear, we will henceforth refer to linear type systems without any coercion between linear and non-linear values as "invariably unique" and type systems adopting Dereliction and Promotion as "properly linear", but still use the term "linear" to group the two of them together, as is often done in the literature since Wadler's first papers. See Figure 2.2 for Wadler's properly linear type system and Figure 2.3 for his invariably unique type system.

Affinity

One common refinement of linear type theory is to allow the use of Weaken, but not Contract, specifically when the type theory only wants to guarantee that a reference is unique, but not that it is used. These type theories are also knows as "affine" [Tov and Pucella, 2011], though the term is often conflated with "linear".

Applications

The various applications of linear type theory include the following:

- Threading a functional program and enforcing an execution order to replace the use of monads for I/O in functional languages with a linear equivalent [de Vries, 2009] [Bernardy et al., 2018] [Brady, 2021]
- Ensuring resource- and memory-safety so that resources and memory cannot be freed multiple times [Weiss et al., 2021]
- Performing efficient in-place updates and enabling the use of arrays in functional languages [de Vries, 2009] [Bernardy et al., 2018]
- Specifying usage protocols for types [Brady, 2021]
- Guiding program synthesis [Brady, 2021]
- Inverting the computation of functions [Matsuda and Wang, 2020]

$$\frac{}{x : \tau \vdash x : \tau}\ \textsc{Var} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (1\ \lambda x : \tau_1.\ e) : \tau_1 \multimap \tau_2}\ {\multimap}\textsc{-Intro} \qquad \frac{\Gamma_1 \vdash e_1 : \tau_1 \multimap \tau_2 \qquad \Gamma_2 \vdash e_2 : \tau_1}{\Gamma_1, \Gamma_2 \vdash (1\ e_1\ e_2) : \tau_2}\ {\multimap}\textsc{-Elim}$$

$$\frac{\Gamma \vdash e : \tau \qquad \tau' \text{ nonlinear}}{\Gamma, x : \tau' \vdash e : \tau}\ \textsc{Weaken} \qquad \frac{\Gamma, x : \tau', x : \tau' \vdash e : \tau \qquad \tau' \text{ nonlinear}}{\Gamma, x : \tau' \vdash e : \tau}\ \textsc{Contract}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \qquad \Gamma \text{ nonlinear}}{\Gamma \vdash (!\ \lambda x : \tau_1.\ e) : \tau_1 \rightarrow \tau_2}\ {\rightarrow}\textsc{-Intro} \qquad \frac{\Gamma_1 \vdash e_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma_2 \vdash e_2 : \tau_1}{\Gamma_1, \Gamma_2 \vdash (!\ e_1\ e_2) : \tau_2}\ {\rightarrow}\textsc{-Elim}$$

Figure 2.1: Original linear type system from Wadler [1990] with separate linear and non-linear terms and types. 1 denotes linear terms, ! denotes non-linear terms. "nonlinear" demands that the full type or context consists only of non-linear →-types.

$$\frac{\Gamma \vdash e : \tau}{\Gamma, x : !\tau' \vdash e : \tau}\ \textsc{Weaken} \qquad \frac{\Gamma, x : !\tau', x : !\tau' \vdash e : \tau}{\Gamma, x : !\tau' \vdash e : \tau}\ \textsc{Contract} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma, x : !\tau_1 \vdash e : \tau_2}\ \textsc{Dereliction} \qquad \frac{!\Gamma \vdash e : \tau}{!\Gamma \vdash e : !\tau}\ \textsc{Promotion}$$

$$\frac{}{x : \tau \vdash x : \tau}\ \textsc{Var} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1.\ e : \tau_1 \multimap \tau_2}\ {\multimap}\textsc{-Intro} \qquad \frac{\Gamma_1 \vdash e_1 : \tau_1 \multimap \tau_2 \qquad \Gamma_2 \vdash e_2 : \tau_1}{\Gamma_1, \Gamma_2 \vdash e_1\ e_2 : \tau_2}\ {\multimap}\textsc{-Elim}$$

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2 \vdash e_2 : \tau_2}{\Gamma_1, \Gamma_2 \vdash (e_1, e_2) : \tau_1 \otimes \tau_2}\ {\otimes}\textsc{-Intro} \qquad \frac{\Gamma_1 \vdash e_1 : \tau_1 \otimes \tau_2 \qquad \Gamma_2, x : \tau_1, y : \tau_2 \vdash e_2 : \tau_3}{\Gamma_1, \Gamma_2 \vdash \texttt{case}\ e_1\ \texttt{of}\ (x, y) \Rightarrow e_2 : \tau_3}\ {\otimes}\textsc{-Elim}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \texttt{left}\ e : \tau_1 \oplus \tau_2}\ {\oplus}\textsc{-Intro-Left} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \texttt{right}\ e : \tau_1 \oplus \tau_2}\ {\oplus}\textsc{-Intro-Right}$$

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \oplus \tau_2 \qquad \Gamma_2, x : \tau_1 \vdash e_2 : \tau_3 \qquad \Gamma_2, y : \tau_2 \vdash e_3 : \tau_3}{\Gamma_1, \Gamma_2 \vdash \texttt{case}\ e_1\ \texttt{of left}\ x \Rightarrow e_2;\ \texttt{right}\ y \Rightarrow e_3 : \tau_3}\ {\oplus}\textsc{-Elim}$$

Figure 2.2: Properly linear type system from Wadler [1991] with support for multiplicative product types (⊗) and multiplicative sum types (⊕).

$$\textsc{Weaken}\ \frac{\Gamma \vdash e : \tau}{\Gamma, x : !\tau' \vdash e : \tau} \qquad \textsc{Contract}\ \frac{\Gamma, x : !\tau', x : !\tau' \vdash e : \tau}{\Gamma, x : !\tau' \vdash e : \tau}$$

$$\textsc{Var}\ \frac{}{x : \tau \vdash x : \tau} \qquad \multimap\text{-}\textsc{Intro}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1.\ e : \tau_1 \multimap \tau_2} \qquad \multimap\text{-}\textsc{Elim}\ \frac{\Gamma_1 \vdash e_1 : \tau_1 \multimap \tau_2 \qquad \Gamma_2 \vdash e_2 : \tau_1}{\Gamma_1, \Gamma_2 \vdash e_1\ e_2 : \tau_2}$$

$$\otimes\text{-}\textsc{Intro}\ \frac{\Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2 \vdash e_2 : \tau_2}{\Gamma_1, \Gamma_2 \vdash (e_1, e_2) : \tau_1 \otimes \tau_2} \qquad \otimes\text{-}\textsc{Elim}\ \frac{\Gamma_1 \vdash e_1 : \tau_1 \otimes \tau_2 \qquad \Gamma_2, x : \tau_1, y : \tau_2 \vdash e_2 : \tau_3}{\Gamma_1, \Gamma_2 \vdash \texttt{case}\ e_1\ \texttt{of}\ (x, y) \Rightarrow e_2 : \tau_3}$$

$$\oplus\text{-}\textsc{Intro-Left}\ \frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \texttt{left}\ e : \tau_1 \oplus \tau_2} \qquad \oplus\text{-}\textsc{Intro-Right}\ \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \texttt{right}\ e : \tau_1 \oplus \tau_2}$$

$$\oplus\text{-}\textsc{Elim}\ \frac{\Gamma_1 \vdash e_1 : \tau_1 \oplus \tau_2 \qquad \Gamma_2, x : \tau_1 \vdash e_2 : \tau_3 \qquad \Gamma_2, y : \tau_2 \vdash e_3 : \tau_3}{\Gamma_1, \Gamma_2 \vdash \texttt{case}\ e_1\ \texttt{of left}\ x \Rightarrow e_2;\ \texttt{right}\ y \Rightarrow e_3 : \tau_3}$$

$$!\text{-}\multimap\text{-}\textsc{Intro}\ \frac{!\Gamma, x : \tau_1 \vdash e : \tau_2}{!\Gamma \vdash \lambda x : \tau_1.\ e : !(\tau_1 \multimap \tau_2)} \qquad !\text{-}\multimap\text{-}\textsc{Elim}\ \frac{\Gamma_1 \vdash e_1 : !(\tau_1 \multimap \tau_2) \qquad \Gamma_2 \vdash e_2 : \tau_1}{\Gamma_1, \Gamma_2 \vdash e_1\ e_2 : \tau_2}$$

$$!\text{-}\otimes\text{-}\textsc{Intro}\ \frac{\Gamma_1 \vdash e_1 : !\tau_1 \qquad \Gamma_2 \vdash e_2 : !\tau_2}{\Gamma_1, \Gamma_2 \vdash (e_1, e_2) : !(!\tau_1 \otimes !\tau_2)} \qquad !\text{-}\otimes\text{-}\textsc{Elim}\ \frac{\Gamma_1 \vdash e_1 : !(!\tau_1 \otimes !\tau_2) \qquad \Gamma_2, x : !\tau_1, y : !\tau_2 \vdash e_2 : \tau_3}{\Gamma_1, \Gamma_2 \vdash \texttt{case}\ e_1\ \texttt{of}\ (x, y) \Rightarrow e_2 : \tau_3}$$

$$!\text{-}\oplus\text{-}\textsc{Intro-Left}\ \frac{\Gamma \vdash e : !\tau_1}{\Gamma \vdash \texttt{left}\ e : !(!\tau_1 \oplus !\tau_2)} \qquad !\text{-}\oplus\text{-}\textsc{Intro-Right}\ \frac{\Gamma \vdash e : !\tau_2}{\Gamma \vdash \texttt{right}\ e : !(!\tau_1 \oplus !\tau_2)}$$

$$!\text{-}\oplus\text{-}\textsc{Elim}\ \frac{\Gamma_1 \vdash e_1 : !(!\tau_1 \oplus !\tau_2) \qquad \Gamma_2, x : !\tau_1 \vdash e_2 : \tau_3 \qquad \Gamma_2, y : !\tau_2 \vdash e_3 : \tau_3}{\Gamma_1, \Gamma_2 \vdash \texttt{case}\ e_1\ \texttt{of left}\ x \Rightarrow e_2;\ \texttt{right}\ y \Rightarrow e_3 : \tau_3}$$

Figure 2.3: Invariably unique type system from Wadler [1991].

2.6 Quantitative Type Theory

Quantitative type theory (QTT) applies the idea of properly linear type theory to dependent type theory.

Context-distribution problem

In linear type theories, the App rule is typically stated in a manner similar to the following in order to distribute the needed amount of judgements to both expressions:

$$\textsc{App}\quad \frac{\Gamma_1 \vdash e_1 : \tau_1 \multimap \tau_2 \qquad \Gamma_2 \vdash e_2 : \tau_1}{\Gamma_1, \Gamma_2 \vdash e_1\ e_2 : \tau_2}$$

When omitting the Exchange rule in dependent type theory, it is not clear that splitting the context into Γ_1 and Γ_2 is always possible, since types in Γ_2 may depend on variables in Γ_1. This seemingly technical issue induces a semantic problem as well: Which occurrences of a variable constitute a use? For example, what about occurrences in types?

Erasure

After this issue was left unsolved for a long time, McBride [2016] resolved it by introducing a third kind of type to linear type theory, resulting in the three kinds of type “linear” (denoted as 1), “non-linear” (denoted as ω) and “erased” (denoted as 0). “Erased” specifies that a type or a term within DTT is not computationally relevant and will be erased by the compiler.

Erased terms or types can only be used in other erased terms or types, and types are always erased. Since an erased term is not computationally relevant, we also do not need to count variable uses in erased terms. Finally, when a linear variable is used, it becomes erased, which justifies the use of 1 to denote linear types and 0 to denote erased types.

Exchange is still omitted, but variables in the context that act as dependencies for other variables in the context are always guaranteed to be erased. In App, since we do not need to count the uses of variables of type 0, we can freely distribute variables of type 1 and ω to either Γ_1 or Γ_2, but duplicate all variables of type 0 in their given order to Γ_1 and Γ_2. In other words, both the technical and the semantic issue are resolved at once.

Quantities instead of types

The implementations of QTT by McBride [2016] and Atkey [2018] use an additional trick to make the description of the type theory more compact.

While linear type theories usually denote linearity on the type τ, quantitative type theory instead denotes it on the binder “:” in $x : \tau$, which leads to 0, 1 and ω not

simply being kinds of types, but "quantities" on the binders $x :_q \tau$ for $q \in \{0, 1, \omega\}$ in the context. There are only linear types, and the quantity on the binder determines the substructural restrictions on the variable.

The typing judgement $\Gamma \vdash e :_\sigma \tau$ is quantified as well to track whether e is being checked in a computationally relevant or irrelevant context, and σ is restricted to $\{0, 1\}$ to ensure admissibility of substitution [Atkey, 2018].

With these tools, $\tau_1 \multimap \tau_2$ is encoded as $(x :_1 \tau_1) \to \tau_2$ and $\tau_1 \otimes \tau_2$ from Figure 2.2 in Section 2.5 is encoded as $(x :_1 \tau_1) \otimes (y :_1 \tau_2) \otimes \mathbb{1}$. With a dependent type-level if-then-else construct, we can also encode $\tau_1 \oplus \tau_2 := (b :_1 \mathbb{B}) \otimes (\text{if } b \text{ then } \tau_1 \text{ else } \tau_2)$ [Grenrus, 2020].

Both McBride [2016] and Atkey [2018] specify a generic framework for adding additional quantities to the type theory, allowing, for example, the additional introduction of a ≤ 1 quantity representing affinity, or more accurate accounting of uses $n > 1$.

The typing rules for quantitative type theory as described by Svoboda [2021] and inspired by Atkey [2018] can be found in figure 2.4.

Demand-based consumption

Finally, note that by denoting the linear quantity on the binder in $(x :_1 \tau_1) \to \tau_2$, we cannot specify a quantity for τ_2 anymore, as we could in linear type theory by using either a linear or non-linear type for the return type. Instead, when checking an application $f\ (g\ e) :_1 \tau_3$ for $f :_1 (x :_q \tau_2) \to \tau_3$, $g :_1 (x :_1 \tau_1) \to \tau_2$ and $e :_1 \tau_1$, we demand q instances of the resources Γ required to check $g\ e :_1 \tau_2$.

In other words, if we need q instances of a return value that requires Γ resources to produce, we instead demand $q \cdot \Gamma$ resources in our context, pretending that we applied the function q times to obtain q instances of the return value. This way, quantities need not be specified on return values, and the resources for the arguments are consumed based on the required amount of the return value. The Promotion rule from section 2.5 is integrated into the rules for other types.

This trick is not inherent to quantitative types and can be applied to linear type systems of any kind by taking $f : \tau_1 \multimap \tau_2$ to mean "f consumes one of τ_1 for every instance of τ_2 that is required", e.g. as in Linear Haskell [Bernardy et al., 2018] or in Ghica and Smith [2014].

However, it is not entirely for free: If one wishes to both use a linear or quantitative type system with Dereliction, as well as guarantee referential uniqueness, then the lack of an annotation on the return type of functions means that we cannot simply annotate every constructor to return a linear type in order to ensure referential uniqueness, as described in section 2.5. Instead, both quantitative and linear type theories that use this trick and want to guarantee referential uniqueness define constructors in a continuation passing manner; e.g. $\text{mkArray} : \mathbb{N} \multimap \alpha \multimap \text{Array } \alpha$ becomes $\text{mkArray} : \mathbb{N} \multimap \alpha \multimap (\text{Array } \alpha \multimap\ !\tau) \multimap\ !\tau$ [Bernardy et al., 2018].

Applications

Quantitative type theory combines the benefits of linear type theory and dependent type theory, leading to far greater capabilities when specifying protocols for types [Brady, 2021]. The introduction of an erasure quantity to combine the two type theories also allows for finer-grained specification of terms that are not computationally relevant but would otherwise be expensive to compute [Brady, 2021].

$$\boxed{\Gamma \vdash} \qquad \textsc{Empty}\ \frac{}{\varnothing \vdash} \qquad \textsc{Extend}\ \frac{\Gamma \vdash \qquad 0\Gamma \vdash S :_0 \mathcal{U}}{\Gamma, x :_q S \vdash}$$

$$\boxed{\Gamma \vdash M :_\sigma S} \qquad \textsc{Var}\ \frac{0\Gamma_1, x :_\sigma S, 0\Gamma_2 \vdash}{0\Gamma_1, x :_\sigma S, 0\Gamma_2 \vdash x :_\sigma S} \qquad \textsc{Universe}\ \frac{0\Gamma \vdash}{0\Gamma \vdash \mathcal{U} :_0 \mathcal{U}}$$

$$\textsc{Conversion}\ \frac{\Gamma \vdash M :_\sigma S \qquad 0\Gamma \vdash S :_0 \mathcal{U} \qquad 0\Gamma \vdash T :_0 \mathcal{U} \qquad S \rightsquigarrow U \leftsquigarrow T}{\Gamma \vdash M :_\sigma T}$$

$$\to\text{-}\textsc{Formation}\ \frac{0\Gamma \vdash S :_0 \mathcal{U} \qquad 0\Gamma, x :_0 S \vdash T :_0 \mathcal{U}}{0\Gamma \vdash (x :_q S) \to T :_0 \mathcal{U}} \qquad \to\text{-}\textsc{Intro}\ \frac{\Gamma, x :_{\sigma q} S \vdash M :_\sigma T}{\Gamma \vdash (\lambda x :_q S.\ M) :_\sigma (x :_q S) \to T}$$

$$\to\text{-}\textsc{Elim}_1\ \frac{\Gamma_1 \vdash M :_\sigma (x :_q S) \to T \qquad \Gamma_2 \vdash N :_1 S}{\Gamma_1 + \sigma q \Gamma_2 \vdash M\ N :_\sigma T} \qquad \to\text{-}\textsc{Elim}_0\ \frac{\Gamma \vdash M :_\sigma (x :_q S) \to T \qquad \sigma q = 0 \qquad 0\Gamma \vdash N :_0 S}{\Gamma \vdash M\ N :_\sigma T}$$

$$\otimes\text{-}\textsc{Formation}\ \frac{0\Gamma \vdash S :_0 \mathcal{U} \qquad 0\Gamma, x :_0 S \vdash T :_0 \mathcal{U}}{0\Gamma \vdash (x :_q S) \otimes T :_0 \mathcal{U}} \qquad \otimes\text{-}\textsc{Intro}_1\ \frac{\Gamma_1 \vdash M :_1 S \qquad \Gamma_2 \vdash N :_\sigma T[M/x]}{\sigma q \Gamma_1 + \Gamma_2 \vdash (M, N) :_\sigma (x :_q S) \otimes T}$$

$$\otimes\text{-}\textsc{Intro}_0\ \frac{\sigma q = 0 \qquad 0\Gamma \vdash M :_0 S \qquad \Gamma \vdash N :_\sigma T[M/x]}{\Gamma \vdash (M, N) :_\sigma (x :_q S) \otimes T}$$

$$\otimes\text{-}\textsc{Elim}\ \frac{\Gamma_1 \vdash M :_\sigma (x :_q S) \otimes T \qquad 0\Gamma_1, z :_0 (x :_q S) \otimes T \vdash U :_0 \mathcal{U} \qquad \Gamma_2, x :_{\sigma q} S, y :_\sigma T \vdash N :_\sigma U[(x, y)/z]}{\Gamma_1 + \Gamma_2 \vdash \texttt{let}_{(x :_q S) \otimes T}\ z@(x, y) = M\ \texttt{in}_U\ N :_\sigma U[M/z]}$$

$$\mathbb{1}\text{-}\textsc{Formation}\ \frac{0\Gamma \vdash}{0\Gamma \vdash \mathbb{1} :_0 \mathcal{U}} \qquad \mathbb{1}\text{-}\textsc{Intro}\ \frac{0\Gamma \vdash}{0\Gamma \vdash () :_\sigma \mathbb{1}} \qquad \mathbb{1}\text{-}\textsc{Elim}\ \frac{\Gamma_1 \vdash M :_\sigma \mathbb{1} \qquad 0\Gamma_1, x :_0 \mathbb{1} \vdash S :_0 \mathcal{U} \qquad \Gamma_2 \vdash N :_\sigma S[()/x]}{\Gamma_1 + \Gamma_2 \vdash \texttt{let}_{\mathbb{1}}\ x@() = M\ \texttt{in}_S\ N :_\sigma S[M/x]}$$

Figure 2.4: Quantitative type theory as described by Svoboda [2021] and inspired by Atkey [2018]. Features a single type universe $\mathcal{U}$. $\rightsquigarrow$ denotes reduction. Quantities $q \in \{0, 1, \omega\}$ satisfy $p + q = q + p$, $0 + q = q$, $\omega + q = \omega$, $1 + 1 = \omega$, $pq = qp$, $0q = 0$, $1q = q$ and $\omega\omega = \omega$.

2.7 Uniqueness Type Theory

Smetsers et al. [1994] and Barendsen and Smetsers [1996] introduce a type system for the Clean programming language with the goal of guaranteeing referential uniqueness to enable many of the applications described in section 2.5. The core idea is to use the linear and non-linear types of linear type systems, but instead of keeping them entirely separate or allowing for a coercion from non-linear types to linear types, the coercion is inverted, allowing for the conversion of linear types to non-linear types.

The motivation for this idea is that both properly linear type systems that guarantee referential uniqueness through careful library design and invariably unique type systems are too restrictive: For some use-cases of referential uniqueness, like destructive updates, it is perfectly acceptable to discard the uniqueness guarantee at some point, because the lack of a static uniqueness guarantee can still be mitigated through other means, as for example in section 2.4, or because it is simply not needed for the remainder of the program.

As long as there is a coercion from linear types to non-linear types, uniqueness type systems refer to linear types as "unique" (tagged with $*$ in the type system) and non-linear types as "non-unique" or "shared" (tagged with ! in the type system). The temporal guarantee becomes inverted: While linear types ensure that a variable is always used exactly once in the future, uniqueness types ensure that a variable has always been used exactly once in the past [de Vries, 2009] [Marshall et al., 2022].

$$\textsc{Shallow-Cast-Left}\ \frac{\Gamma, x : !\tau \vdash e : \tau'}{\Gamma, x : *\tau \vdash e : \tau'} \qquad \textsc{Shallow-Cast-Right}\ \frac{\Gamma \vdash e : *\tau}{\Gamma \vdash e : !\tau}$$

Uniqueness types by library design

Implementations like the one in Linear Haskell [Bernardy et al., 2018] acknowledge the over-restrictiveness of linear type theory and make the coercion from linear to non-linear types a part of their library design as well: If MArray represents an array that is always guaranteed to be linear because its constructor has type $\text{newMArray} : !\mathbb{N} \multimap (\text{MArray}\ \alpha \multimap !\beta) \multimap !\beta$, then the coercion is $\text{freeze} : \text{MArray}\ \alpha \multimap !(\text{Array}\ \alpha)$.

Note that since this coercion requires switching out the entire type of the array, coercing nested arrays in constant time becomes problematic. A recent approach by Spiwack et al. [2022] attempts to mitigate this issue by introducing a system of linear capabilities on top of the linear type system, so that read-only and read+write capabilities are managed and passed around implicitly, freeze turns a read+write MArray into a read-only MArray and reading from a read-only MArray again yields read-only values. The capability-based freeze idea is described in Spiwack [2023]. We will go into some more detail on this approach in section 2.8.

Lambda calculus and uniqueness types

Unfortunately, the original type system of Clean was formulated in terms of graph rewriting and not lambda calculus, which meant that advances by the rest of the type theory research community were difficult to transfer over to uniqueness typing, a deficit that was only resolved much later by de Vries [2009]. In his thesis, de Vries first provides a uniqueness type system based on lambda calculus resembling that of Clean and then iteratively refines it with the goal of introducing higher-rank polymorphism [Jones et al., 2007].

Challenges

In uniqueness type systems, there are a number of challenges that do not appear in linear type systems.

Unique types within shared types The first is that types are inherently less composable. In properly linear type theory, non-linear and linear types can be mixed freely, as long as the provided resources are present to create them. For example, $!(\tau_1 \otimes \tau_2)$ is just a non-linear product, whereas $(!\tau_1) \otimes \tau_2$ is a linear product where the first type is non-linear.

If, on the other hand, the type system is invariably unique and τ_1 is linear, then the former example $!(\tau_1 \otimes \tau_2)$ is malformed, as deconstructing the product and obtaining the linear τ_1 will not actually yield us a guarantee that the corresponding value has not been shared in the past, as e.g. the product could have been shared.

Hence, we must enforce that non-linear or non-unique types cannot contain linear or unique types. In invariably unique type systems, e.g. Wadler's steadfast linear types from Wadler [1991], the answer is usually to enforce this invariant when constructing a value. For example, constructing $!(\tau_1 \otimes \tau_2)$ becomes possible only when both τ_1 and τ_2 are non-linear.

With uniqueness types, the situation is unfortunately more complicated: If it is possible to discard the uniqueness of a value, then the invariant that non-unique types cannot contain unique types does not just need to be enforced at construction, but after discarding the uniqueness of a value as well.

Clean in particular resolves the issue by enforcing the invariant both at construction and at deconstruction; if the type of a value is malformed, then deconstructing it is not allowed. Note that an alternative solution would be that deconstructing it is allowed, but the contained values will again be non-unique.

Higher-order functions Second, both in uniqueness type systems and invariably unique type systems, there is a question of what to do about function closures. When forming $\lambda x.\ e : !(\tau_1 \multimap \tau_2)$, the closure of the function is data that is dragged around by the resulting function. As such, the same considerations as for types $\tau_1 \otimes \tau_2$ apply, i.e. that values of non-linear type (the function) cannot be allowed to contain values

of linear types (anything implicitly contained in the function closure), lest we could duplicate the function value and with that its closure.

In an invariably unique type system without Promotion and Dereliction, this is somewhat easy to resolve: For example, Wadler's steadfast types require that every type in the closure of a function must be non-linear when forming the abstraction.

In a uniqueness type system, this situation is again much more complicated because unique functions can lose their uniqueness. Unlike types $\tau_1 \otimes \tau_2$, we cannot simply resolve it by checking whether a non-unique function contains a unique type in its closure after the uniqueness guarantee has been discarded, as the uniqueness of the elements in the closure is not part of the function type. Resolving this is a difficult challenge and we will delay the discussion of possible solutions to chapter 3.

Uniqueness is not a quantity Finally, it is worth pointing out that uniqueness types cannot simply be added as a quantity to existing quantitative type theories.

The trick described at the end of section 2.6 that allows for specifying whether a value is linear or non-linear only on binders rests on the fact that there is a coercion from non-linear to linear values, so that a function $f : \tau_1 \multimap !\tau_2$ can be coerced to $f : \tau_1 \multimap \tau_2$, after which the amount of required τ_1 is scaled up by the amount of τ_2 resources needed. In other words, uniqueness types are not "quantitative" in the sense that the intuitions for reasoning about amounts of resources do not apply to them.

Instead, one approach to combine uniqueness types with dependent type theory is Graded Modal Dependent Type Theory (GRTT) [Moon et al., 2021], a dependent type theory with a linearly typed substructural base that allows attaching generic modalities to types. Marshall et al. [2022] integrate uniqueness types into Granule [Orchard et al., 2019], a non-dependently-typed precursor to GRTT. Unfortunately, their type system does not allow the use of unique types within other unique types, only within linear types. The uniqueness fragment of the type system can be found in figure 2.5.

$$\textsc{Var}\ \frac{}{!\Gamma, x : \tau \vdash x : \tau} \qquad \multimap\textsc{-Intro}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \multimap \tau_2} \qquad \multimap\textsc{-Elim}\ \frac{\Gamma_1, e_1 : \tau_1 \multimap \tau_2 \qquad \Gamma_2 \vdash e_2 : \tau_1}{\Gamma_1, \Gamma_2 \vdash e_1\ e_2 : \tau_2}$$

$$\textsc{Contract}\ \frac{\Gamma, x : !\tau', x : !\tau' \vdash e : \tau}{\Gamma, x : !\tau' \vdash e : \tau} \qquad \mathbb{1}\textsc{-Intro}\ \frac{}{!\Gamma \vdash () : \mathbb{1}} \qquad \mathbb{1}\textsc{-Elim}\ \frac{\Gamma_1 \vdash e_1 : \mathbb{1} \qquad \Gamma_2 \vdash e_2 : \tau}{\Gamma_1, \Gamma_2 \vdash \texttt{let}\ () = e_1\ \texttt{in}\ e_2 : \tau}$$

$$\otimes\textsc{-Intro}\ \frac{\Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2 \vdash e_2 : \tau_2}{\Gamma_1, \Gamma_2 \vdash (e_1, e_2) : \tau_1 \otimes \tau_2} \qquad \otimes\textsc{-Elim}\ \frac{\Gamma_1 \vdash e_1 : \tau_1 \otimes \tau_2 \qquad \Gamma_2, x : \tau_1, y : \tau_2 \vdash e_2 : \tau_3}{\Gamma_1, \Gamma_2 \vdash \texttt{let}\ (x, y) = e_1\ \texttt{in}\ e_2 : \tau_3}$$

$$\textsc{Dereliction}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma, x : !\tau_1 \vdash e : \tau_2} \qquad \textsc{Promotion}\ \frac{!\Gamma \vdash e : \tau}{!\Gamma \vdash !e : !\tau} \qquad !\textsc{-Elim}\ \frac{\Gamma_1 \vdash e_1 : !\tau_1 \qquad \Gamma_2, x : !\tau_1 \vdash e_2 : \tau_2}{\Gamma_1, \Gamma_2 \vdash \texttt{let}\ !x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

$$\textsc{Shallow-Cast}\ \frac{\Gamma \vdash e : *\tau}{\Gamma \vdash \&e : !\tau} \qquad \textsc{Copy}\ \frac{\Gamma_1 \vdash e_1 : !\tau_1 \qquad \Gamma_2, x : *\tau_1 \vdash e_2 : !\tau_2}{\Gamma_1, \Gamma_2 \vdash \texttt{copy}\ e_1\ \texttt{as}\ x\ \texttt{in}\ e_2 : !\tau_2} \qquad \textsc{Necessitation}\ \frac{\varnothing \vdash e : \tau}{!\Gamma \vdash *e : *\tau}$$

Figure 2.5: Properly linear type system of Marshall et al. [2022] that includes a uniqueness modality. Unique values cannot be contained within unique values, as Shallow-Cast would allow duplicating the inner unique values. As a result, all the built-in container types are linear. Copy allows copying unique values; the non-linearity of τ_2 makes $*$ act as a relative monad [Altenkirch et al., 2015] over !. Weaken is integrated within Var, $\mathbb{1}$-Intro and Necessitation.

2.8 Borrowing

Linear type theory, quantitative type theory and uniqueness type theory all have one inconvenience in common: Every function consumes all of its arguments, and so a function that only reads from a linear or unique argument will lose the reference in the process. As we are working in the context of pure functional programming languages, a reference being "read-only" in a function refers to the fact that the reference does not escape the function.

In most basic formulations of linear and uniqueness type theory, this is resolved by adjusting the encoding of linear and unique functions to manually thread the read-only reference through the program: The function $f : \tau_1 \multimap \tau_2 \multimap \tau_3$, where the second argument does not escape in the return value, becomes $f : \tau_1 \multimap \tau_2 \multimap \tau_2 \otimes \tau_3$. Unfortunately, while this encoding can always be used, it is inconvenient and affects both the type-level encoding and the term-level usage of most functions.

In order to alleviate this issue, many linear type theories implement a notion of "borrowing", i.e. the non-consumption of arguments for which all components are guaranteed not to escape.

Implementations

There are essentially three classes of implementations of borrowing:

1. Applying syntactical restrictions on the types that can be borrowed [Wadler, 1990]
2. Integrating an escape analysis with the type theory [Odersky, 1992] [Kobayashi, 1999] [Aspinall and Hofmann, 2002]
3. Using type-level mechanisms to hide the explicit threading [Spiwack et al., 2022]

Wadler [1990] proposes the notion of a strict `let! (x) y = u in v` expression to borrow a variable `x` in `u`, where `u` is evaluated strictly and none of the components of the type of `x` occur in the type of `u`. If these conditions are fulfilled, then the type system guarantees that `x` cannot be contained in the result of `u`. Unfortunately, this is already quite restrictive, and when polymorphic or erased types come into play, borrowing very rarely works when one wants it to work.

Odersky [1992] implements so-called "observer types", which witness that the variable corresponding to the observer type has been borrowed. Then, upon leaving the scope of the borrow, the implementation checks whether the type of the resulting expression contains any observer types to ensure that none of the borrowed variables escape. Aspinall and Hofmann [2002] and Kobayashi [1999] implement similar ideas.

Spiwack et al. [2022] use a system of linear constraints akin to type classes that can be consumed and returned implicitly by functions in order to remove the explicit threading from the term language of Linear Haskell. These constraints need to be freed manually and explicitly by the user, much like regular Linear Haskell types.

Types subject to these constraints are tagged with an additional variable in order to connect constraints with regular types, and when returning a fresh constraint, an existential quantifier must be used to summon a variable to tag the constraint with. There are constraints for hiding the continuation-passing style when creating functions, constraints for reading and writing arrays, as well as constraints for slices of arrays.

We think that constraint systems are an interesting approach, but that as formulated by Spiwack et al. [2022], there is unfortunately still a large amount of syntactical overhead in the term language, as constraints have to be unpacked explicitly.

Complex borrowing

Finally, linear references may not just get lost when passed as function arguments themselves, but when stored within a function argument as well. For example, in $\mathrm{fst} : \tau_1 \otimes \tau_2 \multimap \tau_1$, τ_1 escapes, but τ_2 is lost. Unfortunately, this issue is much more difficult to resolve, as we cannot retain our unique reference to τ_2 without also retaining our unique reference to $\tau_1 \otimes \tau_2$, with which we will also retain our unique reference to τ_1, a reference that is not unique anymore after application of fst.

Inspired by Rust [Weiss et al., 2021], Spiwack et al. [2022] suggest introducing primitive "lending" functions that grant temporary read access into a type, for example as follows for a linear type `MArray` that is allowed to contain other linear types:

```
lend : MArray α ⊸ !ℕ ⊸ (!α ⊸ β) ⊸ Array α ⊗ β
```

Here, the unique reference to the array is consumed by `lend` and hence unavailable in the continuation of type `!`$\alpha \multimap \beta$. `!`α provides read-only access to the array value indexed by the argument of type `!ℕ`. The constraint mechanism additionally ensures that the argument of type `!`α cannot escape in β, as well as that the explicit threading of `Array` α becomes implicit.

For `MArray`, Spiwack et al. [2022] define another complex borrowing mechanism inspired by Rust: A function `split` allows slicing an array `a` at an index `i`, consuming the original array in the process and producing two linear slices `x := a[0:i)` and `y := a[i:len(a))`, plus a token which witnesses that `x` and `y` are slices of `a`. Then, a function `join` can efficiently put two such slices with an associated witness token back together. Using continuations, we could also again define a lending primitive similar to the following and separately ensure that both arguments of type `!(Array` α`)` do not escape in β:

```
lendSlices : Array α ⊸ !ℕ ⊸ (!(Array α) ⊸ !(Array α) ⊸ β)
   ⊸ Array α ⊗ β
```

Primitive dedicated borrowing operations can be defined for many other types as well. We will not touch on this form of borrowing in this thesis and will instead focus on the simpler notion of borrowing function arguments themselves.

Updating values within values

In section 2.4, we saw that performing destructive updates on values within containers can be problematic for Lean's destructive update mechanism as well. So far, we have only discussed approaches that grant read access to values within containers, but what about write access?

It turns out that granting write access to single elements is rather straight-forward and can be implemented in a similar manner as we did in section 2.4. For arrays and a linear type α, we could define a primitive function `swap : Array` $\alpha \multimap$ `!`$\mathbb{N}$ $\multimap$ $\alpha \multimap$ `Array` $\alpha \otimes \alpha$ that swaps out an element in the array with another and will hence retain the linearity of both. Using this primitive, we can define `update : Array` $\alpha \multimap \alpha \multimap$ `!`$\mathbb{N} \multimap$ `(`$\alpha \multimap \alpha$`)` $\multimap$ `Array` α, where the second argument is a default value that is swapped into the array while the function of type `(`$\alpha \multimap \alpha$`)` is applied. On the other hand, implementing an efficient slicing primitive that allows writing to slices as in Spiwack et al. [2022] would be more difficult, since there is no guarantee that two sub-arrays can be efficiently rejoined without a token witnessing this. The Lean 4 destructive update mechanism struggles with this issue, too.

3 Design Space Exploration

In this chapter, we will explore possible design decisions with the goal of guaranteeing efficient in-place updates. First, we will evaluate linear type theory, quantitative type theory and uniqueness type theory, and argue why we decide for the latter. Second, we will discuss the dimensions in the design space of uniqueness type systems.

3.1 Substructural Framework

Let us first briefly repeat some of the important points from Section 2.5, Section 2.6 and Section 2.7 that are relevant to this section:

1. Properly linear types make a guarantee for the usage of a variable in the future, but not the past.
2. Uniqueness type theory makes a guarantee for the usage of a variable in the past, but not the future.
3. Invariably unique types are always unique and can never discard their uniqueness.
4. Properly linear types can make a guarantee for the past on particular types if all constructors of the type return linear values.
5. Properly linear types can discard the linearity of particular types with special library functions, but are typically not capable of doing so in constant time for nested linear types.
6. Quantitative type theory introduces an "erasure" quantity to specify computationally irrelevant terms in dependent types and count only those in the substructural portion of the type system.
7. Quantitative type theory inherits the properties of proper linearity.
8. Both quantitative and properly linear type theories can use linear functions that return values of linear type to produce values of non-linear type by demanding inputs of non-linear type to the function.
9. Having linear functions consume resources based on demand forces constructors of unique types to be formulated in continuation-passing style.

Properly linear types

Because of 1. and despite 4. and 5., we believe that properly linear types are simply not the right tool for ensuring uniqueness for destructive updates.

Although they can be made to ensure uniqueness with careful library design, the tricks involved often seem to work against the grain of the initial decision to allow for a coercion from non-linear to linear types and are never without sizeable limitations, like the discarding of linearity only being available for a couple of types and nesting of such unique types being complicated.

While Spiwack et al. [2022] manage to remove some of these limitations, they also add an entirely separate system of linear capabilities on top of the linear type system.

Invariably unique types

Meanwhile, due to 3., we think that invariably unique types are too restrictive for safe destructive updates.

We would like our type system to be fairly non-invasive, so as not to bother users when the type system is too weak to enforce the desired property. Hence, since the reference counting system described in Section 2.4 is already suitable to implement safe destructive updates at runtime, we think that it is absolutely crucial that users can fall back to this manual type-less mode of ensuring uniqueness if they cannot get the type system to guarantee the invariants that they need. This is especially pressing because we intend to add a substructural type system to the existing eco-system of Lean, where none of the code is annotated with linearity or uniqueness annotations. As per 2., uniqueness type theory has the desired property.

Term language

Finally, we must decide what the term language of our type system will be.

Due to 7., 8. and 9., existing formulations of quantitative type theory are not readily suited for uniqueness. Graded modal dependent type theory [Moon et al., 2021] is still an active area of research and implementing something along these lines would be far beyond the scope of this thesis. Transferring the core idea of QTT from 6. and removing all the components that make QTT “quantitative” may be possible, but would both increase the complexity of the resulting system due to the extra erasure attribute and require additional adjustments to the existing compiler toolchain to respect erasure.

Because of this, we decide against combining dependent type theory with uniqueness type theory. Instead, we target a derivate of the LCNF language described in Section 2.3 in the later stages of the compiler toolchain when type dependencies have been erased. Type erasure will lose us some analysis precision, but in turn integrating the resulting type system with the Lean compiler toolchain should be much easier.

3.2 Uniqueness Type Systems

In Section 2.7 and Section 2.8, we described two key challenges that come up when designing a uniqueness type system. The first is that non-unique values, including closures of non-unique functions, cannot be allowed to contain unique values. The second is that functions which use an argument in a read-only manner still consume it, thus losing the reference in the process.

3.2.1 Higher-Order Functions

As discussed in Section 2.7, invariably unique type systems ensure that shared containers cannot contain unique values during the construction of a value, whereas Clean ensures it during the deconstruction of a value by disallowing the deconstruction of a shared value if it contains unique values. Higher-order functions complicate this matter because function types typically do not reveal the types of the values in their closure and because implementations usually commit to one particular function pointer when the higher order function is created.

To see why this is an issue, consider a function $f : *(*\alpha \to *(!\beta \to *\alpha))$, using the notation introduced in Section 2.7 where $*$ represents a unique type and $!$ represents a non-unique type. There are two ways in which the uniqueness of the first argument can be leveraged: In the construction of the return value of type $*\alpha$, and in the resulting code for the function f where the first argument may get updated destructively. If we partially apply the first argument, the type becomes $f\ a : *(!\beta \to *\alpha))$ and the information that the first argument was unique is lost, despite a being in the closure of $f\ a$. Now, if we discard the uniqueness of f and apply it twice, then the return type may incorrectly suggest that the return value is unique, and we may even accidentally destructively update the first argument to the function, despite the fact that it is shared between the two function invocations.

There are several possible solutions to this problem, most of which have previously been covered by de Vries [2009].

"Necessarily unique"

The first is to take the approach that Clean uses and disallow discarding the uniqueness of function types altogether. Clean ensures this by introducing another kind of type into its type system, so called "necessarily unique" types. In practice, "necessarily unique" is the same thing that we have been calling "invariably unique" so far: the value is unique when it has been constructed and can never lose its uniqueness. Unfortunately, having support for "necessarily unique" values only in functions creates two additional problems.

First, in a type system with polymorphic types, the fact that functions in particular can be invariably unique also precludes type variables from discarding their uniqueness, as they could be substituted for functions.

And second, if an invariably unique function is stored in a unique container that then discards its uniqueness, we must make the function unavailable during the deconstruction of the value, lest it could have been shared. As argued in Section 2.7, for other unique values in non-unique containers, we do not have to be this restrictive and could instead also discard the uniqueness of the contained unique values during deconstruction.

Furthermore, a perspective put forward by Marshall et al. [2022] is that Clean's "necessarily unique" is too restrictive: For functions with unique values in their closure, we typically do not care about the uniqueness of the function itself, just that the function does not duplicate values in its closure. Hence, functions could really be properly linear, not invariably unique, allowing for some greater flexibility when joining two code paths where one yields a function with a unique value in its closure, whereas the other does not.

Closure typing

The second solution is what de Vries calls "closure typing"—instead of disallowing the discarding of uniqueness of a function altogether, an additional attribute is added to every function type that denotes whether the function contains a unique value in its closure. As a result, when applying the function, the information of whether the closure contains a unique value is not lost, and applying a shared function with a unique value in its closure becomes disallowed.

Higher-rank polymorphism

The third solution outlined by de Vries is to attempt to do away with the coercion altogether and replace it with careful library design, though of another nature than the freeze function in Linear Haskell.

First, de Vries argues that constructors of functions should leverage polymorphism in order to create values of polymorphic kind, e.g. $\text{mkArray} : !\mathbb{N} \to *\text{Array}\ \alpha$ becomes $\text{mkArray} : \forall u.\ !\mathbb{N} \to u(\text{Array}\ \alpha)$, where $u \in \{*, !\}$. However, this is not equivalent to the approach with the coercion, as we must commit to a concrete u when constructing the array, and two code paths cannot use the same array in two separate ways anymore.

To mitigate this, de Vries suggests using higher-rank types so that the constructor is typed as $\text{mkArray} : !\mathbb{N} \to (\forall u.\ u(\text{Array}\ \alpha))$ and two different code paths can instantiate $(\forall u.\ u(\text{Array}\ \alpha))$ in two separate ways. Unfortunately, this is still not equivalent to being able to discard uniqueness: Two code paths can now instantiate the array in two separate ways, but after they instantiate it as needed, they cannot be joined anymore, as the concrete instantiations $*\text{Array}\ \alpha$ and $!\text{Array}\ \alpha$ are not compatible.

Deleveraging uniqueness

Finally, the fourth solution outlined by de Vries, originally due to Harrington [2006], is to allow the application of shared functions with a unique value in their closure, but to degrade the return type of the function to non-unique instead.

As de Vries correctly points out, if this idea is used in a programming language, then ways of leveraging the uniqueness of the function argument other than the uniqueness of the return value must be addressed as well, e.g. the presence of destructive updates in the function. For destructive updates in particular, when creating a higher-order function, one possible implementation is to yield two function pointers for the higher-order function: one for if the function remains unique in which destructive updates are used, and another for if the function becomes non-unique in which no destructive updates are used. Then, when the unique function is forced to discard its uniqueness, the function pointer with destructive updates can be swapped out for a function pointer without destructive updates, and the now violated uniqueness of the function argument cannot be leveraged anymore.

Since Clean uses uniqueness not just for destructive updates, but for threading I/O as well, de Vries argues that the approach is inadequate, as side-effecting functions like $\text{closeFile} : {*}\text{File} \to !\mathbb{B}$ cannot be prevented from leveraging the uniqueness of the $*\text{File}$ argument, and closing a file twice will always be an error. However, we believe that uniqueness types are simply the wrong tool to handle I/O, as it is never desirable to make I/O values shared, and that Clean should instead use invariably unique types for I/O. Uniqueness types are better suited for situations in which the sharedness of a value is still admissible, like destructive updates in memory, where we can always copy a value if it is shared.

Conclusion

We feel that the first approach is overkill. If one introduces linearity into a uniqueness type system for functions, then one should also introduce linearity for all other types, so that linear functions can be nested in linear structures. This is essentially the approach of Marshall et al. [2022], though as stated in Section 2.7, their type system does not support nesting of unique types. This creates lots of extra work for users, and it seems excessive if all we want to do is guarantee safe destructive updates and handle higher-order functions correctly.

The second approach of closure typing also adds plenty of notational overhead to the type theory, as functions now have two separate type annotations, and the idea of not being able to apply a function that is present in the context seems unintuitive to us.

We think that the third approach introduces a lot of complexity in requiring higher-rank types to work, and even then, it does not fully replace the notion of discarding uniqueness.

Despite de Vries' criticism of it, we think that the last approach is the most viable if all we care about are destructive updates of values in memory. However, this approach would require a hefty change to the runtime of Lean, as every unique

higher-order function needs to carry two function pointers around, so that we can switch to the one without destructive updates when the function becomes shared. Because of this, we decided not to implement this approach yet, and will instead require all higher-order functions and the types within them to be non-unique. As this precludes us from making guarantees for monadic code, type classes and idioms using higher-order functions like the one in Section 2.4, this is a considerable limitation that we hope to resolve in the future.

3.2.2 Implicit Coercion

In most descriptions of linear and uniqueness type theory discussed so far, the coercion between non-linear/non-unique and linear/unique types has usually been implicit, e.g. passing a unique value to a shared parameter works, but will discard the uniqueness in the process. For linear type theories this is not an issue, as the coercion from non-linear types to linear types adds structure. However, for uniqueness type theories, the coercion from unique to shared discards the guarantee that the value is unique, and so the user may not want it to happen implicitly.

Additionally, there is a question of when unique types are implicitly coerced. In Clean, this is only possible at function boundaries, and if one wishes to share a variable, then one must choose the respective parameter type accordingly. An explicit coercion operator would give users greater control over when coercion happens.

While the type theory that we will describe in Chapter 4 also uses an implicit coercion, we think it is best to make it explicit when integrating the type theory with Lean 4.

3.2.3 Uniqueness Propagation

As discussed in Section 3.2.1, shared containers cannot be allowed to contain unique values. We have already explored possible approaches for how to ensure this for higher-order functions, but we are still lacking a more general framework for other types.

Subtyping

In Clean, a shared product is allowed to contain unique values, but upon deconstruction or projection, the type system checks that the projected values cannot be unique if the outer value is shared. In other words, if we discard the uniqueness of a product, then we cannot access its fields anymore. This is rather limiting, because we could instead also discard the uniqueness of the fields when we attempt to access them.

Relatedly, there is a question of how deep the subtyping relation induced by the coercion from unique to non-unique types is. In Clean, it is very shallow and only allows discarding the uniqueness of the outer layer, i.e. we cannot pass a value of type $*(*\alpha \times *\beta)$ to a parameter of type $!(!\alpha \times !\beta)$, only a parameter of type $!(*\alpha \times *\beta)$, the fields of which cannot be accessed anymore. Similarly, passing a value of type

$*(*\alpha \times *\beta)$ to a parameter of type $*(!\alpha \times *\beta)$ is not possible either, though doing so is sound. A less shallow subtyping relation would resolve these issues.

If we make the subtyping relation less shallow, then coercions should also propagate the change in the outer attribute to the values contained within, so that $!(*\alpha \times *\beta)$ and $!(!\alpha \times !\beta)$ are not differently annotated but equivalent types, and $!(*\alpha \times *\beta)$ becomes unrepresentable. For types like $*(*\alpha \times *\beta)$, where the attributes are floated out into type arguments, this is straight-forward, as we can propagate the outer sharedness annotation to the inner components of the type.

Algebraic data types

For algebraic data types (ADTs), this is not as straight-forward, as there is a question of what to do with the attributes associated with fields of the ADT. One possibility would be to float out every attribute within the ADT, so that fields use an attribute variable m to refer to a concrete attribute passed to the ADT, after which ! can be propagated directly in the arguments of the ADT, much like in $*(*\alpha \times *\beta)$. However, for large ADTs, this will very quickly accumulate dozens of attribute arguments, leading to a huge notational overhead.

Instead, for our type system that will be described in Chapter 4, we decide that fields with a $*$ attribute are "unique if the outer value is unique". In other words, for the uniqueness annotations within ADTs, sharedness is not propagated directly, only when deconstructing the value and accessing the fields. Since these attributes are not floated out, delaying the propagation does not make us end up with differently annotated types that are equivalent, like it would be the case for $*(*\alpha \times *\beta)$.

3.2.4 Borrowing

As discussed in Section 2.8, most implementations of borrowing use a form of type-driven escape analysis. We believe that introducing extra attributes like observer-types into the type system that users need to keep accurate track of in order to be able to access borrowing is too much of an annotational burden.

Instead, we will make use of the fact that escape analyses are very local in nature; whether a variable escapes or not can be approximated by following the data flow of the variable from the start of the function to the return value. Hence, we implement a data flow analysis [Allen and Cocke, 1976] in Section 4.4. Due to the inherent locality of the analysis, we will not need any type information to run it.

While we will not touch on it further, it is worth pointing out that adding extra annotations describing whether a function parameter is borrowed may still be a good idea for maintenance purposes, despite the fact that we can compute this information. When integrating our type system with Lean 4, it may hence be a good idea to add these annotations as well.

4 Formal Specification

In this chapter, we will provide a formal description of our type theory and all the associated mechanisms required to make it work. Section 4.1 introduces the syntactical material for our types and declares a number of commonly useful utility functions. Section 4.2 defines the syntax of the IR. In Section 4.3, we specify an escape data flow analysis in order to implement the borrowing mechanism described in Section 4.4. Finally, Section 4.5 provides the rules of our type theory.

4.1 Types

In all of the following sections, we use $[x]$ to denote a vector of elements x, otherwise commonly written as $\overline{x}$. We will often lift these brackets over an operation; e.g. the functional code $\mathrm{map}(\oplus, \mathrm{zip}([x],[y]))$ is written as $[x \oplus y]$ for vectors $[x]$ and $[y]$. In derivation rules, we also use $[x]$ for $x \in \mathbb{B}$ to mean $\forall x \in [x].\ x$.

4.1.1 Syntax

$$
\begin{aligned}
x, y, z &\in \mathrm{Var} \\
i &\in \mathrm{Ctor} \\
j &\in \mathrm{Proj} \\
c &\in \mathrm{Const} \\
m &\in \mathrm{Attr} && ::= \ !\ \mid\ * \\
a &\in \mathrm{ADT} && ::= \mu\ x^{\kappa}_{\mathrm{adt}}.\ [[\tau_{\mathrm{field}}(x^{\kappa}_{\mathrm{adt}}, [y^{\tau}])] \to *x^{\kappa}_{\mathrm{adt}}] \\
A &\in \mathrm{ADTConst} \\
\gamma &\in \mathrm{ADTDecls} && = \mathrm{ADTConst} \rightharpoonup \mathrm{ADT} \\
\tau &\in \mathrm{AttrType} && ::= m\ x^{\kappa}\ \mid\ x^{\tau}\ \mid\ m\ \blacksquare\ \mid\ m\ A\ [\tau_{\mathrm{arg}}]\ \mid\ !\ [\tau_{\mathrm{param}}] \to \tau_{\mathrm{ret}} \\
\delta_{\tau} &\in \mathrm{FunTypes} && = \mathrm{Const} \rightharpoonup [\mathrm{AttrType}] \times \mathrm{AttrType}
\end{aligned}
$$

Ctor and Proj denote the constructors and fields within a constructor, respectively. Const designates function names. Attr contains the attributes that are the main subject of our type theory; shared (!) and unique ($*$).

ADTs

Since the Lean 4 compiler erases type dependencies, we will limit ourselves to types that look like potentially recursive algebraic data types.

In $\mu\ x^\kappa_{\text{adt}}.\ [[\tau_{\text{field}}(x^\kappa_{\text{adt}}, [y^\tau])] \to *x^\kappa_{\text{adt}}]$, x^κ_{adt} is the variable we use to refer back to the ADT itself, $[[\tau_{\text{field}}(x^\kappa_{\text{adt}}, [y^\tau])] \to *x^\kappa_{\text{adt}}]$ is a vector of constructors and $[\tau_{\text{field}}(x^\kappa_{\text{adt}}, [y^\tau])]$ denotes the types of the fields of the constructor, where τ_{field} is parametrized by the variable x^κ_{adt} representing the ADT itself, as well as a vector $[y^\tau]$ of type parameters of the ADT.

Constructors, projections and type parameters are assumed to be enumerated by intervals $[0, n)$, and so we use the following notation:

$$
\begin{aligned}
(\mu\ x^\kappa_{\text{adt}}.\ [[\tau_{\text{field}}(x^\kappa_{\text{adt}}, [y^\tau])] \to *x^\kappa_{\text{adt}}])_i &\coloneqq [[\tau_{\text{field}}(x^\kappa_{\text{adt}}, [y^\tau])] \to *x^\kappa_{\text{adt}}]_i \\
([\tau_{\text{field}}(x^\kappa_{\text{adt}}, [y^\tau])] \to *x^\kappa_{\text{adt}})_j &\coloneqq [\tau_{\text{field}}(x^\kappa_{\text{adt}}, [y^\tau])]_j \\
[y^\tau]_x &\coloneqq x
\end{aligned}
$$

As Lean 4 code commonly interacts with external types and external code via its foreign function interface (FFI), we cannot assume that we can access an ADT declaration for every type. To cope with this, ADTs are instead identified by an ADTConst, the mapping of which is maintained in a global and partial function $\gamma \in \text{ADTDecls}$. ADTConsts $A \notin \text{dom}(\gamma)$ that appear in the program are regarded as external. Lastly, we demand that all $A \in \text{dom}(\gamma)$ are fully propagated, i.e. that $\forall i\ j.\ \text{propagate}(\gamma(A)_{ij}) = \gamma(A)_{ij}$ for the definition of propagate below.

AttrType

AttrType contains our types. $m\ x^\kappa$ and x^τ are the two kinds of variables that can occur only within an ADT; self-referring variables x^κ have an associated (fixed) attribute and the variable only represents the parameterless portion of a type, while variables x^τ can denote any type parameter $\tau \in \text{AttrType}$.

$m\ \blacksquare$ is an erased type, $m\ A\ [\tau_{\text{arg}}]$ is an ADT (or external type) A parametrized by type arguments $[\tau_{\text{arg}}]$, and $!\ [\tau_{\text{param}} \to \tau_{\text{ret}}]$ denotes the type of a higher-order function.

Finally, δ_τ provides the parameter and return types for all functions in the program, including external ones. This is a reasonable assumption because we can simply assign a type $[!\ \kappa_{\text{param}}] \to\ !\ \kappa_{\text{ret}}$ for Lean 4 functions with an unattributed function type $[\kappa_{\text{param}}] \to\ \kappa_{\text{ret}}$. For $\delta_\tau(c) = ([\tau_{\text{param}}], \tau_{\text{ret}})$, we also demand that all the types are fully propagated, i.e. $\forall \tau_{\text{param}} \in [\tau_{\text{param}}].\ \text{propagate}(\tau_{\text{param}}) = \tau_{\text{param}} \land \text{propagate}(\tau_{\text{ret}}) = \tau_{\text{ret}}$ for the definition of propagate below.

Throughout this thesis, we assume that functions c have already been type-checked without annotations pre-erasure by the Lean type checker.

4.1.2 Propagation

A function weaken makes types shared and propagates sharedness inwards through the type:

$$
\begin{aligned}
&\text{weaken} : \text{AttrType} \to \text{AttrType}\\
&\text{weaken}(m\ x^{\kappa}) = !\ x^{\kappa}\\
&\text{weaken}(x^{\tau}) = x^{\tau}\\
&\text{weaken}(m\ \blacksquare) = !\ \blacksquare\\
&\text{weaken}(m\ A\ [\tau_{\text{arg}}]) = !\ A\ [\text{weaken}(\tau_{\text{arg}})]\\
&\text{weaken}(!\ [\tau_{\text{param}}] \to \tau_{\text{ret}})\\
&\qquad = !\ [\text{weaken}(\tau_{\text{param}})] \to \text{weaken}(\tau_{\text{ret}})
\end{aligned}
$$

We will henceforth denote applications of weaken using $!\tau \coloneqq \text{weaken}(\tau)$. We will need this function whenever we have to make a type shared and we will avoid using it for the types of fields that have not been substituted yet.

$$
\begin{aligned}
&\text{propagate} : \text{AttrType} \to \text{AttrType}\\
&\text{propagate}(m\ x^{\kappa}) = m\ x^{\kappa}\\
&\text{propagate}(x^{\tau}) = x^{\tau}\\
&\text{propagate}(m\ \blacksquare) = m\ \blacksquare\\
&\text{propagate}(*\ A\ [\tau_{\text{arg}}]) = *\ A\ [\text{propagate}(\tau_{\text{arg}})]\\
&\text{propagate}(!\ A\ [\tau_{\text{arg}}]) = !\ A\ [!\tau_{\text{arg}}]\\
&\text{propagate}(!\ [\tau_{\text{param}}] \to \tau_{\text{ret}})\\
&\qquad = !\ [!\tau_{\text{param}}] \to !\tau_{\text{ret}}
\end{aligned}
$$

Using propagate, we ensure that unique types are made shared if they are contained within a shared type, since a value within another value cannot be guaranteed to be unique if the outer value is already shared.

We use the following notation for substitution in $a = \mu\ x^{\kappa}_{\text{adt}}.\ [[\tau_{\text{field}}(x^{\kappa}_{\text{adt}}, [y^{\tau}])] \to *x^{\kappa}_{\text{adt}}]$:

$$a\{A, [\tau]\} \coloneqq \mu\ x^{\kappa}_{\text{adt}}.\ [[\text{propagate}(\tau_{\text{field}}[A\ [\tau]/x^{\kappa}_{\text{adt}}][[\tau]/[y^{\tau}]])] \to *x^{\kappa}_{\text{adt}}]$$

4.1.3 Definitional Nuances

It is worth pointing out a number of semantic nuances in both the definitions of our types and the function propagate above:

- If we know that a type is unique, we can always throw away this guarantee and make it shared, as described in Section 2.7.

- In m ■, ■ could be any other type, potentially parametrized by any other attributed type if ■ used to be $A\ [\tau_{\text{arg}}]$. We must ensure that our type theory can deal with this kind of erasure.
- We must always ensure that types remain fully propagated.
- Within an ADT declaration a, we do not know how to propagate x^τ, as it depends on the concrete type argument. Instead, we ensure that types become fully propagated when substituting type variables for type arguments using our definition of $a\{A, [\tau]\}$.
- While we can propagate within a given ADT field or within any other given type, we cannot propagate from $!\ A\ [\tau_{\text{arg}}]$ into the fields within $\gamma(A)$, as not all the attributes in $\gamma(A)$ are floated out to arguments in $!\ A\ [\tau_{\text{arg}}]$, only those in the type arguments $[\tau_{\text{arg}}]$. To alleviate this issue, we take an attribute $\star$ in a field within $\gamma(A)$ to mean "unique if the outer value is unique" and enforce this property in our type rules for projection on $m\ A\ [\tau_{\text{arg}}]$.
- Higher-order functions are always shared, so we do not need to worry about covariance or contravariance. This is a considerable limitation: Lean 4 code uses higher-order functions very liberally to encode type classes, monads, as well as some performance idioms related to the Counting Immutable Beans optimization described in Section 2.4. See Section 3.2.1 for possible approaches to alleviate this issue in future work.
- Since external types have no associated declaration, if we want to gather information about the type, we must rely on auxiliary information provided by users at the FFI. We will need this kind of auxiliary information in Section 4.3 and Section 4.4.

4.1.4 Utilities

We will now proceed to declare some convenient auxiliary functions.

$$
\begin{aligned}
&\text{weakenInner} : \text{AttrType} \to \text{AttrType} \\
&\text{weakenInner}(m\ x^\kappa) &&= m\ x^\kappa \\
&\text{weakenInner}(x^\tau) &&= x^\tau \\
&\text{weakenInner}(m\ \blacksquare) &&= m\ \blacksquare \\
&\text{weakenInner}(m\ A\ [\tau_{\text{arg}}]) &&= m\ A\ [!\tau_{\text{arg}}] \\
&\text{weakenInner}(!\ [\tau_{\text{param}}] \to \tau_{\text{ret}}) &&= !\ [\tau_{\text{param}}] \to \tau_{\text{ret}}
\end{aligned}
$$

With weakenInner, we leave the outer attribute intact but weaken every inner type. This will be useful when dealing with erased types m ■: When casting m ■ to

another type τ, we want τ to retain the outer attribute m, but we cannot make any guarantees for the inner attributes, and so we weaken them. We will avoid using it for the types of fields that have not been substituted yet.

$$
\begin{aligned}
&\text{strengthen} : \text{AttrType} \to \text{AttrType} \\
&\text{strengthen}(m\ x^{\kappa}) && = *\ x^{\kappa} \\
&\text{strengthen}(x^{\tau}) && = x^{\tau} \\
&\text{strengthen}(m\ \blacksquare) && = *\ \blacksquare \\
&\text{strengthen}(m\ A\ [\tau_{\text{arg}}]) && = *\ A\ [\text{strengthen}(\tau_{\text{arg}})] \\
&\text{strengthen}(!\ [\tau_{\text{param}}] \to \tau_{\text{ret}}) && = !\ [\tau_{\text{param}}] \to \tau_{\text{ret}}
\end{aligned}
$$

Strengthening makes every attribute within a type unique that can be made unique. We will use strengthen for inferring the type arguments of $m\ A\ [\tau_{\text{arg}}]$ at construction: If a type parameter variable is not assigned by any constructor argument, we can strengthen it. We will also avoid using it for the types of fields that have not been substituted yet.

$$
\begin{aligned}
&\text{attr} : \text{AttrType} \rightharpoonup \text{Attr} \\
&\text{attr}(m\ x^{\kappa}) && = m \\
&\text{attr}(m\ \blacksquare) && = m \\
&\text{attr}(m\ A\ [\tau_{\text{arg}}]) && = m \\
&\text{attr}(!\ [\tau_{\text{param}}] \to \tau_{\text{ret}}) && = !
\end{aligned}
$$

The function attr simply yields the outer attribute of any $\tau \neq x^{\tau}$.

4.1.5 Subtyping

Finally, whenever we pass a type τ_1 to a type τ_2, we must ask ourselves whether τ_1 can be applied to τ_2. The type structure must be the same, but it should be possible to throw away the uniqueness attribute of types within τ_1. Hence, we use $m_1 \succcurlyeq_m m_2 :\Leftrightarrow m_1 = * \vee m_2 = !$ to denote attribute subtyping and define a subtyping relation $\succcurlyeq$ for fully propagated types τ_1 and τ_2 as follows:

$$
\boxed{\tau_1 \succcurlyeq \tau_2} \qquad \frac{m_1 \succcurlyeq_m m_2}{m_1\ \blacksquare \succcurlyeq m_2\ \blacksquare} \qquad \frac{m_1 \succcurlyeq_m m_2}{m_1\ x^{\kappa} \succcurlyeq m_2\ x^{\kappa}} \qquad \frac{}{x^{\tau} \succcurlyeq x^{\tau}}
$$

$$
\frac{}{!\ [\tau_{\text{param}}] \to \tau_{\text{ret}} \succcurlyeq !\ [\tau_{\text{param}}] \to \tau_{\text{ret}}} \qquad \frac{m_1 \succcurlyeq_m m_2 \qquad [\tau_{\text{arg}_1} \succcurlyeq \tau_{\text{arg}_2}]}{m_1\ A\ [\tau_{\text{arg}_1}] \succcurlyeq m_2\ A\ [\tau_{\text{arg}_2}]}
$$

Note that if higher-order functions could be unique, we would have to account for covariance and contravariance in this definition.

4.2 Intermediate Representation

For our model of the IR, we use a mixture of the IR described by Ullrich and de Moura [2020] and the newly implemented LCNF, both detailed in Section 2.3.

4.2.1 Syntax

$$
\begin{aligned}
\mathtt{e} \in \text{Expr} \; &::= \mathtt{c}\ \mathtt{[y]} \mid \mathtt{pap}\ \mathtt{c}\ \mathtt{[y]} \mid \mathtt{x}\ \mathtt{y} \mid \mathtt{(A}\ \mathtt{[}\tau\mathtt{?])}.\mathtt{ctor}_{\mathtt{i}}\ \mathtt{[y]} \\
&\quad \mid \mathtt{A}.\mathtt{proj}_{\mathtt{ij}}\ \mathtt{y} \\
\mathtt{F} \in \text{FnBody} &::= \mathtt{ret}\ \mathtt{x} \mid \mathtt{let}\ \mathtt{x} := \mathtt{e};\ \mathtt{F} \mid \mathtt{case}\ \mathtt{x}\ \mathtt{of}\ \mathtt{[F]} \\
&\quad \mid \mathtt{A}.\mathtt{case'}\ \mathtt{x}\ \mathtt{of}\ \mathtt{[ctor}_{\mathtt{i}}\ \mathtt{[y]} \Rightarrow \mathtt{F]} \\
f \in \text{Fn} \; &::= \lambda\ \mathtt{[y]}.\ \mathtt{F} \\
\delta \in \text{Program} &= \text{Const} \rightharpoonup \text{Fn}
\end{aligned}
$$

Expr and FnBody are similar to Lean's IR, except for our definition of `proj` and `ctor`, as well as the addition of a new instruction `case'`. As in Lean's IR, a global and partial $\delta \in$ Program assigns function declarations to constants. All $c \notin \text{dom}(\delta)$ that occur in the program are assumed to be external functions.

Projection

`proj` is provided not just with the projection `j` as in Lean's IR, but also the constructor `i`. As the code generation ensures that `proj` calls always occur after `case` within the same function if the type has multiple constructors or on its own if the type only has a single constructor, we can easily compute `i` by walking back from the $\mathtt{proj}_{\mathtt{j}}\ \mathtt{y}$ call either to the start of the function to set $\mathtt{i} = 0$ or to a `case x of [...]` instruction, where we choose `i` as the index of the branch that we are walking back from.

Construction

`ctor` takes an additional vector of explicit attributed type arguments $\mathtt{[}\tau\mathtt{?]}$, where `?` refers to each explicit argument being optional.

Since users do not provide them, Lean can provide us with type arguments $\mathtt{[}\kappa\mathtt{]}$ for the constructor call, but not any of the attributes, and so we must infer them from the types of arguments provided in `[y]`. But since there may be type arguments to `A` that occur only in the other constructors for `A`, we cannot infer all of them, and so they must be provided explicitly. However, type arguments that do not occur in `[y]` are also not subject to any uniqueness constraints, and so we can instantiate them as strongly as possible.

Subsequently, the attributes in $\mathtt{[}\tau\mathtt{?]}$ can be chosen arbitrarily: If the type argument occurs in `[y]`, we can infer the type together with its attributes, and if it does not occur in `[y]`, the type τ_e must be provided in $\mathtt{[}\tau\mathtt{?]}$, but the corresponding attributes can be chosen as given by $\text{strengthen}(\tau_e)$.

Deconstruction

The `case` and `proj` combination turns out to be unwieldy for substructural type systems: When we use `let z := A.proj`$_{ij}$ `y; F` on a unique value to obtain another unique value, the contained value now exists both in `z` and in `y`, i.e. uniqueness is violated. The solution to this issue would be that `A.proj`$_{ij}$ `y` consumes our unique value `y` so that it is not available in `F` anymore.

However, it is very common that we would like to access multiple fields of `y` in succession, which we will not be able to do now that `y` is consumed. So, instead, the typical solution to this issue in substructural type systems is not to access fields via projections, but using a single destructuring pattern match that yields all fields of the type in one go and consumes the variable associated with the type. This is exactly what `case'` does as well, and while the instruction does not exist in Lean's IR, it does exist in Lean 4's LCNF.

Regardless, even in LCNF, structures are still accessed via projections and not using a destructuring pattern match. To alleviate this final issue, we implement a compromise in Section 4.5 which ensures that we can use multiple projections on `y`, but not use it in any other manner.

Omissions

Finally, there are a number of omissions from our IR compared to the IR implemented in Lean. Most notably, there are instructions to work with join points, which we could implement as functions in our IR. However, it is worth noting that join points, like auxiliary functions c generated by the Lean compiler, do not necessarily have an associated user-provided type $\delta_\gamma(c)$. We will not provide a general type inference mechanism in this thesis, but will briefly discuss the topic in Section 6.1.

4.3 Escape Analysis

As described in Section 2.8, borrowing in functional languages is closely related to escape analysis; if nothing within a shared parameter escapes, then we do not have to make a unique argument to that parameter shared, as the caller is guaranteed to still hold the only reference to the value in question when the called function returns. Instead of unloading this additional burden of tracking the data flow of variables and fields to the user, we implement a data flow analysis.

4.3.1 Syntax

$$
\begin{aligned}
n, m &\in \mathbb{N} \\
s, t, v &\in \text{Tag} &&::= \#\text{const c} \mid \#\text{case i} \mid \#\text{app} \mid \#\text{param n} \\
q &\in \text{Escapee} &&::= x_{[ij]}@[t]? \\
\delta_{q_e} &\in \text{ExternFunEscapees} = \text{Const} \rightharpoonup 2^{\text{Escapee}}
\end{aligned}
$$

Escapees are the subject of our escape analysis and represent the elements of the sets that we compute, i.e. sets of escapees denote the elements of the lattice underlying our data flow analysis. Each escapee has an associated variable x, a field index $[ij]$ represented by a vector of Ctor × Proj tuples and an optional vector of tags that describes the path to the parameter an escapee was spawned from, if the escapee came from a function call. The need for the vector of tags will become obvious later, and until then it can just be understood as an identifier that identifies the location in the code where escapees from function calls were spawned. We write $x_{[ij]}$ for escapees without a tag, $x_{[ij]}@[t]$ for escapees with a tag and $x_{[ij]}@[t]?$ for escapees that either have a tag attached or do not have a tag attached.

Since external funtions do not have a function body that we can analyze, a global and partial function δ_{q_e} allows specifying the set of all escapees for these functions. For external functions $c \notin \mathrm{dom}(\delta_{q_e})$ we will assume all parameters and all fields thereof to escape.

4.3.2 Computing Escapees

Using abstract interpretation [Cousot and Cousot, 1977], we compute a fixed point of the following mutually recursive equations $[\![\cdot]\!]_Q(\cdot)$ and δ_Q, which we will explain in detail along the way. The first parameter of $[\![\cdot]\!]_Q(\cdot)$ is the portion of the function body that we want to compute the escapees for and the second parameter denotes the vector of tags thus far from the start of the function to this portion of the function body.

Escapees of function bodies

$$
\begin{aligned}
&[\![\cdot]\!]_Q(\cdot) : \mathrm{FnBody} \times [\mathrm{Tag}] \to 2^{\mathrm{Escapee}} \\
&[\![\texttt{ret x}]\!]_Q([t]) = \{\texttt{x}_{[]}\} \\
&[\![\texttt{case x of [F]}]\!]_Q([t]) \;=\; \bigcup_n [\![\texttt{[F]}_n]\!]_Q(\#\mathrm{case\ n} :: [t]) \\
&[\![\texttt{A.case' x of [ctor}_\mathrm{i}\texttt{ [y]} \Rightarrow \texttt{F]}]\!]_Q([t]) \;=\; \bigcup_n [\![\texttt{[F]}_n]\!]_Q(t') \\
&\qquad \cup \left\{\texttt{x}_{nm::[kj]}@[s]? \mid (\texttt{[y]}_m)_{[kj]}@[s]? \in [\![\texttt{[F]}_n]\!]_Q(t') \land m \in [0, |\texttt{[y]}|)\right\} \\
&\qquad\qquad \text{where } t' \coloneqq \mathrm{case\ n} :: [t] \\
&[\![\texttt{let x = c [y]; F}]\!]_Q([t]) = Q_F \\
&\qquad \cup \begin{cases}
Q & \exists [nm].\ \texttt{x}_{[nm]}@[v]? \in Q_F \land \texttt{c} \in \mathrm{dom}(\delta_Q) \\
\{\texttt{y}_{[]} \mid \texttt{y} \in \texttt{[y]}\} & \exists [nm].\ \texttt{x}_{[nm]}@[v]? \in Q_F \land \texttt{c} \notin \mathrm{dom}(\delta_Q) \land \texttt{c} \notin \mathrm{dom}(\delta) \\
\varnothing & \exists [nm].\ \texttt{x}_{[nm]}@[v]? \in Q_F \land \texttt{c} \notin \mathrm{dom}(\delta_Q) \land \texttt{c} \in \mathrm{dom}(\delta) \\
\varnothing & \forall [nm].\ \texttt{x}_{[nm]}@[v]? \notin Q_F
\end{cases} \\
&\qquad\qquad \text{where } Q \coloneqq \left\{(\texttt{[y]}_z)_{[ij]}@(\#\mathrm{param\ z} :: [t]) \mid z_{[ij]}@[s]? \in \delta_Q(\texttt{c})\right\} \\
&\qquad\qquad \text{and } Q_F \coloneqq [\![\texttt{F}]\!]_Q(\#\mathrm{app} :: [t])
\end{aligned}
$$

$$
\begin{aligned}
&[\![\texttt{let x = pap c [y]; F}]\!]_Q([t]) = [\![\texttt{F}]\!]_Q([t]) \\
&\quad \cup \begin{cases} \{\texttt{y}_{[]} \mid \texttt{y} \in \texttt{[y]}\} & \exists [nm].\ \texttt{x}_{[nm]}@[v]? \in [\![\texttt{F}]\!]_Q([t]) \\ \varnothing & \forall [nm].\ \texttt{x}_{[nm]}@[v]? \notin [\![\texttt{F}]\!]_Q([t]) \end{cases} \\
&[\![\texttt{let x = y z; F}]\!]_Q([t]) = [\![\texttt{F}]\!]_Q([t]) \\
&\quad \cup \begin{cases} \{\texttt{y}_{[]}, \texttt{z}_{[]}\} & \exists [nm].\ \texttt{x}_{[nm]}@[v]? \in [\![\texttt{F}]\!]_Q([t]) \\ \varnothing & \forall [nm].\ \texttt{x}_{[nm]}@[v]? \notin [\![\texttt{F}]\!]_Q([t]) \end{cases} \\
&[\![\texttt{let x = (A [}\tau\texttt{?]).ctor}_\texttt{i}\texttt{ [y]; F}]\!]_Q([t]) = [\![\texttt{F}]\!]_Q([t]) \\
&\quad \cup \begin{cases} \{\texttt{y}_{[]} \mid \texttt{y} \in \texttt{[y]}\} & \texttt{x}_{[]}@[v]? \in [\![\texttt{F}]\!]_Q([t]) \\ \{(\texttt{[y]}_j)_{[nm]}@[s]? \mid \texttt{x}_{\texttt{i}j::[nm]}@[s]? \in [\![\texttt{F}]\!]_Q([t])\} & \texttt{x}_{[]}@[v]? \notin [\![\texttt{F}]\!]_Q([t]) \end{cases} \\
&[\![\texttt{let x = A.proj}_\texttt{ij}\texttt{ y; F}]\!]_Q([t]) = [\![F]\!]_Q([t]) \\
&\quad \cup \begin{cases} \{\texttt{y}_{\texttt{ij}::[kl]}@[s]? \mid \texttt{x}_{[kl]}@[s]? \in [\![\texttt{F}]\!]_Q([t])\} & \exists [nm].\ \texttt{x}_{[nm]}@[v]? \in [\![\texttt{F}]\!]_Q([t]) \\ \varnothing & \forall [nm].\ \texttt{x}_{[nm]}@[v]? \notin [\![\texttt{F}]\!]_Q([t]) \end{cases}
\end{aligned}
$$

In `ret x`, only `x` itself escapes. For `case x of [F]`, we determine the escapees of each $\texttt{F} \in \texttt{[F]}$ and compute the resulting union of all escapees. In `A.case' x of [` $\texttt{ctor}_\texttt{i}$ `[y] ⇒ F]`, we use the same idea as for `case`, but must also transfer escapees concerning `[y]` over to `x`: Each escapee $(\texttt{[y]}_m)_{[kj]}@[s]?$ in branch n corresponding to constructor n is converted to an escapee $\texttt{x}_{nm::[kj]}@[s]?$.

For all `let x = e; F` function bodies, we will always have a case stating that if `x` does not escape in `F`, then we need not generate any additional escapees for `e`.

Application `let x = c [y]; F` is the most tricky since it is where our analysis recurses with escapees for `c`. If we have already computed escapees for `c` or they are specified in δ_{q_e}, i.e. $\texttt{c} \in \mathrm{dom}(\delta_Q)$, we take all escapees $z_{[ij]}@[s]?$ for parameters z from δ_Q and rename them to the corresponding arguments $\texttt{[y]}_z$. If $\texttt{c} \notin \mathrm{dom}(\delta_{q_e})$ is external, all `[y]` are assumed to escape. Finally, if we have not already computed the escapees for `c` but are expected to do so in the future because `c` is not external, we yield the bottom element of our lattice $\bot = \varnothing$.

When creating a higher-order function using `pap c [y]`, we assume that all `[y]` escape if the resulting higher-order function escapes. The same is true for higher-order function application `y z`: if the result escapes, then so may `y` and `z`. Note that creating an escapee for `y`, i.e. the higher-order function itself, is important, because if the higher-order function containing all the previously-applied arguments or the return value of the function escapes, we need to know that the previously-applied arguments may escape too, and so we propagate this bit of information backwards using the escapee for `y`.

$\texttt{(A [}\tau\texttt{?]).ctor}_\texttt{i}$ `[y]` and $\texttt{A.proj}_\texttt{ij}$ `y` are once again fairly straight-forward. If the ADT resulting from a constructor call escapes, then so do all of its fields, and if only particular fields of constructor `i` escape, then the respective escapees $\texttt{x}_{\texttt{i}j::[nm]}@[s]?$ must be translated to escapees for $\texttt{[y]}_j$ by removing the $\texttt{i}j$ field. Other escapees $\texttt{x}_{kj::[nm]}@[s]?$ for $k \neq \texttt{i}$ do not need to be translated. For $\texttt{A.proj}_\texttt{ij}$ `y`, we use the

same idea as for `case'` and translate the escapees for the projection to ones for `y`.

Post-processing

Next, we will define a couple of post-processing functions to make our application of abstract interpretation to the mutually recursive $[\![\cdot]\!]_Q\,(\cdot)$ and δ_Q terminate.

$$\begin{aligned}&\mathrm{fd} : [\mathrm{Var}] \times 2^{\mathrm{Escapee}} \to 2^{\mathrm{Escapee}}\\&\mathrm{fd}([y], Q) = \left\{x_{[ij]}@[t]? \mid x_{[ij]}@[t]? \in Q \wedge x \in [y])\right\}\end{aligned}$$

With fd, we remove all dead escapees for a function by keeping only those that correspond to function parameters $[y]$. This is mainly useful for performance because $[\![\cdot]\!]_Q\,(\cdot)$ accumulates escapees for all variables in a function, even local ones.

$$\begin{aligned}&\mathrm{fs} : 2^{\mathrm{Escapee}} \to 2^{\mathrm{Escapee}}\\&\mathrm{fs}(Q) = \{q \mid q \in Q \wedge \neg\exists q' \in Q.\ q \neq q' \wedge q' \subset q\}\end{aligned}$$

Here, $x_{[i_1 j_1]}@[t_1]? \subset x_{[i_1 j_1]{+}{+}[i_2 j_2]}@[t_2]?$ asserts that $x_{[i_1 j_1]}@[t_1]?$ subsumes $x_{[i_1 j_1]{+}{+}[i_2 j_2]}@[t_2]?$. Hence, fs removes all escapees which are subsumed by another escapee.

$$\begin{aligned}&\equiv_t : \mathrm{Escapee} \times \mathrm{Escapee} \to \mathbb{B}\\&x_{[ij]}@[s]? \equiv_t y_{[kl]}@[v]? :\Leftrightarrow [s]? = [v]?\end{aligned}$$

$$\begin{aligned}&\mathrm{collapse} : \mathrm{Escapee} \times \mathrm{Escapee} \rightharpoonup \mathrm{Escapee}\\&\mathrm{collapse}(x_{[i_1 j_1]}@[t]?, x_{[i_2 j_2]}@[t]?) = x_{\mathrm{lcp}([i_1 j_1],[i_2 j_2])}@[t]?\end{aligned}$$

$$\begin{aligned}&\mathrm{ct} : 2^{\mathrm{Escapee}} \rightharpoonup 2^{\mathrm{Escapee}}\\&\mathrm{ct}(Q) = \left\{\mathrm{fold}(\mathrm{collapse}, \left[x_{[ij]}@[s]\right]) \mid \left[x_{[ij]}@[s]\right] \in Q/{\equiv_t}\right\}\end{aligned}$$

The function ct is the main tool that makes our escape analysis terminate and finally uses the tags that we have been keeping track of in $[\![\cdot]\!]_Q\,(\cdot)$. The key idea is that we take escapees with the same vector of tags, i.e. equivalence classes in $\left[x_{[ij]}@[s]\right] \in Q/{\equiv_t}$, and collapse them so that we get an escapee with a field that is the longest common prefix (lcp) of all the fields of escapees in an equivalence class. Note that in such an equivalence class, as all escapees have been created from the same parameter at the same call site, all these escapees must use the same variable.

Without ct, the corresponding lattice over 2^{Escapee} does not have finite height: In an escapee $x_{[ij]}@[t]?$, we can bound x by all the variables that are possible in the program and $[t]?$ by every single call site in the program, but the field $[ij]$ may

diverge, e.g. when attempting to run the escape analysis on a recursive function with a `List`-type, in which case we will keep prepending fields to the respective escapee, and they will not subsume one another. For example, consider the following function `List.get?` for an ADT $\gamma(\text{List}) = \mu\ \text{List}.\ \ast\ \text{List}\ \mid\ [\alpha, \ast\text{List}] \to \ast\text{List}$:

```
def List.get? : List α → Nat → Option α
  | [],        _      => Option.none
  | x :: _,    0      => Option.some x
  | _ :: xs,   n + 1  => List.get? xs n
```

If we were to apply our escape analysis without ct, we would obtain an infinite set of escapees $\{\texttt{xs}_{[10]}, \texttt{xs}_{[11,10]}, \texttt{xs}_{[11,11,10]}, \ldots\}$, where 10 denotes the head of the cons constructor, and 11 denotes the tail of the cons constructor.

Collapsing the escapees with the same tag effectively bounds the lattice: Since there are only finitely many call sites, if the escape analysis is executed on a program where it would otherwise diverge, it must necessarily eventually visit the same call site twice and yield an escapee with the same variable but a different field. Collapsing all these escapees from the same call site thus computes a more general escapee that subsumes all the previous escapees from that call site, ensuring that we can iteratively reduce the field in which we were diverging up to a bound of [], where we are guaranteed to terminate. This is a form of widening [Blanchet, 2002] where both the previous and the current fixed-point iteration are part of our set and get widened together post-hoc.

In Section 5.3, we will see that this form of widening is too aggressive when analyzing the escapees of a recursive function on a recursive type.

Escapees of functions

$$\delta_Q : \text{Const} \rightharpoonup 2^{\text{Escapee}}$$

$$\delta_Q(c) = \begin{cases} \text{fd}(\texttt{[y]}, \text{fs}(\text{ct}([\![\text{F}]\!]_Q\,([\#\text{const c}])))) & c \in \text{dom}(\delta) \wedge \delta(c) = \lambda\ \texttt{[y]}.\ \texttt{F} \\ \delta_{q_e}(c) & c \notin \text{dom}(\delta) \wedge c \in \text{dom}(\delta_{q_e}) \end{cases}$$

Finally, δ_Q computes the escapees of every function in the program and uses δ_{q_e} to obtain escapees for some external functions.

4.4 Borrowing

When checking whether a parameter can be borrowed, we do not need to check whether all fields escape, only fields that are unique in an argument that is applied to the parameter. For example, if we are borrowing a value of type $\ast\text{Triple}\ [\ast\text{Array}\ [\ast\text{Array}\ [!\ \blacksquare]], \ast\text{Array}\ [!\ \blacksquare],\ !\ \blacksquare]$ to a parameter of type $!\text{Triple}\ [!\text{Array}\ [!\text{Array}\ [!\ \blacksquare]],\ !\text{Array}\ [!\ \blacksquare],\ !\ \blacksquare]$, it is acceptable if values within any of the fields typed by $!\ \blacksquare$ escape. The same is true for fields that are always shared regardless of a type parameter in the declaration of an ADT. In this section, we will

define a function $\mathbb{B}_{\cdot}(\cdot,\cdot)$ that tells us which parameters of a function can be borrowed when the function is applied. Henceforth, we will use $[x] \leq_+ [y]$ to denote that $[x]$ is a prefix of $[y]$.

4.4.1 Syntax

$$
\begin{aligned}
r_{*_e} &\in \text{ExternUniqueFieldResult} &&::= *_r \;\mid\; !_r \;\mid\; ?_r\ x^\tau\ [ij] \\
f_{*_e} &\in \text{ExternUniqueField} &&::= [ij](x^\tau)? \\
\gamma_{*_e} &\in \text{ExternUniqueFields} &&= \text{ADTConst} \rightharpoonup 2^{\text{ExternUniqueField}}
\end{aligned}
$$

For external types $A \notin \text{dom}(\gamma)$, we cannot compute which fields are unique if their outer value is unique. Obtaining this information even for external types is useful because it allows us to specify which escapees are the relevant ones for a given external type and automatically check the adherence to this specification for the escapees provided for external functions, as well as handle fields for which uniqueness depends on a type parameter.

Elements of ExternUniqueFieldResult denote the result for queries that ask whether a specific field is unique: it can be unique ($*_r$), shared ($!_r$) or its uniqueness can depend on a type parameter x^τ ($?_r\ x^\tau\ [ij]$). We will see what the auxiliary $[ij]$ is used for in the definition of eu below.

ExternUniqueField is used to specify that a specific field $[ij]$ is unique, with the caveat that its uniqueness may depend on a type parameter x^τ. We write $[ij]$ when the uniqueness of a unique field does not depend on a type parameter, $[ij](x^\tau)$ when uniqueness does depend on a type parameter and $[ij](x^\tau)?$ when uniqueness may or may not depend on a type parameter.

Finally, a global and partial function γ_{*_e} specifies the full and nonempty tree of unique fields for a given ADTConst by its leafs, i.e. there are no $[ij](x^\tau)?$, $[kl](y^\tau)? \in \gamma_{*_e}(A)$ s.t. $[ij] \neq [kl]$ but $[ij] \leq_+ [kl]$ or $[kl] \leq_+ [ij]$. The tree must include the unique fields of all dependencies. For external types $A \notin \text{dom}(\gamma_{*_e})$, we assume that all fields are unique.

4.4.2 Unique Fields

External uniqueness

$$
\begin{aligned}
&\text{eu} : \text{ADTConst} \times [\text{Ctor} \times \text{Proj}] \rightharpoonup \text{ExternUniqueFieldResult} \\
&\text{eu}(A, p) = \\
&\qquad \begin{cases}
?_r\ x^\tau\ [kl] & A \in \text{dom}(\gamma_{*_e}) \wedge \exists [ij](x^\tau) \in \gamma_{*_e}(A).\ \exists [kl].\ [ij]{+}{+}[kl] = p \\
*_r & A \in \text{dom}(\gamma_{*_e}) \wedge \exists [ij](x^\tau) \in \gamma_{*_e}(A).\ p \leq_+ [ij] \wedge p \neq [ij] \\
*_r & A \in \text{dom}(\gamma_{*_e}) \wedge \exists [ij] \in \gamma_{*_e}(A).\ p \leq_+ [ij] \vee [ij] \leq_+ p \\
!_r & A \in \text{dom}(\gamma_{*_e}) \wedge \neg\exists [ij](x^\tau)? \in \gamma_{*_e}(A).\ p \leq_+ [ij] \vee [ij] \leq_+ p \\
*_r & A \notin \text{dom}(\gamma_{*_e})
\end{cases}
\end{aligned}
$$

eu ("external unique") computes the ExternUniqueFieldResult for a given external type and a path p to a field. If the field is somewhere on the interior of the tree induced by γ_{*_e}, we assume that the field is unique. Otherwise, if the field is a leaf or points to a field within a leaf, there are two cases: Either the uniqueness of the field depends on a type parameter, in which case we yield $?_r\ x^\tau\ [kl]$ with $[kl]$ being the remaining path within the leaf, or it does not, in which case we return that the field is unique. Only if the field is not within the tree or within one of the leafs do we return that the field is shared.

Internal uniqueness

$$
\begin{aligned}
&\text{isUnique} : \text{AttrType} \times [\text{Ctor} \times \text{Proj}] \rightharpoonup \mathbb{B} \\
&\text{isUnique}(*\ \blacksquare, \text{path}) && = \top \\
&\text{isUnique}(*\ A\ [\tau_{\text{arg}}], []) && = \top \\
&\text{isUnique}(*\ A\ [\tau_{\text{arg}}], \text{path}@((i,j) :: \text{rest})) = \\
&\qquad \begin{cases}
\text{isUnique}(\gamma(A)\{A, [\tau_{\text{arg}}]\}_{ij}, \text{rest}) & A \in \text{dom}(\gamma) \\
\top & A \notin \text{dom}(\gamma) \wedge \text{eu}(A, \text{path}) = *_r \\
\bot & A \notin \text{dom}(\gamma) \wedge \text{eu}(A, \text{path}) = !_r \\
\text{isUnique}([\tau_{\text{arg}}]_{x^\tau}, [kl]) & A \notin \text{dom}(\gamma) \wedge \text{eu}(A, \text{path}) = ?_r\ x^\tau\ [kl]
\end{cases} \\
&\text{isUnique}(!\ \blacksquare, \text{path}) && = \bot \\
&\text{isUnique}(!\ A\ [\tau_{\text{arg}}], \text{path}) && = \bot \\
&\text{isUnique}(!\ [\tau_{\text{param}}] \to \tau_{\text{ret}}, \text{path}) && = \bot
\end{aligned}
$$

With isUnique, we compute whether a given field is unique in a given type. If the path points into an erased type, we assume that it is always unique, as we do not know anything about the type in question. Otherwise, if an attribute is shared, then every field within the type in question must also be shared. Finally, for ADTDecls A there are several cases: If A is an ADT, we can proceed with the field denoted in the path after substitution eliminates variables in the ADT. If it is an external type, we use the information by eu and proceed with $[kl]$ in $[\tau_{\text{arg}}]_{x^\tau}$ if the result is $?_r\ x^\tau\ [kl]$, bouncing back and forth between external types and type arguments.

$$
\begin{aligned}
&\gamma_* : \text{ADTConst} \times [\text{AttrType}] \to 2^{[\text{Ctor} \times \text{Proj}]} \\
&\gamma_*(A, [\tau_{\text{arg}}]) = \{p \mid \text{isUnique}(*\ A\ [\tau_{\text{arg}}], p) = \top\}
\end{aligned}
$$

In γ_*, we accumulate all the unique fields of a given type with a given vector of type arguments.

4.4.3 Borrowed Parameters

$$
\begin{aligned}
&\mathbb{B}_Q(x, \tau_{\text{arg}}, \tau_{\text{param}}) : 2^{\text{Escapee}} \times \text{Var} \times \text{AttrType} \times \text{AttrType} \to \mathbb{B} \\
&\mathbb{B}_Q(x, *\ A\ [\tau_{\text{arg}}], !\ A\ [\tau'_{\text{arg}}]) = \forall x_{[ij]}@[t]? \in Q.\ [ij] \notin \gamma_*(A, [\tau_{\text{arg}}]) \\
&\mathbb{B}_Q(x, *\ A\ [\tau_{\text{arg}}], !\ \blacksquare) \quad = \forall x_{[ij]}@[t]? \in Q.\ [ij] \notin \gamma_*(A, [\tau_{\text{arg}}]) \\
&\mathbb{B}_Q(x, *\ \blacksquare, !\ A\ [\tau'_{\text{arg}}]) \quad = \forall x_{[ij]}@[t]? \in Q.\ [ij] \notin \gamma_*(A, [\text{strengthen}(\tau'_{\text{arg}})]) \\
&\mathbb{B}_Q(x, *\ \blacksquare, !\ \blacksquare) \quad = \forall [ij].\ x_{[ij]}@[t]? \notin Q \\
&\mathbb{B}_Q(x, \tau_{\text{arg}}, \tau_{\text{param}}) \quad = \bot \qquad \text{otherwise}
\end{aligned}
$$

$\mathbb{B}_Q(x, \tau_{\text{arg}}, \tau_{\text{param}})$ combines our escape analysis and the information we have gathered about unique fields in order to check whether a function parameter can be borrowed. Q denotes the set of escapees belonging to the function parameter x for which we want to check whether it is borrowed. τ_{arg} denotes the type of the argument that is supplied by the caller and τ_{param} denotes the type of the function parameter. We need the type of the supplied argument because the uniqueness of a field can depend on the concrete attributes for type arguments that are set in the caller.

When the argument to the function is an ADT of type $*\ A\ [\tau_{\text{arg}}]$, we check that the parameter x itself does not escape and that none of its unique fields escape. In doing so, we use $[\tau_{\text{arg}}]$, not $[\tau'_{\text{arg}}]$, since fields that are unique because the corresponding type argument is unique cannot be allowed to escape.

If the type in the argument provided by the caller has been erased to $*\ \blacksquare$, there are two cases, depending on whether the type of the function parameter has been erased as well. If it has not been erased, we have no information about the attributes in the type arguments of the erased function argument type and must hence assume all of them to be unique, which is why we strengthen the shared type arguments $[\tau'_{\text{arg}}]$ of the function parameter type. If it has been erased, then we do not even know the name of the ADT at hand and must resort to checking whether any field whatsoever escapes.

$$
\begin{aligned}
&\mathbb{B}_\cdot(\cdot, \cdot) : \text{Const} \times \text{Var} \times \text{AttrType} \to \mathbb{B} \\
&\mathbb{B}_c(x, \tau_{\text{arg}}) = \begin{cases} \mathbb{B}_{\delta_Q(c)_x}(x, \tau_{\text{arg}}, [\tau_{\text{param}}]_x) & c \in \text{dom}(\delta_Q) \\ \varnothing & c \notin \text{dom}(\delta_Q) \end{cases} \\
&\qquad \text{where } \delta_\tau(c) = ([\tau_{\text{param}}], \tau_{\text{ret}}) \\
&\qquad \text{and } \delta_Q(c)_x := \{ y_{[ij]}@[t]? \mid y_{[ij]}@[t]? \in \delta_Q(c) \land y = x \}
\end{aligned}
$$

With $\mathbb{B}_\cdot(\cdot, \cdot)$, we obtain a total function that tells us which parameters can be borrowed when applied with a type τ_{arg} for each function in the program. If we have no escape information for a function, then no parameter can be borrowed.

4.5 Type-Checking

In this section we will finally define the rules of our type theory.

4.5.1 Syntax

$$\begin{aligned} Z &\in \text{ZeroedFields} &&= \text{Var} \times \text{Ctor} \times \text{Proj} \to \mathbb{B} \\ \Gamma &\in \text{Context} &&::= [\,] \mid \Gamma, x : \tau \end{aligned}$$

ZeroedFields will be used for the mechanism described in the discussion of the `case'` instruction in Section 4.2. As we want to enable the use of multiple projections on the same variable within a function, we must track which fields have already been projected, disallow repeated projections of the same field and disallow using the variable in any way other than projecting from it. $Z \in \text{ZeroedFields}$ will be used to track this information. For convenience we also declare the following functions that check whether a variable has no zeroed fields and zero a field if the corresponding inner attribute is unique.

$$\begin{aligned} &\text{nz} : \text{ZeroedFields} \times \text{Var} \to \mathbb{B} \\ &\text{nz}(Z, x) = \neg \exists i\ j.\ Z(x, i, j) = \top \end{aligned}$$

$$\begin{aligned} &\text{zero} : \text{ZeroedFields} \times \text{Attr} \times \text{Var} \times \text{Ctor} \times \text{Proj} \to \text{ZeroedFields} \\ &\text{zero}(Z, m, x, i, j) = \begin{cases} Z[(x, i, j) \mapsto \top] & m = * \\ Z & m = ! \end{cases} \end{aligned}$$

We assume Γ to be a multiset, i.e. we track duplicate judgements, but not the order of the context. Note that the latter would be required in dependent type theory, as the order of type dependencies must be retained.

4.5.2 Type Theory

In the following, we will progressively introduce the rules of our type theory and explain them along the way. Whenever a variable `x` is used in any meaningful way other than projection, we demand $\text{nz}(Z, \texttt{x})$ so that it cannot be used if any field has been projected in the past.

Programs

$$\boxed{\vdash \delta_\tau} \qquad \frac{\begin{array}{l}\textsc{Program} \\ \forall c \in \text{dom}(\delta_\tau) \cap \text{dom}(\delta) \text{ s.t. } \delta(c) = \lambda\ [\texttt{y}].\ \texttt{F} \wedge \delta_\tau(c) = ([\tau_{\text{param}}], \tau_{\text{ret}}). \\ \qquad \varnothing; [\texttt{y} : \tau_{\text{param}}] \vdash \texttt{F} : \tau_{\text{ret}} \end{array}}{\vdash \delta_\tau}$$

The PROGRAM rule states that in order to check a program δ_τ, we check that each function $c \in \mathrm{dom}(\delta)$ adheres to its function type.

Substructurality

$$\boxed{Z;\Gamma \vdash \mathtt{F} : \tau}$$

$$\textsc{Duplicate}\quad \frac{Z;\Gamma,\mathtt{x} : !\tau, \mathtt{x} : !\tau \vdash \mathtt{F} : \tau_{\text{ret'}}}{Z;\Gamma,\mathtt{x} : !\tau \vdash \mathtt{F} : \tau_{\text{ret'}}} \qquad \textsc{Forget}\quad \frac{Z;\Gamma \vdash \mathtt{F} : \tau_{\text{ret'}}}{Z;\Gamma,\mathtt{x} : !\tau \vdash \mathtt{F} : \tau_{\text{ret'}}}$$

$$\textsc{Cast}\quad \frac{\tau \succcurlyeq \tau' \qquad \mathrm{nz}(Z,\mathtt{x}) \qquad Z;\Gamma,\mathtt{x} : \tau' \vdash \mathtt{F} : \tau_{\text{ret'}}}{Z;\Gamma,\mathtt{x} : \tau \vdash \mathtt{F} : \tau_{\text{ret'}}}$$

DUPLICATE and FORGET allow manipulating variables of shared type in the context as if the context was a set and structural. Note that $!\tau$ is an application of the weaken function defined in Section 4.1 and that unique variables cannot be manipulated in this manner; their exact amount in the context needs to be tracked. The CAST rule allows applying the subtyping relation also defined in Section 4.1.

Erasure

$$\blacksquare\text{-}\textsc{Cast}\quad \frac{Z;\Gamma,\mathtt{x} : \mathrm{weakenInner}(\tau) \vdash \mathtt{F} : \tau_{\text{ret'}}}{Z;\Gamma,\mathtt{x} : \mathrm{attr}(\tau)\ \blacksquare \vdash \mathtt{F} : \tau_{\text{ret'}}} \qquad \blacksquare\text{-}\textsc{Erase}\quad \frac{Z;\Gamma,\mathtt{x} : \mathrm{attr}(\tau)\ \blacksquare \vdash \mathtt{F} : \tau_{\text{ret'}}}{Z;\Gamma,\mathtt{x} : \tau \vdash \mathtt{F} : \tau_{\text{ret'}}}$$

■-CAST and ■-ERASE enable us to work with erased types: We can cast to and from any type while retaining the outer attribute, but have to make all the inner attributes shared in the process.

Control flow

$$\textsc{Ret}\quad \frac{\mathrm{nz}(Z,\mathtt{x})}{Z;\Gamma,\mathtt{x} : \tau_{\text{ret'}} \vdash \mathtt{ret\ \ x} : \tau_{\text{ret'}}} \qquad \textsc{Case}\quad \frac{\mathrm{nz}(Z,\mathtt{x}) \qquad [Z;\Gamma,\mathtt{x} : \tau \vdash \mathtt{F} : \tau_{\text{ret'}}]}{Z;\Gamma,\mathtt{x} : \tau \vdash \mathtt{case\ \ x\ \ of\ \ [F]} : \tau_{\text{ret'}}}$$

The RET and CASE rules are straight-forward: For RET, we need a matching variable with a matching type in our context, and for CASE, we check every branch. It is worth pointing out that in RET, there is no issue with throwing away the rest of the context Γ, as all variables can always be made shared and then discarded using WEAKEN, and that in CASE, `x` does not need to be consumed as `case` is read-only.

$$\textsc{Case'-!}\quad \frac{\begin{array}{c}\mathrm{nz}(Z,\mathtt{x}) \qquad \mathtt{A} \in \mathrm{dom}(\gamma) \\ \gamma(\mathtt{A})\{\mathtt{A},[\tau_{\text{arg}}]\} = \mu\ x^{\kappa}_{\text{adt}}.\ [[\tau_{\text{field}}] \to *x^{\kappa}_{\text{adt}}] \qquad [Z;\Gamma,[\mathtt{y} : !\tau_{\text{field}}] \vdash \mathtt{F} : \tau_{\text{ret'}}]\end{array}}{Z;\Gamma,\mathtt{x} : !\ \mathtt{A}\ [\tau_{\text{arg}}] \vdash \mathtt{A.case'\ \ x\ \ of\ \ [ctor_i\ \ [y]\ \ \Rightarrow\ \ F]} : \tau_{\text{ret'}}}$$

CASE'-$*$

$$\frac{\mathrm{nz}(Z,\mathtt{x}) \qquad \mathtt{A} \in \mathrm{dom}(\gamma) \qquad \gamma(\mathtt{A})\{\mathtt{A},[\tau_{\mathrm{arg}}]\} = \mu\ x^{\kappa}_{\mathrm{adt}}.\ [[\tau_{\mathrm{field}}] \to *x^{\kappa}_{\mathrm{adt}}] \qquad [Z;\Gamma,[\mathtt{y}:\tau_{\mathrm{field}}] \vdash \mathtt{F}:\tau_{\mathrm{ret'}}]}{Z;\Gamma,\mathtt{x}:*\ \mathtt{A}\ [\tau_{\mathrm{arg}}] \vdash \mathtt{A.case'\ x\ of\ [ctor_i\ [y]\ \Rightarrow\ F]} : \tau_{\mathrm{ret'}}}$$

The CASE'-rules work similar to CASE, except that the variable that is being matched on is consumed and that we need to add the variables associated with the constructor in a specific branch to the context. If the value we are matching on is shared, then the newly created variables must be shared as well. This is the essence of $*$ in ADTs meaning "unique if the outer value is unique".

Application

LET-APP

$$\frac{\begin{array}{c}[\mathrm{nz}(Z,\mathtt{y})] \qquad \delta_\tau(\mathtt{c}) = ([\tau'_{\mathrm{param}}],\tau_{\mathrm{ret}}) \\ \forall x \text{ s.t. } \mathbb{B}_{\mathtt{c}}(x,[\tau_{\mathrm{param}}]_x) = \top.\ ![\tau_{\mathrm{param}}]_x \rightsquigarrow_{\blacksquare} [\tau'_{\mathrm{param}}]_x \\ \forall x \text{ s.t. } \mathbb{B}_{\mathtt{c}}(x,[\tau_{\mathrm{param}}]_x) = \bot.\ [\tau_{\mathrm{param}}]_x \rightsquigarrow_{\blacksquare} [\tau'_{\mathrm{param}}]_x \\ Z;\Gamma,\{[\mathtt{y}:\tau_{\mathrm{param}}]_x \mid \mathbb{B}_{\mathtt{c}}(x,[\tau_{\mathrm{param}}]_x) = \top\},\mathtt{z}:\tau_{\mathrm{ret}} \vdash \mathtt{F}:\tau_{\mathrm{ret'}}\end{array}}{Z;\Gamma,[\mathtt{y}:\tau_{\mathrm{param}}] \vdash \mathtt{let\ z\ :=\ c\ [y];\ F}:\tau_{\mathrm{ret'}}}$$

In LET-APP, our argument types need to match the parameter types: Borrowed arguments are allowed to have arbitrary attributes, while non-borrowed arguments need to match exactly. Since $\mathbb{B}_{\mathtt{c}}$ benefits from having more type information available, we allow the application of the erasure rules within LET-APP instead of requiring its application beforehand, using the relation $\rightsquigarrow_{\blacksquare}$:

EQUAL $\dfrac{}{\tau \rightsquigarrow_{\blacksquare} \tau}$ ERASE $\dfrac{\mathrm{attr}(\tau) = m}{\tau \rightsquigarrow_{\blacksquare} m\ \blacksquare}$ CAST $\dfrac{\mathrm{attr}(\tau) = m}{m\ \blacksquare \rightsquigarrow_{\blacksquare} \mathrm{weakenInner}(\tau)}$

We obtain the result of the function call in our new context. Non-borrowed arguments are consumed, borrowed ones are retained.

LET-PAP-FULL

$$\frac{[\mathrm{nz}(Z,\mathtt{y})] \qquad \delta_\tau(\mathtt{c}) = ([\tau_{\mathrm{param}}],\tau_{\mathrm{ret}}) \qquad |[\mathtt{y}]| = |[\tau_{\mathrm{param}}]| \qquad Z;\Gamma,z:!\tau_{\mathrm{ret}} \vdash \mathtt{F}:\tau_{\mathrm{ret'}}}{Z;\Gamma,[\mathtt{y}:!\tau_{\mathrm{param}}] \vdash \mathtt{let\ z\ :=\ pap\ c\ [y];\ F}:\tau_{\mathrm{ret'}}}$$

LET-PAP-PART

$$\frac{\begin{array}{c}[\mathrm{nz}(Z,\mathtt{y})] \qquad \delta_\tau(\mathtt{c}) = ([\tau_{\mathrm{param}}],\tau_{\mathrm{ret}}) \qquad |[\mathtt{y}]| = |[\tau_{\mathrm{param}_1}]| < |[\tau_{\mathrm{param}}]| \\ [\tau_{\mathrm{param}_1}]{+}{+}[\tau_{\mathrm{param}_2}] = [\tau_{\mathrm{param}}] \qquad Z;\Gamma,\mathtt{z}:!\ [\tau_{\mathrm{param}_2}] \to \tau_{\mathrm{ret}} \vdash \mathtt{F}:\tau_{\mathrm{ret'}}\end{array}}{Z;\Gamma,[\mathtt{y}:!\tau_{\mathrm{param}_1}] \vdash \mathtt{let\ z\ :=\ pap\ c\ [y];\ F}:\tau_{\mathrm{ret'}}}$$

For `pap`, there are two separate rules, one for a `pap` call that is effectively a full application, and one for a proper partial application. As our higher-order functions are always shared, our context needs to contain matching shared arguments that are consumed in the process. Depending on the rule, we either obtain a new higher-order function or the shared return value of the function. Note that when leveraging

uniqueness for destructive updates in an implementation, we need to ensure that the function yielded by `pap c [y]` does not rely on the uniqueness of its arguments and must generate a variant of `c` that does not rely on uniqueness instead.

$$\dfrac{\text{nz}(Z, \texttt{y}) \qquad Z; \Gamma, \texttt{z} : !\tau_{\text{ret}} \vdash \texttt{F} : \tau_{\text{ret'}}}{Z; \Gamma, \texttt{x} : !\ \tau_{\text{param}} \to \tau_{\text{ret}}, \texttt{y} : !\tau_{\text{param}} \vdash \texttt{let z := x y; F} : \tau_{\text{ret'}}}\ \text{Let-VarApp-Full}$$

$$\dfrac{\text{nz}(Z, \texttt{y}) \qquad |[\tau_{\text{param'}}]| \geq 1 \qquad Z; \Gamma, \texttt{z} : !\ [\tau_{\text{param'}}] \to \tau_{\text{ret}} \vdash \texttt{F} : \tau_{\text{ret'}}}{Z; \Gamma, \texttt{x} : !\ (\tau_{\text{param}} :: [\tau_{\text{param'}}]) \to \tau_{\text{ret}}, \texttt{y} : !\tau_{\text{param}} \vdash \texttt{let z := x y; F} : \tau_{\text{ret'}}}\ \text{Let-VarApp-Part}$$

For Let-VarApp, there is a similar split as for Let-Pap: Depending on whether we have applied all the arguments of a higher-order function, we either get a new higher-order function with the argument applied or the shared return value of the function.

Construction and projection

$$\dfrac{[\text{nz}(Z, \texttt{y})] \qquad \gamma(\texttt{A})\{\texttt{A}, [\tau'_{\text{arg}}]\}_{\texttt{i}} = [\tau] \to *x^{\kappa}_{\text{adt}} \qquad Z; \Gamma, \texttt{z} : *\ \texttt{A}\ [\tau'_{\text{arg}}] \vdash \texttt{F} : \tau_{\text{ret'}}}{Z; \Gamma, [\texttt{y} : \tau] \vdash \texttt{let x = (A } [\tau_{\text{arg}}?]\texttt{).ctor}_{\text{i}}\ \texttt{[y]; F} : \tau_{\text{ret'}}}\ \text{Let-Ctor-}*$$

$$\dfrac{[\text{nz}(Z, \texttt{y})] \qquad \gamma(\texttt{A})\{\texttt{A}, [!\tau'_{\text{arg}}]\}_{\texttt{i}} = [\tau] \to *x^{\kappa}_{\text{adt}} \qquad Z; \Gamma, \texttt{z} : !\ \texttt{A}\ [!\tau'_{\text{arg}}] \vdash \texttt{F} : \tau_{\text{ret'}}}{Z; \Gamma, [\texttt{y} : !\tau] \vdash \texttt{let x = (A } [\tau_{\text{arg}}?]\texttt{).ctor}_{\text{i}}\ \texttt{[y]; F} : \tau_{\text{ret'}}}\ \text{Let-Ctor-!}$$

Let-Ctor-$*$ and Let-Ctor-! create a unique and shared value respectively. Let-Ctor-! is needed because for a value that is only used in a shared manner, we do not need to demand unique constructor arguments. Note that in these rules, $[\tau_{\text{arg}}?]$ is ignored, it is not clear how we must choose $[\tau'_{\text{arg}}]$ and whether we should apply Let-Ctor-$*$ or Let-Ctor-! for typing to succeed. We will resolve this in Section 5.2.

$$\dfrac{\begin{array}{c} Z(\texttt{y}, \texttt{i}, \texttt{j}) = \bot \qquad \tau_{\text{field}} = \gamma(\texttt{A})\{\texttt{A}, [\tau_{\text{arg}}]\}_{\texttt{ij}} \\ \text{zero}(Z, \text{attr}(\tau_{\text{field}}), \texttt{y}, \texttt{i}, \texttt{j}); \Gamma, \texttt{y} : *\ \texttt{A}\ [\tau_{\text{arg}}], \texttt{z} : \tau_{\text{field}} \vdash \texttt{F} : \tau_{\text{ret'}} \end{array}}{Z; \Gamma, \texttt{y} : *\ \texttt{A}\ [\tau_{\text{arg}}] \vdash \texttt{let z = A.proj}_{\text{ij}}\ \texttt{y; F} : \tau_{\text{ret'}}}\ \text{Let-Proj-}*$$

$$\dfrac{Z(\texttt{y}, \texttt{i}, \texttt{j}) = \bot \qquad Z; \Gamma, \texttt{y} : !\ \texttt{A}\ [\tau_{\text{arg}}], \texttt{z} : !\gamma(\texttt{A})\{\texttt{A}, [\tau_{\text{arg}}]\}_{\texttt{ij}} \vdash \texttt{F} : \tau_{\text{ret'}}}{Z; \Gamma, \texttt{y} : !\ \texttt{A}\ [\tau_{\text{arg}}] \vdash \texttt{let z = A.proj}_{\text{ij}}\ \texttt{y; F} : \tau_{\text{ret'}}}\ \text{Let-Proj-!}$$

Finally, our Let-Proj rules are only applicable if the specific field has not been projected yet and retain the variable that we project from. We also obtain the field in our new context. If the ADT is shared, then we need to make the new field shared as well, but do not have to zero the projected field, whereas otherwise, we obtain the field with its actual attribute, but have to zero it.

$$\boxed{\vdash \delta_\tau} \quad \textsc{Program}\ \frac{\forall c \in \mathrm{dom}(\delta_\tau) \cap \mathrm{dom}(\delta) \text{ s.t. } \delta(c) = \lambda\ \texttt{[y]}.\ \texttt{F} \wedge \delta_\tau(c) = ([\tau_{\text{param}}], \tau_{\text{ret}}).\quad \varnothing; [\texttt{y} : \tau_{\text{param}}] \vdash \texttt{F} : \tau_{\text{ret}}}{\vdash \delta_\tau}$$

$$\boxed{Z; \Gamma \vdash \texttt{F} : \tau}$$

$$\textsc{Duplicate}\ \frac{Z; \Gamma, \texttt{x} : !\tau, \texttt{x} : !\tau \vdash \texttt{F} : \tau_{\text{ret'}}}{Z; \Gamma, \texttt{x} : !\tau \vdash \texttt{F} : \tau_{\text{ret'}}} \qquad \textsc{Forget}\ \frac{Z; \Gamma \vdash \texttt{F} : \tau_{\text{ret'}}}{Z; \Gamma, \texttt{x} : !\tau \vdash \texttt{F} : \tau_{\text{ret'}}} \qquad \textsc{Cast}\ \frac{\tau \succcurlyeq \tau' \quad \mathrm{nz}(Z, \texttt{x}) \quad Z; \Gamma, \texttt{x} : \tau' \vdash \texttt{F} : \tau_{\text{ret'}}}{Z; \Gamma, \texttt{x} : \tau \vdash \texttt{F} : \tau_{\text{ret'}}}$$

$$\blacksquare\textsc{-Cast}\ \frac{Z; \Gamma, \texttt{x} : \mathrm{weakenInner}(\tau) \vdash \texttt{F} : \tau_{\text{ret'}}}{Z; \Gamma, \texttt{x} : \mathrm{attr}(\tau)\ \blacksquare \vdash \texttt{F} : \tau_{\text{ret'}}} \qquad \blacksquare\textsc{-Erase}\ \frac{Z; \Gamma, \texttt{x} : \mathrm{attr}(\tau)\ \blacksquare \vdash \texttt{F} : \tau_{\text{ret'}}}{Z; \Gamma, \texttt{x} : \tau \vdash \texttt{F} : \tau_{\text{ret'}}}$$

$$\textsc{Ret}\ \frac{\mathrm{nz}(Z, \texttt{x})}{Z; \Gamma, \texttt{x} : \tau_{\text{ret'}} \vdash \texttt{ret}\ \ \texttt{x} : \tau_{\text{ret'}}} \qquad \textsc{Case}\ \frac{\mathrm{nz}(Z, \texttt{x}) \quad [Z; \Gamma, \texttt{x} : \tau \vdash \texttt{F} : \tau_{\text{ret'}}]}{Z; \Gamma, \texttt{x} : \tau \vdash \texttt{case}\ \ \texttt{x}\ \ \texttt{of}\ \ \texttt{[F]} : \tau_{\text{ret'}}}$$

$$\textsc{Case'-!}\ \frac{\mathrm{nz}(Z, \texttt{x}) \quad \texttt{A} \in \mathrm{dom}(\gamma) \quad \gamma(\texttt{A})\{\texttt{A}, [\tau_{\text{arg}}]\} = \mu\ x^{\kappa}_{\text{adt}}.\ [[\tau_{\text{field}}] \to *x^{\kappa}_{\text{adt}}] \quad [Z; \Gamma, [\texttt{y} : !\tau_{\text{field}}] \vdash \texttt{F} : \tau_{\text{ret'}}]}{Z; \Gamma, \texttt{x} : !\ \texttt{A}\ [\tau_{\text{arg}}] \vdash \texttt{A.case'}\ \ \texttt{x}\ \ \texttt{of}\ \ \texttt{[ctor}_{\texttt{i}}\ \ \texttt{[y]}\ \ \Rightarrow\ \texttt{F]} : \tau_{\text{ret'}}}$$

$$\textsc{Case'-}*\ \frac{\mathrm{nz}(Z, \texttt{x}) \quad \texttt{A} \in \mathrm{dom}(\gamma) \quad \gamma(\texttt{A})\{\texttt{A}, [\tau_{\text{arg}}]\} = \mu\ x^{\kappa}_{\text{adt}}.\ [[\tau_{\text{field}}] \to *x^{\kappa}_{\text{adt}}] \quad [Z; \Gamma, [\texttt{y} : \tau_{\text{field}}] \vdash \texttt{F} : \tau_{\text{ret'}}]}{Z; \Gamma, \texttt{x} : *\ \texttt{A}\ [\tau_{\text{arg}}] \vdash \texttt{A.case'}\ \ \texttt{x}\ \ \texttt{of}\ \ \texttt{[ctor}_{\texttt{i}}\ \ \texttt{[y]}\ \ \Rightarrow\ \texttt{F]} : \tau_{\text{ret'}}}$$

$$\textsc{Let-App}\ \frac{\begin{array}{c}[\mathrm{nz}(Z, \texttt{y})] \quad \delta_\tau(\texttt{c}) = ([\tau'_{\text{param}}], \tau_{\text{ret}}) \\ \forall x \text{ s.t. } \mathbb{B}_{\texttt{c}}(x, [\tau_{\text{param}}]_x) = \top.\ ![\tau_{\text{param}}]_x \rightsquigarrow_{\blacksquare} [\tau'_{\text{param}}]_x \\ \forall x \text{ s.t. } \mathbb{B}_{\texttt{c}}(x, [\tau_{\text{param}}]_x) = \bot.\ [\tau_{\text{param}}]_x \rightsquigarrow_{\blacksquare} [\tau'_{\text{param}}]_x \\ Z; \Gamma, \{[\texttt{y} : \tau_{\text{param}}]_x \mid \mathbb{B}_{\texttt{c}}(x, [\tau_{\text{param}}]_x) = \top\}, \texttt{z} : \tau_{\text{ret}} \vdash \texttt{F} : \tau_{\text{ret'}}\end{array}}{Z; \Gamma, [\texttt{y} : \tau_{\text{param}}] \vdash \texttt{let}\ \ \texttt{z}\ \ \texttt{:=}\ \ \texttt{c}\ \ \texttt{[y]};\ \ \texttt{F} : \tau_{\text{ret'}}}$$

$$\textsc{Let-Pap-Full}\ \frac{[\mathrm{nz}(Z, \texttt{y})] \quad \delta_\tau(\texttt{c}) = ([\tau_{\text{param}}], \tau_{\text{ret}}) \quad |\texttt{[y]}| = |[\tau_{\text{param}}]| \quad Z; \Gamma, z : !\tau_{\text{ret}} \vdash \texttt{F} : \tau_{\text{ret'}}}{Z; \Gamma, [\texttt{y} : !\tau_{\text{param}}] \vdash \texttt{let}\ \ \texttt{z}\ \ \texttt{:=}\ \ \texttt{pap}\ \ \texttt{c}\ \ \texttt{[y]};\ \ \texttt{F} : \tau_{\text{ret'}}}$$

$$\frac{\begin{array}{c}[\mathrm{nz}(Z,\mathtt{y})] \quad \delta_\tau(\mathtt{c}) = ([\tau_{\text{param}}],\tau_{\text{ret}}) \quad |[\mathtt{y}]| = |[\tau_{\text{param}_1}]| < |[\tau_{\text{param}}]| \\ [\tau_{\text{param}_1}]{+}{+}[\tau_{\text{param}_2}] = [\tau_{\text{param}}] \quad Z;\Gamma,\mathtt{z} : !\ [\tau_{\text{param}_2}] \to \tau_{\text{ret}} \vdash \mathtt{F} : \tau_{\text{ret'}}\end{array}}{Z;\Gamma,[\mathtt{y} : !\tau_{\text{param}_1}] \vdash \mathtt{let\ z\ :=\ pap\ c\ [y];\ F} : \tau_{\text{ret'}}}\ \text{LET-PAP-PART}$$

$$\frac{\mathrm{nz}(Z,\mathtt{y}) \quad Z;\Gamma,\mathtt{z} : !\tau_{\text{ret}} \vdash \mathtt{F} : \tau_{\text{ret'}}}{Z;\Gamma,\mathtt{x} : !\ \tau_{\text{param}} \to \tau_{\text{ret}},\mathtt{y} : !\tau_{\text{param}} \vdash \mathtt{let\ z\ :=\ x\ y;\ F} : \tau_{\text{ret'}}}\ \text{LET-VARAPP-FULL}$$

$$\frac{\mathrm{nz}(Z,\mathtt{y}) \quad |[\tau_{\text{param'}}]| \geq 1 \quad Z;\Gamma,\mathtt{z} : !\ [\tau_{\text{param'}}] \to \tau_{\text{ret}} \vdash \mathtt{F} : \tau_{\text{ret'}}}{Z;\Gamma,\mathtt{x} : !\ (\tau_{\text{param}} :: [\tau_{\text{param'}}]) \to \tau_{\text{ret}},\mathtt{y} : !\tau_{\text{param}} \vdash \mathtt{let\ z\ :=\ x\ y;\ F} : \tau_{\text{ret'}}}\ \text{LET-VARAPP-PART}$$

$$\frac{[\mathrm{nz}(Z,\mathtt{y})] \quad \gamma(\mathtt{A})\{\mathtt{A},[\tau'_{\text{arg}}]\}_{\mathtt{i}} = [\tau] \to *x^{\kappa}_{\text{adt}} \quad Z;\Gamma,\mathtt{z} : *\ \mathtt{A}\ [\tau'_{\text{arg}}] \vdash \mathtt{F} : \tau_{\text{ret'}}}{Z;\Gamma,[\mathtt{y} : \tau] \vdash \mathtt{let\ x\ =\ (A\ }[\tau_{\text{arg}}?]\mathtt{).ctor}_{\mathrm{i}}\ \mathtt{[y];\ F} : \tau_{\text{ret'}}}\ \text{LET-CTOR-}*$$

$$\frac{[\mathrm{nz}(Z,\mathtt{y})] \quad \gamma(\mathtt{A})\{\mathtt{A},[!\tau'_{\text{arg}}]\}_{\mathtt{i}} = [\tau] \to *x^{\kappa}_{\text{adt}} \quad Z;\Gamma,\mathtt{z} : !\ \mathtt{A}\ [!\tau'_{\text{arg}}] \vdash \mathtt{F} : \tau_{\text{ret'}}}{Z;\Gamma,[\mathtt{y} : !\tau] \vdash \mathtt{let\ x\ =\ (A\ }[\tau_{\text{arg}}?]\mathtt{).ctor}_{\mathrm{i}}\ \mathtt{[y];\ F} : \tau_{\text{ret'}}}\ \text{LET-CTOR-!}$$

$$\frac{\begin{array}{c}Z(\mathtt{y},\mathtt{i},\mathtt{j}) = \bot \quad \tau_{\text{field}} = \gamma(\mathtt{A})\{\mathtt{A},[\tau_{\text{arg}}]\}_{\mathtt{ij}} \\ \mathrm{zero}(Z,\mathrm{attr}(\tau_{\text{field}}),\mathtt{y},\mathtt{i},\mathtt{j});\Gamma,\mathtt{y} : *\ \mathtt{A}\ [\tau_{\text{arg}}],\mathtt{z} : \tau_{\text{field}} \vdash \mathtt{F} : \tau_{\text{ret'}}\end{array}}{Z;\Gamma,\mathtt{y} : *\ \mathtt{A}\ [\tau_{\text{arg}}] \vdash \mathtt{let\ z\ =\ A.proj}_{\text{ij}}\ \mathtt{y;\ F} : \tau_{\text{ret'}}}\ \text{LET-PROJ-}*$$

$$\frac{Z(\mathtt{y},\mathtt{i},\mathtt{j}) = \bot \quad Z;\Gamma,\mathtt{y} : !\ \mathtt{A}\ [\tau_{\text{arg}}],\mathtt{z} : !\gamma(\mathtt{A})\{\mathtt{A},[\tau_{\text{arg}}]\}_{\mathtt{ij}} \vdash \mathtt{F} : \tau_{\text{ret'}}}{Z;\Gamma,\mathtt{y} : !\ \mathtt{A}\ [\tau_{\text{arg}}] \vdash \mathtt{let\ z\ =\ A.proj}_{\text{ij}}\ \mathtt{y;\ F} : \tau_{\text{ret'}}}\ \text{LET-PROJ-!}$$

Figure 4.1: All typing rules of our type system at a glance.

5 Implementation

We have implemented a type checker for the type theory described in Chapter 4 using Lean 4 at `https://github.com/mhuisi/Uniq`, targeting the model IR from Section 4.2. Note that we have not yet integrated our checker into Lean 4 itself. Unless otherwise stated below, our implementation follows the formal description in Chapter 4.

5.1 Deviations From Specification

To represent the global and partial functions γ, δ_τ, δ, δ_{q_e}, γ_{*_e}, as well as δ_Q, we use a map type based on red-black trees. On the other hand, the function γ_* that yields unique fields is not represented at all because the codomain is usually infinite, and instead we use isUnique for every escapee in isBorrowed directly.

In order to perform the data flow analysis in Section 4.3, we use Kosaraju's algorithm [Sharir, 1981] to compute strongly connected components in the call graph of functions $c \in \mathrm{dom}(\delta)$, traverse the resulting graph of strongly-connected components in reverse topological sort and then iteratively compute $\delta_Q(c)$ within each strongly connected component of mutually recursive functions until we reach a fixed point. Kosaraju's algorithm seems to be more suited to compute strongly-connected components in a functional language because it needs to maintain less state than Tarjan's algorithm [Tarjan, 1972].

We implement the type checker for the type theory in Section 4.5 as follows:

- $Z \in$ ZeroedFields is implemented as a red-black tree based set and Γ as a red-black tree based map from Var to AttrType, not a multiset. Since we know that variables are either unique or shared, all variables in Γ can either be duplicated and forgotten freely, or there is only one of them in the context. Hence, we do not need to track the exact amount of each variable in the context, only whether it is unique or shared.

- The type rules are implemented by matching on the program and applying the corresponding unique rule.

- When checking whether a type τ_1 is applicable to a type τ_2 in any rule, we do not check for equality, but instead a relation $\tau_1 \rightsquigarrow \tau_2$ that determines whether τ_1 can be made equal to τ_2 after applying the Cast, ■-Cast and ■-Erase rules.

- When using Cast, ■-Cast and ■-Erase, it is important that we replace the old type with the new one, lest we could retain a unique attribute in our context after it has already been made shared. Hence, after checking whether $\tau_1 \rightsquigarrow \tau_2$ for $x : \tau_1 \in \Gamma$ in a rule, we set $\Gamma[x \mapsto \tau_2]$ to update the context. This can be understood as applying Cast, ■-Cast and ■-Erase as needed just before the application of the rule.

- When encountering an erased type in one of the Case' or Let-Proj rules where we have no expected type to convert to using $\tau_1 \rightsquigarrow \tau_2$, we use the ADT designated in the instruction and propagate the erasedness to the type arguments of the ADT. Additionally, we assume each type parameter to be shared, so as to correctly implement the ■-Cast rule.

- To eliminate Duplicate and Forget, we integrate their application carefully into the other rules by not consuming shared variables in rules that would otherwise consume them.

- We infer the constructor type arguments and the constructor return type to decide how to apply Let-Ctor-$*$ and Let-Ctor-! according to our description below in Section 5.2.

- When applying a vector of arguments with types $[\tau_1]$ to parameters of types $[\tau_2]$ for $[x : \tau_1] \subseteq \Gamma$, we apply Cast, ■-Cast and ■-Erase while consuming unique variables and updating the context iteratively, so that e.g. `c x x` fails for $\mathtt{x} : * \blacksquare$ and $\mathtt{c} : * \blacksquare \to * \blacksquare \to \ldots$ or $\mathtt{c} : ! \blacksquare \to * \blacksquare \to \ldots$.

5.2 Constructor Type Inference

In this section, we will discuss methods of inference in order to decide how and when to apply the Let-Ctor-$*$ and Let-Ctor-! rules in Section 4.5. All methods discussed here are approximations, i.e. the Let-Ctor-$*$ and Let-Ctor-! rules defined here using inference can lead to a type error when the corresponding rules in Section 4.5 will not. However, errors of this kind should be easy to understand and easy to resolve locally in user code.

Type arguments

As discussed briefly in the description of the `ctor` instruction in Section 4.2, we must infer the attributes in a `(A [τ?]).ctor`$_i$ `[y]` call since Lean cannot provide them and because we need them to decide how to apply the Let-Ctor-$*$ and Let-Ctor-! rules in Section 4.5.2.

We will do so in two steps: First, we assign types to type variables based on the types of the arguments provided in the context Γ for `[y]`. Then, we use the explicit type arguments provided by the user in `[τ?]` to fill the remaining variables that could not be inferred and choose their attributes as strongly as possible, since type

arguments that could not be inferred from the types of `[y]` are also not assigned by the types of `[y]`.

Together, this ensures that no attributes for type arguments must be provided explicitly by the user, as they can either be inferred or chosen as strongly as possible.

$$f \sqcup g : (M \rightharpoonup N) \times (M \rightharpoonup N) \rightharpoonup (\mathrm{dom}(f) \cup \mathrm{dom}(g) \rightharpoonup N)$$

$$(f \sqcup g)(x) = \begin{cases} f(x) & x \in \mathrm{dom}(f) \wedge x \notin \mathrm{dom}(g) \\ g(x) & x \notin \mathrm{dom}(f) \wedge x \in \mathrm{dom}(g) \\ f(x) & x \in \mathrm{dom}(f) \cap \mathrm{dom}(g) \wedge f(x) = g(x) \end{cases}$$

Note that $(f, g) \notin \mathrm{dom}(\cdot \sqcup \cdot)$ if $\exists x \in \mathrm{dom}(f) \cap \mathrm{dom}(g).\ f(x) \neq g(x)$.

$$\begin{aligned}
&\mathrm{inferVars}' : \mathrm{AttrType} \times \mathrm{AttrType} \rightharpoonup (\mathrm{Var} \rightharpoonup \mathrm{AttrType}) \\
&\mathrm{inferVars}'(m\ x^{\kappa}, \tau) && = \varnothing \\
&\mathrm{inferVars}'(x^{\tau}, \tau) && = \{x^{\tau} \mapsto \tau\} \\
&\mathrm{inferVars}'(m_1\ \blacksquare, m_2\ \blacksquare) && = \varnothing \\
&\mathrm{inferVars}'(m_1\ A\ [\tau_{\mathrm{arg}_1}], m_2\ A\ [\tau_{\mathrm{arg}_2}]) && = \bigsqcup_i \mathrm{inferVars}'([\tau_{\mathrm{arg}_1}]_i, [\tau_{\mathrm{arg}_2}]_i) \\
&\mathrm{inferVars}'(!\ [\tau_{\mathrm{param}_1}] \to \tau_{\mathrm{ret}_1}, !\ [\tau_{\mathrm{param}_2}] \to \tau_{\mathrm{ret}_2}) = \\
&\qquad \bigsqcup_i \mathrm{inferVars}'([\tau_{\mathrm{param}_1}]_i, [\tau_{\mathrm{param}_2}]_i) \sqcup \mathrm{inferVars}'(\tau_{\mathrm{ret}_1}, \tau_{\mathrm{ret}_2})
\end{aligned}$$

$$\begin{aligned}
&\mathrm{inferVars} : [\mathrm{AttrType}] \times [\mathrm{AttrType}] \rightharpoonup (\mathrm{Var} \rightharpoonup \mathrm{AttrType}) \\
&\mathrm{inferVars}([], []) && = \varnothing \\
&\mathrm{inferVars}(\tau_1 :: \mathrm{rest}_1, \tau_2 :: \mathrm{rest}_2) && = \mathrm{inferVars}'(\tau_1, \tau_2) \sqcup \mathrm{inferVars}(\mathrm{rest}_1, \mathrm{rest}_2)
\end{aligned}$$

The function inferVars' takes an expected type with type variables and a provided type and computes an assignment of types to variables s.t. the first type becomes the second after substitution. In doing so, it ignores self-variables $m\ x^{\kappa}$ and uniqueness attributes. As a result, inferVars' yielding an assignment by itself cannot guarantee that the first type is equal to the second one after substitution, and we must perform another check after substituting all the variables to obtain this guarantee.

Note that if there is a conflicting assignment for any variable in $\mathrm{inferVars}'(\tau_1, \tau_2)$ or $\mathrm{inferVars}([\tau_1], [\tau_2])$, then $\cdot \sqcup \cdot$ propagates its partiality to inferVars' and subequently to inferVars, i.e. $(\tau_1, \tau_2) \notin \mathrm{dom}(\mathrm{inferVars}')$ and $([\tau_1], [\tau_2]) \notin \mathrm{dom}(\mathrm{inferVars})$.

$$\begin{aligned}
&\mathrm{pickTypes} : [\mathrm{AttrType?}] \times [\mathrm{AttrType?}] \rightharpoonup [\mathrm{AttrType}] \\
&\mathrm{pickTypes}([], []) && = [] \\
&\mathrm{pickTypes}(\tau_e? :: \mathrm{rest}_e, \tau_i :: \mathrm{rest}_i) && = \tau_i :: \mathrm{pickTypes}(\mathrm{rest}_e, \mathrm{rest}_i) \\
&\mathrm{pickTypes}(\tau_e :: \mathrm{rest}_e, - :: \mathrm{rest}_i) && = \mathrm{strengthen}(\tau_e) :: \mathrm{pickTypes}(\mathrm{rest}_e, \mathrm{rest}_i)
\end{aligned}$$

Here, we use – to denote that an optional AttrType is not present. With pickTypes, we implement the mechanism that inferred types are preferred if they exist, and otherwise an explicit type is used and strengthened. Note that if $\exists i.\ [\tau_1]_i = [\tau_2]_i = -$, then $([\tau_1], [\tau_2]) \notin \mathrm{dom}(\mathrm{pickTypes})$.

$$\begin{aligned}
&\mathrm{inferTypeArgs} : \mathrm{ADT} \times \mathrm{Ctor} \times [\mathrm{AttrType}] \times [\mathrm{AttrType?}] \rightharpoonup [\mathrm{AttrType}]\\
&\mathrm{inferTypeArgs}(a, i, [\tau_{\mathrm{arg}}], [\tau_e?]) = \mathrm{pickTypes}([\tau_e?], [\mathrm{inferred}(y^\tau)])\\
&\qquad \text{where } a_i = [\tau_{\mathrm{field}}(x^\kappa_{\mathrm{adt}}, [y^\tau])] \to *x^\kappa_{\mathrm{adt}}\\
&\qquad \text{and inferred} = \mathrm{inferVars}([\tau_{\mathrm{field}}(x^\kappa_{\mathrm{adt}}, [y^\tau])], [\tau_{\mathrm{arg}}]])
\end{aligned}$$

Lastly, using inferTypeArgs, we infer type arguments for a constructor i in an ADT a with types $[\tau_{\mathrm{arg}}]$ for some vector of arguments `[y]` from a context Γ and user-provided explicit argument types $[\tau_e?]$. Once again, both inferVars and pickTypes propagate their partiality to inferTypeAgs.

Return types

Furthermore, we must infer whether the result of a `(A [τ?]).ctor_i [y]` call is used uniquely. If it is, we want the resulting type to be unique, and otherwise, it can be shared. This is important because we do not need to demand unique types for arguments `[y]` if the ADT created by a `ctor` call is used only in a shared manner.

$$\begin{aligned}
&\mathrm{uu} : \mathrm{FnBody} \times \mathrm{AttrType} \to 2^{\mathrm{Var}}\\
&\mathrm{uu}(\texttt{ret x}, \tau_{\mathrm{ret}}) = \begin{cases} \{\texttt{x}\} & \mathrm{attr}(\tau_{\mathrm{ret}}) = * \\ \varnothing & \mathrm{attr}(\tau_{\mathrm{ret}}) = ! \end{cases}\\
&\mathrm{uu}(\texttt{case x of [F]}, \tau_{\mathrm{ret}}) = \bigcup_i \mathrm{uu}(\texttt{[F]}_i, \tau_{\mathrm{ret}})\\
&\mathrm{uu}(\texttt{A.case' x of [ctor}_\texttt{i}\texttt{ [y] ⇒ F]}, \tau_{\mathrm{ret}}) = \bigcup_i \mathrm{uu}(\texttt{[F]}_i, \tau_{\mathrm{ret}})\\
&\qquad \cup \begin{cases} \{\texttt{x}\} & \exists \texttt{y} \in \texttt{[y]}.\ \texttt{y} \in \mathrm{uu}(\texttt{[F]}_i, \tau_{\mathrm{ret}}) \\ \varnothing & \neg\exists \texttt{y} \in \texttt{[y]}.\ \texttt{y} \in \mathrm{uu}(\texttt{[F]}_i, \tau_{\mathrm{ret}}) \end{cases}\\
&\mathrm{uu}(\texttt{let x = c [y]; F}, \tau_{\mathrm{ret}}) = \mathrm{uu}(\texttt{F}, \tau_{\mathrm{ret}})\\
&\qquad \cup \{\texttt{[y]}_i \mid \mathrm{attr}([\tau_{\mathrm{param}}]_i) = * \wedge \delta_\tau(c) = ([\tau_{\mathrm{param}}], \tau'_{\mathrm{ret}})\}\\
&\mathrm{uu}(\texttt{let x = pap c [y]; F}, \tau_{\mathrm{ret}}) = \mathrm{uu}(\texttt{F}, \tau_{\mathrm{ret}})\\
&\mathrm{uu}(\texttt{let x = y z; F}, \tau_{\mathrm{ret}}) = \mathrm{uu}(\texttt{F}, \tau_{\mathrm{ret}})\\
&\mathrm{uu}(\texttt{let x = (A [}\tau\texttt{?]).ctor}_\texttt{i}\texttt{ [y]; F}, \tau_{\mathrm{ret}}) = \mathrm{uu}(\texttt{F}, \tau_{\mathrm{ret}})\\
&\qquad \cup \begin{cases} \{\texttt{[y]}_j \mid \exists x^\tau.\ \gamma(\texttt{A})_{\texttt{i}j} = x^\tau \vee \mathrm{attr}(\gamma(\texttt{A})_{\texttt{i}j}) = *\} & \texttt{x} \in \mathrm{uu}(\texttt{F}, \tau_{\mathrm{ret}}) \\ \varnothing & \texttt{x} \notin \mathrm{uu}(\texttt{F}, \tau_{\mathrm{ret}}) \end{cases}\\
&\mathrm{uu}(\texttt{let x = A.proj}_\texttt{ij}\texttt{ y; F}, \tau_{\mathrm{ret}}) = \mathrm{uu}(\texttt{F}, \tau_{\mathrm{ret}}) \cup \begin{cases} \{\texttt{y}\} & \texttt{x} \in \mathrm{uu}(\texttt{F}, \tau_{\mathrm{ret}}) \\ \varnothing & \texttt{x} \notin \mathrm{uu}(\texttt{F}, \tau_{\mathrm{ret}}) \end{cases}
\end{aligned}$$

In the `A.case' x of ...` case, we denote `x` as being used uniquely if any of its fields are used uniquely in any of the branches. Arguments to a constructor $\texttt{ctor}_\mathrm{i}$ are regarded as being used uniquely if the constructor itself is used uniquely and the argument type is either unique or a type variable, in which case we assume the argument as being used uniquely. Note that it would be possible to infer this more accurately by inferring which type the type variable will be assigned to from an expected type. If a projection $\texttt{A.proj}_\mathrm{ij}\ \texttt{x}$ is used uniquely, then `x` is used uniquely as well.

Adjusted constructor rules

Using these inference mechanisms, we can define Let-Ctor-$*$ and Let-Ctor-! rules that make it clear which one of the two is applicable and how $[\tau'_{\text{arg}}]$ are chosen:

Let-Ctor-$*$

$$\frac{\begin{array}{c}[\mathrm{nz}(Z,\mathtt{y})] \qquad (\gamma(\mathtt{A}),\mathtt{i},[\tau],[\tau_{\text{arg}}?]) \in \mathrm{dom}(\mathrm{inferTypeArgs}) \\ [\tau'_{\text{arg}}] = \mathrm{inferTypeArgs}(\gamma(\mathtt{A}),\mathtt{i},[\tau],[\tau_{\text{arg}}?]) \\ \gamma(\mathtt{A})\{\mathtt{A},[\tau'_{\text{arg}}]\}_\mathtt{i} = [\tau] \to *x^{\kappa}_{\text{adt}} \qquad \mathtt{x} \in \mathrm{uu}(\mathtt{F},\tau_{\text{ret'}}) \qquad Z;\Gamma,\mathtt{z} : *\ \mathtt{A}\ [\tau'_{\text{arg}}] \vdash \mathtt{F} : \tau_{\text{ret'}}\end{array}}{Z;\Gamma,[\mathtt{y}:\tau] \vdash \texttt{let x = (A [}\tau_{\text{arg}}\texttt{?]).ctor}_\mathrm{i}\ \texttt{[y]; F} : \tau_{\text{ret'}}}$$

Let-Ctor-!

$$\frac{\begin{array}{c}[\mathrm{nz}(Z,\mathtt{y})] \qquad (\gamma(\mathtt{A}),\mathtt{i},[!\tau],[\tau_{\text{arg}}?]) \in \mathrm{dom}(\mathrm{inferTypeArgs}) \\ [\tau'_{\text{arg}}] = \mathrm{inferTypeArgs}(\gamma(\mathtt{A}),\mathtt{i},[!\tau],[\tau_{\text{arg}}?]) \\ \gamma(\mathtt{A})\{\mathtt{A},[\tau'_{\text{arg}}]\}_\mathtt{i} = [\tau] \to *x^{\kappa}_{\text{adt}} \qquad \mathtt{x} \notin \mathrm{uu}(\mathtt{F},\tau_{\text{ret'}}) \qquad Z;\Gamma,\mathtt{z} : !\ \mathtt{A}\ [!\tau'_{\text{arg}}] \vdash \mathtt{F} : \tau_{\text{ret'}}\end{array}}{Z;\Gamma,[\mathtt{y}:!\tau] \vdash \texttt{let x = (A [}\tau_{\text{arg}}\texttt{?]).ctor}_\mathrm{i}\ \texttt{[y]; F} : \tau_{\text{ret'}}}$$

Let-Ctor-$*$ is applicable if `x` is used in a unique manner. After inferring the type arguments using inferTypeArgs, we substitute them in $\gamma(\mathtt{A})$ and check whether the resulting types for the fields match those of our initial arguments that we used for inference. This is because while $(\gamma(\mathtt{A}), i, [\tau], [\tau_{\text{arg}}?]) \in \mathrm{dom}(\mathrm{inferTypeArgs})$ is always true if a type τ is applicable to the corresponding constructor argument, it may still be true if τ is not applicable to the constructor argument, as inferTypeArgs is ignoring attributes and self-variables x^{κ}_{adt} in $\gamma(\mathtt{A})$. When applying the rule, we consume the arguments and gain a new unique instance of the ADT type in return.

Let-Ctor-! is similar and applicable if `x` is not used in a unique manner, in which case the new instance of the ADT is created as shared and we only need to demand the arguments `[y]` to be shared.

5.3 Examples

In this section, we will cover the implementations of a few Lean functions, some of which we have seen previously, and demonstrate features of our type checker using them as examples.

List.map

Recall our implementation of `List.map` from Section 2.1, augmented with uniqueness annotations where we want types to be unique:

```
def List.map (f : α → β) : *List α → *List β
  | []      => []
  | x :: xs => f x :: map f xs
```

`List.map` translates to the following IR code:

```
List.map f xs = List.case' xs of
  ctor₀ ⇒
    let nil := (List [! ■]).ctor₀;
    ret nil
  ctor₁ hd tl ⇒
    let hd' := f hd;
    let tl' := List.map f tl;
    let r := (List [! ■]).ctor₁ hd' tl';
    ret r
```

We have the following global context:

$$\begin{aligned}\gamma &= \{\text{List} \mapsto \mu\ \text{List}.\ \ast\ \text{List}\ \mid\ [\alpha, \ast\text{List}] \to \ast\text{List}\}\\ \delta_\tau &= \{\texttt{List.map} \mapsto ([!(!\ \blacksquare \to\ !\ \blacksquare), \ast\text{List}\ [!\ \blacksquare]] \to \ast\text{List}\ [!\ \blacksquare])\}\\ \gamma_{\ast_e} &= \varnothing\\ \delta_{q_e} &= \varnothing\end{aligned}$$

Given this context, we can check $\vdash \delta_\tau$. Since `List.map` creates a new list, we can also type it using any combination of outer List attributes. The inner input attribute can be made unique, but we cannot retain the uniqueness through the application of the function of type $!(!\ \blacksquare \to\ !\ \blacksquare)$, which is why the inner output attribute must always be shared.

Our escape analysis produces the following output:

$$\begin{aligned}\delta_Q(\texttt{List.map}) &= \left\{\texttt{f}_{[]}, \texttt{f}_{[]}@[t_1], \texttt{xs}_{[10]}, \texttt{xs}_{[11]}@[t_2]\right\}\\ &\text{where } t_1 = [\#\text{param } 0, \#\text{app}, \#\text{case } 1, \#\text{const } \texttt{List.map}]\\ &\quad\text{and } t_2 = [\#\text{param } 1, \#\text{app}, \#\text{case } 1, \#\text{const } \texttt{List.map}]\end{aligned}$$

Hence, even if the outer input List attribute was shared, we would not be able to borrow the argument: Our escape analysis approximates that the tail of `xs` can escape, which would be unique if we were to borrow a unique List to `xs`.

List.get?

We will now cover our implementation of `List.get?` from Section 2.1 to demonstrate constructor argument type inference and an issue with our implementation of bor-

rowing, augmented with uniqueness annotations where we want types to be unique:

```
def List.get? : List α → Nat → *Option α
  | [],         _       => Option.none
  | x :: _,     0       => Option.some x
  | _ :: xs, n + 1 => List.get? xs n
```

It translates to the following IR code:

```
List.get? xs i = List.case' of
  ctor₀ ⇒
    let none := (Option [! ■]).ctor₀;
    ret none
  ctor₁ hd tl ⇒
    let zero := Nat.zero;
    let ieqzero := Nat.eq i zero;
    case ieqzero of
      true ⇒
        let predi := Nat.pred i;
        let r := List.get? tl predi;
        ret r
      false ⇒
        let some := (Option [-]).ctor₁ hd;
        ret some
```

Our types are as follows:

$$\gamma = \begin{Bmatrix} \text{List} \mapsto \mu\ \text{List}. & *\text{List} \mid [\alpha, *\text{List}] \to *\text{List} \\ \text{Option} \mapsto \mu\ \text{Option}. & *\text{Option} \mid \alpha \to *\text{Option} \end{Bmatrix}$$

$$\delta_\tau = \begin{Bmatrix} \texttt{List.get?} \mapsto [!\text{List}\ [!\ \blacksquare],\ !\text{Nat}\ []] & \to *\text{Option}\ [!\ \blacksquare] \\ \texttt{Nat.zero} \mapsto [] & \to !\text{Nat}\ [] \\ \texttt{Nat.eq} \mapsto [!\text{Nat}\ [],\ !\text{Nat}\ []] & \to !\text{Bool}\ [] \\ \texttt{Nat.pred} \mapsto [!\text{Nat}\ []] & \to !\text{Nat}\ [] \end{Bmatrix}$$

For simplicity, we assume that Nat and Bool are external types. Furthermore, we provide no escapee information, using both $\gamma_{*_e} = \varnothing$ and $\delta_{q_e} = \varnothing$.

With these declarations, we can check $\vdash \delta_\tau$. Note that the type for the `Option.some` call can be inferred. Our escape analysis computes the following information:

$$\delta_Q(\texttt{List.get?}) = \{\texttt{xs}_{[10]}, \texttt{xs}_{[11]}@[t]\} \text{ for some } t$$

This means that a unique argument that is passed to `xs` cannot be borrowed, as one of its unique fields $\texttt{xs}_{[11]}$ is approximated to escape by our escape analysis. Our method of widening is too aggressive because the underlying lattice lacks expressivity — if we could widen the escapees $\texttt{xs}_{[11,10]}$ and $\texttt{xs}_{[10]}$ to something like $\texttt{xs}_{[*,10]}$ or $\texttt{xs}_{[(11)*,10]}$ instead, where $*$ represents any path in the former and a Kleene star in the latter, then we could exclude $\texttt{xs}_{[11]}$ as an escapee. We will briefly revisit this idea in Section 6.3.

Array.transpose

Finally, we will cover a more complex example that shows off nested uniqueness attributes and external types. We intend to check the following function that transposes a two-dimensional square matrix:

```
def Array.T! (xs : *Array (*Array α)) (i j : Nat)
  : *Array (*Array α) :=
  let n := Array.size xs
  if i >= n then
    xs
  else if j >= n then
    Array.T! xs (i + 1) (i + 2)
  else
    let (xs, xs_i) := Array.swap xs i #[]
    let (xs, xs_j) := Array.swap xs j #[]
    let x := Array.get! xs_i j
    let x' := Array.get! xs_j i
    let xs_i := Array.set! xs_i j x'
    let xs_j := Array.set! xs_j i x
    let xs := Array.set'! xs i xs_i
    let xs := Array.set'! xs j xs_j
    Array.T! xs i (j + 1)
```

We assume that `Array.T!` is always called with a square matrix using `i` = 0 and `j` = 1 at the start. For brevity, we omit the equivalent IR code.

We have the following algebraic data types:

$$\gamma = \left\{ \begin{matrix} \text{Bool} \mapsto \mu\ \text{Bool}.\ \ast\text{Bool} \ \mid\ \ast\text{Bool} \\ \text{Tuple} \mapsto \mu\ \text{Tuple}.\ [\alpha, \beta] \to \ast\text{Tuple} \end{matrix} \right\}$$

The types we assign to constants in the program in the `Array` namespace are as follows:

$$\delta_\tau(\texttt{Array.}\ast) = \left\{ \begin{array}{rl} \texttt{T!} \mapsto & [\ast\text{Array } [\ast\text{Array } [!\ \blacksquare]],\ !\text{Nat } [],\ !\text{Nat } []] \\ & \to \ast\text{Array } [\ast\text{Array } [!\ \blacksquare]] \\ \texttt{size} \mapsto & [!\text{Array } [!\ \blacksquare]] \to !\text{Nat } [] \\ \texttt{empty} \mapsto & [] \to \ast\text{Array } [\ast\ \blacksquare] \\ \texttt{swap} \mapsto & [\ast\text{Array } [\ast\text{Array } [!\ \blacksquare]],\ !\text{Nat } [], \ast\text{Array } [!\ \blacksquare]] \\ & \to \ast\text{Tuple } [\ast\text{Array } [\ast\text{Array } [!\ \blacksquare]], \ast\text{Array } [!\ \blacksquare]] \\ \texttt{get!} \mapsto & [!\text{Array } [!\ \blacksquare],\ !\text{Nat } []] \to !\ \blacksquare \\ \texttt{set!} \mapsto & [\ast\text{Array } [!\ \blacksquare],\ !\text{Nat } [],\ !\ \blacksquare] \to \ast\text{Array } [!\ \blacksquare] \\ \texttt{set'!} \mapsto & [\ast\text{Array } [\ast\ \blacksquare],\ !\text{Nat } [], \ast\ \blacksquare] \to \ast\text{Array } [\ast\ \blacksquare] \end{array} \right\}$$

For operations on natural numbers, we have:

$$\delta_\tau(\texttt{Nat.*}) = \left\{ \begin{array}{ll} \texttt{one} \mapsto [] & \to \,!\text{Nat } [] \\ \texttt{two} \mapsto [] & \to \,!\text{Nat } [] \\ \texttt{geq} \mapsto [!\text{Nat } [],\ !\text{Nat } []] & \to \,!\text{Bool } [] \\ \texttt{add} \mapsto [!\text{Nat } [],\ !\text{Nat } []] & \to \,!\text{Nat } [] \end{array} \right\}$$

`Array` and `Nat` are external types. For `Array`, `size` and `get!` we provide the following escape information:

$$\gamma_{*_e} = \{\text{Array} \mapsto [00](\alpha)\}$$
$$\delta_{q_e} = \left\{\texttt{Array.size} \mapsto \varnothing, \texttt{Array.get!} \mapsto \{\texttt{xs}_{[00]}\}\right\}$$

Here, α denotes the first type parameter of `Array` and `xs` denotes the first parameter of `Array.get!`.

Using these definitions, we can check $\vdash \delta_\tau$. The following aspects are noteworthy about this example:

- `xs` is successfully borrowed in the applications of `Array.size` and `Array.get!`. Without borrowing, this definition would not type-check, as `xs` would be consumed on application of these functions.
- The `Array.swap` function ensures that the inner uniqueness is retained by swapping in a unique value that takes the place of the value in the array. When using the `Array.swap` function, we must be careful to maintain `i ≠ j`, as we would otherwise access the same index twice.
- Since we lack polymorphism, we have two separate `Array.set!` functions, depending on whether the inner attribute is shared or unique.

6 Future Work

In this chapter, we will briefly cover work that is still left to be done in order to integrate our type theory with Lean 4, as well as work that would greatly improve the user-experience of working with our type theory.

6.1 Type Inference

We have not covered the topic of type inference at all thus far, but some amount of type inference is necessary to integrate our type theory with Lean 4. During the compilation process, the Lean 4 compiler toolchain may create additional auxiliary functions that users cannot provide type annotations for:

- In monadic control flow, Lean may create join points and other auxiliary functions to represent complex control flow (`break`, `continue`, early `return`, `if` blocks).
- As argued in Section 4.2, functions that act as join points may get replaced by dedicated join point instructions that callers can directly jump to.
- If a function does not use all of its parameters, then an auxiliary function using only live parameters is created. Note that this may occur often because of LCNF's type erasure step.

Fortunately, in all of these cases, we have access to plenty of surrounding context to determine what the respective uniqueness attributes for an auxiliary function should be.

Nonetheless, it would be nice to have some amount of type inference for the uniqueness attributes of user-written functions as well. Generally, we want parameters that can be borrowed in the sense of Section 4.4 to be shared. On the other hand, for parameters that escape, there is no principal type: Depending on whether callers have a unique or shared value at hand, they would want the parameter type to match the given attribute that they have available.

Hence, without polymorphism, one possible approach would be to infer the strongest possible type for a given function and then automatically create auxiliary functions that represent possible weakenings of this strongest possible type, the most obvious weakening being a function where all parameters are shared and the return type is shared as well.

With polymorphism, principal types may again become available and could directly be inferred if the inference algorithm is sufficiently sophisticated.

6.2 Integration With Lean 4

In Chapter 5, we implement a type checker for our type theory and a model implementation of the IR described in Section 4.2, but do not integrate it with Lean 4 itself. In the future, the following steps need to be taken to finish the integration:

1. A form of type inference that can deal with the kinds of auxiliary functions described in Section 6.1 needs to be implemented.

2. A notation to provide uniqueness annotations in types needs to be implemented. Types annotated with `*` are unique, types without an annotation are shared. The notation should set the `mdata` field of Lean 4's `Expr`, which can be used to store auxiliary data.

3. The `mdata` field needs to be handled correctly in the compiler toolchain so that it is preserved until our type checker runs at the end of the pipeline. As of the writing of this thesis, the function that converts Lean types to LCNF types erases expressions annotated with `mdata`.

4. The functions δ_{q_e} from Section 4.3 and γ_{*_e} from Section 4.4 need to be provided using a custom annotation in Lean 4's annotation mechanism, where declarations, including stubs for external functions and types, can be provided with auxiliary data by users.

5. Lean's LCNF must be translated to our model IR and our model type system while preserving information about the mapping.

6. Our type checker must be adjusted to produce reasonable error codes. As of now, it only yields a boolean indicating whether a given environment type-checks. Then, if there is a uniqueness type error, using the aforementioned mapping, the error for our model IR must be translated to a corresponding LCNF error.

7. In addition to checking whether a given environment type-checks, our type checker should accumulate information about the scope in which a variable is unique. After translating this information back to LCNF, it can be used in subsequent optimizations, for example eliminating the reference count check described in Section 2.4.

6.3 Borrowing

In Section 2.8, we saw that implementing more powerful borrowing mechanisms would be useful. In Section 5.3, we found that our escape analysis is of limited utility when working with recursive types. Hence, in order to implement a user-friendly uniqueness type theory, it is essential that the topic of borrowing is re-visited in the future.

Spiwack et al. [2022] and Weiss et al. [2021] may provide some guidance for implementing a borrowing mechanism that handles more cases than the one described in this thesis.

Another possible approach would be to augment the escapees in Section 4.3 with a Kleene star and a union operator, or an "any path"-operator, and have $\mathbb{B}.(\cdot, \cdot, \cdot)$ from Section 4.4 generate patterns that describe the uniqueness of the fields of an algebraic data type. Then, checking whether any unique fields escape amounts to checking whether the intersection of these patterns is nonempty.

6.4 Higher-Order Functions

We have discussed the topic of higher-order functions in uniqueness type systems at length in Section 2.7 and Section 3.2.1, but chose to disregard uniqueness of functions for now in Section 4.1 by assuming every higher-order function as shared.

This situation is far from optimal because Lean code uses higher-order functions for type classes, for monadic code, as well as for tricks that guarantee uniqueness for unique values within other unique values, like the `update` trick discussed in Section 2.4.

As described in Section 3.2.1, we believe that the approach that deleverages the uniqueness of values in a function closure is the most viable, though it requires a change to the runtime in that higher-order functions need to store two function pointers as opposed to just one. Additionally, several other components in Chapter 4 must be adjusted as well, since subtyping must now account for co- and contravariant type parameters and propagation must be capable of propagating through higher-order functions. Similarly, borrowing most be adjusted with unique functions in mind.

6.5 Polymorphism

Another topic we have not touched on at all is attribute polymorphism. Part of the reason for this is that polymorphism results in somewhat unintuitive behaviour if the coercion between unique and shared values is implicit, as in our system in Chapter 4, since polymorphic functions may both silently cast to a shared value and then propagate the sharedness through the rest of the code. In this case, using different function names would unveil the mistake early on.

However, if the coercion is made explicit, then mistakes would always be spotted early on, as a unique value can never be passed to a shared parameter unless users acknowledge it with an explicit instruction. Then, shared and unique functions would not need to be named differently, and polymorphism would be a useful thing to have.

What follows are some brief and incomplete thoughts on what would be required to make polymorphism work:

- In order to be substitutable for both a unique and a shared attribute, variables

that are polymorphic in their attribute can only be passed to polymorphic parameters, but must otherwise be used uniquely.

- Variables polymorphic in their attribute cannot be updated destructively.
- Borrowed parameters should always be shared, not polymorphic, since both shared and unique values can be passed to the parameter, while it is still allowed to share the borrowed parameter within the function body as long as it does not escape.
- There needs to be a mechanism to connect the uniqueness of two attribute variables and state that "this component of the return type can be unique if this parameter is unique", similar to Clean's $\leq$ operator or the boolean connectives discussed by de Vries [2009].
- Uniqueness attributes in ADTs should propagate the attribute variable of the outer value when the respective field is accessed.
- Ideally, the fact that shared types cannot contain unique ones should not have to be explicitly reflected everywhere in the type annotation of a function; it should be assumed implicitly, or at least the annotational clutter that is present in de Vries' implementations of polymorphism should somehow be reduced.

An alternative to polymorphism would be to provide facilities that generate function declarations for all valid attribute annotations after type erasure. Since there are only two attributes and users likely do not care much about the concrete uniqueness annotations except that they should make their code type-check, generating functions for all possible annotations may be a worthwhile compromise, especially as systems that support attribute polymorphism tend to produce fairly complex annotations.

6.6 Proof of Soundness

In all of the previous sections, we have only argued informally about the soundness of our type system, i.e. that variables of unique type are indeed referenced uniquely. A formal proof of soundness of our system is future work.

7 Related Work

In this chapter, we will briefly cover other approaches that are similar to ours.

Linear Haskell is a lazy functional language with support for properly linear types, but no borrowing. It supports an `MArray` type with efficient destructive updates and a function `freeze` that can convert a linear `MArray` to a non-linear `Array`. As described in Section 2.6, `MArray` constructors use continuation passing. Additionally, an `MArray` itself is not allowed to contain linear values. Linear Haskell also has experimental support for multiplicity polymorphism and multiplicity inference.

Spiwack et al. [2022] resolve some of these shortcomings by introducing a language of linear capabilities on top of Linear Haskell. The uniqueness of a type is still a library design decision, but constructors do not need to be stated in continuation-passing style anymore. Arrays are allowed to contain other unique types and there is a borrowing mechanism both for unique types within unique types and for unique types that are used in a read-only manner in functions.

Unfortunately, linear constraints still have to be unpacked explicitly in the term language and the resulting system is quite complex. We do however think that the general idea of decoupling values from their linear capabilities and enabling libraries to provide complex capabilities of their own may prove to be a fruitful avenue of research.

Idris 2 [Brady, 2021] is a dependently typed functional language with support for quantitative type theory. As such, its substructural type system has the same properties as that of Linear Haskell, but also supports an erasure quantity. Dependent type theory enables some additional applications, like specifying linear usage protocols and simulating session types [Honda, 1993].

Idris, the precursor to Idris 2, has support for what the authors call “uniqueness types” as well [Brady, 2017]. Idris’ “uniqueness types” are what we call “invariably unique types” in Section 2.5, though Idris’ linear types need not be consumed and are hence affine. Compound data types are either declared to be inherently unique or non-unique, and there is no coercion between the two. It also supports a restrictive notion of borrowing that allows pattern matching on borrowed values, but no further inspection of the values. Due to the lack of an erasure quantity, the combination of dependent and linear types is limited.

ATS [Shi and Xi, 2013] is a dependently typed language with support for an extensive foreign function interface and invariably unique types to ensure safe resource-

usage. Pointers to resources consist of two components: the reference to the resource, as well as a linear value which witnesses that the resource has not been freed yet.

There does not appear to be a safe borrowing mechanism or a safe coercion between unique and shared types, but library functions can assert that a reference is not consumed by a function. There also seems to be support for syntactic sugar that reduces the amount of explicit linear values being passed around by implicitly managing a context of linear values, similar in spirit to the approach of Spiwack et al. [2022].

Clean [Smetsers et al., 1994] is the functional language that introduced the concept of uniqueness typing. It supports uniqueness types, polymorphism for uniqueness attributes, as well as type inference. de Vries [2009] provides a type-theoretical description for Clean that uses lambda calculus as its term language.

As previously discussed in Chapter 3, Clean uses invariably unique types for higher-order functions. Its borrowing mechanism follows Wadler [1990] and only allows borrowing in expressions that produce primitive types, so that borrowed variables are guaranteed not to escape. Clean uses uniqueness for destructive updates, as well as safe I/O.

Cogent [O'Connor et al., 2021] is a non-recursive functional language designed for systems programming with extensive FFI capabilities, delegating I/O and recursive programs to C. It features a certifying compiler that produces proofs that the generated program is a refinement of the original Cogent program using an Isabelle/HOL [Nipkow et al., 2002] embedding. With these certificates, it is possible to prove Cogent programs correct within Isabelle/HOL.

It also features invariably unique types to ensure safe resource usage and enable destructive updates, as well as a borrowing mechanism based on observer types [Odersky, 1992], polymorphism and type inference. Since it uses uniqueness to ensure safe resource use, there is no coercion from unique to non-unique types. Higher-order functions are always fully applied and cannot capture variables in their closure.

Granule [Orchard et al., 2019] is a functional programming language that acts as a framework for linear types, indexed types and graded modal types. Marshall et al. [2022] add uniqueness types to Granule as a separate modality and use them for efficient destructive updates. Their type system does not support nested uniqueness types, borrowing, type inference or uniqueness polymorphism. Higher-order functions are implemented using properly linear types.

Futhark [Henriksen et al., 2017] is a functional data-parallel language intended for GPU code. It uses an alias analysis in order to support efficient in-place updates on arrays only: If an array has been aliased in the past, then all of its aliases may not be used after the in-place update. This allows for some greater flexibility, where

arrays can become temporarily aliased. Across function boundaries, invariably unique types are used to ensure the uniqueness of arrays, not the full aliasing information.

There are dedicated type system rules for common array operations and variable uses are classified as “consuming” or “observing”. This allows Futhark to implement a form of borrowing where observing and consuming operations are used in an alternating manner, so that an array can only be observed arbitrarily often before it is consumed, after which it cannot be observed or consumed anymore.

There is no notion of uniqueness polymorphism or type inference. There is also no support for letting users implement custom consumers of higher-order functions. The built-in higher-order operators always fully apply the higher-order functions that they are passed and arrays in the closure of the higher-order function are not allowed to be consumed within it.

Affe [Radanne et al., 2020] is an impure functional language with support for properly linear and affine types, borrowing, polymorphism and type inference. Higher-order functions that contain linear values in their closure are again linear.

It supports the notion of an “explicit borrow” where the borrowee is allowed to update the borrowed value, but is still forced to use it linearly. This is necessary because Affe is impure; in a purely functional language, we would instead just return the updated value regardless of whether we are using a substructural type system or not.

For a detailed comparison of Affe with other ML-like languages that support linear types, such as System F° [Mazurak et al., 2010], Alms [Tov and Pucella, 2011], Quill [Morris, 2016] and Mezzo [Balabonski et al., 2016], as well as the imperative programming languages Rust [Weiss et al., 2021], Vault [DeLine and Fähndrich, 2001] and Plaid [Garcia et al., 2014], we refer to Radanne et al. [2020].

Linear Dafny [Li et al., 2022] is an imperative language with invariably unique types and support for an SMT backend [Barrett and Tinelli, 2018] that is leveraged to reason about general program properties and aliasing when the linear type system cannot provide the required guarantees.

It supports a traditional borrowing mechanism in the style of observer types [Odersky, 1992] where observation propagates outwards, as well as arbitrarily mixing unique and shared types, the latter of which is ensured to be safe by delegating an aliasing proof obligation to the SMT backend.

Since Lean is also a general proof language, we consider the approach of Linear Dafny very relevant to Lean as well. In order to implement something similar for Lean, uniqueness types would have to be made compatible with Lean’s dependent type theory and we would have to embed a model of Lean that allows reasoning about the aliasing of Lean values within Lean itself. Niu et al. [2022] may provide some guidance for augmenting dependently typed languages with cost functions.

8 Conclusion

We have provided an overview for the design space of substructural type theory with the goal of ensuring destructive updates in the Lean 4 theorem prover and evaluated existing approaches, concluding that uniqueness type theory is most suited for ensuring destructive updates.

Using our evaluation, we have designed a uniqueness type theory of our own and implemented a type checker for it using the Lean 4 programming language at `https://github.com/mhuisi/Uniq`. Our type theory targets a combined model of Lean 4's intermediate representations. It supports uniqueness types, algebraic data types, erased types, external declarations, non-shallow subtyping for uniqueness attributes and a notion of borrowing. To implement the latter, we have designed and implemented an escape data flow analysis that computes an over-approximation of both parameters and fields of parameters that escape in a given function.

Our type theory lacks support for uniqueness attributes in higher-order functions, type inference, as well as support for attribute polymorphism. Our escape analysis produces non-satisfactory results on recursive functions over recursive types, inhibiting the borrowing of arguments to such functions. Integrating our type theory with the Lean 4 compiler is future work. We have not formally evaluated the soundness of our type theory.

For higher-order functions, we have evaluated all existing approaches known to us and made a recommendation for an approach that we think is the most suitable one to implement in the future. For borrowing, we have made suggestions to improve the implementation provided in this thesis. For the topics of type inference, polymorphism and Lean 4 integration, we have outlined steps that need to be taken in order to complete the implementation thereof.

Bibliography

F. E. Allen and J. Cocke. A Program Data Flow Analysis Procedure. *Communications of the ACM*, 19(3):137, March 1976. ISSN 0001-0782. doi: 10.1145/360018.360025.

Thosten Altenkirch, James Chapman, and Tarmo Uustalu. Monads need not be endofunctors. *Logical Methods in Computer Science*, 11(1):3, March 2015. ISSN 18605974. doi: 10.2168/LMCS-11(1:3)2015. arXiv:1412.7148 [cs, math].

David Aspinall and Martin Hofmann. Another Type System for In-Place Update. In *Programming Languages and Systems*, volume 2305, pages 36–52. Springer, Berlin, Heidelberg, 2002. ISBN 978-3-540-43363-7 978-3-540-45927-9. doi: 10.1007/3-540-45927-8_4. Series Title: Lecture Notes in Computer Science.

Robert Atkey. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 56–65, Oxford United Kingdom, July 2018. ACM. ISBN 978-1-4503-5583-4. doi: 10.1145/3209108.3209189.

Lennart Augustsson. Implementing Haskell overloading. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 65–73, Copenhagen Denmark, July 1993. ACM. ISBN 978-0-89791-595-3. doi: 10.1145/165180.165191.

Jeremy Avigad, Leonardo de Moura, Soonho Kong, and Sebastian Ullrich. *Theorem Proving in Lean 4*. 2022. URL `https://leanprover.github.io/theorem_proving_in_lean4`.

Thibaut Balabonski, François Pottier, and Jonathan Protzenko. The Design and Formalization of Mezzo, a Permission-Based Programming Language. *ACM Transactions on Programming Languages and Systems*, 38(4):14:1–14:94, August 2016. ISSN 0164-0925. doi: 10.1145/2837022.

Erik Barendsen and Sjaak Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6(6):579–612, December 1996. ISSN 0960-1295, 1469-8072. doi: 10.1017/S0960129500070109. Publisher: Cambridge University Press.

Clark Barrett and Cesare Tinelli. Satisfiability Modulo Theories. In *Handbook of Model Checking*, pages 305–343. Springer International Publishing, Cham, 2018. ISBN 978-3-319-10575-8. doi: 10.1007/978-3-319-10575-8_11.

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language. *Proceedings of the ACM on Programming Languages*, 2(POPL): 1–29, January 2018. ISSN 2475-1421. doi: 10.1145/3158093. arXiv:1710.09756 [cs].

Bruno Blanchet. Introduction to Abstract Interpretation. November 2002. URL `https://bblanche.gitlabpages.inria.fr/absint.pdf`.

Edwin Brady. Idris 2: Quantitative Type Theory in Practice. *arXiv:2104.00480 [cs]*, April 2021.

Edwin Charles Brady. Type-Driven Development of Concurrent Communicating Systems. *Computer Science*, 18(3), July 2017. ISSN 2300-7036. doi: 10.7494/csci.2017.18.3.1413. Number: 3.

Mario Carneiro. The Type Theory of Lean. Master's thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, April 2019. URL `https://github.com/digama0/lean-type-theory/releases/download/v1.0/main.pdf`.

Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming*, 6(2):195–244, March 1996. ISSN 1469-7653, 0956-7968. doi: 10.1017/S0956796800001660. Publisher: Cambridge University Press.

Pritam Choudhury, Harley Eades III, Richard A. Eisenberg, and Stephanie C. Weirich. A Graded Dependent Type System with a Usage-Aware Semantics (extended version). *arXiv:2011.04070 [cs]*, January 2021.

David Thrane Christiansen. *Functional Programming in Lean*. 2023. URL `https://leanprover.github.io/functional_programming_in_lean/`.

John Cocke. Global Common Subexpression Elimination. In *Proceedings of a symposium on Compiler optimization*, pages 20–24, New York, NY, USA, July 1970. Association for Computing Machinery. ISBN 978-1-4503-7386-9. doi: 10.1145/800028.808480.

Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, January 1977. Association for Computing Machinery. ISBN 978-1-4503-7350-0. doi: 10.1145/512950.512973.

Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28*, Lecture Notes in Computer Science, pages 625–635, Cham, 2021. Springer International Publishing. ISBN 978-3-030-79876-5. doi: 10.1007/978-3-030-79876-5_37.

Edsko de Vries. *Making Uniqueness Typing Less Unique*. PhD thesis, Trinity College (Dublin, Ireland). School of Computer Science & Statistics, 2009. URL `http://www.tara.tcd.ie/handle/2262/90081`.

Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 59–69, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 978-1-58113-414-8. doi: 10.1145/378795.378811.

Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. Foundations of Typestate-Oriented Programming. *ACM Transactions on Programming Languages and Systems*, 36(4):12:1–12:44, October 2014. ISSN 0164-0925. doi: 10.1145/2629609.

Dan R. Ghica and Alex I. Smith. Bounded Linear Types in a Resource Semiring. In Zhong Shao, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 331–350, Berlin, Heidelberg, 2014. Springer. ISBN 978-3-642-54833-8. doi: 10.1007/978-3-642-54833-8_18.

Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, January 1987. ISSN 0304-3975. doi: 10.1016/0304-3975(87)90045-4.

Oleg Grenrus. Dependent Linear types in QTT, December 2020. URL `https://oleg.fi/gists/posts/2020-12-18-dependent-linear.html`.

Dana Harrington. Uniqueness logic. *Theoretical Computer Science*, 354(1):24–41, March 2006. ISSN 0304-3975. doi: 10.1016/j.tcs.2005.11.006.

Fritz Henglein and Jesper Jørgensen. Formally Optimal Boxing. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '94, pages 213–226, New York, NY, USA, February 1994. Association for Computing Machinery. ISBN 978-0-89791-636-3. doi: 10.1145/174675.177874.

Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely Functional GPU-Programming with Nested Parallelism and In-Place Array Updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 556–571, New York, NY, USA, June 2017. Association for Computing Machinery. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062354.

Kohei Honda. Types for Dyadic Interaction. In *CONCUR'93*, Lecture Notes in Computer Science, pages 509–523, Berlin, Heidelberg, 1993. Springer. ISBN 978-3-540-47968-0. doi: 10.1007/3-540-57208-2_35.

Thomas Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science, pages 190–203, Berlin, Heidelberg, 1985. Springer. ISBN 978-3-540-39677-2. doi: 10.1007/3-540-15975-4_37.

Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4-5):393–434, July 2002. ISSN 1469-7653, 0956-7968. doi: 10.1017/S0956796802004331. Publisher: Cambridge University Press.

Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the Rules: Rewriting as a practical optimisation technique in GHC. In *2001 ACM SIGPLAN*, page 203, 2001.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(1):1–82, January 2007. ISSN 0956-7968, 1469-7653. doi: 10.1017/S0956796806006034.

Naoki Kobayashi. Quasi-Linear Types. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '99*, pages 29–42, San Antonio, Texas, United States, 1999. ACM Press. ISBN 978-1-58113-095-9. doi: 10.1145/292540.292546.

C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, March 2004. doi: 10.1109/CGO.2004.1281665.

Jialin Li, Andrea Lattuada, Yi Zhou, Jonathan Cameron, Jon Howell, Bryan Parno, and Chris Hawblitzel. Linear Types for Large-Scale Systems Verification. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):69:1–69:28, April 2022. doi: 10.1145/3527313.

Daniel Marshall, Michael Vollmer, and Dominic Orchard. Linearity and Uniqueness: An Entente Cordiale. In Ilya Sergey, editor, *Programming Languages and Systems*, volume 13240, pages 346–375. Springer International Publishing, Cham, 2022. ISBN 978-3-030-99335-1 978-3-030-99336-8. doi: 10.1007/978-3-030-99336-8_13. Series Title: Lecture Notes in Computer Science.

Kazutaka Matsuda and Meng Wang. Sparcl: A Language for Partially-Invertible Computation. *Proceedings of the ACM on Programming Languages*, 4(ICFP):1–31, August 2020. ISSN 2475-1421. doi: 10.1145/3409000.

Luke Maurer, Paul Downen, Zena M. Ariola, and Simon Peyton Jones. Compiling without Continuations. *ACM SIGPLAN Notices*, 52(6):482–494, June 2017. ISSN 0362-1340. doi: 10.1145/3140587.3062380.

Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. Lightweight Linear Types in System F°. In *Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation*, pages 77–88, Madrid Spain, January 2010. ACM. ISBN 978-1-60558-891-9. doi: 10.1145/1708016.1708027.

Conor McBride. I Got Plenty o' Nuttin'. In *A List of Successes That Can Change the World*, volume 9600, pages 207–233. Springer International Publishing, Cham, 2016. ISBN 978-3-319-30935-4 978-3-319-30936-1. doi: 10.1007/978-3-319-30936-1_12. Series Title: Lecture Notes in Computer Science.

Benjamin Moon, Harley Eades III, and Dominic Orchard. Graded Modal Dependent Type Theory. *arXiv:2010.13163 [cs]*, February 2021.

J. Garrett Morris. The Best of Both Worlds: Linear Functional Programming without Compromise. *ACM SIGPLAN Notices*, 51(9):448–461, September 2016. ISSN 0362-1340. doi: 10.1145/3022670.2951925.

Tobias Nipkow, Markus Wenzel, Lawrence C. Paulson, Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen, editors. *Isabelle/HOL*, volume 2283 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, 2002. ISBN 978-3-540-43376-7 978-3-540-45949-1. doi: 10.1007/3-540-45949-9.

Yue Niu, Jonathan Sterling, Harrison Grodin, and Robert Harper. A Cost-Aware Logical Framework. *Proceedings of the ACM on Programming Languages*, 6(POPL): 9:1–9:31, January 2022. doi: 10.1145/3498670.

Martin Odersky. Observers for Linear Types. In *ESOP '92*, volume 582, pages 390–407. Springer Berlin Heidelberg, Berlin, Heidelberg, 1992. ISBN 978-3-540-55253-6 978-3-540-46803-5. doi: 10.1007/3-540-55253-7_23. Series Title: Lecture Notes in Computer Science.

Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. Quantitative Program Reasoning with Graded Modal Types. *Proceedings of the ACM on Programming Languages*, 3(ICFP):110:1–110:30, July 2019. doi: 10.1145/3341714.

Liam O'Connor, Zilin Chen, Christine Rizkallah, Vincent Jackson, Sidney Amani, Gerwin Klein, Toby Murray, Thomas Sewell, and Gabriele Keller. Cogent: uniqueness types and certifying compilation. *Journal of Functional Programming*, 31:e25, 2021. ISSN 0956-7968, 1469-7653. doi: 10.1017/S095679682100023X.

Gabriel Radanne, Hannes Saffrich, and Peter Thiemann. Kindly Bent to Free Us. *Proceedings of the ACM on Programming Languages*, 4(ICFP):103:1–103:29, August 2020. doi: 10.1145/3408985.

M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, January 1981. ISSN 0898-1221. doi: 10.1016/0898-1221(81)90008-0.

Rui Shi and Hongwei Xi. A linear type system for multicore programming in ATS. *Science of Computer Programming*, 78(8):1176–1192, August 2013. ISSN 0167-6423. doi: 10.1016/j.scico.2012.09.005.

Sjaak Smetsers, Erik Barendsen, Marko van Eekelen, and Rinus Plasmeijer. Guaranteeing Safe Destructive Updates through a Type System with Uniqueness Information for Graphs. In *Graph Transformations in Computer Science*, Lecture Notes in Computer Science, pages 358–379, Berlin, Heidelberg, 1994. Springer. ISBN 978-3-540-48333-5. doi: 10.1007/3-540-57787-4_23.

Arnaud Spiwack. Linear Constraints: The Problem With O(1) Freeze, January 2023. URL `https://www.tweag.io/blog/2023-01-26-linear-constraints-freeze/`.

Arnaud Spiwack, Csongor Kiss, Jean-Philippe Bernardy, Nicolas Wu, and Richard A. Eisenberg. Linearly Qualified Types: Generic Inference for Capabilities and Uniqueness. *Proceedings of the ACM on Programming Languages*, 6(ICFP):137–164, August 2022. ISSN 2475-1421. doi: 10.1145/3547626.

Tomáš Svoboda. Additive Pairs in Quantitative Type Theory. Master's thesis, Charles University, 2021. URL `https://dspace.cuni.cz/bitstream/handle/20.500.11956/127263/120390849.pdf`.

Robert Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, June 1972. ISSN 0097-5397. doi: 10.1137/0201010.

Matúš Tejiščák. *Erasure in Dependently Typed Programming*. PhD thesis, University of St Andrews, March 2019. URL `https://ziman.functor.sk/media/thesis.pdf`.

Jesse A. Tov and Riccardo Pucella. Practical Affine Types. *ACM SIGPLAN Notices*, 46(1):447–458, January 2011. ISSN 0362-1340. doi: 10.1145/1925844.1926436.

Sebastian Ullrich and Leonardo de Moura. Counting Immutable Beans: Reference Counting Optimized for Purely Functional Programming, March 2020. arXiv:1908.05647 [cs].

Sebastian Ullrich and Leonardo de Moura. Beyond Notations: Hygienic Macro Expansion for Theorem Proving Languages, April 2022. arXiv:2001.10490 [cs].

Philip Wadler. Linear Types can Change the World!, 1990. URL `https://cs.ioc.ee/ewscs/2010/mycroft/linear-2up.pdf`.

Philip Wadler. Is there a use for linear logic? *ACM SIGPLAN Notices*, 26(9):255–273, May 1991. ISSN 0362-1340. doi: 10.1145/115866.115894.

Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed. Oxide: The Essence of Rust, October 2021. arXiv:1903.00982 [cs].

Erklärung

Hiermit erkläre ich, Marc Huisinga, dass ich die vorliegende Masterarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

______________________________ ______________________________

Ort, Datum Unterschrift

Acknowledgements

I am thankful to Muhammet Gümüş, Markus Himmel and Paul Reichert for working through a draft of this thesis and providing helpful feedback. I am especially grateful to Achim Kriso for our repeated conversations about substructural type theory, without which I would have lacked many of the insights that went into the design of the type theory presented here. I would also like to thank my thesis advisor Sebastian Ullrich for getting me interested in Lean, always having an open ear and providing guidance for anything Lean-related, all the way through my bachelor's thesis, my master's studies and eventually my master's thesis. Finally, I am very grateful to my parents for supporting me all the way through university.