MSc in Computer Science 2020-21

Project Dissertation

Project Dissertation title:	Dependent Types with Borrowing
Term and year of submission:	Trinity Term, 2021
Candidate Number:	1047352
Word count:	24,218 words

Abstract

Dependent type theory and sub-structural typing are two of the most influential ideas in modern computer science research. Sub-structural types have recently found widespread mainstream applications as part of the Rust programming language in the form of *ownership types* [13], which extend affine types with the concept of "borrowing." Rust's type system allows us to check memory safety, thread safety, and complex user-defined invariants (e.g., holding a mutex lock while accessing a resource) at zero runtime cost. Recent work, such as by Krishnaswami et al. [12], has allowed us to integrate linear types with dependent types, to obtain linear dependent types (LDTs). This project aims to generalize existing work on linear dependent types by designing and implementing a simple language integrating ownership and dependent types, i.e., "dependent types with borrowing," with the main aim of exploring the possible design space for type systems supporting this functionality. To do so, we build on Krishnaswami et al.'s separation of the space of (dependent) intuitionistic functions and linear functions by extending the latter with a notion of "borrowing" a value for a "lifetime." We then further generalize the previous work by making linear function types (partially) dependent: minor tweaks to this system ensure that we retain properties amenable to low-level implementation (e.g., fixed binary representation for return values, avoiding the need for boxing with standard calling conventions).

Contents

Chapter 1

Introduction

This chapter attempts to explain and motivate the main objective of this project, namely, to explore the design space of dependent ownership types via the design of the experimental programming language `isotope` and it's type system. In particular, we describe the applications of dependent typing and ownership typing, and previous work on combining dependent types with linear types, which ownership types may be viewed as a generalization of. We then describe some of the potential applications of a combined system of dependent and ownership types, and proceed to describe our objectives in the design of `isotope`. Finally, we lay out the organization of the remainder of the thesis, and finish with a summary of our main achievements.

1.1 Motivation

We can view a type system as having two primary functions: assigning types to values and restricting where they can be used based on their types. From a correctness standpoint, this enables weeding out large classes of trivial "round peg in a square hole" errors, such as attempting to add an integer to a string, even before running a program. Such information can also significantly improve the quality of compiled programs by enabling implementors to make numerous optimizations, such as unboxing and static dispatch. Sub-structural type systems go further by restricting how often we may use a given term. For example, we can model exhaustible resources like memory allocations and open file handles by restricting values of certain types to be used at most once; such types are called *affine*. Languages such as Rust and C++ use such a mechanism via the "RAII" pattern to provide a semblance of automatic memory management without requiring a garbage collector. Similarly, by requiring that values of certain types, such as file handles and error codes, are observed at least once, we can avoid common bugs such as resource leaks and ignored error conditions; such types are called *relevant*. *Linear type systems* are sub-structural type systems that include support for affine and relevant types (a type which is both affine and relevant is called *linear*).

Linear typing shows promise as a tool to efficiently implement traditionally stateful operations, such as array updates, in a purely functional setting, without recourse to "magical" state monads and their underlying unsafe operations [4, 20]. In particular, a linear type system can allow us to model effects (such as exceptions) as relevant types, hard-to-clone resources (such as arrays and file handles) as affine types, and hence, state, or effect on resources, as linear types. In doing so, we can "embed" imperative programming into a purely functional language with, at least theoretically, no loss in performance, without losing the benefits of, e.g., referential transparency and immutability.

Dependent type systems, on the other hand, generalize simple type systems by allowing types to depend on terms. For example, where we were previously limited to, e.g., declaring `x` to be an array of integers, we can now state it is an array of length `n + 3` with first element

`2 * n`, where `n` is some runtime variable. By doing so, we can give precise, compiler-verified specifications of the functional behavior of a term in its type alone. Via the Curry-Howard correspondence, we can regard terms in a dependently-typed language as mathematical proofs, and indeed the primary application of dependent type theory today is as the formal underpinning for proof assistants such as Coq [17].

By extending Benton's linear/non-linear calculus [3] to support type dependency, Krishnaswami et al. [12] have demonstrated an integration of linear and dependent types. Encoding imperative programs as linearly typed terms via a system of locations and reference capabilities (in other words, pointers and the permission to access pointees), they proceeded to demonstrate an internal, "proofs as programs" methodology for imperative programming based off this system [12]. This strategy, however, requires that we manually prove all pointer access valid, which can very quickly become very cumbersome. Moreover, common patterns, such as aliased references, become rather complex to implement: say we have a function that takes a pair of pointers and a reference capability for each. If we want to pass in the same pointer twice, we will not be able to since reference capabilities are linear. While this is a feature for functions like `memmove`, for which aliased arguments is undefined behavior, it is not great for something like "compute the dot product of two vectors." Of course, such patterns are *possible* to implement, but it is tedious and manual.

Outside the world of purely functional programming, one of the most promising applications of sub-structural typing in modern programming language design is the Rust language's system of *ownership typing* [13, 9, 10]. By extending an affine type system with the ability to "borrow" resources without consuming them, the Rust language allows the programmer to perform complex, machine-checked resource management without using a garbage collector, allowing statically-checked memory safety and thread safety without runtime performance overhead. In particular, ownership types allow seamlessly modeling data structures such as mutexes, reference-counted pointers, containers, and channels in a type-safe way.

Unfortunately, for some low-level tasks and primitives, Rust programmers must drop down to "`unsafe`" code, which allows the manipulation of raw pointers but, consequently, comes with all the risks of C [13, 9, 10], as ownership and borrowing rules are left unchecked. This functionality is used "under the hood" to implement many of Rust's core data structures, including vectors (growable arrays), hash maps, mutexes, and reference-counted allocations, which expose a safe interface statically checked by Rust's type system. Rust promises that assuming the `unsafe` foundations of a program are sound, the program itself is free of undefined behavior *regardless* of the actions performed by any "safe" code. Proving this guarantee, along with formalizing the exact requirements for `unsafe` code to be considered sound, was the main aim of the RustBelt project [9].

The goal of this thesis is to generalize the work in Krishnaswami et al. [12] to the integration of dependent types and ownership types, which we call "dependent types with borrowing." In doing so, we hope to address the problems mentioned above with pointer-based modeling of imperative programs by, like Rust, performing the majority of such checking automatically within the type system rather than by manually juggling capabilities. More interestingly, since ownership types are a strict generalization of linear types, manual separation logic like management of locations and reference capabilities should still be possible in a system of dependent ownership types. Therefore, instead of dropping down to `unsafe` code, it should be possible to drop down to these more primitive operators instead and hence verify the safety of "`unsafe` blocks" without using external tools. Eventually, we hope such a system is a step towards an easy-to-use, safe, Rust-like systems programming with verified foundations that anyone can extend without dealing with complicated proof assistants and external tools.

To address this goal, we introduce the novel concepts of *instants* and *constraint sets*, which we use to encode ownership types, and attempt to design a simple dependently typed language around this paradigm. The main aim of this thesis is to explore the design space for such languages; consequently, we focus on experimenting with different possible features and implementations rather than formalizing a fixed set of rules. Nevertheless, we do attempt to have our work be at least amenable to formalization, and consequently, we provide a semi-formal exposition of our calculus in terms of typing rules.

1.2 Objectives

The main objective of this project is to explore design space around the integration of ownership types and dependent types. To do so, we introduce the "`isotope`" language, a simple dependently-typed lambda calculus supporting a notion of "borrowing" values, which we call the "dependent types with borrowing" model of dependent ownership types. This is implemented by introducing the novel notions of *instants* and *constraint sets*, which we use to encode a system of ownership typing. We intend `isotope` to be a minimal, experimental implementation of this design to study its theoretical and practical properties. We aim to give a semi-formal theoretical description of `isotope`, motivated by examples throughout, and explore some of its basic properties and potential extensions, concluding with the current status of an (unfinished) concrete implementation of the `isotope` language in Rust.

1.3 Thesis Structure

This thesis is composed of 6 chapters (including this introduction), which describe the background theory, typing rules and a sketch of a compilation algorithm for `isotope`, our implementation work thus far, and a conclusion.

- In Chapter 2, we cover the preliminary knowledge and background theory necessary to understand `isotope`. Beginning with a brief overview of the untyped lambda calculus in Section 2.1, we cover simple type theory and substructural typing in Section 2.2. We then give a brief review of dependent type theory, in the style of the Calculus of Constructions (borrowing from [17]), in Section 2.3. Changing gears, we give a brief description of the C programming language and it's basic features, as well as the problem of undefined behaviour, in Section 2.4. Finally, we give an overview of the Rust language and it's system of ownership types in Section 2.5.

- In Chapter 3, we cover the core `isotope` language and type system, and consider some example programs. Beginning with an informal overview of the language in Section 3.1, we give a rule-by-rule account of the type system and grammar of the language in subsequent sections.

- In Chapter 4, we give a sketch of an algorithm to lower `isotope` programs, as defined in Chapter 3, to a low-level C-like pseudocode.

- In Chapter 5, we describe the current status of the Rust implementation of `isotope`, and give a brief overview of some of the practical differences between the theory and the concrete language.

- In Chapter 6, we reflect on our current progress and discuss possible further work, completing the `isotope` interpreter and compiler, including support for nontermination, and including support for proper heap manipulation.

1.4 Achievements

1. Define a programming language and type system, `isotope`, which integrates dependent types and ownership types via the usage of instants and constraint sets, in Chapter 3.
2. Invent and implement an algorithm, the **borrowck** algorithm from Section 3.4, for checking constraint sets for consistency, as a functional analog to Rust's `borrowck` pass.
3. Give a sketch of an algorithm for compiling `isotope` programs into a low-level C-like pseudocode in Chapter 4.
4. Partially implement a type-checker and interpreter for `isotope`, as detailed in Chapter 5, in the Rust programming language.

Chapter 2

Background and Preliminaries

In this chapter, we cover the background and preliminary materials for the main body of the thesis. We briefly review the untyped lambda calculus in Section 2.1 and simple types and the Curry-Howard correspondence in Section 2.2. In Section 2.3, we give an overview of dependent type theory in the form of a summarized presentation of the Calculus of Inductive Constructions, the underpinnings of the Coq theorem prover. Section 2.3 will later form the core of the intuitionistic typing rules we introduce in Chapter 3 when formally developing our type theory. Changing gears, in Section 2.4, we briefly review the C programming language and some basic systems programming concepts. In Section 2.5, we compare and contrast this with a basic overview of Rust and its system of ownership types.

2.1 The Lambda Calculus

2.1.1 Terms

The lambda calculus is among the simplest intuitive examples of a programming language, and hence, makes a good setting for developing and analyzing type theories. As we will extensively use the lambda calculus in this thesis, we will review its essential properties. The set of lambda calculus terms, Λ, is usually defined inductively as follows:

$$\frac{x \in \mathcal{V}}{x \in \Lambda}\,\textsf{Var} \qquad \frac{s \in \Lambda \quad t \in \Lambda}{st \in \Lambda}\,\textsf{App} \qquad \frac{x \in \mathcal{V} \quad s \in \Lambda}{\lambda x.s \in \Lambda}\,\textsf{Abs} \tag{2.1}$$

where $\mathcal{V}$ is an infinite set of *variables*. We will use $s \equiv t$ to describe syntactic equality between terms in Λ. We assume application is left associative, i.e. $xyz \equiv (xy)z$. For brevity, we will write $\lambda xyz.t$ to mean $\lambda x.\lambda y.\lambda z.t$. Our goal is to interpret abstractions $\lambda x.s$ as functions via substitution: $(\lambda x.s)t$ should be equal to "s with t substituted for x." To make this rigorous, we define *substitution* of terms inductively, for all $x \in \mathcal{V}$ and $t \in \Lambda$ as follows:

$$\begin{aligned}
 & & x[t/x] &\equiv t \\
\forall y \in \mathcal{V} \setminus \{x\}, & & y[t/x] &\equiv y \\
\forall l, r \in \Lambda, & & (lr)[t/x] &\equiv (l[t/x])(r[t/x]) \\
\forall s \in \Lambda, & & (\lambda x.s)[t/x] &\equiv (\lambda x.s) \\
\forall s \in \Lambda, \forall y \in \mathcal{V} \setminus \{x\}, & & (\lambda y.s)[t/x] &\equiv \lambda y.s[t/x]
\end{aligned} \tag{2.2}$$

We define the set of *free variables*, i.e. those not bound by a λ-abstraction, of a term as follows:

$$\begin{aligned} & \mathsf{fv}(x) && = \{x\} \\ \forall l, r \in \Lambda, \quad & \mathsf{fv}(lr) && = \mathsf{fv}(l) \cup \mathsf{fv}(r) \\ \forall s \in \Lambda, \forall x \in \mathcal{V}, \quad & \mathsf{fv}(\lambda x.s) && = \mathsf{fv}(s) \setminus \{x\} \end{aligned} \tag{2.3}$$

One downside of the inductive definition in Equation 2.1 is that $\lambda y.y \not\equiv \lambda x.x$, even though they only differ in the name of a bound variable. We do not want this, and so we quotient Λ by *α-conversion*, [1] i.e. the renaming of bound variables:

$$\forall x, y \in \mathcal{V}, \forall s \in \Lambda, y \notin \mathsf{fv}(\lambda x.s) \implies \lambda x.s \equiv \lambda y.s[y/x] \tag{2.4}$$

We will take on faith that substitution $s[t/x]$ is well-defined under α-conversion.

2.1.2 Reduction

struct VecU8 u = double_vec(v); Given a binary relation R on Λ, we may define *reduction* with respect to R, $\rightarrow_R$, as follows:

$$\frac{(s,t) \in R}{s \rightarrow_R t} \qquad \frac{s \rightarrow_R s'}{st \rightarrow_R s't}\,\mathsf{Left} \qquad \frac{t \rightarrow_R t'}{st \rightarrow_R st'}\,\mathsf{Right} \qquad \frac{s \rightarrow_R s'}{\lambda x.s \rightarrow_R \lambda x.s'}\,\mathsf{Abs} \tag{2.5}$$

We think of $\rightarrow_R$ as defining a single step of reduction: we will denote the transitive and reflexive closure of $\rightarrow_R$ as $\twoheadrightarrow_R$, and the equivalence relation obtained by quotienting Λ by $\rightarrow_R$ as $=_R$. We say a term is in R-normal form if $\neg\exists t \in \Lambda, s \rightarrow_R t$, i.e. it cannot be reduced further.

---

[1] In our actual implementation, we represent lambda terms by *de-Bruijn indices*, which automatically quotient under α-conversion, but we will used the named formalism in our background exposition.

We can use this machinery to give a basic semantics to the lambda calculus by thinking of *abstractions* $\lambda x.s$ as *functions*, which may be applied by *substitution*, via the β-reduction relation below:

$$\beta = \{((\lambda x.s)t, s[t/x]) : x \in \mathcal{V},\ s, t \in \Lambda\} \tag{2.6}$$

It turns out that this is enough to give us a Turing-complete programming language, in which terms are evaluated by reducing to β-normal form: we can encode arbitrary data structures and control flow as functions in this calculus. As a very simple example, we may represent the Boolean values true and false by "projection functions" as follows:

$$\mathsf{true} \equiv \lambda xy.x, \quad \mathsf{false} \equiv \lambda xy.y \implies \mathsf{if} \equiv \lambda ctf.ctf \tag{2.7}$$

This recovers the usual control flow

$$\mathsf{if}\ \mathsf{true}\ l\ r \twoheadrightarrow_\beta l, \qquad \mathsf{if}\ \mathsf{false}\ l\ r \twoheadrightarrow_\beta r \tag{2.8}$$

Unfortunately, not all "equal" functions (in the mathematical sense) are actually β-equal: as a trivial example, we have

$$\forall f \in \mathcal{V}, \forall s \in \Lambda, (\lambda x.fx)s =_\beta fs,\ \text{but}\ \lambda x.fx \neq_\beta f \tag{2.9}$$

i.e. $=_\beta$ does not satisfy *function extensionality*. We'd like to automatically detect as many such equalities as possible, but we need to worry about *consistency*: a reduction relation R is *inconsistent* if

$$\forall s, t \in \Lambda, s =_R t \tag{2.10}$$

For example, we trivially have that $\beta \cup \{(\mathsf{true}, \mathsf{false})\}$ is inconsistent, as

$$\forall s, t \in \Lambda, \mathsf{true}\ s\ t \twoheadrightarrow_\beta s, \quad \mathsf{false}\ s\ t \twoheadrightarrow_\beta t \tag{2.11}$$

We will again take on faith that β is indeed consistent. It turns out that simply considering all reductions of the form

$$\eta = \{(\lambda x.fx, f) : f \in \mathcal{V}\} \tag{2.12}$$

yields a consistent relation $\beta\eta = \beta \cup \eta$. Indeed, $\beta\eta$ is actually *maximally consistent*: if s, t are closed $\beta\eta$-normal terms then $\beta\eta \cup \{(s, t)\}$ is inconsistent [11], so, in a sense, it's the best we can do!

Another issue to consider is that not all terms have a β-normal form; for example,

$$\mathbf{\Omega} = (\lambda x.xx)(\lambda x.xx) \rightarrow_\beta \mathbf{\Omega} \tag{2.13}$$

reduces to itself, and hence has no β-normal form. We call a term with an R-normal form *weakly R-normalizable*. Some terms have a β-normal form, and yet do not necessarily reduce to it; for example,

$$(\lambda x.y)\mathbf{\Omega} \rightarrow_b y \tag{2.14}$$

has a normal form, y, but we could also just infinitely reduce the $\mathbf{\Omega}$ on the right. On the other hand, a term which always reduces to a normal form after a finite number of steps is called *strongly R-normalizable*.

In the following sections, we will discuss lambda calculi with "a notion of reduction" →, to which rules will be added (such as, e.g., Equation 2.26). In reality, we are defining a notion of reduction R, to which these equations are added; we implicitly extend the definition of $\to_R$ to recursively consider components of expressions like (a, b). We will, in these cases, use $=$ to mean $=_R$, with $\equiv$ reserved as usual for syntactic equality.

2.2 Simple Types

In this section, we give an overview of the *simply typed lambda calculus*, and consider various possible extensions to this theory, including substructural lambda calculi and the addition of ground types and data structures, such as the natural numbers $\mathbb{N}$ or the tensor product $\otimes$. This is probably the simplest example of a *type system*, and, consequently, is used as a basis for much of modern research into type theory.

2.2.1 Simple Type Theory

We will now consider a simple type system for the lambda calculus, which we will later use as a base to introduce features such as type dependency and sub-structural typing. We define the set of *simple types* $\mathcal{T}$ over a set of *atomic types* $\mathcal{A}$ as follows:

$$\frac{A \in \mathcal{A}}{A \in \mathcal{T}}\,\textsf{Atomic} \qquad \frac{A \in \mathcal{T} \quad B \in \mathcal{T}}{A \multimap B \in \mathcal{T}}\,\textsf{Arrow} \tag{2.15}$$

The arrow operator is right associative, i.e. $A \multimap B \multimap C = A \multimap (B \multimap C)$. We define a *typing context* Γ to be a *multiset* of assignments $x : A$ of variables $x \in \mathcal{V}$ to types $A \in \mathcal{T}$ which is *consistent*; i.e., if $x : A \in \Gamma$ and $x : A' \in \Gamma$ then $A = A'$. The *catenation* of two contexts, Γ, Δ is defined if, as a multiset, it is consistent. We proceed to define *typing judgements* for terms $s \in \Lambda$ of the form $\Gamma \vdash x : A$. We provide the following rules:

$$\frac{}{x : A \vdash x : A}\,\mathsf{Var} \qquad \frac{\Gamma \vdash s : A \multimap B \quad \Delta \vdash t : A}{\Gamma, \Delta \vdash st : B}\,\mathsf{App} \qquad \frac{\Gamma, x : A \vdash s : B}{\Gamma \vdash \lambda x.s : A \multimap B}\,\mathsf{Abs} \tag{2.16}$$

Type-checking a simple term $\lambda xf.fx$ using these rules, we obtain deduction tree

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : A \vdash x : A}\,\mathsf{Var} \quad \dfrac{}{f : A \multimap B \vdash f : A \multimap B}\,\mathsf{Var}}{x : A, f : A \multimap B \vdash fx : B}\,\mathsf{App}}{x : A \vdash \lambda f.fx : (A \multimap B) \multimap B}\,\mathsf{Abs}}{\vdash \lambda xf.fx : A \multimap (A \multimap B) \multimap B}\,\mathsf{Abs} \tag{2.17}$$

The rules in Equation 2.16, however, are not complete: in their current form every term appearing in a mapping Γ must be used exactly once: this is called a *linear* type system. This means that functions which discard their arguments, like the projection $\lambda xy.x$, or functions which use them multiple times, like $\lambda fx.fxx$, will not type-check! Such functions, in general, are called *nonlinear*. Specifically

Definition 1 (Linear/affine/relevant terms)**.** *A term $s \in \Lambda$ is called* affine *if every variable in s is used at most once, and* relevant *if every term is used at least once. A term which is both affine and relevant is called* linear*. A term which is not linear is* nonlinear*. Formally:*

- *$\forall x \in \mathcal{V}$, x is both affine and relevant, i.e., linear*
- *Given $s, t \in \Lambda$,*
  - *st is affine if s and t are both affine and $\mathsf{fv}(s) \cap \mathsf{fv}(t) = \varnothing$.*
  - *st is relevant if s and t are both relevant.*
- *Given $s \in \Lambda$,*

  - $\lambda x.s$ *is affine if* s *is affine*

  - $\lambda x.s$ *is relevant if* s *is relevant and* $x \in \mathsf{fv}(s)$

We make the following claim

Claim 1. *In the type system with only the rules given by Equation 2.16, if* $\vdash s : T$ *for some* s*, then* s *is linear.*

If we want to allow discarding arguments (i.e., terms which are not necessarily *relevant*), we must introduce a *weakening* rule, as follows:

$$\frac{\Gamma \vdash s : A}{\Gamma, x : B \vdash s : A}\,\textsf{Weakening} \tag{2.18}$$

This leaves us with an *affine* type system, where terms can be used *at most* once, and we can now typecheck

$$\cfrac{\cfrac{\cfrac{\cfrac{}{x : A \vdash x : A}\,\textsf{Var}}{x : A, y : B \vdash x : A}\,\textsf{Weakening}}{x : A \vdash \lambda y.x : B \multimap A}\,\textsf{Abs}}{\vdash \lambda xy.x : A \multimap B \multimap A}\,\textsf{Abs} \tag{2.19}$$

In particular, we make the following claim

Claim 2. *In the type system with only the rules given by Equations 2.16 and 2.18, if* $\vdash s : T$ *for some* s*, then* s *is affine.*

On the other hand, if we want to allow duplicating arguments, we must introduce a *contraction* rule, as follows:

$$\frac{\Gamma, x : A, x : A \vdash s : B}{\Gamma, x : A \vdash s : B}\,\textsf{Contraction} \tag{2.20}$$

and we can now typecheck

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{f : A \multimap A \multimap B \vdash f : A \multimap A \multimap B}\,\textsf{Var} \quad \dfrac{}{x : A \vdash x : A}\,\textsf{Var}}{f : A \multimap A \multimap B, x : A \vdash fx : A \multimap B}\,\textsf{App} \quad \dfrac{}{x : A \vdash x : A}\,\textsf{Var}}{f : A \multimap A \multimap B, x : A, x : A \vdash fxx : B}\,\textsf{App}}{f : A \multimap A \multimap B, x : A \vdash fxx : B}\,\textsf{Contraction}}{f : A \multimap A \multimap B \vdash \lambda x.fxx : A \multimap B}\,\textsf{Abs}}{\vdash \lambda fx.fxx : (A \multimap A \multimap B) \multimap A \multimap B}\,\textsf{Abs} \tag{2.21}$$

This leaves us with a *relevant* type system, where terms must be used *at least* once. In particular, we make the following claim

Claim 3. *In the type system with only the rules given by Equations 2.16 and 2.20, if* $\vdash s : T$ *for some* s*, then* s *is relevant.*

If we introduce both Weakening and Contraction from Equations 2.18 and 2.20 respectively, there is no restriction on how terms may be used, so we have an *intuitionistic* type system. We will generally denote the arrow in an intuitionistic type system as $\rightarrow$ (rather than $\multimap$). [2]

2.2.2 Ground Types

In the untyped lambda calculus, we often encode datatypes, such as booleans and integers, as pure functions, as we demonstrated for booleans in Section 2.1.2. While this remains possible in a typed language, it is often practically more convenient to introduce a set of *ground types*, such as bool and $\mathbb{N}$. Each such type is introduced with a set of *introduction rules*, such as

$$\dfrac{}{\textsf{true} : \textsf{bool}}\,\textsf{true-intro} \qquad \dfrac{}{\textsf{false} : \textsf{bool}}\,\textsf{false-intro} \qquad \dfrac{}{0 : \mathbb{N}}\,\textsf{0-intro} \qquad \dfrac{n : \mathbb{N}}{\textsf{s}n : \mathbb{N}}\,\textsf{succ-intro} \tag{2.22}$$

---

[2] Note that in some works, $\multimap$ denotes the *linear arrow*, for which neither Weakening or Contraction hold, while $\rightarrow$ denotes the *intuitionistic arrow*, for which both hold. Here, however, $\multimap$ means *either* the linear arrow, affine arrow, relevant arrow, or intuitionistic arrow, while $\rightarrow$ continues to specifically denote the intuitionistic arrow.

where s is the *successor operation*, i.e. $\mathsf{s}n = n+1$. This allows us to define integer literals such as $1 = \mathsf{s}0$, $2 = \mathsf{s}1 = \mathsf{s}(\mathsf{s}0)$, and so on. Similarly, we can define *eliminators* such as

$$\frac{\Gamma \vdash b : \mathsf{bool} \quad \Delta \vdash t : A \quad \Delta \vdash f : A}{\Gamma, \Delta \vdash \mathsf{if}\ b\ t\ f : A}\ \mathsf{bool\text{-}elim} \qquad \mathsf{if}\ \mathsf{true}\ t\ f \to t \qquad \mathsf{if}\ \mathsf{false}\ t\ f \to f \tag{2.23}$$

We could follow a similar approach for $\mathbb{N}$ (and, in Section 2.3.8, we will), or, alternatively, we could define a collection of builtin operators, such as $+ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ or $==: \mathbb{N} \to \mathbb{N} \to \mathsf{bool}$, and associated reduction rules.

2.2.3 Product Types

We may also want to add in additional types corresponding to common data structures, such as tuples of elements. To do so, we may add more structure to $\mathcal{T}$ by introducing additional *type formers*, such as the *tensor product* of types A and B, $A \otimes B$, which has typing rules

$$\frac{A \in \mathcal{T} \quad B \in \mathcal{T}}{A \otimes B \in \mathcal{T}}\ \otimes\text{-form} \qquad \frac{\Gamma \vdash a : A \quad \Delta \vdash b : B}{\Gamma, \Delta \vdash (a, b) : A \otimes B}\ \otimes\text{-intro} \tag{2.24}$$

$$\frac{\Gamma \vdash e : A \times B \quad \Delta, a : A, b : B \vdash c : C}{\Gamma, \Delta \vdash \mathsf{let}\ (a, b) = e\ \mathsf{in}\ c : C}\ \otimes\text{-elim} \tag{2.25}$$

and reduction rule

$$\mathsf{let}\ (a', b') = (a, b)\ \mathsf{in}\ c \ \to\ c[a/a'][b/b'] \tag{2.26}$$

This lets us define, for example, the tensor product of morphisms internally as follows:

$$\forall f : A \to C, g : B \to D, f \otimes g = \lambda e.\mathsf{let}\ (a, b) = e\ \mathsf{in}\ (fa, fb) : A \otimes B \to C \otimes D \tag{2.27}$$

However, if we define the usual projection operators as follows

$$\forall i \in \{1, 2\}, \pi_i \equiv \lambda e.\mathsf{let}\ (x_1, x_2) = e\ \mathsf{in}\ x_i \tag{2.28}$$

we require $\mathsf{Weakening}$ to derive the expected type

$$\dfrac{\dfrac{\dfrac{}{e : A_1 \times A_2 \vdash e : A_1 \otimes A_2}\,\mathsf{Var} \qquad \dfrac{\dfrac{}{x_i : A_i \vdash x_i : A_i}\,\mathsf{Var}}{x_1 : A_1, x_2 : A_2 \vdash x_i : A_i}\,\mathsf{Weakening}}{e : A_1 \otimes A_2 \vdash \mathsf{let}\ (x_1, x_2) = e\ \mathsf{in}\ x_i : A_i}\,\otimes\textsf{-elim}}{\vdash \pi_i : A_1 \times A_2 \to A_i}\,\mathsf{Abs} \tag{2.29}$$

and, from this, the expected typing rules

$$\dfrac{\begin{array}{c}\vdots \\ \vdash \pi_i : A_1 \times A_2 \to A_i\end{array} \quad \Gamma \vdash e : A_1 \times A_2}{\Gamma \vdash \pi_i e : A_i}\,\mathsf{App} \tag{2.30}$$

Alternatively, we may introduce a *conjunction* type equipped with projection operators

$$\dfrac{A \in \mathcal{T} \quad B \in \mathcal{T}}{A \& B \in \mathcal{T}}\,\&\textsf{-form} \qquad \dfrac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma, \Delta \vdash (a, b) : A \& B}\,\&\textsf{-intros} \qquad \dfrac{\Gamma \vdash e : A_1 \& B_2}{\Gamma \vdash \pi_i e : A_i}\,\&\textsf{-elim} \tag{2.31}$$

In this case, we could define [3]

$$\lambda ef.\mathsf{let}\ (a, b) = e\ \mathsf{in}\ fab \equiv \lambda ef.f(\pi_1 e)(\pi_2 e) \tag{2.32}$$

---

[3] We could directly implement Equation 2.26 by taking $\mathsf{let}\ (a, b) = e\ \mathsf{in}\ c \equiv c[\pi_1 e/a][\pi_2 e/b]$, but this would require some lemmas on typing subterms. In general, we can get the same behaviour using Equation 2.32 by writing $(\lambda ef.\mathsf{let}\ (a, b) = e\ \mathsf{in}\ fab)(\lambda ab.c)$, and this actually has the type of a (categorical) eliminator for a Cartesian product.

we require Contraction to derive the expected type

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{e : A\&B \vdash e : A\&B}\,\textsf{Var}}{e : A\&B \vdash \pi_1 e : A}\,\textsf{\&-elim}}{e : A\&B, f : A \to B \to C \vdash f(\pi_1 e) : B \to C}\,\textsf{App} \quad \dfrac{\dfrac{}{e : A\&B \vdash e : A\&B}\,\textsf{Var}}{e : A\&B \vdash \pi_2 e : B}\,\textsf{\&-elim}}{e : A\&B, e : A\&B, f : A \to B \to C \vdash f(\pi_1 e)(\pi_2 e) : C}\,\textsf{App}}{e : A\&B, f : A \to B \to C \vdash f(\pi_1 e)(\pi_2 e) : C}\,\textsf{Contraction}}{e : A\&B \vdash \lambda f.f(\pi_1 e)(\pi_2 e) : (A \to B \to C) \to C}\,\textsf{Abs}}{\vdash \lambda ef.f(\pi_1 e)(\pi_2 e) : (A\&B) \to (A \to B \to C) \to C}\,\textsf{Abs} \tag{2.33}$$

Hence, in intuitionistic logic, & and ⊗ are "the same thing" (i.e., isomorphic); we hence write *both* as the *Cartesian product* ×.

2.2.4 Coproduct Types

Similarly, given types $A + B$, we may define their *coproduct*, or *sum*, $A + B$, to have typing rules

$$\dfrac{A \in \mathcal{T} \quad B \in \mathcal{T}}{A + B \in \mathcal{T}}\,\textsf{+-form} \qquad \dfrac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{l}\ a : A + B}\,\textsf{inl-intro} \qquad \dfrac{\Gamma \vdash b : B}{\Gamma \vdash \mathsf{r}\ b : A + B}\,\textsf{inr-intro} \tag{2.34}$$

We interpret this type as being "either A or B;" consequently, we may introduce eliminator

$$\dfrac{\Gamma \vdash e : A + B \quad \Delta, a : A \vdash c_a : C \quad \Delta, b : B \vdash c_b : C}{\Gamma, \Delta \vdash \mathsf{case}_+\ e\ \{\mathsf{l}\ a \mapsto c_a\ \mathsf{r}\ b \mapsto c_b\} : C}\,\textsf{+-elim} \tag{2.35}$$

with rules

$$\mathsf{case}_+\ (\mathsf{l}\ a')\ \{\mathsf{l}\ a \mapsto c_a\ \mathsf{r}\ b \mapsto c_b\} = c_a[a'/a], \tag{2.36}$$

$$\mathsf{case}_+\ (\mathsf{r}\ b')\{\mathsf{l}\ a \mapsto c_a\ \mathsf{r}\ b \mapsto c_b\} = c_b[b'/b] \tag{2.37}$$

This gives us a basic system of *algebraic data types*, though we postpone a discussion of inductive types to Section 2.3.

2.2.5 Linear and Nonlinear Type Theory

In a linear type system, we check linearity via the properties of the typing context. Of course, we may have certain types, such as bool, for example, which we do not want to be linear; for these particular types, we could introduce contraction rules and weakening rules as follows:

$$\frac{\Gamma, x : \mathsf{bool}, x : \mathsf{bool} \vdash s : A}{\Gamma, x : \mathsf{bool} \vdash s : A}\,\textsf{bool-cntr} \qquad \frac{\Gamma \vdash s : A}{\Gamma, x : \mathsf{bool} \vdash s : A}\,\textsf{bool-weak} \tag{2.38}$$

We could instead make a type, e.g., affine but not relevant, by only including Contraction but not Weakening, or vice versa. In general, we may also be interested in what objects we could define should the linearity restriction be lifted for a certain *object* or function argument, or perhaps some portion of a data structure (e.g., the left argument of a tuple); to do so, we can introduce the *exponential type*

$$\frac{\Gamma, x : A \vdash s : B}{\Gamma, x :!A \vdash s : B}\,\textsf{exp-intro} \qquad \frac{\Gamma, x :!A, x :!A \vdash s : B}{\Gamma, x :!A \vdash s : B}\,\textsf{exp-cntr} \qquad \frac{\Gamma \vdash s : B}{\Gamma, x :!A \vdash s : B}\,\textsf{exp-weak} \tag{2.39}$$

Now, for example, we could make true and false to be intuitionistic by defining, e.g., true :!bool, but still allowing *other* terms in the same type to be linear. Indeed, we can use the exponential to derive an embedded version of intuitionistic type theory inside linear type theory by using the *Girard encoding*

$$A \to B \equiv !A \multimap B \tag{2.40}$$

Similarly, we could define a Cartesian product

$$A \times B \equiv !(A \& B) \tag{2.41}$$

and so on.

Instead of introducing an exponential type, we could also consider a type system with separate intuitionistic and linear contexts equipped with functors F, G, which take intuitionistic terms and types into linear terms and types, and vice versa (with separate linear type formers $\otimes$, $\times$, etc., and intuitionistic type formers $\times$, etc.). This approach is Benton's linear/non-linear calculus [3], which Krishnaswami et al. extend to support type dependency in the *intuitionistic* function space [12].

Alternatively, we could define *linearity on the arrow* (resp. affinity/relevancy), by, in an intuitionistic type system, admitting the type $A \multimap B$ for 1-linear functions $f : A \to B$, with a 1-linear function defined as follows

Definition 2 (Linear/relevant/affine term)**.** *Given a variable x, a term $s \in \Lambda$ is* relevant in x *if x appears at least once in s, i.e. $x \in \mathsf{fv}(s)$, and* affine in x *if x appears at most once in s. A term s is* linear in x *if it is both affine and relevant in x, i.e. uses x exactly once. A function f is n-*affine *if it uses its n^{th} argument at most once, and n-*relevant *if it uses it's argument at least once; if f is both affine and relevant, i.e., uses it's argument exactly once, we call it n-*linear*.*

In particular, given a variable x, we provide the following mutually inductive definition:

- *For all variables y, y is affine in x, and is both an n-affine and n-relevant function for any n.*

- *A lambda function* $\lambda y.s$ *(assuming* $y \neq x$*, renaming as necessary) is*
  - *affine in* x *if* s *is affine in* x
  - 1*-affine if* s *is affine in* y
  - 1*-relevant if* s *is relevant in* y
  - $n+1$*-affine if* s *is* n*-affine*
  - $n+1$*-relevant if* s *is* n*-relevant*
- *An application* st *is*
  - *affine in* x *if*
    - s *is an affine function, and is* 1*-affine in* x*.*
    - t *is affine in* x*.*
    - $x \notin \mathsf{fv}(s) \cap \mathsf{fv}(t)$*.*
  - n*-affine if* s *is* $n+1$*-affine*
  - n*-relevant if* s *is* $n+1$*-relevant*

Note that a term s *is linear/affine/relevant w.r.t. Definition 1 if*

- *For all* n*,* s *is* n*-linear/affine/relevant*
- *For all* $x \in \mathsf{fv}(s)$*,* s *is linear/affine/relevant in* x

This is the approach used by Linear Haskell [4]. One advantage of this approach is that our typing judgments become easier to work with, as intuitionistic rules mean we do not need to worry about resource management while performing deductions, linearity instead being a separate syntactic check. This approach also makes it easier to mix linear and nonlinear functions. On the other hand, dealing with linearity in a particular index (e.g., 5-linearity) can be challenging. Our type system in Chapter 3 will use a combination of these approaches.

2.2.6 Intuitionistic Conventions

In an intuitionistic setting, we will apply rules such as Var, Weakening, and Contraction implicitly except when we want to explicitly demonstrate their necessity. We will also tend to express typing rules using only one typing context. For example, we would express App as

$$\frac{\Gamma \vdash s : A \to B \quad \Gamma \vdash t : A}{\Gamma \vdash st : B}\,\textsf{App-int} \tag{2.42}$$

This is equivalent to the original rule in Equation 2.16, as we have

$$\begin{array}{c} \dfrac{\dfrac{\Gamma \vdash s : A \to B \quad \Gamma \vdash t : A}{\Gamma, \Gamma \vdash st : B}\,\textsf{App}}{\vdots}\,\textsf{Contraction} \\ \Gamma \vdash st : B \end{array} \qquad \dfrac{\begin{array}{c} \dfrac{\Gamma \vdash s : A \to B}{\vdots}\,\textsf{Weakening} \\ \Gamma, \Delta \vdash s : A \to B \end{array} \qquad \begin{array}{c} \dfrac{\Delta \vdash t : A}{\vdots}\,\textsf{Weakening} \\ \Gamma, \Delta \vdash t : A \end{array}}{\Gamma, \Delta \vdash st : B}\,\textsf{App-int} \tag{2.43}$$

2.3 Dependent Types

Type dependency, or allowing types to depend on terms, enables programmers to give precise functional specifications of a program's behavior in its type signature. As an example, consider a function $f : \mathbb{N} \to [\mathbb{N}]$, where $[A]$ denotes the type of lists of elements of A. We may know that, say, the length of the list $f(n)$ is always $2n + 3$, but there's no real way to encode that in the type signature of the function. Consequently, we may have to include many unnecessary length checks in our code, impacting performance, or face the possibility of bugs or even undefined behavior. However, with type dependency, we can address this problem by simply giving f the dependent function type $\Pi n : \mathbb{N}.[\mathbb{N}; 2n+3]$, where $[A; n]$ is the type of arrays of length n. In this section, we will give a description of the Calculus of Inductive Constructions, a dependently typed lambda calculus extended with *inductive types*, which give a general formalism for the data types described in Section 2.2.

2.3.1 Typing Universes

Rather than introduce a set of atomic type variables, we instead introduce a family of universe types $\mathcal{U}_i$ for $i \in \mathbb{N}$, with typing rule

$$\frac{}{\vdash \mathcal{U}_i : \mathcal{U}_{i+1}}\ \mathcal{U}\text{-form} \tag{2.44}$$

Unfortunately, we cannot introduce just a single typing universe $\mathcal{U} : \mathcal{U}$ (i.e., "type in type"), as due to *Girard's paradox*, this would allow us to produce, for any type A, a term $\bot : A$ (i.e., an infinite loop), which would lead to our type theory's interpretation as a logic via the Curry-Howard correspondence (see Subsection 2.3.6) to be inconsistent (as we could provide a "proof" of every proposition, though it would be non-normalizing) [5].

We will consider any term $A : \mathcal{U}_i$ to be a valid type, and hence allow typing judgements of the form $a : A$ (i.e., we assume *Russell-style* typing universes). Since we want to emulate the $\mathcal{U} : \mathcal{U}$ case as closely as possible (while still maintaining consistency), we want $A : \mathcal{U}_i$ to imply $A : \mathcal{U}_{i+1}$; while this is often simply introduced as a single rule (e.g., in [18]) , we will instead introduce a *subtyping judgement* $A <: B$ with rules

$$\frac{}{A <: A} \qquad \frac{}{\mathcal{U}_i <: \mathcal{U}_{i+1}} \qquad \frac{A <: B \quad a : A}{a : B} \tag{2.45}$$

In spirit, $A <: B$ means "anything with type A can be substituted anywhere we expect a term with type B."

Alternatively, we could use *Tarski-style* universes, in which $A : \mathcal{U}_i$ is not a valid type but rather corresponds to a type $\mathrm{El}_i(A)$, recovering a strict separation between types and terms. In this case, we could introduce functions $j_i : \mathcal{U}_i \to \mathcal{U}_{i+1}$ with $\mathrm{El}_i(A) = \mathrm{El}_{i+1}(j_i A)$

2.3.2 Dependent Function Types

We can now introduce the *dependent function type*, or Π-type,

$$\frac{\Gamma \vdash A : \mathcal{U}_i \quad \Delta, x : A \vdash B : \mathcal{U}_i}{\Gamma \vdash \Pi x : A, V : \mathcal{U}_i}\ \textsf{Π-form} \tag{2.46}$$

We then provide *introduction* and *elimination* rules

$$\frac{\Gamma \vdash s : B}{\Gamma \vdash \lambda x.s : \Pi x : A.B}\ \textsf{Π-intro} \qquad \frac{\Gamma \vdash s : \Pi x : A.B \quad \Gamma \vdash t : A}{\Gamma \vdash st : B[t/x]}\ \textsf{Π-elim} \tag{2.47}$$

Note this allows us to reinterpret the intuitionistic arrow $A \to B$ as shorthand for the dependent function type $\Pi - : A.B$; in this case, the rules $\textsf{Abs}$ and $\textsf{App}$ from Equation 2.47 correspond exactly to the simply typed ones in Equation 2.16. We can now, for example, type the polymorphic identity function $\lambda A.\lambda x.x$ as follows:

$$\frac{\dfrac{A : \mathcal{U}_i, x : A \vdash x : A}{A : \mathcal{U}_i \vdash \lambda x.x : A \to A}\ \textsf{Π-intros}}{\vdash \lambda A.\lambda x.x : \Pi A : \mathcal{U}_i.A \to A}\ \textsf{Π-intros} \tag{2.48}$$

2.3.3 Subtyping and Variance

As an element of a dependent function type $f : \Pi x : A.B$ takes an argument a of type A, and returns a result of type $B[a/x]$, it follows that if $A' <: A$, we should be able to replace f with a function of type $g : \Pi : x : A'.B$. This yields rule

$$\frac{A' <: A}{\Pi x : A.B <: \Pi x : A'.B} \tag{2.49}$$

Note that this judgement "flips" the order of the subtyping relation; hence, the argument type parameter of the type former Π is called *contravariant*.Conversely, *assuming* that $B <: B'$ implies that $B[a/x] <: B'[a/x]$ (a property that we will not prove for this type system), then we could also replace f with a function of type $g : \Pi x : A.B'$ to get a result of type $B'[a/x]$ (which, via subtyping, we may assume can be interpreted as a term of type $B[a/x]$). This yields rule

$$\frac{B <: B'}{\Pi x : A.B <: \Pi x : A.B'} \tag{2.50}$$

Here the order of the subtyping relation is preserved; hence, the result type parameter of the type former Π is *covariant*. On the other hand, if we defined a type former $\mathsf{Id}\ A \equiv A \to A$, A appears in both a *contravariant* (the left) and *covariant* (the right) position; hence, for us to have $\mathsf{Id}\ A <: \mathsf{Id}\ B$, we would require $A <: B$ and $B <: A$, i.e., that A and B were equivalent when considered as types. Such a parameter is called *invariant*.

2.3.4 Booleans

In the Section 2.3.2, we defined Π-types as dependent analogs to the simple function type $A \to B$; similarly, we may define dependent analogs of Section 2.2's ground types and data types as follows. For example, as in Section 2.2.2, we may introduce a type of booleans

$$\frac{}{\vdash \mathsf{bool} : \mathcal{U}_1}\,\textsf{bool-form} \qquad \frac{}{\vdash \mathsf{true} : \mathsf{bool}}\,\textsf{true-intro} \qquad \frac{}{\vdash \mathsf{false} : \mathsf{bool}}\,\textsf{false-intro} \tag{2.51}$$

We introduce *pattern matching* via the *case statement*

$$\frac{\Gamma \vdash F : \mathsf{bool} \to \mathcal{U}_i \quad \Gamma \vdash t : F\ \mathsf{true} \quad \Gamma \vdash f : F\ \mathsf{false} \quad \Gamma \vdash b : \mathsf{bool}}{\Gamma \vdash \mathsf{case}_{\mathsf{bool}}\ b\ \{\mathsf{true} \mapsto t, \mathsf{false} \mapsto f\} : F\ b}\,\textsf{bool-elim} \tag{2.52}$$

having reduction rules

$$\mathsf{case}_{\mathsf{bool}}\ \mathsf{true}\ \{\mathsf{true} \mapsto t, \mathsf{false} \mapsto f\} \to t \qquad \mathsf{case}_{\mathsf{bool}}\ \mathsf{false}\ \{\mathsf{true} \mapsto t, \mathsf{false} \mapsto f\} \to f \tag{2.53}$$

Notice that, unlike the if-statement from Section 2.2.2, this allows the type of the true and false branches to differ, e.g., the following term typechecks

$$\begin{aligned} &\lambda x.\mathsf{case}_{\mathsf{bool}}\ x \left\{ \begin{array}{ll} \mathsf{true} & \mapsto \mathsf{true}, \\ \mathsf{false} & \mapsto \lambda x.x \end{array} \right\} \\ &\qquad : \Pi x : \mathsf{bool}.\mathsf{case}_{\mathsf{bool}}\ x\ (\lambda - .\mathcal{U}_1) \left\{ \begin{array}{ll} \mathsf{true} & \mapsto \mathsf{bool}, \\ \mathsf{false} & \mapsto \mathsf{bool} \to \mathsf{bool} \end{array} \right\} \end{aligned} \tag{2.54}$$

As demonstrated above, we may recover the original if-statement behaviour by setting F to be a constant function $\lambda - .A$ for some $A : \mathcal{U}_i$. We may hence define

$$\mathsf{if}\ b\ t\ f \equiv \mathsf{case}\ b\ \{\mathsf{true} \mapsto t, \mathsf{false} \mapsto f\} \tag{2.55}$$

Logical operations may then be defined in the obvious manner, e.g.

$$\mathsf{not} \equiv \lambda x.\mathsf{if}\ x\ \mathsf{false}\ \mathsf{true} : \mathsf{bool} \to \mathsf{bool} \tag{2.56}$$

$$\mathsf{and} \equiv \lambda xy.\mathsf{if}\ x\ y\ \mathsf{false} : \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool} \tag{2.57}$$

$$\mathsf{or} \equiv \lambda xy.\mathsf{if}\ x\ \mathsf{true}\ y : \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool} \tag{2.58}$$

2.3.5 Dependent Pair Types

Similarly, we may define Σ-types as dependent analogs to the Cartesian product type $A \times B$ from Section 2.2.3. In particular, we introduce typing rules

$$\frac{\Gamma \vdash A : \mathcal{U}_i \quad \Gamma, x : A \vdash B : \mathcal{U}_i}{\Gamma \vdash \Sigma x : A.B : \mathcal{U}_i}\ \Sigma\text{-form} \qquad \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B[a/x]}{\Gamma \vdash (a, b) : \Sigma x : A.B}\ \Sigma\text{-intro} \tag{2.59}$$

with eliminator

$$\frac{\Gamma \vdash F : \Sigma x : A.B \to \mathcal{U}_i \quad \Gamma \vdash e : \Sigma x : A.B \quad \Gamma, a : A, b : B[a/x] \vdash c : F(a, b)}{\Gamma \vdash \mathsf{case}_\Sigma\ e\ \{(a, b) \mapsto c\} : F\ e}\ \Sigma\text{-elim} \tag{2.60}$$

having reduction rule

$$\mathsf{case}_\Sigma\ (a', b')\ \{(a, b) \mapsto c\} \to c[a'/a][b'/b] \tag{2.61}$$

Note this is, in essence, the let-statement from Section 2.2.3 if we choose $F = \lambda - .A$ for some constant $A : \mathcal{U}_i$; as a consequence, we may define

$$\mathsf{let}\ (a, b) = e\ \mathsf{in}\ c \equiv \mathsf{case}_\Sigma\ e\ \{(a, b) \mapsto c\} \tag{2.62}$$

We may hence define the Cartesian product eliminators π_1, π_2 in the usual manner as in Section 2.2.3, i.e., using Equation 2.28. Their typing derivations, however, are different; in particular, we have

$$\dfrac{\dfrac{\Gamma \vdash \lambda - .A : \Sigma x : A.B \to \mathcal{U}_i \quad \Gamma, a : A \vdash a : A}{\Gamma, e : \Sigma x : A.B \vdash \mathsf{let}\ (a, b) = e\ \mathsf{in}\ a : A}\ \Sigma\text{-elim}}{\Gamma \vdash \pi_1 \equiv \lambda e.\mathsf{let}\ (a, b) = e\ \mathsf{in}\ a : \Sigma x : A.B \to A}\ \Pi\text{-intro} \tag{2.63}$$

$$\dfrac{\dfrac{\Gamma \vdash \pi_1 : \Sigma x : A.B \to \mathcal{U}_i \quad \Gamma, a : A, b : B[a/x] \vdash b : B[\pi_1(a, b)/x]}{\Gamma, e : \Sigma x : A.B \vdash \mathsf{let}\ (a, b) = b\ \mathsf{in}\ a : A}\ \Sigma\text{-elim}}{\Gamma \vdash \pi_1 \equiv \lambda e.\mathsf{let}\ (a, b) = e\ \mathsf{in}\ a : \Sigma x : A.B \to A}\ \Pi\text{-intro} \tag{2.64}$$

Hence, as in Section 2.3.2, we may define $A \times B \equiv \Sigma - : A.B$ to get the usual intuitionistic typing rule for the Cartesian product. Similarly, we may define a coproduct type $A + B$ as

$$A + B \equiv \Sigma b : \mathsf{bool}.\mathsf{if}\ b\ A\ B, \qquad \mathsf{l} \equiv \lambda l.(\mathsf{true}, l) \qquad \mathsf{r} \equiv \lambda r.(\mathsf{false}, r) \tag{2.65}$$

with the case_+-statement from Section 2.2.4 defined by

$$\mathsf{case}_+\ e\ \{\mathsf{r}\ a \mapsto c_a, \mathsf{l}\ b \mapsto c_b\} \equiv \mathsf{case}_\Sigma\ e\ \{(x, v) \mapsto (\mathsf{if}\ x\ (\lambda a.c_a)\ (\lambda b.c_b))\ v\} \tag{2.66}$$

with typing derivation

$$\dfrac{\begin{array}{c}\Gamma \vdash e : A + B \\ \Gamma \vdash F : A + B \to \mathcal{U}_i\end{array} \quad \dfrac{\dfrac{\dfrac{\Gamma, a : A \vdash c_a : \mathsf{F}(\mathsf{l}\ a)}{\Gamma \vdash \lambda a.c_a : \Pi a : A.F(\mathsf{l}\ a)} \quad \dfrac{\Gamma, b : B \vdash c_b : \mathsf{F}(\mathsf{r}\ b)}{\Gamma \vdash \lambda b.c_b : \Pi b : B.F(\mathsf{r}\ b)}}{\Gamma, x : \mathsf{bool} \vdash \mathsf{if}\ x\ (\lambda a.c_a)\ (\lambda b.c_b) : \mathsf{if}\ x\ (\Pi a : A.F(\mathsf{l}\ a))\ (\Pi b : B.F(\mathsf{r}\ b))}}{\Gamma, x : \mathsf{bool}, v : \mathsf{if}\ x\ A\ B \vdash (\mathsf{if}\ x\ (\lambda a.c_a)\ (\lambda b.c_b))\ v : F\ (x, v)}}{\Gamma \vdash \mathsf{case}_\Sigma\ e\ \{(x, v) \mapsto (\mathsf{if}\ x\ (\lambda a.c_a)\ (\lambda b.c_b))\ v\} : F\ e} \tag{2.67}$$

Choosing $F \equiv \lambda - .C$ gives an intuitionistic form of the elimination rule in Equation 2.35.

2.3.6 Identity Types

While dependent type theory is already powerful with Σ, Π, and ground types, we still cannot really express many *propositions* like "x is even." To do so, we can introduce the *identity type* $\mathsf{Id}_A\ x\ y$ for terms $x, y : A$, with typing rules

$$\frac{\Gamma \vdash A : \mathcal{U}_i}{\Gamma \vdash \mathsf{Id}_A : A \to A \to \mathcal{U}_i}\ \textsf{Id-form} \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{refl}_A\ a : \mathsf{Id}_A\ a\ a}\ \textsf{Id-intro} \tag{2.68}$$

We will leave the type A out when it is clear from the context, writing, e.g., Id instead of Id_A.

We view elements of $\mathsf{Id}_A\ x\ y$ as *evidence* that $x = y$; the single constructor refl_A represents the *axiom* that, for any $a : A$, $a = a$, and therefore we can produce an element $\mathsf{refl}_A\ a : \mathsf{Id}_A\ a\ a$. This seems useless, as we already *know* that $a = a$, but becomes useful when it interacts with other axioms. For example, say we wanted to prove that "for all b in bool, either $b = \mathsf{true}$ or $b = \mathsf{false}$." If we had such a proof, then *given* b, we would have evidence that *either* $b = \mathsf{true}$ or $b = \mathsf{false}$. If we translate "evidence that $a = b$" as $\mathsf{Id}_A\ a\ b$, "either P or Q" as $P + Q$, and "given $x : A$, P" as $\Pi x : A.P$, we can interpret a term of type

$$\Pi b : \mathsf{bool}.\mathsf{Id}\ b\ \mathsf{true} + \mathsf{Id}\ b\ \mathsf{false} \tag{2.69}$$

as such a proof. And indeed, we may construct such a term, hence proving this fact, as follows:

$$\dfrac{\dfrac{\begin{array}{cc} & \vdots \\ & \vdash \mathsf{inl}\ (\mathsf{refl\ true}) : \mathsf{Id\ true\ true} + \mathsf{Id\ true\ false} \\ \vdots & \vdots \\ \vdash \lambda x.\mathsf{Id}\ x\ \mathsf{true} + \mathsf{Id}\ x\ \mathsf{false} : \mathsf{bool} \to \mathcal{U}_1 & \vdash \mathsf{inr}\ (\mathsf{refl\ false}) : \mathsf{Id\ false\ true} + \mathsf{Id\ false\ false} \end{array}}{b : \mathsf{bool} \vdash \mathsf{if}\ b\ (\mathsf{inl}\ (\mathsf{refl\ true}))\ (\mathsf{inr}\ (\mathsf{refl\ false})) : \mathsf{bool}.\mathsf{Id}\ b\ \mathsf{true} + \mathsf{Id}\ b\ \mathsf{false}}}{\vdash D \equiv \lambda b.\mathsf{if}\ b\ (\mathsf{inl}\ (\mathsf{refl\ true}))\ (\mathsf{inr}\ (\mathsf{refl\ false})) : \Pi b : \mathsf{bool}.\mathsf{Id}\ b\ \mathsf{true} + \mathsf{Id}\ b\ \mathsf{false}} \tag{2.70}$$

The interesting thing about this proof is that it's *computationally relevant*, i.e., it not only states that a fact is true, but carries data regarding the truth of the fact. In this case, for example, we can take

$$\pi_1 : \mathsf{Id}\ b\ \mathsf{true} + \mathsf{Id}\ b\ \mathsf{false} \to \mathsf{bool} \tag{2.71}$$

to extract a simple Boolean value telling us whether we have $b = \mathsf{true}$ or $b = \mathsf{false}$; we could then prove, e.g., that

$$\forall b \in \mathsf{bool}, \pi_1 D = b \tag{2.72}$$

i.e., construct a term

$$P : \Pi b : \mathsf{bool}.\mathsf{Id}\ (\pi_1 D)\ b \tag{2.73}$$

In this sense, inhabitants of a type become proofs of a proposition, and proofs of a proposition become inhabitants of a type, i.e., we interpret *propositions as types* (and vice versa). Taking this analogy further, we can interpret, e.g., Σ-types $\Sigma x : A.P$ as existential quantifiers $\exists x \in A.P$ ("a term x, along with evidence that P holds"), $\times$ as logical conjunction, $\to$ as implication, and so on.

Unfortunately, with the tools we have so far, we lack the tools to *use* the propositions we can prove either in other propositions or in programs. For example, given $x = y$, we have no way to conclude that $f(x) = f(y)$ (i.e., applicativity). To remedy this situation, we introduce *path induction*, with the following typing rules

$$\frac{\Gamma \vdash D : \Pi xy : A.\mathsf{Id}_A\ x\ y \to \mathcal{U}_i \quad \Gamma \vdash p : \mathsf{Id}_A\ x\ y \quad \Gamma, a : A \vdash d : D\ a\ a\ \mathsf{refl}_A\ a}{\Gamma \vdash \mathsf{case}_{\mathsf{Id}_A}\ p\ \{\mathsf{refl}_A\ a \mapsto d\} : D\ x\ y\ p}\ \textsf{Id-elim} \tag{2.74}$$

This, in essence, says that "anything we can do with $a : A$ and a proof that $a = a$ we can do with $x, y : A$ and a proof that $x = y$." This allows us to prove, e.g., applicativity, as follows

$$\cfrac{\cfrac{\begin{array}{c}\vdots\\ \Gamma, f : A \to B \vdash D : \Pi xy : A.\mathsf{Id}_A\ x\ y \to \mathcal{U}_i\end{array} \quad \cfrac{\cfrac{}{\Gamma, f : A \to B, a : A \vdash f\ a : B}}{\Gamma, f : A \to B, a : A \vdash \mathsf{refl}_B\ f(a) : \mathsf{Id}_B\ (f\ a)\ (f\ a)}}{\Gamma, f : A \to B, x : A, y : A, p : \mathsf{Id}_B\ x\ y \vdash \mathsf{case}_{\mathsf{Id}_A}\ p\ \{\mathsf{refl}_A\ a \mapsto \mathsf{refl}_B\ (f\ a)\} : \mathsf{Id}_B\ (f\ x)\ (f\ y)}}{\Gamma \vdash \lambda fxyp.\mathsf{case}_{\mathsf{Id}_A}\ p\ \{\mathsf{refl}_A\ a \mapsto \mathsf{refl}_B\ f(a)\} : \Pi f : A \to B.\Pi xy : A.\mathsf{Id}_A\ x\ y \to \mathsf{Id}_B\ (f\ x)\ (f\ y)} \tag{2.75}$$

where $D \equiv \lambda xyp.\mathsf{Id}_B\ (f\ x)\ (f\ y)$. We complete our analogy between propositions and types by providing a type representing *truth* $\top$, the *unit type* $\mathbf{1}$, and *falsity* $\bot$, the *empty type* $\mathbf{0}$, with the following rules

$$\frac{}{\mathbf{0} : \mathcal{U}_1} \qquad \frac{}{\mathbf{1} : \mathcal{U}_1} \qquad \frac{}{() : \mathbf{1}} \tag{2.76}$$

The unit type has eliminator

$$\frac{\Gamma; C \vdash F : \mathbf{1} \to \mathcal{U}_i \quad \Gamma; C \vdash d : F() \quad \Gamma; C \vdash u : \mathbf{1}}{\Gamma; C \vdash \mathsf{case}_{\mathbf{1}}\ i\ \{() \mapsto d\} : F()} \tag{2.77}$$

with reduction rule

$$\mathsf{case}_{\mathbf{1}}\ i\ \{() \mapsto d\} \to d \tag{2.78}$$

In particular, anything which is true for it's single element () is (obviously) true for the entire

Set Theory	**Lambda Calculus**
$\forall x \in A.P$	$\Pi x : A.P$
$\exists x \in A.P$	$\Sigma x : A.P$
$P \implies Q$	$P \to Q$
$P \wedge Q$	$P \times Q$
$P \vee Q$	$P + Q$
$a = b$	$\mathsf{Id}_A\ a\ b$
$\bot$	$\mathbf{0}$
$\top$	$\mathbf{1}$
$\neg P$	$P \to \mathbf{0}$

Figure 2.1: The correspondence between set-theoretic *propositions* and type-theoretic *types*.

type. On the other hand, the empty type has the interesting eliminator

$$\frac{\Gamma; C \vdash F : \mathbf{0} \to \mathcal{U}_i \quad \Gamma; C \vdash c : \mathbf{0}}{\Gamma; C \vdash \mathsf{case}_{\mathbf{0}}\ c\ \{\} : T\ c} \tag{2.79}$$

Note there are no cases to handle, since there is no way to construct an element of $\mathbf{0}$. There also is not a reduction rule, since if we have a closed term of this form, something's gone wrong! This represents the *principle of explosion*: if we can conclude $\bot$, we can conclude anything (here, any type T, as we can simply define $F = \lambda - .T$). This also allows us to define *negation*: as $\neg P \iff P \implies \bot$, we can define the negation of a type A to be the function type $A \to \mathbf{0}$; in particular, this implies that if we can prove A, we can prove anything. The entire correspondence we described is given in Figure 2.1.

2.3.7 Natural Numbers

We now want to *inductive types* and *recursive functions* on these types. We will begin with the canonical example of the *natural numbers* $\mathbb{N} : \mathcal{U}_1$. We provide the following typing rules, as in Section 2.2.2:

$$\frac{}{\vdash \mathbb{N} : \mathcal{U}_1}\ \mathsf{\mathbb{N}\text{-form}} \quad \frac{}{\vdash 0 : \mathbb{N}}\ \mathsf{0\text{-intro}} \qquad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathsf{s}\ n : \mathbb{N}}\ \mathsf{s\text{-intro}} \tag{2.80}$$

As usual, we provide an *eliminator* with a case for each rule

$$\frac{\Gamma \vdash F : \mathbb{N} \to \mathcal{U}_i \quad \Gamma \vdash n : \mathbb{N} \quad \Gamma \vdash z : F(0) \quad \Gamma, n' : \mathbb{N} \vdash s : F(\mathsf{s}n')}{\Gamma \vdash \mathsf{case}_{\mathbb{N}}\ F\ n\ \{0 \mapsto z, \mathsf{s}\ n' \mapsto s\} : F(n)}\ \mathsf{\mathbb{N}\text{-elim}} \tag{2.81}$$

with the reduction rules

$$\mathsf{case}_{\mathbb{N}}\ F\ 0\ \{0 \mapsto z, \mathsf{s}\ n' \mapsto s\} \to z \tag{2.82}$$

$$\mathsf{case}_{\mathbb{N}}\ F\ (\mathsf{s}n)\ \{0 \mapsto z, \mathsf{s}\ n' \mapsto s\} \to s[n/n'] \tag{2.83}$$

In general, when F is clear from the context (e.g., the constant function $F = \lambda - .\mathbb{N}$), we may leave it out, giving syntax

$$\mathsf{case}_{\mathbb{N}}\ n\ \{0 \mapsto -, \mathsf{s}\ n' \mapsto s\} \tag{2.84}$$

We can now define, e.g., a *predecessor function*

$$\lambda n.\mathsf{case}_{\mathbb{N}}\ n\ \{0 \mapsto 0, \mathsf{s}\ n \mapsto n\} \tag{2.85}$$

Unfortunately, the tools we currently have are not enough to define arithmetic operations, such as addition. For that, we have to introduce fixpoints.

2.3.8 Fixpoints

To be able to define recursive functions, we introduce *fixpoints*, with the following definition:

Definition 3 (Fixpoint Definition)**.** *Given symbols* $f_1, ..., f_n$ *and terms* $F_1, ..., F_n$*, we define a* fixpoint definition *to be an expression of the form* $\mathsf{Fix}([f_j : F_j = D_j])$*, which we will view as mutually recursively defining each symbol* f_j *of type* F_j *in terms of the other symbols* f_k*. For a fixpoint definition* $\mathcal{F}$*, we introduce terms* $\mathcal{F} :: f_j$ *corresponding to each symbol* f_j*. Where there is no risk of confusion, we will write* f_j *to mean* $\mathcal{F} :: f_j$*.*

Naively, we would introduce the following typing rule

$$\frac{(\Gamma, (f_i : F_i)_i \vdash D_j : F_j)_j}{\Gamma \vdash \mathsf{Fix}([f_j : F_j = D_j]) :: f_k : F_k} \tag{2.86}$$

in short, "assuming each symbol f_i is of type F_i, if each definition D_j is of type F_j in Γ, then the recursive definition $f_j : F_j = D_j$ in terms of $\{f_1, ..., f_n\}$ is valid." We introduce the obvious reduction rule

$$\mathsf{Fix}([f_j : F_j = D_j]) :: f_i \to D_i[\mathsf{Fix}([f_j : F_j = D_j])f_j/f_j]_j \tag{2.87}$$

Unfortunately, this makes all but the most trivial fixpoints no longer strongly normalizing, as we can simply "unfold" their definitions an arbitrary number of times. This is usually handled by introducing an explicit reduction discipline (to avoid infinite loops when checking terms for equality), such as, e.g., only unfolding full recursive calls to functions on closed terms. This is mainly, however, an implementation detail, and hence out of the scope of this section.

We may now define arithmetic operators inductively; for example, addition is given by

$$\mathsf{Fix}([\mathsf{add} = \lambda nm.\mathsf{case}_{\mathbb{N}}\ m\ \{0 \mapsto n, \mathsf{s}\ m \mapsto \mathsf{s}(\mathsf{add}\ n\ m)\}]) \tag{2.88}$$

i.e. “to add m to n, add 1 to n m times.” When fully applied, this reduces as expeced, e.g.

$$\mathsf{add}\ 2\ 2 \to \mathsf{s}(\mathsf{add}\ 1\ 2) \to \mathsf{s}(\mathsf{s}(\mathsf{add}\ 0\ 2)) \to \mathsf{s}(\mathsf{s}(2)) \equiv 4 \tag{2.89}$$

Similarly, we may perform *proofs by induction*; for example, we may perform the following inductive proof:

Claim 4. $\forall n.\mathsf{add}\ 0\ n = n$

Proof. Define the proposition $P(n) = [0 + n = n]$. We have

- $0 + 0 = 0$, and hence $P(0)$.
- By definition, $0 + \mathsf{s}\ n = \mathsf{s}\ (0 + n)$. Assuming $P(n)$, we have $\mathsf{s}(0 + n) = \mathsf{s}n$, and hence by transitivity we have $P(\mathsf{s}n)$

Hence, by induction $\forall n \in \mathbb{N}, P(n)$. □

Translating this into dependent type theory, we could write

$$\begin{aligned}&\mathsf{Fix}([\mathsf{leftunit} : \Pi n : \mathbb{N}.\mathsf{Id}_{\mathbb{N}}\ (\mathsf{add}\ 0\ n)\ n = \\ &\qquad \lambda n.\mathsf{case}_{\mathbb{N}}\ n\ \{0 \mapsto n, \mathsf{s}\ m \mapsto \mathsf{trans}_{\mathbb{N}}\ (\mathsf{refl}_{\mathbb{N}}\ (\mathsf{add}\ 0\ (\mathsf{s}\ m))\ (\mathsf{leftunit}\ n)\}])\end{aligned} \tag{2.90}$$

Here, the base case is represented by the 0 branch of the case-statement, while the inductive case is represented by the $\mathsf{s}m$ branch, with the use of the inductive hypothesis represented by the recursive call to leftunit.

Unfortunately, this naive theory is a little *too* powerful: we could easily define, e.g., an element of $\mathbf{0}$

$$\mathsf{Fix}([\boldsymbol{\Omega} : \mathbf{0} = \boldsymbol{\Omega}]) \tag{2.91}$$

In other words, the logic defined in 2.3.6 becomes inconsistent. For most practical uses, this is actually fine, as dependent type theoy is used as a programming language, rather than a logic. If we want to give a specification of our programs in dependent type theory, however, this is disastrous, as we can now simply prove false specifications.

It turns out, however, that we can identify *inconsistent terms*, such as that given in Equation 2.91, with *nonterminating terms* [2]. In particular, we give reduction rules

Definition 4 (Terminating)**.** *We say a term s is terminating if there exists n such that, for any closed, normalized terms $t_1, ..., t_n$, $s\ t_1\ ...\ t_n$ is normalizable.*

Definitions like Equation 2.91 are then ruled out as, like in Equation 2.13, the only possible reduction is $\boldsymbol{\Omega} \to \boldsymbol{\Omega}$, as would nonterminating functions like

$$\mathsf{Fix}([\boldsymbol{\omega} : \mathbb{N} \to \mathbf{0} = \lambda n.\boldsymbol{\omega}\ n]) \tag{2.92}$$

Unfortunately, it turns out that checking whether a term is terminating or not is, in general, undecidable, as it reduces to the Halting Problem. However, it is possible to define a *conservative* check which, while rejecting some terminating programs as well, successfully rejects all nonterminating programs. In particular, we could restrict our attention to *primitive recursive* definitions, which, in essence, only allow recursive calls on arguments derived unchanged from

case statements (this always terminates, since such terms always have a constructor removed, and a term can only have a finite number of constructors to remove). For example; add is primitive recursive, though

$$\mathsf{Fix}([\mathsf{halfadd} = \lambda nm.\mathsf{case}_{\mathbb{N}}\ m\ \{0 \mapsto n, \mathsf{s}\ m \mapsto \mathsf{case}_{\mathbb{N}}\ m\ \{0 \mapsto n, \mathsf{s}\ m \mapsto \mathsf{s}(\mathsf{halfadd}\ 0\ m)\}\}]) \tag{2.93}$$

is not, though both terminate, as the latter strips off two s per recursive call. To accomodate this case, we can allow *structurally recursive* definitions, which allow stripping off multiple constructors in one go; it is possible to generalize this to mutual structural recursion, allowing mutually recursive fixpoints. Agda uses this strategy [2], which is described in the paper [1]. Alternatively, we could emulate Coq and use a strategy based off *guarded fixpoints*, which admits all structurally recursive definitions (and hence, all primitively recursive definitions) [17].

We will not describe on a particular termination checker, but rather assume that we are given one which recognizes at least all cases of *primitive recursion*, and will only consider a fixpoint well-defined if it passes the termination checker. We write the resulting rule as, for $\mathcal{F} \equiv \mathsf{Fix}([f_j : F_j = D_j])$,

$$\frac{(\Gamma, (f_i : F_i)_i \vdash D_j : F_j)_j \quad \Gamma \vdash \mathcal{F}\ \mathsf{terminating}}{\Gamma; C \vdash \mathcal{F} :: f_k : F_k} \tag{2.94}$$

where the termination checker used is provided $\mathcal{F}$ and Γ as arguments.

2.3.9 Inductive Types

So far, we have introduced a hodgepodge of types, such as Id_A, $\mathbb{N}$, and bool. It turns out, we may define all these types via the more primitive concept of an *inductive type*. In this section, we present a formalism for inductive types using the *Calculus of Inductive Constructions* (CoC), which is the basis of the theorem provers Coq [17] and Agda [14]agda-ref, derived from the Coq reference manual [17]. We begin with a naive definition

Definition 5 (Inductive Definition)**.** *An* inductive definition *is an expression of the form*

$$\mathsf{Ind}(\Gamma_I := \Gamma_C) \tag{2.95}$$

where $\Gamma_I = [I_j : A_j], \Gamma_C = [c_{jk} : C_{jk}]$ *are sets of type bindings such that each* A_j *is an arity and each* C_{jk} *is a type of constructor for* I_j*.*

Each such definition introduces a set of *inductive type families* $I_1, ..., I_j$, which may be defined in terms of themselves, similarly to a fixpoint. An *arity* is simply the type of a family of types, i.e. a function $\Pi a : A.\Pi b.B. \ ... \ \mathcal{U}_i$, while a *type of constructor* is a valid type for a constructor for some member of a type family. More formally:

Definition 6 (Type of constructor)**.** *If I is a variable, we say that C is a* type of constructor of I with parameters $\mathsf{params}_I(C)$ *if*

- $C = I \ t_1 \ ... \ t_n$*; in this case we define* $\mathsf{params}_I(C) = []$
- $C = \Pi x : U.V$ *where V is a type of constructor of I; in this case we define*

$$\mathsf{params}_I(C) = U : \mathsf{params}_I(V) \tag{2.96}$$

For example, $\Pi A : \mathcal{U}_i.\Pi x : IA.\Pi a : A.IA$ is a type of constructor, with

$$\begin{aligned}
&\mathsf{params}_I(\Pi A : \mathcal{U}_i.\Pi x : IA.\Pi a : A.IA) \\
&= A : \mathsf{params}_I(\Pi x : IA.\Pi a : A.IA) \\
&= A : (IA) : \mathsf{params}_I(\Pi a : A.IA) \\
&= A : (IA) : A : \mathsf{params}_I(IA) \\
&= [A, IA, A] \qquad (2.97)
\end{aligned}$$

while $\mathcal{U}_i \to I \to \mathcal{U}_i$ is *not* a type of constructor for I, as the output is not a member of the type family I.

Definition 7 (Arity)**.** *A type T is an* arity of sort $\mathcal{U}_i$ *if*

- $T = \mathcal{U}_i$
- $T = \Pi x : U.V$ *where U is an arity of sort $\mathcal{U}_i$*

A type T is an arity *if there exists a universe $\mathcal{U}_i$ such that T is an arity of sort $\mathcal{U}_i$.*

For example, $A \to \mathcal{U}_2$ and $\mathcal{U}_2$ are arities of sort $\mathcal{U}_2$ while $A \to B \to \mathcal{U}_3$ is an arity of sort $\mathcal{U}_3$, while $\Pi A : \mathcal{U}_i.A \to A$ is *not* an arity, since the output is not contained in a typing universe $\mathcal{U}_i$.

Unfortunately, as with fixpoints, a naive definition of inductive types opens the door to contradictions. While we will not give a detailed example, this has to do with Cantor's paradox [17]: in particular, consider the inductive definition

$$\mathsf{Ind}([\mathsf{cantor} : \mathcal{U}_1] := [\mathsf{power} : (\mathsf{cantor} \to \mathsf{bool}) \to \mathsf{cantor}, \mathsf{base} : \mathsf{cantor}]) \qquad (2.98)$$

Interpreting this naively as an inductive definition of a set, we would require that the set of terms of type cantor contained all functions $\mathsf{cantor} \rightarrow \mathsf{bool}$, which is impossible, as the cardinality of this set is that of the power set of cantor (which, by Cantor's theorem, is strictly larger than the cardinality of cantor). Thankfully, we can avoid such cardinality troubles by introducing the somewhat abstract notion of *strict positivity*:

Definition 8 (Strict Positivity)**.** *Let T be a dependent type. We say T* satisfies the positivity condition *for a variable X if*

- $T = X\ t_1\ ...\ t_n$ *where* $\forall i, X \notin \mathsf{fv}(t_i)$
- $T = \Pi x : U.V$ *and*
  - *X occurs strictly positively in U*
  - *V satisfies the positivity condition for X*

We say X occurs strictly positively *in T if*

- $X \notin \mathsf{fv}(T)$
- $T = X\ t_1\ ...\ t_n$ *and* $\forall i, X \notin \mathsf{fv}(t_i)$
- $T = \Pi x : U.V$ *where* $X \notin \mathsf{fv}(U)$ *and X occurs strictly positively in V.*
- $T = I\ a_1\ ...\ a_m\ t_1\ ...\ t_p$ *where*
  - $\forall i, X \notin \mathsf{fv}(t_i)$
  - *I is an inductive definition of the form*

$$\mathsf{Ind}(I : A := [c_j : C_j]) \tag{2.99}$$

(in particular, is not mutually inductive and has m parameters) and

  - $\forall i = 1..p, X \notin \mathsf{fv}(t_i)$
  - $\forall j = 1..m, C_j[p_1/a_1]...[p_m/a_m]$ *satistfies the nested positivity condition for* X*.*

If I *is an inductive definition with* m *parameters, the type of constructor* T satisfies the nested positivity condition *for a constant* X *if:*

- $T = I\ b_1\ ...\ b_m\ u_1\ ...\ u_p$ *and* $\forall i, X \notin \mathsf{fv}(u_i)$
- $T = \Pi x : U.V$*, and*
  - X *occurs strictly positively in* U
  - V *satisfies the nested positivity condition for* X

For example, X occurs strictly positively in $\mathbb{N} \to X$ and $(\mathbb{N} \to X) \to X$, but not in $X \to \mathbb{N}$ or $(X \to \mathbb{N}) \to X$. It turns out that, as long as the types of constructor in our inductive definition are restricted to being strictly positive in each inductively defined type, our theory remains consistent [17]. We may hence define a well-formed inductive definition as follows:

Definition 9 (Well-formed Inductive Definition)**.** *We will say an inductive definition* $\mathcal{I} \equiv \mathsf{Ind}([I_j : A_j] := [c_{jk} : C_{jk}])$ *is* well-formed *in* Γ *if*

- *Each* A_j *is a type in* Γ*, i.e.* $\exists i, \Gamma \vdash A_j : \mathcal{U}_i$.
- *Each* C_{jk} *is a type of constructor of* I_j *which satisfies the positivity condition for all types in the definition* $I_1, ..., I_J$
- *Each* A_j *is an arity of sort* s_j
- $\forall j, k, \Gamma, I_1 : A_1, ..., I_J : A_J \vdash C_{jk} : s_j$

We will write this judgement as

$$\Gamma \vdash \mathcal{I} \text{ ok} \tag{2.100}$$

We introduce typing rules

$$\frac{\Gamma \vdash \mathcal{I} \text{ ok}}{\Gamma \vdash \mathcal{I} :: I_j : A_j}\,\textsf{Ind} \qquad \frac{\Gamma \vdash \mathcal{I} \text{ ok}}{\Gamma \vdash \mathcal{I} :: c_{jk} : C_{jk}[\mathcal{I} :: I_\ell / I_\ell]_{\ell=1..J}}\,\textsf{Cons} \tag{2.101}$$

When there is no risk of confusion, we will often omit judgements of the form $\Gamma \vdash \mathcal{I}$ ok *and write* N *for* $\mathcal{I} :: N$*.*

Using this definition, we may define the ground types we have previously seen as inductive types as follows

$$\mathsf{Ind}([\mathbf{0} : \mathcal{U}_1] := []) \tag{2.102}$$

$$\mathsf{Ind}([\mathbf{1} : \mathcal{U}_1] := [() : \mathbf{1}]) \tag{2.103}$$

$$\mathsf{Ind}([\mathsf{bool} : \mathcal{U}_1] := [\mathsf{true} : \mathsf{bool}, \mathsf{false} : \mathsf{bool}]) \tag{2.104}$$

$$\mathsf{Ind}([\Sigma x : A.B : \mathcal{U}_i] := [(,) : \Pi a : A.B.\Sigma x : A.B]) \tag{2.105}$$

$$\mathsf{Ind}([\mathbb{N} : \mathcal{U}_1] := [0 : \mathbb{N}, \mathsf{s} : \mathbb{N} \to \mathbb{N}]) \tag{2.106}$$

$$\mathsf{Ind}([\mathsf{Id} : A \to A \to \mathcal{U}_i] := [\mathsf{refl} : \Pi a : A.\mathsf{Id}\ A\ a\ a]) \tag{2.107}$$

We can also use Ind to define propositions by giving the *evidence* these propositions contain. For example, we can define "$a \leq b$" to mean *either* evidence $a = b$, or evidence $b \equiv \mathsf{s}\ b'$

combined with evidence $a \leq b'$, to get the predicate

$$\mathsf{Ind}\left([\mathsf{Le} : \mathbb{N} \to \mathbb{N} \to \mathcal{U}_1] := [\mathsf{le} : \Pi nm : \mathbb{N}.\mathsf{Id}\ n\ m \to \mathsf{Le}\ n\ m, \mathsf{lts} : \Pi nm : \mathbb{N}.\mathsf{Lt}\ n\ m \to \mathsf{Le}\ n\ (\mathsf{s}m)]\right) \tag{2.108}$$

We can even define mutually inductive propositions such as

$$\mathsf{Ind}\left(\begin{bmatrix} \mathsf{even} : \mathbb{N} \to \mathcal{U}_1, \\ \\ \mathsf{odd} : \mathbb{N} \to \mathcal{U}_1 \end{bmatrix} := \begin{bmatrix} \mathsf{even0} : & \mathsf{even}\ 0, \\ \mathsf{evenS} : & \Pi n : \mathbb{N}.\mathsf{odd}\ n \to \mathsf{even}\ (\mathsf{s}n) \\ \mathsf{oddS} : & \Pi n : \mathbb{N}.\mathsf{even}\ n \to \mathsf{odd}\ (\mathsf{s}n) \end{bmatrix}\right) \tag{2.109}$$

i.e., “$\mathsf{s}n$ is even if n is odd and odd if n is even, and 0 is even.” We may also provide an alternative definition of $A + B$ as follows:

$$\mathsf{Ind}([A + B : \mathcal{U}_i] := [\mathsf{inl} : A \to A + B, \qquad \mathsf{inr} : B \to A + B]) \tag{2.110}$$

Unfortunately, these two definitions are not equated by the theory; for that, we’d require the *univalence axiom* [18]. Hence, we will have to pick one (though, since they’re isomorphic, it does not matter much, and conversion is trivial). We can derive computation rules in a natural way:

Definition 10 (Pattern Matching)**.** *Given an inductive definition* $\mathcal{I} = \mathsf{Ind}([I_j : A_j] := [c_{jk} :$

$C_{jk}]$), *for each* I_j, *we define an* case statement $\mathcal{I} :: \mathsf{case}_{I_j}$ *with typing rule*

$$\frac{\begin{array}{rl}\Gamma & \vdash \mathcal{I}\ \mathsf{ok} \\ \Gamma & \vdash i : \mathcal{I} :: I_j \\ \Gamma & \vdash F : \mathcal{I} :: I_j \to \mathcal{U}_i \\ (\Gamma, (p_n : P_n)_{n=1..N_k}; C & \vdash r_k : F(c_{jk}\ p_1\ ...\ p_{N_k}))_k\end{array}}{\Gamma \vdash \mathcal{I} :: \mathsf{case}_{I_j}\ i\ \{(c_{jk}\ p_1\ ...\ p_{N_k} \mapsto r_k)_k\} : F(i)}\ \textsf{case} \tag{2.111}$$

where

$$P_k = \mathsf{param}_I(C_{jk}), \qquad N_k = \mathsf{len}(P_k)$$

We introduce reduction rules, for $k = 1, ..., n$,

$$\begin{aligned}&\mathcal{I} :: \mathsf{case}_{I_j}\ (\mathcal{I} :: c_{jk}\ a_1\ ...\ a_{N_k})\ \{c_{j1}\ p_1\ ...\ p_{N_1} \mapsto r_1, ..., c_{jK_j}\ p_1\ ...\ p_{N_{K_j}} \mapsto r_{K_j}\} \\ &\qquad \to r_k[a_n/p_n]_{\ell=1,...,N_k}\end{aligned} \tag{2.112}$$

Where $\mathcal{I}$ *is clear from the context, we may omit it.*

This yields the usual rules for $\mathsf{if} \equiv \mathsf{case}_{\mathsf{bool}}$, $\mathsf{case}_{\mathbb{N}}$, case_{Σ}, and so on, as special cases.

2.4 The C Language

In this section, we will change gears from type theory and give a brief review of the C programming language, along with some basic systems programming concepts, which we will use later in the thesis. We begin with Section 2.4.1, in which we describe why we include C in our background exposition and as the target language for the compilation algorithm Chapter

4. In Section 2.4.2, we give a brief description of the C programming language and it's basic features, while in 2.4.3, we describe one of the major drawbacks of programming in C, namely, undefined behaviour.

2.4.1 Why C

C serves as the prototypical example of a systems programming language, often being referred to as "portable assembly." Indeed, rather than target an actual assembly language or purpose-built intermediate representation such as LLVM, many compilers (including the original implementations of Eiffel, C++, and Modula 3 [15]) opt to instead target C so as to be able to generate high-performance, bare-metal code not dependent on any particular runtime without needing to worry about complex platform and implementation-specific details such as register allocation or instruction scheduling while taking advantage of the formidable development effort spent on optimizing the compilation of C programs [15, 8]. Another advantage of targeting C is the relative simplicity of explaining the target code a compiler produces by simply presenting output in C, which is usually *far* easier to read than assembly or most intermediate representations. Hence, while we decided C was not the best choice for an actual implementation of `isotope`, we will use C to describe our compilation algorithm for simplicity and ease of explanation.

2.4.2 C Basics

C is a statically-typed, procedural language, with a somewhat weak type discipline in which most types can be freely cast to each other, often implicitly (having evolved from an untyped language [15]). As C provides direct low-level access to computer memory through the use of

```c
#include<stdio.h>

int main(void) {
    puts("hello, world");
    return 0;
}
```

Figure 2.2: As is traditional, a “hello, world” program in C

pointers, it is often beneficial to think of C types as simply a method of viewing a chunk of memory with a given size (exposed by the `sizeof` expression) and alignment (exposed by the `alignof` expression). This is the perspective we will adopt for most of this thesis.

C provides a small selection of builtin *scalar types*, such as `int` and `long`. Types can be aliased using `typedef`s: for portability, we often instead use type definitions from the standard library's `stddef.h`, such as `uint32_t` (for an unsigned 32-bit integer), instead of directly using C's integer types (which may have different widths on different platforms). Especially important is the type of *bytes* or *characters*, `char`. C also supports the construction of compound *record types* (`struct`s) and *union types* (`union`s) from these basic components, as well as *array types* `T[`n`]`. C has a well defined ABI for each sort of type supported by the language:

- Structs are guaranteed to lay out their fields in the order they are defined, with appropriate padding between fields such that each field starts at an address with the proper alignment. For example, assuming that, on a given platform, `int` had size 4 and alignment 4, while `short` had size 2 and alignment 2, the struct

  ```c
  struct my_struct {
      short first_field;
      int second_field;
  ```

```
};
```

would have size 8 and alignment 4, with `first_field` taking up the first 2 bytes, followed by 2 bytes of padding, and `second_field` taking up the remaining 4 bytes.

- Unions are, in essence, simply a block of memory large enough to hold any of a collection of type, which are all stored right at the front of the union. For example, assuming the sizes and alignment for `int` and `short` given above, the union

```
union my_union {
    short first_variant;
    int second_variant;
};
```

would have size 4 and alignment 4, with `first_field` taking up the first 2 bytes and `second_field` taking up the whole 4 bytes (and overlapping with `first_field`)

- Arrays `T`[n] are simply blocks of memory large enough to hold n aligned copies of type `T`; e.g., if `T` had size 16 and alignment 4, then `T`[4] would have size 64 and alignment 4. On the other hand, if `U` had size 14 and alignment 8, then `U`[5] would have size 80 and alignment 8; note that it must skip forwards by 16 bytes for every value of `U` (rather than 14) since `U` has a size which is not divisible by it's alignment.

Where C gets interesting is in it's support for unrestricted manipulation of raw pointers to memory. In particular, given any type `T`, we may construct a *pointer type* `T`* representing a pointer to memory assumed to hold a value of type `T`, though this can be *null* (or otherwise *dangling*) and point to invalid memory, or *uninitialized*, in which case writes are defined but reads are not. We can *dereference* a pointer to get the value it points to, and we may perform

```c
#include<stdlib.h>
#include<string.h>

void memory_leak() {
    char* memory = (char*)malloc(8192);
    strcpy(
        memory,
        "This memory is malloc‘ed, but never free‘d;\n"
        "this is not incorrect, but does cause a memory leak!\n"
    );
}
```

Figure 2.3: A C function which leaks some memory.

arithmetic operations on pointers too, for example, get the next address in memory. More interestingly, we may *cast* pointers from one type to another, getting, in essence, “multiple ways of viewing the same memory.”

Indeed, many of C’s operations are, under the hood, encoded as simple pointer manipulations. For example, array indexing notation is actually syntax sugar for pointer arithmetic, as in Figure 2.4. Similarly, heap allocations are exposed via raw pointers in C, with programmers required to perform *manual memory management*: dynamic memory allocations must be explicitly made with a call to `malloc`, and then freed with a call to `free`. If this latter step is omitted, as in Figure 2.3, the program has a *memory leak*; this is not necessarily a bug, but can cause the program’s memory to continually grow over time and could eventually cause the host system to run out of RAM). More dangerous is when memory is accessed after being `free`’d, or read before being initialized, or `free`’d twice, as these situations trigger undefined behaviour.

```
int array_sugared() {
    int array[5] = {1, 2, 3, 4, 5};
    return array[3];
}

int array_desugared() {
    int array[5] = {1, 2, 3, 4, 5};
    return *(array + 3);
}

int strange_sugared() {
    int array[5] = {1, 2, 3, 4, 5};
    return 3[array];
}

int strange_desugared() {
    int array[5] = {1, 2, 3, 4, 5};
    return *(3 + array);
}
```

Figure 2.4: The function `array_sugared` desugars to `array_desugared`, and is equivalent to `strange_sugared` (which desugars to `strange_desugared`), since `i[j]` is actually just syntax sugar for `*(i + j)` and hence is (unexpectedly) commutative.

2.4.3 Undefined Behaviour

While the low level control over memory layout and contents that C provides is powerful, as we will see, it is also dangerous, as we can easily trigger *undefined behaviour* (UB), oftentimes without any warning from the compiler. Undefined behaviour is an especially worrisome class of bug, as it means that the compiler is theoretically unconstrained in what it can output [7]. In particular, the compiler can choose one day to output exactly the program you intended, while another day, on another platform, it can write out a program which deletes your hard drive. Hence, UB tends to lead to particularly insidious bugs, which appear nondeterministically on only some platforms (hindering reproducibility) and in seemingly illogical places (e.g., impossible behaviour being caused by undefined behaviour in a completely separate source file due to an incorrect compiler optimization). The situation is made dire by the fact that even state-of-the art C compilers, such as `clang`, will often fail to warn users even obvious cases of undefined behaviour, even at the highest warning levels. Coupled with the fact that nearly every operation in C involves pointers, and hence has a high chance of risking UB (not to mention that even seemingly innocuous operations, such as signed integer overflow, are actually technically UB according to the C standard [7]), memory safety bugs pose a serious challenge for maintaining code correctness and security.

As a trivial example of UB, consider the (contrived) code in Figure 2.5, in which a pointer to a stack variable `char x` is returned by the function `dangling_return` and subsequently dereferenced.

```c
char* dangling_return(char x) {
    return &x;
}

char deref_dangling(char x) {
    // Undefined behaviour: dereferencing a dangling pointer!
    return *dangling_return(x);
}
```

Figure 2.5: Code which triggers UB due to dereferencing a dangling pointer

When we give this program as input to `clang` (version 11.0.0-2, with flags `-Wall`, `-Wpedantic`), we get the following encouraging warning:

```
dangling-return.c:2:13: warning: address of stack memory associated with parameter
    ’x’ returned [-Wreturn-stack-address]
    return &x;
```

```
            ^

1 warning generated.
```

Unfortunately, this is a particularly simple case of undefined behaviour to detect. If we attempt to, instead of using memory which has just been popped from the stack, use memory which has just been freed, as in the following code

```c
#include<stdlib.h>

int use_after_free(int my_value) {
    int* ptr = (int*)malloc(sizeof(int));
    *ptr = my_value;
    free(ptr);
    // Undefined behaviour: dereferencing a freed pointer!
    return *ptr;
}
```

Figure 2.6: Code which triggers UB due to use-after-free

no error is produced by either `clang` or `gcc` (version 10.3.0, with flags `-Wall`, `-Wpedantic`), even with the high warning settings described below. Similarly, in Figure 2.7, `clang` does not warn about 2 of the 3 instances of undefined behaviour (though it *does* warn about unused variables)! Clearly, therefore, the programmer cannot rely on compiler warnings to avoid UB.

```c
void array_out_of_bounds() {
  int my_array[3] = {1, 4, 5};
  // Undefined behaviour: array index out of bounds!
  my_array[3] = 7;
}

void pointer_out_of_bounds() {
  int my_array[3] = {1, 4, 4};
  // Implicit cast from int[] to *int; pointer is to beginning of array
  int *my_ptr = my_array;
  // All good: in-bounds pointer arithmetic is defined
  int *in_bounds = my_ptr + 2;
  *in_bounds = 5; // my_array is now {1, 4, 5}
  // All good: creating a pointer one-past-the-end of an array is defined
  int *one_past_the_end = my_ptr + 3;
  // Undefined behaviour: constructing a pointer outside an allocation!
  // Yes, really... C11 standard, 6.5.6p8
  int *outside_alloc = one_past_the_end + 1;
  // Undefined behaviour: accessing a pointer which does not point to valid memory!
  *one_past_the_end = 7;
}
```

```
index-out-of-bounds.c:4:3: warning: array index 3 is past the end of the array (which contains 3 elements) [-Warray-bounds]
  my_array[3] = 7;
  ^        ~
index-out-of-bounds.c:2:3: note: array 'my_array' declared here
  int my_array[3] = {1, 4, 5};
  ^
index-out-of-bounds.c:18:8: warning: unused variable 'outside_alloc' [-Wunused-variable]
  int *outside_alloc = one_past_the_end + 1;
       ^
2 warnings generated.
```

Figure 2.7: Code demonstrating undefined behaviour caused by dereferencing out-of-bounds pointers, as well as creating out-of-bounds pointers via pointer arithmetic, which, surprisingly, is UB *even if those pointers are never subsequently accessed* (see 6.5.6p8 in [7]). The warning displayed is produced by `clang` version 11.0.0-2 with flags `-Wall`, `-Wpedantic`.

2.5 Rust and Ownership Types

In this section, we give a basic overview of the Rust programming language, which we compare and constrast to C. In particular, in Section 2.5.1, we give a brief description of the Rust languages. Then, in Section 2.5.2, we give some examples of Rust programs, and an overview of how Rust's system of *ownership types* allows statically eliminating the kinds of error we discussed in Section 2.4.3.

2.5.1 Rust 101

Rust is a systems programming language designed to take full advantage of modern hardware while automatically verifying the "safety," i.e. freedom from undefined behaviour, of a large set of programs assuming only the correctness of a (hopefully) small, trusted base of "`unsafe`" code [13]. This is done by restricting C-like manipulation of pointers in "safe" (i.e., not `unsafe`) code to use *references*, which are *statically* verified to uphold invariance such as liveness and uniqueness by the *borrow checker* [13]. This approach is unique in that it theoretically imposes no no runtime overhead (and, potentially, due to the additional information available to the compiler, there may even be a *boost* in runtime performance), instead statically verifying manual memory management [13]. This is achieved in a way which is mostly transparent to the user by automatically inserting calls to destructors via the *resource acquisition is initialization* (RAII) pattern, which eliminates the majority of memory leaks (but see Figure 2.9). To make this possible (and, unlike in C++, avoid unexpected calls to a `clone` method!), Rust uses a system of affine types; the combination of affine types extended with borrowing is called *ownership typing*.

Rust is an imperative language, but has a distinct "functional flavor," being in a sense a

```rust
fn main() {
    println!("hello, world")
}
```

Figure 2.8: A “hello, world” program in Rust

member of the ML family of languages; for example, there is full support for algebraic data types (via Rust’s `enum`s) and pattern matching (via the `match` statement). This expressivity can often make Rust programming feel like programming in a higher-level language which just happens to support ownership types, and can allow programmers to encode interesting abstractions and invariants (such as, e.g., that a reference to a variable read from inside a mutex cannot outlive the mutex guard) naturally in the type system, where they can be automatically verified. Unfortunately, Rust does not yet have a complete formal specification, though recent work, such as the RustBelt [9] and Stacked Borrows [10] projects, have focused on giving proper definitions for the behaviour of `unsafe` code and Rust’s memory model respectively.

In particular, we introduce

- `struct`s, which are analogues to C’s `struct`s except that they have undefined representation (unless one explicitly annotates them with `#repr(C)`)
- `union`s, which are analogues to C’s `union`s
- Arrays `[T;n]`, which again are analogous to C’s arrays `T[n]`
- `enum`s, which implement algebraic data types by acting as *tagged unions*: a union stored along with a *tag* indicating which variant is currently being stored. This allows us to safely pattern match on Rust `enum`s.

Unlike C, however, Rust types are by default *affine*: they can be used at most once. Rust types must explicitly be marked `Copy` to be allowed unrestricted nonlinear use. In addition, Rust types may be marked `Drop`, in which case, via the RAII pattern, they will automatically be passed to a user-defined destructor on going out of scope unless explicitly leaked with `std::mem::forget` or leaked accidentally in certain rare situations such as a cycle of references (see Figure 2.9). This lets one implement "nearly linear" types, though Rust explicitly does *not* guarantee that safe code always calls destructors.

In Rust, the main way of accessing memory is via *reference types* `&'a T` parametrized by *lifetimes* `'a` (often omitted as Rust can usually infer it), representing a pointer to `T` which is guaranteed to be valid for the "lifetime" `'a`. We can hence freely dereference such a pointer without risking UB. What's interesting is that, for some term `t` of affine type `T`, we can take a reference `r = &t` to `t` without consuming `t`; the type of this reference is automatically taken to be `&'a T`, where `'a` is the lifetime for which `t` is live, i.e., the range of time until `t` is consumed. Since lifetimes are inferred, and coercion between lifetimes is almost always handled automatically, when the lifetime system is combined with well-designed abstractions, Rust can statically verify a huge class of programs. This combination of affine typing and borrowing is called *ownership typing*.

Risky pointer arithmetic operations, such as array indexing, are generally performed "under the hood" by safe operations provided by the Rust libraries, which make use of the guarantees upheld by the strong type system Rust provides. For example, given a reference `a: &[T]` to a *slice*, which "under the hood" is simply a size `len(a)` and a pointer `first(a)` to `T` guaranteed to point to the beginning of a valid array of `len(a)` values of type `T` (with no guarantees if `len(a) = 0`, the array indexing

```rust
let x = a[i] + 5;
```

desugars to the C code

```
int x;
if i < len(a) {
    x = *(a + 5)
} else {
    panic("error message")
}
```

where `panic` crashes the program. While such “safe” desugarings introduce minor overhead, the optimizer can usually remove the majority of it [13]. Rust makes the promise that, if all the `unsafe` code in a program is *sound*, i.e. contains no UB, then the program itself is sound, regardless of what the safe code does [9]. Hence, a program which relies only on a small core of trusted `unsafe` code, such as the standard library, can be considered extremely safe.

2.5.2 Borrowing

Rust’s programming model allows us to replace C’s unreliable warnings with guaranteed errors, so long as we restrict ourselves to safe code. For example, translating the code in Figure 2.5 to Rust gives the code in 2.10,

```rust
/// This function allocates and then explicitly frees memory
fn explicit_drop() {
    let memory: Vec<u8> = Vec::with_capacity(8192);
    println!("{:?}", memory);
    std::mem::drop(memory);
}

/// This function does *not* leak memory, even though memory
/// is not explicitly freed, due to RAII
fn no_leak() {
    let memory: Vec<u8> = Vec::with_capacity(8192);
    println!("{:?}", memory);
    // An implicit call to ‘std::mem::drop(memory)‘ is inserted here,
    // since _memory is never destroyed in the function body, even
    // though it is used.
}

/// This function intentionally leaks memory in safe Rust
fn intentional_leak() {
    let memory: Vec<u8> = Vec::with_capacity(8192);
    println!("{:?}", memory);
    // The destructor of ‘memory‘ is never called,
    // since it is consumed by ‘forget‘
    std::mem::forget(memory);
}
```

Figure 2.9: Memory leaks in safe Rust

```rust
fn dangling_return(x: char) -> &char {
    &x
}

fn deref_dangling(x: char) -> char {
    *dangling_return(x)
}
```

Figure 2.10: The code in Figure 2.5 translated into Rust, along with the error message produced

which fails to compile, yielding the error message

```
error[E0106]: missing lifetime specifier
 --> dangling-return.rs:1:32
  |
1 | fn dangling_return(x: char) -> &char {
  |                                ^ expected named lifetime parameter
  |
  = help: this function’s return type contains a borrowed value with
    an elided lifetime, but the lifetime cannot be derived from the
    arguments
help: consider using the ‘’static‘ lifetime
  |
1 | fn dangling_return(x: char) -> &’static char {
  |                                ^^^^^^^^
```

This tells us, in essence, that the lifetime of the value being returned cannot be inferred by

Rust. Rust only ever infers lifetimes using function signatures (i.e., it ignores function bodies when doing lifetime inference), so this makes sense. It furthermore suggests that we use the `'static` lifetime, i.e., the lifetime encompassing the entire runtime of the program. However, if we attempt this, giving the program in Figure 2.11

```
fn dangling_return_try_fix(x: char) -> &'static char {
    &x
}
```

Figure 2.11: The code in Figure 2.5 translated into Rust with full lifetime annotations

we get error message

```
error[E0515]: cannot return reference to function parameter ‘x‘
 --> dangling-return-try-fix.rs:2:5
  |
2 |     &x
  |     ^^ returns a reference to data owned by the current function
```

mirroring the warning `clang` gives us for 2.5. So far so good.

Translating the use-after-free bug in Figure 2.6 to use Rust's `Box`, which is a zero-cost abstraction representing a single, owned (and therefore affine) allocation, we obtain the program in Figure 2.12

```rust
fn use_after_free(my_value: i32) -> i32 {
    let ptr = Box::new(my_value);
    std::mem::drop(ptr);
    *ptr
}
```

Figure 2.12: The code in Figure 2.6 translated into Rust

which, when compiled, yields error message

```
error[E0382]: use of moved value: ‘ptr‘
 --> use-after-free.rs:4:5
  |
2 |     let ptr = Box::new(my_value);
  |         --- move occurs because ‘ptr‘ has type ‘Box<i32>‘, which does not
   implement the ‘Copy‘ trait
3 |     std::mem::drop(ptr);
  |                    --- value moved here
4 |     *ptr
  |     ^^^^ value used here after move
```

As we can see, Rust realizes that, since `ptr` is of an affine type, passing it to `drop` destroys it, and hence, it cannot be used after that point. Note that `clang` did not even give us a warning here, as it cannot really track ownership implementation.

Now, consider the program in Figure 2.13. This program consumes a vector, and returns a hash of some slice of that vector and the vector back, doubled. Consider the reordering of this program in Figure 2.14. This fails to compile since, now, it requires that the variable `s` is used after the creation of `u`, which consumes `u`. This is an error, since `s` borrows from `v`, and hence cannot be used once `v` is destroyed. As we can see in Figure 2.14, Rust not only catches this mistake but provides an informative error message. On the other hand, in Figure 2.15, we translate both programs into C: they are both accepted without warning by `clang`.

As we can see, ownership typing provides a powerful mechanism to automatically verify a large class of programs. However, under the hood, safe abstractions like `Box` and `Vec` must be built from `unsafe` pieces. Furthermore, while ownership typing allows us to verify that (safe) code is *sound*, it does *not* let us actually verify any *properties* of the code (e.g., that it always returns even numbers on odd inputs, or even that it always terminates), unlike in dependent type theory. By integrating ownership types with dependent types, it should theoretically be possible to verify `unsafe` code within the language itself, i.e., without the use of external tools, while at the same time potentially verifying specifications about safe code, all without requiring the full burden of dependent verification for, e.g., every single pointer access (since most can simply be verified by checking lifetimes automatically). We're quite far from that now, but by exploring the design space for integrating ownership and dependent types, we hope that `isotope` can help work towards this goal.

```rust
/// This function doubles the value of it’s argument
fn double_vec(v: Vec<u8>) -> Vec<u8>;

/// This function returns some slice of the underlying vector
fn slice_vec(v: &Vec<u8>) -> &[u8];

/// This function takes the sum of it’s argument
fn hash(s: &[u8]) -> u64;

fn double_sum(v: Vec<u8>) -> (Vec<u8>, u64) {
    let s: &[u8] = slice_vec(&v);
    let h: u64 = sum(s);
    let u = double_vec(v);
    (u, h)
}
```

Figure 2.13: A simple Rust program which consumes a vector, and returns a hash of some slice of that vector and the vector back, doubled.

```rust
fn double_sum_bad(v: Vec<u8>) -> (Vec<u8>, u64) {
    let s: &[u8] = slice_vec(&v);
    let u = double_vec(v);
    let h: u64 = sum(s);
    (u, h)
}
```

```
error[E0505]: cannot move out of ‘v‘ because it is borrowed
 --> src/lib.rs:12:24
   |
11 |     let s: &[u8] = slice_vec(&v);
   |                              -- borrow of ‘v‘ occurs here
12 |     let u = double_vec(v);
   |                        ^ move out of ‘v‘ occurs here
13 |     let h: u64 = sum(s);
   |                      - borrow later used here
```

Figure 2.14: The program in Figure 2.13 with statements re-ordered so it fails to compile, along with the error message produced.

```c
#include<stddef.h>
#include<stdint.h>

struct VecU8 {
    size_t capacity;
    size_t len;
    uint8_t* data;
};

struct SliceU8 {
    size_t len;
    const uint8_t* data;
};

struct VecU8_and_U64 {
    struct VecU8 u;
    uint64_t h;
};

struct VecU8 double_vec(struct VecU8 v);

struct SliceU8 slice_vec(const struct VecU8* v);

uint64_t sum(struct SliceU8 s);

struct VecU8_and_U64 double_sum(struct VecU8 v) {
    struct SliceU8 s = slice_vec(&v);
    uint64_t h = sum(s);
    struct VecU8 u = double_vec(v);
    struct VecU8_and_U64 r = { .u =  u, .h =  h };
    return r;
}

struct VecU8_and_U64 double_sum_bad(struct VecU8 v) {
    struct SliceU8 s = slice_vec(&v);
    struct VecU8 u = double_vec(v);
    uint64_t h = sum(s);
    struct VecU8_and_U64 r = { .u =  u, .h =  h };
    return r;
}
```

Figure 2.15: The program in Figure 2.13, along with it's invalid re-ordering in Figure 2.14, translated into C. Both functions compile without any warnings or errors, but `double_sum_bad` exhibits unspecified behaviour: the sum h may be doubled (if `double_vec` overwrites the vector in-place) or even trigger undefined behaviour (if, e.g., `double_vec` reallocates v).

Chapter 3

Language

In this chapter, we describe the `isotope` language and give a semi-formal definition of it's typing rules and an *internal* denotational semantics for the language, in the form of a *denotation operator* which takes language terms to a small, CoIC-like intuitionistic fragment of `isotope` meant to represent their "mathematical value." In Section 3.1, we motivate the `isotope` language and give a high-level overview of it's core innovations; namely, the concepts of *instants* and *constraint sets*, and how they can be used to both typecheck and compile `isotope` terms. Then, in Section 3.2, we give an account of the basic typing judgements of the `isotope` language. In Section 3.4, we give typing rules for instants, and introduce the *borrow checking algorithm*, one of the main contributions of this thesis, in full detail. Then, in Section 3.5, we give an account of how the concepts in the previous section, Section 3.4, can allow us to give rules for machine function types and fixpoints. Finally, in Section 3.6, we give an account of `isotope`'s data structures; namely, inductive data declarations, primitive types such as integers, and type formers such as arrays.

3.1 Introduction

Say we wanted to convert the code in Figure 2.13 into a purely functional language. Inventing a Rust-like syntax, we would write something like

$$
\begin{aligned}
\mathtt{double\_sum} \equiv \mathsf{fn}(v : \mathsf{Vec}\ \mathtt{u8}) \mapsto\ & \mathsf{let}\ s = \mathtt{slice\_vec}\ \mathsf{c}v\ \&v\ \mathsf{in} \\
& \mathsf{let}\ h = \mathtt{sum}\ \mathsf{c}v\ s\ \mathsf{in} \\
& \mathsf{let}\ d = \mathtt{double\_vec}\ v\ \mathsf{in}\ (d, h) \\
& : \mathsf{Fn}(\mathsf{Vec}\ \mathtt{u8}) \to (\mathsf{Vec}\ \mathtt{u8}, \mathtt{u8})
\end{aligned}
\tag{3.1}
$$

This program has the data dependence graph in Figure 3.1

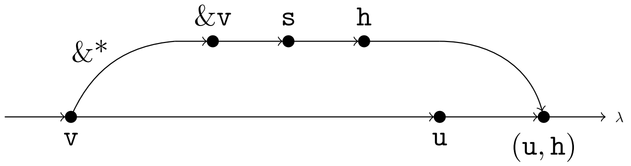


Figure 3.1: The data dependency graph of the program in Equation 3.1

Now, say we wanted to lower this program to a C-like language; naively, we could try traversing the data dependence graph and, at each node, adding a local variable set to the value of that node. The issue is that there are multiple possible such traversals of the graph; in particular, we could compile `u`

- After `h`; this is equivalent to the code in Figure 2.13

- Between `s` and `h`; this is equivalent to the code in Figure 2.14, which, as we discussed, gives an error in Rust and undefined behaviour in C.

- Between `&v` and `s`, which is another wrong re-ordering.

and be in compliance with the graph. What we need to do is recognize that the graph *really* looks like Figure 3.2

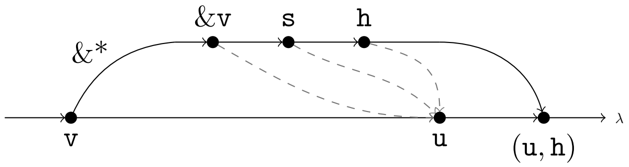


Figure 3.2: The data dependency graph of the program in Equation 3.1, annotated with borrowing dependencies (gray dotted lines).

Here, we insert gray dotted lines between `&v` and `u`, since the former borrows resources (`v`) used by the latter, `s` and `u`, for the same reason, and `h` and `u`, since constructing `h` *uses* resources which borrow from resources (`v`) which are destroyed by `u`, and hence all of `&v`, `s`, and `h` must be scheduled before `u`, even though there is no data dependency between them. Rust, in essence, generates graphs like that in Figure 3.2 and then checks that they are consistent with the program's execution order; if we're working in a functional language like the one used in Equation 3.1, we only need to check that an order *exists* (i.e., that the extended data dependence graph is *acyclic*); we may then use this order to compile our program (we go into more detail in Chapter 4).

Unfortunately, it's quite difficult to generate such graphs in a principled way, as we need to analyze

- Which variables borrow from other variables
- Which variables destroy other variables
- Which variables use other variables

and not necessarily only locally: here, for example, `h` does not borrow from `v`, but since it uses `s` which does, it must go before `u`, requiring at least some nonlocal analysis, which is complicated. Rust does so by introducing a system of logical constraints on the lifetimes of variables; the approach we will choose is similar, but instead based off graph theory.

In particular, rather than consider a data dependence graph directly, we consider a graph of *instants* in time. In particular, for each *resource* in the graph, we single out 3 important instants:

- The *beginning* of a resource, which is the *first possible time* it can be used.
- The *end* of the resource, which is the *last possible time* it can be used.
- The *consumption* of a resource, which is when it *is* used. Naturally, this is after the beginning, and before the end (and, therefore, the beginning is also before the end, by transitivity).

We then get the following graph

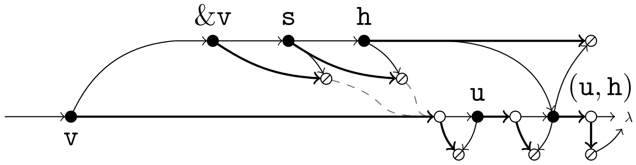


Figure 3.3: The TDG of the program in Figure 2.13. We represent the *beginning* of a resource with a black node, the *end* of a resource with a crossed-out node, and the *consumption* of a resource, if different from the end, as a white node. We represent virtual dependencies induced by borrowing as gray, dotted lines, and edges between the beginning, consumption, and end of a given node as thick lines.

While this graph *looks* quite complicated, it's actually very easy to derive using only *local* information. In particular:

- If x depends on y but does not consume it, then x must be compiled after y but before y is destroyed and hence no longer available. So the beginning of x is between the beginning of y and the consumption of y (see Figure 3.5b).

- If x uses up y, then x must be (fully) compiled after y is destroyed, and yet this point must not be such that y would be impossible to use (because, e.g., it depended on some underlying resource which was destroyed). So therefore, the beginning of x is between the consumption of y and the end of y (see Figure 3.5a).
- If x *borrows from* y, then it's impossible to use x in any way after y has been destroyed; in other words, the end of x is before the consumption of y. In the case of a *direct* borrow of x, e.g., the expression $\&x$, we get Figure 3.5c.

In other words, we need to walk the graph of the program, and build up *constraints* between the *instants* within the program. These constraints are intimately tied to where, and whether, variables are used; hence, instead of the usual strategy for implementing sub-structural types (which are necessary for ownership typing) by removing rules for manipulating typing contexts (as in Section 2.2), it makes sense to handle sub-structurality as an additional restriction along with, and directly determining, the set of constraints between instants.

This leads us to directly the core idea behind `isotope`: to define a type theory which, separate from the typing context, builds up a set of *constraints* which, by handling both sub-structurality and the ordering and creation of *instants*, implements a system of Rust-like *ownership types*. Once our constraint set is built up, we may then check it for *consistency* to make sure our program can compile down to a valid low-level, C-like program; the constraint set we have built up can then be directly used to build this program by, in essence, simply traversing it as a graph and, every time we come across the *beginning* of a resource, compiling it. Once we have gotten to this point, there's very little stopping us from introducing *type dependency*, and hence, being left with a system of integrated dependent and ownership types, i.e., the goal of the thesis: dependent types with borrowing.

This chapter, then, deals with the main contribution of this thesis: an experimental attempt at defining a set of typing rules for dependent, constraint-based ownership typing. The main aim of defining these rules is to explore the design space for possible type theories of this form: the rules themselves are not final, and can probably be significantly simplified. However, we have tried to be relatively formal, so as to lay the foundation for a type theory which could eventually be proved fully consistent. For this reason, we have attempted to define `isotope`'s type theory as an extension of the CoIC, and introduced *internal* denotations for all terms in `isotope`'s CoIC-like fragment; a proof of consistency might look something like "the CoIC-like fragment of `isotope` is equiconsistent with the CoIC, and `isotope` is equiconsistent with it's CoIC-like fragment; therefore, `isotope` is consistent."

3.2 Typing Judgements and Constraints

We now begin defining the core `isotope` language. The full grammar is deferred to Figure 3.4, but we first present the calculus rule by rule.

3.2.1 Typing Contexts

We begin by defining `isotope` typing contexts as follows:

Definition 11 (Typing Context)**.** *A* typing context Γ *is a set of judgements of the form*

- $x : A$ *where* x *is a variable and* A *is a term, meaning "the variable* x *is of type* A*"*
- $x \leftarrow: A$ *where* x *is a variable and* A *is a term, meaning "the variable* x *is a named term of type* A*"*

$$\begin{array}{rl}
\langle \mathit{expr} \rangle ::= & x \mid \langle \mathit{expr} \rangle\ \langle \mathit{expr} \rangle \mid \lambda x.\ \langle \mathit{expr} \rangle \mid \Pi x :\langle \mathit{expr} \rangle.\langle \mathit{expr} \rangle \mid \langle \mathit{expr} \rangle^{\perp} \mid \mathcal{U}_i \\
\mid & \langle \mathit{indef} \rangle{::}n \mid \langle \mathit{indef} \rangle{::}\mathsf{case}_n\ \langle \mathit{expr} \rangle\ \{\langle \mathit{case} \rangle,\ ...,\ \langle \mathit{case} \rangle\} \mid \langle \mathit{fixdef} \rangle{::}f \\
\mid & \mathcal{M}_i\ \langle \mathit{instant} \rangle \mid \mathsf{Fn}_c\ \langle \mathit{blargs} \rangle \to \langle \mathit{expr} \rangle \mid \mathsf{Closure}\ (\langle \mathit{block} \rangle,...,\langle \mathit{block} \rangle) \\
\mid & \forall\ \langle \mathit{ivar} \rangle\ \langle \mathit{expr} \rangle \mid \hat{\lambda}\ \langle \mathit{ivar} \rangle\ \langle \mathit{expr} \rangle \mid \langle \mathit{expr} \rangle\ \langle \mathit{instant} \rangle \mid \mathsf{Ref}\ \langle \mathit{expr} \rangle\ \langle \mathit{instant} \rangle \\
\mid & [\langle \mathit{expr} \rangle;\ \langle \mathit{expr} \rangle] \\
\mid & \langle \mathit{expr} \rangle\ \mathsf{in}\ \langle \mathit{expr} \rangle\ \mathsf{since} \mid \langle \mathit{expr} \rangle\ \mathsf{as}\ \langle \mathit{expr} \rangle \\
\mid & \&\ \langle \mathit{expr} \rangle \mid \&\mathsf{move}\ \langle \mathit{expr} \rangle \\
\mid & \mathsf{let}\ x = \langle \mathit{expr} \rangle\ \mathsf{in}\ \langle \mathit{expr} \rangle \mid \mathsf{const}\ x = \langle \mathit{expr} \rangle\ \mathsf{in}\ \langle \mathit{expr} \rangle \\
\mid & \langle \mathit{indef} \rangle{::}\mathsf{match}_n\ \langle \mathit{expr} \rangle\ \{\ \langle \mathit{block} \rangle,\ ...,\ \langle \mathit{block} \rangle\ \} \\
\mid & \mathsf{fn}_c\ \langle \mathit{block} \rangle \mid \mathsf{close}\ \langle \mathit{block} \rangle \\
\mid & \delta\ \langle \mathit{expr} \rangle \mid \mathsf{D}\ \langle \mathit{expr} \rangle \mid \mathsf{d}\ \langle \mathit{expr} \rangle \mid \mathsf{T}\ \langle \mathit{expr} \rangle \mid \mathsf{t}\ \langle \mathit{expr} \rangle \mid \mathsf{sizeof}\ \langle \mathit{expr} \rangle \mid \mathsf{alignof}\ \langle \mathit{expr} \rangle \\
\mid & \langle \mathit{primitive} \rangle
\end{array}$$

$$\langle \mathit{case} \rangle ::= c\ p_1, ..., p_n \mapsto$$

$$\langle \mathit{indef} \rangle ::= \mathsf{Ind}(\langle \mathit{sigs} \rangle := \langle \mathit{sigs} \rangle) \mid \mathsf{Data}(\langle \mathit{sigs} \rangle := \langle \mathit{sigs} \rangle)$$

$$\langle \mathit{sigs} \rangle ::= [p_1 : \langle \mathit{expr} \rangle,\ ...,\ p_n : \langle \mathit{expr} \rangle]$$

$$\langle \mathit{fixdef} \rangle ::= \mathsf{Fix}([f_1 = F_1,\ ...,\ f_n = F_n]) \mid \mathsf{Phi}([f_1 = F_1,\ ...,\ f_n = F_n])$$

$$\langle \mathit{block} \rangle ::= \langle \mathit{blargs} \rangle \mapsto \langle \mathit{expr} \rangle$$

$$\langle \mathit{blargs} \rangle ::= p_1\text{: } \langle \mathit{expr} \rangle,\ ...,\ p_n\text{: } \langle \mathit{expr} \rangle$$

$$\langle \mathit{ivar} \rangle ::= \text{`}a \in [\ \langle \mathit{instant} \rangle,\ \langle \mathit{instant} \rangle\ ] \mid \text{`}a \preceq \langle \mathit{instant} \rangle \mid \text{`}a \succeq \langle \mathit{instant} \rangle$$

$$\langle \mathit{instant} \rangle ::= \text{`}a \mid \infty \mid \langle \mathit{instant} \rangle \wedge \langle \mathit{instant} \rangle \mid \langle \mathit{instant} \rangle \vee \langle \mathit{instant} \rangle \mid \mathsf{is}\ \langle \mathit{instant} \rangle \mid \mathsf{ip}\ \langle \mathit{instant} \rangle$$

Figure 3.4: A simple AST grammar for the `isotope` language

- $x \leftarrow a : A$ *where* x *is a variable and* a, A *are terms, meaning "the variable* x *is a named term of type* A *defined to be* a*"*

- $‘\alpha \mathsf{R} R$ *means "the variable* $‘\alpha$ *is an instant satisfying constraint* $\mathsf{R}R$*", where* $\mathsf{R} \in \{“\succeq”, “\preceq”\}$ *and* R *is an instant or* $\mathsf{R} = “\in”$ *and* $R = [‘a, ‘b]$ *where* a, b *are instants.*

For contexts Γ, Δ*, we will take* Γ, Δ *to mean the union of the contexts as sets. If* Γ *is a context and* J *is a judgement, we will define* $\Gamma, J = \Gamma, \{J\}$*.*

Given a variable x*, we will write* $x \in \Gamma$ *to mean "there exists a judgement* $x : A$*,* $x \leftarrow: A$*, or* $x \leftarrow a : A$ *in* Γ*." Similarly, given an instant variable* $‘\alpha$*, we will write* $‘\alpha \in \Gamma$ *to mean "there exists a judgement* $‘\alpha \mathsf{R} R$ *in* Γ*."*

Unlike in Chapter 2, we introduce three new kinds of judgement: $x \leftarrow: A$, $x \leftarrow a : A$, and $‘\alpha \mathsf{R} R$. $x \leftarrow: A$ is really a bit of a hack; it's simply there to ensure that terms which appear in instants are each given a unique identifier (their variable name x). $x \leftarrow a : A$ is similar, but also simplifies our treatment of let-statements in an ownership typed setting. $‘\alpha \mathsf{R} R$ is simply the judgement to introduce a new lifetime variable $‘\alpha$ assumed to satisfy a given constraint.

Given a typing context Γ, we define the *symmetric form of* Γ*,* $\mathsf{sym}(\Gamma)$, as follows:

$$\mathsf{sym}(A, B) = \mathsf{sym}(A), \mathsf{sym}(B) \tag{3.2}$$

$$\mathsf{sym}(x : A) = x : A, \qquad \mathsf{sym}(x \leftarrow: A) = x \leftarrow: A, \qquad \mathsf{sym}(x \leftarrow a : A) = x \leftarrow a : A \tag{3.3}$$

$$\mathsf{sym}(‘\alpha \in [‘a, ‘b]) = ‘\alpha \in [‘a, ‘b] \tag{3.4}$$

$$\mathsf{sym}(‘\alpha \preceq ‘a) = \mathsf{sym}(‘\alpha \succeq ‘a) = \varnothing \tag{3.5}$$

3.2.2 Constraints

The key innovation of the `isotope` language is the introduction of *constraints*, which are used to model Rust-like ownership types. When typing machine terms, we will build up sets of constraints, which are then checked at function definition boundaries (this is the *borrow check*) to enforce that ownership rules are not violated. Intuitionistic values and type formers are, for the most part, transparent to this process. Later, in Chapter 4, we will show how the data structures built up during this phase can be used to compile `isotope` into a low level language (we will use C as an example, but the real implementation uses LLVM).

We begin by defining an *instant constraint set*, which, in essence, is a set of ordering relations which can be imposed on the instants in a term

Definition 12 (Instant Constraint Set)**.** *An* instant constraint set I *is a relation from the set of instants to itself which is either* finite *(represented as a finite set of judgements of the form* $`a \preceq `b$*) or* discrete *(i.e.,* $\forall `a, `b, `aI`b$*, represented by the symbol* $!$*). We define a partial order on instant constraint sets as follows*

$$I \preceq I' \iff I \subseteq I' \tag{3.6}$$

Given an individual instant $`a$*, we write* $`a \in I$ *to mean* $\exists `b, (`a, `b) \in I \lor (`b, `a) \in I$*.*

Given an instant constraint set I*, we define it's* constrained variable set *to be given by*

$$\mathsf{rv}(C) = \bigcup_{(`a,`b)\in I} \mathsf{fv}(`a) \cup \mathsf{fv}(`b) \tag{3.7}$$

We define it's free variable set $\mathsf{fv}(I)$ *to be given by* $\varnothing$ *if* $I =!$*, and* $\mathsf{rv}(I)$ *otherwise.*

Given an instant constraint set I, we denote it's transitive closure *(not reflexive) as I^+, which we also take to be a constraint set.* [1] *We define an instant constraint set's underlying* equivalence constraint *$I^=$ to be given by*

$$I^= = \{(`a, `b) : (`a, `b) \in I^+ \wedge (`b, `a) \in I^+\} \tag{3.8}$$

This is again an instant constraint set (and not reflexive).

We write the union of instant constraint sets as either I, I' or $I \cup I'$.

We couple this with *usage constraints*, which consist of consumptions (usages) and simple dependencies (reads), via the following definitions:

Definition 13 (Usage Bag)**.** *A* usage bag *U is a finite* multiset *of uses $\mathsf{u}(r)$ and partial uses $\mathsf{p}(r)$ of resources r, written $\mathsf{u}(r_1), ..., \mathsf{u}(r_n), \mathsf{p}(r'_1), ..., \mathsf{p}(r'_n)$, or the expression !, which is assumed to contain an infinite number of copies of every resource. The catenation (i.e., multiset sum) of usage bags is written U, U'. For general values s, we define $\mathsf{u}(s) = \{\mathsf{u}(v) : v \in \mathsf{fv}(s)\}$, and likewise for $\mathsf{p}(s)$. We define the* union *of usage bags $U \cup U'$ to be given by the smallest such bag containing both U and U' (i.e., the count of any element is the maximum of the counts in U and U', rather than the sum).*

Definition 14 (Read Set)**.** *A* read set *R is a finite* set *of resources r, written $\mathsf{r}(r_1), ..., \mathsf{r}(r_n)$, or the expression !, which is assumed to contain every resource. The union of resource sets is written either R, R' or $R \cup R'$. For general values s, we define $\mathsf{r}(s) = \{\mathsf{r}(v) : v \in \mathsf{fv}(s)\}$.*

We may now define a constraint set as a combination of an instant constraint set and a set of usage constraints (i.e., a usage bag and a read set), as follows:

---

[1] If the closure was reflexive, I^+ would be always be infinite and yet may not be equal to !, so I^+ would not be a valid instant constraint set

Definition 15 (Constraint Set)**.** *A* constraint set *is a tuple* $C = (I, U, R)$*, where* I *is an instant constraint set,* U *is a usage bag, and* R *is a read set. We define a partial order on constraint sets given by the lattice order, i.e.,*

$$C = (I, U, R) \subseteq C' = (I', U', R') \iff I \subseteq I' \wedge U \subseteq U' \wedge R \subseteq R' \tag{3.9}$$

and define the maximal *constraint set* $!$ *to be given by* $(!, \varnothing, !, !)$*, and the* empty *constraint set* $\varnothing$ *to be given by* $(\varnothing, \varnothing, \varnothing, \varnothing)$*. The* catenation *of constraint sets is written* C, C'*, and defined as*

$$(I, U, R), (I', U', R') = ((I, I'), (U, U'), (R, R')) = ((I \cup I'), (U, U'), (R \cup R')) \tag{3.10}$$

while the union *of constraint sets is written* C, C' *and defined as*

$$(I, U, R) \cup (I', U', R') = ((I \cup I'), (U \cup U'), (R \cup R')) = ((I, I'), (U \cup U'), (R, R')) \tag{3.11}$$

As syntax sugar, we will treat instant constraint sets I *as constraint sets* $(I, \varnothing, \varnothing, \varnothing)$*, usage bags* U *as constraint sets* $(\varnothing, \varnothing, U, \varnothing)$*, and read sets as constraint sets* $(\varnothing, \varnothing, \varnothing, R)$*; in particular, we will allow writing constraint sets in the format*

$$\mathsf{u}(r_2), \mathsf{r}(r_3), \mathsf{p}(r_9), `\alpha \preceq `\beta, \mathsf{u}(r_4) \tag{3.12}$$

and so on. We define

$$\mathsf{insts}((I, U, R)) = I, \quad \mathsf{usage}((I, U, R)) = U, \quad \mathsf{read}((I, U, R)) = R \tag{3.13}$$

In general, we will annotate our typing contexts with a constraint set as follows:

Definition 16 (Constrained Context)**.** *A* constrained context *is a typing context* Γ *together with a constraint set* C*, written* $\Gamma; C$*.*

As in Section 2.2, we have a weakening rule

$$\frac{\Gamma; C \vdash P \quad \Gamma \subseteq \Gamma' \quad C \preceq C'}{\Gamma'; C' \vdash P} \tag{3.14}$$

which we will often apply implicitly.

3.2.3 Equality, Annotations, and Free Variables

As in Section 2.1, direct comparison of terms is represented by $\equiv$, with implicit quotienting under α-conversion. Given a term or instant t, we define it's *free variable set* $\mathsf{fv}(t)$ to be the set of free variables the term depends on. We define substitution as usual. In particular, we have

- For a term variable x, $\mathsf{fv}(x) = \{x\}$, $x[a/x] \equiv a$, and $x[a/y] \equiv x$.
- For an instant variable $`\alpha$, $\mathsf{fv}(`\alpha) = \{`\alpha\}$, $x[`a/`\alpha] \equiv `a$, and $`\alpha[`a/`\beta] = `\alpha$.

We assume the following fact:

Claim 5. *If* $x \notin \mathsf{fv}(t)$*, then* $t[a/x] \equiv t$*.*

In general, we apply a vague "α-conversion convention" in which, for simplicity, we assume that variable names never collide, silently performing α-conversion where necessary to do so. [2] We define a *fully annotated term* to be a term t, a term A, and a full derivation that $t : A$

---

[2] In the actual implementation, there are no variable names, as we use de-Bruijn indices, but for notational convenience we use a name-based convention.

(in particular, we may access the types assigned to subterms of t at points in the derivation, as well as, e.g., the type of A itself). We define the *constant variable set* of a fully annotated term $\mathsf{cv}(t : A)$, which are the variables which must be held constant for a term to be valid. In general, $\mathsf{cv}(t : A) \subseteq \mathsf{fv}(t) \cup \mathsf{fv}(A)$; we will often write $\mathsf{cv}(t)$ as shorthand for $\mathsf{cv}(t : A)$.

Similarly, we define a *reduction relation* to be a relation R from the set of terms to itself (as well as, potentially, from the set of instants to itself), with $\to_R$ being the congruence of R with term formers (i.e., $s \to_R s' \implies st \to_R s't$, and so on). We define $\twoheadrightarrow_R$ as the transitive, reflexive closure of $\to_R$ and $=_R$ as the symmetric closure of $\to_R$. Where a reduction relation R depends on a context $\Gamma; C$ such that

$$\Gamma \subseteq \Gamma' \land C \subseteq C' \implies R(\Gamma; C) \subseteq R(\Gamma'; C') \tag{3.15}$$

we call such a relation *contextual*; in this case, we write

$$\Gamma; C \vdash (s, t) \in R \iff (s, t) \in R(\Gamma; C), \qquad (s, t) \in R \iff \vdash (s, t) \in R \tag{3.16}$$

$$\Gamma; C \vdash s \to_R t \iff s \in to_{R(\Gamma;C)} t, \qquad s \to_R t \iff \vdash s \to_R t \tag{3.17}$$

and likewise for $\twoheadrightarrow_R$ and $=_R$. We define a privileged contextual reduction relation i, and in general take $(\to) = (\to_\mathsf{i})$, $(\twoheadrightarrow) = (\twoheadrightarrow_\mathsf{i})$, and $(=) = (=_\mathsf{i})$; that is,

$$\Gamma; C \vdash a = b \iff \Gamma; C \vdash a =_\mathsf{i} b \tag{3.18}$$

and so on. Note that if we have a reduction rule of the form

$$\frac{P}{t \to_R s} \tag{3.19}$$

this implies a rule of the form

$$\frac{P}{\Gamma; C \vdash t \to_R s} \tag{3.20}$$

by Equation 3.16. Given a relation R, we define it's *reduced variable set* to be given by

$$\mathsf{rfv}_R(t) = \bigcap_{s \twoheadrightarrow_R t} \mathsf{fv}(s) \tag{3.21}$$

As shorthand, we have $\mathsf{rfv}(t) = \mathsf{rfv}_{\mathsf{i}(\varnothing)}(t)$.

3.2.4 Basic Judgements

We may now introduce the basic typing judgements making up `isotope`'s type theory. Unfortunately, there are quite a few of them: simplifying `isotope` is a major potential area for future work.

(a) $\Gamma; C$ ok, meaning "the constrained context $\Gamma; C$ is *well-formed*," i.e., the definitions in Γ can be ordered in an acyclic manner such that each is well-typed and well-defined, and all the instants in C are defined in Γ." This has basic axiom

$$\frac{}{\varnothing; \varnothing \ \mathsf{ok}} \tag{3.22}$$

In general, for all the following judgements, we will implicitly assume that any contexts appearing in either the premises or conclusions must be well-formed for the judgement to be valid. We define

$$\Gamma\ \mathsf{ok} \iff \Gamma; !\ \mathsf{ok} \tag{3.23}$$

(b) $\Gamma; C \vdash A\ \mathsf{type}(\ell)$, meaning “$A$ is a type with linearity ℓ in context Γ,” where

$$\ell \in \mathsf{Lin} = \{\mathsf{linear}, \mathsf{affine}, \mathsf{relevant}, \mathsf{lifetime}\} \tag{3.24}$$

(see Equation 3.48). We have basic axiom

$$\frac{\Gamma; C \vdash A\ \mathsf{type}(\ell) \quad \ell \preceq \ell'}{\Gamma; C \vdash r\ \mathsf{type}(\ell')} \tag{3.25}$$

where

$$\frac{\ell \in \mathsf{Lin}}{\vdash \ell : \mathsf{Lin}} \qquad \frac{\Gamma; C \vdash \ell : \mathsf{Lin}}{\Gamma; C \vdash \ell \preceq \mathsf{linear}} \qquad \frac{\Gamma; C \vdash \ell : \mathsf{Lin}}{\Gamma; C \vdash \mathsf{scalar} \preceq \ell} \tag{3.26}$$

If ℓ is not specified, i.e. we have $\Gamma; C \vdash A\ \mathsf{type}$. we assume $\ell = \mathsf{linear}$. We give basic axioms

$$\frac{\Gamma; C \vdash A\ \mathsf{type} \quad x \notin \Gamma}{\Gamma, x : A; C\ \mathsf{ok}} \qquad \frac{\Gamma; C \vdash A\ \mathsf{type} \quad x \notin \Gamma}{\Gamma, x \leftarrow: A; C\ \mathsf{ok}} \tag{3.27}$$

(c) $\Gamma; C \vdash x \leftarrow: A$, meaning “given constraints C, the variable x is a named term of type A in Γ.” This judgement has basic axioms

$$\frac{}{\Gamma, x \leftarrow a : A; C \vdash x \leftarrow: A} \qquad \frac{}{\Gamma, x \leftarrow: A; C \vdash x \leftarrow: A} \qquad \frac{\Gamma; C \vdash A\ \mathsf{type} \quad x \notin \Gamma}{\Gamma, x \leftarrow: A; C\ \mathsf{ok}} \tag{3.28}$$

(d) $\Gamma; C \vdash a : A$, meaning “given constraints C, the term a is of type A in Γ.” This judgement has basic axioms

$$\frac{}{\Gamma, x : A; C \vdash x : A} \qquad \frac{\Gamma; C \vdash x \leftarrow: A}{\Gamma; C \vdash x : A} \qquad \frac{\Gamma; C \vdash a : A \quad x \notin \Gamma}{\Gamma, x \leftarrow a : A; C \ \mathsf{ok}} \tag{3.29}$$

(e) $\Gamma; C \vdash \text{‘}a \ \mathsf{inst}(v)$, meaning “$a$ is an instant in Γ with variance v,” where $v = \{-1, 0, 1\}$, meaning “contravariant,” “invariant,” and “covariant” respectively. For brevity, we write -1 as $-$ and 1 as $+$. This judgement has basic axioms

$$\frac{}{\Gamma, \text{‘}\alpha \mathsf{R} R; C \vdash \text{‘}\alpha \ \mathsf{inst}(\mathsf{variance}(\mathsf{R}))} \qquad \frac{\Gamma; C \vdash \text{‘}a \ \mathsf{inst}(0)}{\Gamma; C \vdash \text{‘}a \ \mathsf{inst}(v)} \tag{3.30}$$

$$\frac{\Gamma; C \vdash \text{‘}a \ \mathsf{inst}(+)}{\Gamma, \alpha \preceq \text{‘}a; C \ \mathsf{ok}} \qquad \frac{\Gamma; C \vdash \text{‘}a \ \mathsf{inst}(-)}{\Gamma, \alpha \succeq \text{‘}a; C \ \mathsf{ok}} \qquad \frac{\Gamma; C \vdash \text{‘}a \ \mathsf{inst}(0) \quad \Gamma; C \vdash \text{‘}b \ \mathsf{inst}(0)}{\Gamma, \alpha \in [\text{‘}a, \text{‘}b]; C \ \mathsf{ok}} \tag{3.31}$$

where

$$\mathsf{variance}(\text{“} \preceq \text{”}) = -, \qquad \mathsf{variance}(\text{“} \in \text{”}) = 0, \qquad \mathsf{variance}(\text{“} \succeq \text{”}) = + \tag{3.32}$$

We define

$$\frac{\Gamma; C \vdash \text{‘}a \ \mathsf{inst}(v)}{\Gamma; C \vdash \text{‘}a \ \mathsf{inst}} \tag{3.33}$$

with basic axioms

$$\frac{\Gamma; C \vdash \text{‘}a \ \mathsf{inst} \quad \Gamma; C \vdash \text{‘}b \ \mathsf{inst}}{\Gamma; C, \text{‘}a \preceq \text{‘}b \ \mathsf{ok}} \tag{3.34}$$

(f) $\Gamma; C \vdash a \ \mathsf{rsrc}(A, \ell)$, meaning “given constraints C, a is a *resource* of type A with linearity

ℓ in context Γ," where $\ell \in \{\mathsf{linear}, \mathsf{affine}, \mathsf{relevant}, \mathsf{lifetime}\}$ We have basic axioms

$$\frac{\Gamma; C \vdash r\ \mathsf{rsrc}(A, \ell) \quad \ell \preceq \ell'}{\Gamma; C \vdash r\ \mathsf{rsrc}(A, \ell')} \qquad \frac{\Gamma; C \vdash A\ \mathsf{type}(\ell) \quad \Gamma; C \vdash x \leftarrow: A}{\Gamma, C \vdash x\ \mathsf{rsrc}(A, \ell)} \tag{3.35}$$

We define

$$\frac{\Gamma; C \vdash r\ \mathsf{rsrc}(A, \ell)}{\Gamma; C \vdash r\ \mathsf{rsrc}} \tag{3.36}$$

which has basic axioms

$$\frac{\Gamma; C \vdash r\ \mathsf{rsrc}}{\Gamma; C, \mathsf{o}(r)\ \mathsf{ok}} \qquad \frac{\Gamma; C \vdash r\ \mathsf{rsrc}}{\Gamma; C, \mathsf{u}(r)\ \mathsf{ok}} \qquad \frac{\Gamma; C \vdash r\ \mathsf{rsrc}}{\Gamma; C, \mathsf{r}(r)\ \mathsf{ok}} \tag{3.37}$$

We define shorthand

$$\frac{\Gamma; C \vdash x\ \mathsf{rsrc}(\mathsf{affine})}{\Gamma; C \vdash x\ \mathsf{irrel}} \qquad \frac{\Gamma; C \vdash x\ \mathsf{rsrc}(\mathsf{affine})}{\Gamma; C \vdash x\ \mathsf{copy}} \tag{3.38}$$

(g) $\Gamma; C \vdash a\ \mathsf{obsv}(x)$, meaning "given constraints C, a is a term which *observes* x in Γ," where x is a variable. This has basic axioms

$$\frac{\Gamma; ! \vdash x\ \mathsf{irrel}}{\Gamma; C \vdash a\ \mathsf{obsv}(x)}, \qquad \frac{}{\Gamma; C \vdash x\ \mathsf{obsv}(x)} \tag{3.39}$$

(h) $\Gamma; C \vdash A/B$ means "given constraints C, A is a substitute for B in Γ." When there are no constraints, we may simply write $\Gamma \vdash A/B$. This judgement has basic axioms

$$\frac{\Gamma; C \vdash A = B}{\Gamma; C \vdash A/B} \qquad \frac{\Gamma; C \vdash A/B \quad \Gamma; C \vdash B/C}{\Gamma; C \vdash A/C} \qquad \frac{\Gamma; C \vdash A/B \quad \Gamma; C \vdash A : C}{\Gamma; C \vdash B : C} \tag{3.40}$$

$$\frac{\Gamma; C \vdash A/B \quad \Gamma; C \vdash C : D \quad \Gamma; C \vdash AC : E}{\Gamma; C \vdash AC/BC} \tag{3.41}$$

$$\frac{\Gamma; C \vdash A/B \quad \Gamma; C \vdash `a \ \mathsf{inst} \quad \Gamma; C \vdash A`a : D}{\Gamma; C \vdash A`a/B`a} \tag{3.42}$$

(i) $\Gamma; C \vdash A <: B$ means “given constraints C, A is a subtype of B in Γ.” When there are no constraints, we may simply write $\Gamma \vdash A <: B$. This judgement has basic axioms

$$\frac{\Gamma; C \vdash A \ \mathsf{type} \quad \Gamma; C \vdash B \ \mathsf{type} \quad \Gamma; C \vdash A/B}{\Gamma; C \vdash A <: B} \qquad \frac{\Gamma; C \vdash A <: B \quad \Gamma; C \vdash a : A}{\Gamma; C \vdash a : B} \tag{3.43}$$

We will often apply the rules in bullet (h) along with those in Equation 3.43 implicitly.

(j) $\Gamma; C \vdash `a \preceq `b$ means “given constraints C, $`a$ is always before $`b$ in Γ.” This judgement has basic axioms

$$\frac{\Gamma; C \vdash `a \ \mathsf{inst}}{\Gamma; C \vdash `a \preceq `a} \qquad \frac{\Gamma; C \vdash `a \preceq `b \quad \Gamma; C \vdash `b \preceq `c}{\Gamma; C \vdash `a \preceq `c} \qquad \frac{\Gamma; C \vdash `a \preceq `b \quad \Gamma; C \vdash `b \preceq `a}{\Gamma; C \vdash `a = `b} \tag{3.44}$$

$$\frac{}{\Gamma, `\alpha \preceq `a; C \vdash `\alpha \preceq `a} \qquad \frac{}{\Gamma, `\alpha \succeq `a; C \vdash `a \preceq `\alpha} \tag{3.45}$$

$$\frac{}{\Gamma, `\alpha \in [`a, `b]; C \vdash `\alpha \preceq `b} \qquad \frac{}{\Gamma, `\alpha \in [`a, `b]; C \vdash `a \preceq `\alpha} \tag{3.46}$$

We may write $\Gamma; C \vdash `b \preceq `a$ as $\Gamma; C \vdash `a \succeq `b$. Similarly, we write $\Gamma; C \vdash `a \in [`b, `c]$ to mean $\Gamma; C \vdash `b \preceq `a$ and $\Gamma; C \vdash `a \preceq `c$.

3.3 Intuitionistic Type Theory

We now define the core intuitionistic type theory underlying `isotope`, which is simply a slightly modified version of the Calculus of Inductive Constructions.

3.3.1 Universes

Intuitionistic terms are equipped with a hierarchy of intuitionistic universes $\mathcal{U}_i$ parametrized by linearities ℓ, with the following typing rules [3]

$$\frac{}{\vdash \mathcal{U}_i : \mathsf{Lin} \to \mathcal{U}_{i+1}\ \mathsf{scalar}} \qquad \frac{i \leq j \quad \ell \preceq \ell'}{\vdash \mathcal{U}_i\ \ell / \mathcal{U}_j\ \ell'} \qquad \frac{\Gamma; ! \vdash A : \mathcal{U}_i\ \ell}{\Gamma; C \vdash A\ \mathsf{type}(\ell)} \tag{3.47}$$

where Lin is the inductive type (see Section 3.3.3) given by

$$\mathsf{Ind}([\mathsf{Lin} : \mathcal{U}_1] := [\mathsf{scalar} : \mathsf{Lin}, \mathsf{affine} : \mathsf{Lin}, \mathsf{relevant} : \mathsf{Lin}, \mathsf{linear} : \mathsf{Lin}]) \tag{3.48}$$

While universes are parametrized by linearities, any term can be used nonlinearly within an intuitionistic term, as linearity is enforced by constraint sets C and we are usually able to use $C =!$ in the intuitionistic setting (which disables all restrictions). In general, we write $\mathcal{U}_i$ as shorthand for $\mathcal{U}_i$ scalar where there is no risk of confusion. A type $A : \mathcal{U}_i \ell$ for $\ell \neq \mathsf{scalar}$ is called a *mixed type*, as it may contain both intuitionistic and machine components.

---

[3] The type $A \to B$ is defined in Section 3.3.2

3.3.2 Dependent Function Types

As in Section 2.3.2, we introduce typing rules for dependent function types

$$\frac{\Gamma; ! \vdash A : \mathcal{U}_i\ \ell \quad \Gamma, x : A; ! \vdash B : \mathcal{U}_i\ \ell}{\Gamma; C \vdash \Pi x : A.B : \mathcal{U}_i\ \ell} \qquad \frac{\Gamma, x : A; C \vdash s : B \quad x \notin \mathsf{fv}(C)}{\Gamma; C \vdash \lambda x.s : \Pi x : A.B} \tag{3.49}$$

with $A \to B \equiv \Pi - : A.B$. We introduce *application* of dependent functions

$$\frac{\Gamma; C \vdash f : \Pi x : A.B \quad \Gamma; C' \vdash a : A}{\Gamma; C, C' \vdash fa : B[a/x]} \tag{3.50}$$

which allows us to define the β*-reduction* relation

$$\beta = \{((\lambda x.s)t, s[t/x]) : s, t \in \Lambda, x \in \mathcal{V}\} \tag{3.51}$$

We have that $\beta \subseteq \mathsf{i}$, implying in particular that $s \to_\beta s' \implies s \to s'$. These terms have variable sets

$$\mathsf{v}(\Pi x : A.B) = \mathsf{v}(A) \cup (\mathsf{v}(B) \setminus \{x\}), \qquad \mathsf{v}(\lambda x.s) = \mathsf{v}(s) \setminus \{x\}, \qquad \mathsf{v}(st) = \mathsf{v}(s) \cup \mathsf{v}(t) \tag{3.52}$$

for $\mathsf{v} = \mathsf{fv}, \mathsf{cv}$ and reduction axioms

$$\frac{A \to_R A'}{\Pi x : A.B \to_R \Pi x : A'.B} \qquad \frac{B \to_R B'}{\Pi x : A.B \to_R \Pi x : A.B'} \qquad \frac{s \to_R s'}{\lambda x.s \to_R \lambda x.s'} \tag{3.53}$$

$$\frac{s \to_R s'}{st \to_R s't} \qquad \frac{t' \to_R t}{st \to_R st'} \tag{3.54}$$

3.3.3 Inductive Types and Pattern Matching

We repeat the definition of inductive definition from Section 2.3.9

Definition 17 (Inductive Definition)**.** *An* inductive definition *is an expression of the form*

$$\mathsf{Ind}(\Gamma_I := \Gamma_C) \tag{3.55}$$

where $\Gamma_I = [I_j : A_j], \Gamma_C = [c_{jk} : C_{jk}]$ *are sets of type bindings such that each* A_j *is an arity (see Definition 7) and each* C_{jk} *is a type of constructor for* I_j *(see Definition 6)* [4]*.*

We adapt Definition 9 for a well-formed inductive definition from Section 2.3.9 to `isotope`'s type theory as follows

Definition 18 (Well-formed Inductive Definition)**.** *We will say an inductive definition* $\mathcal{I} \equiv \mathsf{Ind}([I_j : A_j] := [c_{jk} : C_{jk}])$ *is* well-formed *in* $\Gamma; C$ *if*

- *Each* A_j *is a type in* $\Gamma; C$*, i.e.* $\Gamma; C \vdash A_j\ \mathsf{type}$*.*
- *Each* C_{jk} *is a type of constructor of* I_j *which satisfies the positivity condition (see Definition 8) for all types in the definition* $I_1, ..., I_J$
- *Each* A_j *is an arity of sort* s_j *(see Definition 7)*
- $\forall j, k, \Gamma, I_1 : A_1, ..., I_J : A_J; C \vdash C_{jk} : s_j$

We will write this judgement as

$$\Gamma; C \vdash \mathcal{I}\ \mathsf{ok} \tag{3.56}$$

---

[4] Note that Definition 7 refers to universes $\mathcal{U}_i$; here, as previously described, we take that to mean $\mathcal{U}_i\ \mathsf{scalar}$. In particular, this disallows including mixed types in any inductive types, which, for now, is an intentional simplification.

We introduce typing rules

$$\frac{\Gamma; C \vdash \mathcal{I}\ \mathsf{ok}}{\Gamma; C \vdash \mathcal{I} :: I_j : A_j}\ \mathit{Ind} \qquad \frac{\Gamma; C \vdash \mathcal{I}\ \mathsf{ok}}{\Gamma; C \vdash \mathcal{I} :: c_{jk} : C_{jk}[\mathcal{I} :: I_\ell / I_\ell]_{\ell=1..J}}\ \mathit{Cons} \tag{3.57}$$

When there is no risk of confusion, we will often omit judgements of the form $\Gamma \vdash \mathcal{I}\ \mathsf{ok}$ *and write* N *for* $\mathcal{I} :: N$.

Similarly, we adapt the definition of pattern-matching from Section 2.3.9 as follows:

Definition 19 (Pattern Matching)**.** *Given an inductive definition* $\mathcal{I} = \mathsf{Ind}([I_j : A_j] := [c_{jk} : C_{jk}])$*, for each* I_j*, we define an* case statement $\mathcal{I} :: \mathsf{case}_{I_j}$ *with typing rule*

$$\frac{\begin{array}{rl} \Gamma; C & \vdash \mathcal{I}\ \mathsf{ok} \\ \Gamma; C & \vdash i : \mathcal{I} :: I_j \\ \Gamma; C & \vdash F : \mathcal{I} :: I_j \to \mathcal{U}_i \\ (\Gamma, (p_n : P_n)_{n=1..N_k}; C & \vdash r_k : F(c_{jk}\ p_1\ \ldots\ p_{N_k}))_k \end{array}}{\Gamma; C \vdash \mathcal{I} :: \mathsf{case}_{I_j}\ i\ \{(c_{jk}\ p_1\ \ldots\ p_{N_k} \mapsto r_k)_k\} : F(i)}\ \mathit{case} \tag{3.58}$$

where (see Definition 6)

$$P_k = \mathsf{param}_I(C_{jk}), \qquad N_k = \mathsf{len}(P_k)$$

To interpret pattern matching, we introduce a reduction rule $\iota \subseteq \mathsf{i}$*, such that we have, for*

$k = 1, ..., n$,

$$(\mathcal{I} :: \mathsf{case}_{I_j}\ (\mathcal{I} :: c_{jk}\ a_1\ ...\ a_{N_k})\ \{c_{j1}\ p_1\ ...\ p_{N_1} \mapsto r_1, ..., c_{jK_j}\ p_1\ ...\ p_{N_{K_j}} \mapsto r_{K_j}\},$$
$$r_k[a_n/p_n]_{\ell=1,...,N_k}) \in \iota \quad (3.59)$$

Where $\mathcal{I}$ is clear from context, we may omit it.

We define

$$\mathsf{v}(\mathsf{Ind}([I_j : A_j] := [c_{jk} : C_{jk}])) = \left(\bigcup_j \mathsf{v}(A_j) \cup \bigcup_k \mathsf{v}(C_{jk})\right) \setminus \{(I_j)_j\} \quad (3.60)$$

for $\mathsf{v} = \mathsf{fv}, \mathsf{cv}$, and, similarly, introduce reduction rules

$$\frac{i \to_R i'}{\mathsf{case}_{I_j}\ i\ \{(c_{jk}\ p_1\ ...\ p_{N_k} \mapsto r_k)_k\} \to_R \mathsf{case}_{I_j}\ i'\ \{(c_{jk}\ p_1\ ...\ p_{N_k} \mapsto r_k)_k\}} \quad (3.61)$$

Similarly, where $k \neq \ell \implies r_k \equiv r'_k$, we have

$$\frac{r_\ell \to_R r'_\ell}{\mathsf{case}_{I_j}\ i\ \{(c_{jk}\ p_1\ ...\ p_{N_k} \mapsto r'_k)_k\} \to_R \mathsf{case}_{I_j}\ i\ \{(c_{jk}\ p_1\ ...\ p_{N_k} \mapsto r'_k)_k\}} \quad (3.62)$$

We introduce basic inductive types

$$\mathsf{Ind}([\mathbf{0} : \mathcal{U}_1] := []) \tag{3.63}$$

$$\mathsf{Ind}([\mathbf{1} : \mathcal{U}_1] := [() : \mathbf{1}]) \tag{3.64}$$

$$\mathsf{Ind}([\mathsf{bool} : \mathcal{U}_1] := [\mathsf{true} : \mathsf{bool}, \mathsf{false} : \mathsf{bool}]) \tag{3.65}$$

$$\mathsf{Ind}([\mathbb{N} : \mathcal{U}_1] := [0 : \mathbb{N}, \mathsf{s} : \mathbb{N} \to \mathbb{N}]) \tag{3.66}$$

$$\mathsf{Ind}([\mathsf{Id}_A : \mathcal{U}_i] := [\mathsf{refl}_A : \Pi a : A.\mathsf{Id}_A\ a\ a]) \text{ where } A : \mathcal{U}_i \tag{3.67}$$

$$\mathsf{Ind}([\mathsf{Maybe}\ A : \mathcal{U}_i]) := [\mathsf{Just}_A : A \to \mathsf{Maybe}\ A, \mathsf{Nothing}_A : \mathsf{Maybe}\ A] \text{ where } A : \mathcal{U}_i \tag{3.68}$$

$$\mathsf{Ind}([\Sigma a : A.B : \mathcal{U}_i] := [(,) : \Pi a : A.B \to \Sigma a : A.B]) \text{ where } A, B : \mathcal{U}_i \tag{3.69}$$

We assume these are always in scope, in addition, we omit subscript A when there is no risk of confusion (e.g., writing Id or refl instead of Id_A or refl_A). We assume the existence of the following functions on inductive types, with a proper definition (and, hopefully, interpreter optimization):

- Arithmetic on $\mathbb{N}$, e.g. $\mathsf{add}, \mathsf{sub}, \mathsf{mod}$, etc. We will often write this with operators, e.g. $+, -$.

- Logic on bool (e.g. $\mathsf{and}, \mathsf{or}$)

- A predicate $\mathsf{Le} : \mathbb{N} \to \mathbb{N} \to \mathcal{U}_1$, where $\mathcal{L}]\ n\ m$ means $n \leq m$. Similar predicates Lt for $<$, and so on.

- Wrapping n-bit arithmetic on $\mathsf{B}_n \equiv \Sigma m : \mathbb{N}.\mathsf{Lt}\ m\ 2^m$, of the form add_n, sub_n, etc, along with a wrapping n-bit truncation function $\mathsf{trunc}_n : \mathbb{N} \to \mathsf{B}_n$ such that

$$\forall \text{closed } b \in \mathsf{B}_n . \mathsf{trunc}_n(\pi_1 b) = b \tag{3.70}$$

and n-bit operations on B_n, e.g. $\mathsf{and}_n, \mathsf{shr}_n$ and xor_n. We will often write these with operators, e.g. $+, -, \oplus$.

- Defining $\mathsf{F}_n \equiv \Sigma m : \mathbb{N}.\mathsf{Lt}\ m\ n$ (in particular, $\mathsf{B}_n = \mathsf{F}_{2^n}$), we assume the existence of functions $\mathsf{switch}_A : A \to ... \to A \to \mathsf{F}_n \to A$ which, given n values of type A, return the n^{th}.

3.3.4 Fixpoints

As in Section 2.3.8, we define a fixpoint definition as follows:

Definition 20 (Fixpoint Definition)**.** *Given symbols $f_1, ..., f_n$ and terms $F_1, ..., F_n$, we define a* fixpoint definition *to be an expression of the form* $\mathsf{Fix}([f_j = F_j])$*, which we will view as mutually recursively defining each symbol f_j in terms of the other symbols f_k. For a fixpoint definition $\mathcal{F}$, we introduce terms $\mathcal{F} :: f_j$ corresponding to each symbol f_j. Where there is no risk of confusion, we will write f_j to mean $\mathcal{F} :: f_j$.*

The typing rule is the same, namely, for $\mathcal{F} \equiv \mathsf{Fix}([f_j : F_j = D_j])$

$$\frac{(\Gamma, (f_i : F_i)_i; ! \vdash D_j : F_j)_j \quad \Gamma \vdash \mathcal{F}\ \mathsf{terminating}}{\Gamma; C \vdash \mathcal{F} :: f_k : F_k} \tag{3.71}$$

where the termination checker used is unspecified but guaranteed to support at least primitive recursion.

3.4 Instants and Borrow Checking

In this section, we define *instants.* We then demonstrate how types and terms may be parametrized by instants using the $\forall$ and $\hat{\lambda}$ operators. Finally, we introduce *machine universes* parametrized by instants and linearities.

3.4.1 Instants

The core idea of `isotope` is the introduction of *instants*, which we use as building blocks to define a notion of a value's *lifetime*. In particular, given a *resource* r, we introduce instants representing the *beginning* $\mathsf{b}r$, *consumption* $\mathsf{c}r$, and *end* $\mathsf{e}r$ of the resource via the following rules

$$\frac{\Gamma; C \vdash a\ \mathsf{rsrc}}{\Gamma; C \vdash \mathsf{b}a\ \mathsf{inst}(0)} \qquad \frac{\Gamma; C \vdash a\ \mathsf{rsrc}}{\Gamma; C \vdash \mathsf{c}a\ \mathsf{inst}(0)} \qquad \frac{\Gamma; C \vdash a\ \mathsf{rsrc}}{\Gamma; C \vdash \mathsf{e}a\ \mathsf{inst}(0)} \tag{3.72}$$

We introduce basic ordering rules

$$\frac{\Gamma; C \vdash a : A}{\Gamma; C \vdash \mathsf{b}a \preceq \mathsf{c}a} \qquad \frac{\Gamma; C \vdash a : A}{\Gamma; C \vdash \mathsf{c}a \preceq \mathsf{e}a} \tag{3.73}$$

We may now describe moments in time such as, e.g., "the time a is consumed" ($\mathsf{c}a$) or "the time a is fully constructed" ($\mathsf{b}a$), but still cannot describe instants such as "some time after both a and b are fully constructed, but before c is consumed." To do so, we introduce the *meet* $`a \wedge `b$ and *join* $`a \vee `b$ of instants $`a, `b$, representing the *latest* instant before $`a$ and $`b$

and the *earliest* instant after ‘a and ‘b respectively. We have introduction rules [5]

$$\frac{\Gamma; C \vdash {}`a\ \mathsf{inst}(v) \quad \Gamma; C \vdash {}`a\ \mathsf{inst}(v)}{\Gamma; C \vdash {}`a \wedge {}`b\ \mathsf{inst}(v)} \qquad \frac{\Gamma; C \vdash {}`a\ \mathsf{inst}(v) \quad \Gamma; C \vdash {}`b\ \mathsf{inst}(v)}{\Gamma; C \vdash {}`a \vee {}`b\ \mathsf{inst}(v)} \tag{3.74}$$

with variable sets

$$\mathsf{v}({}`a \wedge {}`b) = \mathsf{v}({}`a \vee {}`b) = \mathsf{v}({}`a) \cup \mathsf{fv}({}`b) \tag{3.75}$$

for $\mathsf{v} = \mathsf{fv}, \mathsf{cv}$, and reduction rules

$$\frac{`a \to_R {}`a'}{`a \wedge {}`b \to_R {}`a' \wedge {}`b} \qquad \frac{`b \to_R {}`b'}{`a \wedge {}`b \to_R {}`a \wedge {}`b'} \qquad \frac{`a \to_R {}`a'}{`a \wedge {}`b \to_R {}`a' \vee {}`b} \qquad \frac{`b \to_R {}`b'}{`a \wedge {}`b \to_R {}`a \vee {}`b'} \tag{3.76}$$

We introduce ordering rules

$$\frac{\Gamma; C \vdash {}`a_1\ \mathsf{inst}(v) \quad \Gamma; C \vdash {}`a_2\ \mathsf{inst}(v)}{\Gamma; C \vdash {}`a_1 \wedge {}`a_2 \preceq {}`a_i} \qquad \frac{\Gamma; C \vdash {}`b \preceq {}`a_1 \quad \Gamma; C \vdash {}`b \preceq {}`a_2}{\Gamma; C \vdash {}`b \preceq {}`a_1 \wedge {}`a_2} \tag{3.77}$$

$$\frac{\Gamma; C \vdash {}`a_1\ \mathsf{inst}(v) \quad \Gamma; C \vdash {}`a_2\ \mathsf{inst}(v)}{\Gamma; C \vdash {}`a_i \preceq {}`a_1 \vee {}`a_2} \qquad \frac{\Gamma; C \vdash {}`a_1 \preceq {}`b \quad \Gamma; C \vdash {}`a_2 \preceq {}`b}{\Gamma; C \vdash {}`a_1 \vee {}`a_2 \preceq {}`b} \tag{3.78}$$

In particular, these imply that $\wedge$ and $\vee$ are commutative, as

$$\frac{\dfrac{\dfrac{}{\Gamma; C \vdash {}`a \wedge {}`b \preceq {}`b} \quad \dfrac{}{\Gamma; C \vdash {}`a \wedge {}`b \preceq {}`a}}{\Gamma; C \vdash {}`a \wedge {}`b \preceq {}`b \wedge {}`a} \qquad \dfrac{\dfrac{}{\Gamma; C \vdash {}`b \wedge {}`a \preceq {}`a} \quad \dfrac{}{\Gamma; C \vdash {}`b \wedge {}`a \preceq {}`b}}{\Gamma; C \vdash {}`b \wedge {}`a \preceq {}`a \wedge {}`b}}{\Gamma; C \vdash {}`a \wedge {}`b = {}`b \wedge {}`a} \tag{3.79}$$

---

[5] Note in particular that we do not allow taking the meet ($\wedge$) or join ($\vee$) of instants with mixed variances; e.g., if ‘a $\mathsf{inst}(+)$ and ‘b $\mathsf{inst}(-)$, then ‘$a \wedge$ ‘b is invalid. On the other hand, as invariant instants ‘a $\mathsf{inst}(0)$ can be coerced to either covariant or contravariant instants, we may freely take their meet and join with either.

and likewise for $\vee$. We also introduce units for the $\wedge$ and $\vee$ via the following rules

$$\frac{}{\vdash \infty\ \mathsf{inst}(0)} \qquad \frac{}{\vdash -\infty\ \mathsf{inst}(0)} \tag{3.80}$$

with empty free variable sets and ordering rules

$$\frac{\Gamma; C \vdash `a\ \mathsf{inst}}{\Gamma; C \vdash `a \preceq \infty} \qquad \frac{\Gamma; C \vdash `a\ \mathsf{inst}}{\Gamma; C \vdash -\infty \preceq' a} \tag{3.81}$$

giving $`a \wedge \infty = \infty \wedge `a = `a \vee -\infty = -\infty \vee `a = `a$. As we wish to be able to express strict constraints $`a \prec `b$, we introduce *successor* and *predecessor* instants

$$\frac{\Gamma; C \vdash `a\ \mathsf{inst}(v)}{\Gamma; C \vdash \mathsf{is}`a\ \mathsf{inst}(v)} \qquad \frac{\Gamma; C \vdash `a\ \mathsf{inst}(v)}{\Gamma; C \vdash \mathsf{ip}`a\ \mathsf{inst}(v)} \tag{3.82}$$

with variable sets

$$\mathsf{v}(\mathsf{is}`a) = \mathsf{v}(\mathsf{ip}`a) = \mathsf{v}(`a) \tag{3.83}$$

for $\mathsf{v} = \mathsf{fv}, \mathsf{cv}$, and reduction rules

$$\frac{`a \to_R `a'}{\mathsf{is}`a \to_R \mathsf{is}`a'} \qquad \frac{`a \to_R `a'}{\mathsf{is}`a \to_R \mathsf{ip}`a'} \tag{3.84}$$

and ordering rules

$$\frac{\Gamma; C \vdash `a\ \mathsf{inst}}{\Gamma; C \vdash `a \preceq \mathsf{is}`a} \qquad \frac{\Gamma; C \vdash \mathsf{is}`a \preceq `a}{\Gamma; C \vdash `a = \infty} \qquad \frac{\Gamma; C \vdash `a\ \mathsf{inst}}{\Gamma; C \vdash \mathsf{ip}`a \preceq `a} \qquad \frac{\Gamma; C \vdash `a \preceq \mathsf{ip}`a}{\Gamma; C \vdash `a = -\infty} \tag{3.85}$$

We also introduce the ability to create new, opaque instants $‘x(u)$ from an infinite set of names (we assume each name is unique, assigning human-readable names as a convenience),

$$\frac{}{\vdash ‘x(u)\ \mathsf{inst}(0)} \tag{3.86}$$

where $u \in \{0, 1\}$ denotes whether this instant is *separated*. If a constraint set equates two separated instants, we will consider it to be inconsistent.

We distinguish opaque instants from instant variables such as $‘\alpha$ by, conventionally, using Latin letters rather than Greek letters. Opaque instants do *not* show up in the free variable set of an instant.

3.4.2 Instant Parameters

Now that we have introduced a basic calculus of instants, we can define the type former $\forall$ to allow *universal quantification* over instants to have the following introduction rules

$$\frac{\Gamma, ‘\alpha \mathsf{R} R; C \vdash T : \mathcal{U}_i \quad ‘\alpha \notin \mathsf{sfv}(C)}{\Gamma; C \vdash \forall ‘\alpha \mathsf{R} R.T : \mathcal{U}_i} \qquad \frac{\Gamma, ‘\alpha \mathsf{R} R; C \vdash t : T \quad ‘\alpha \notin \mathsf{sfv}(C)}{\Gamma; C \vdash \hat{\mathsf{\lambda}} ‘\alpha \mathsf{R} R.t : \forall ‘a \mathsf{R} R.T} \tag{3.87}$$

Universally quantified terms have variable sets

$$\mathsf{v}(\forall ‘\alpha \mathsf{R} R.T) = \mathsf{v}(R) \cup (\mathsf{v}(T) \setminus \{‘\alpha\}) \tag{3.88}$$

for $\mathsf{v} = \mathsf{cv}, \mathsf{fv}$ and reduction rules

$$\frac{T \to_R T'}{\forall ‘\alpha \mathsf{R} R.T \to_R \forall ‘\alpha \mathsf{R} R.T'} \qquad \frac{R \to_R R'}{\forall ‘\alpha \mathsf{R} R.T \to_R \forall ‘\alpha \mathsf{R} R'.T} \tag{3.89}$$

where

$$\frac{`a \to_R `a'}{[`a, `b] \to_R [`a', `b]} \qquad \frac{`b \to_R `b'}{[`a, `b] \to_R [`a, `b']} \tag{3.90}$$

To allow specializing terms quantified by instants, we provide rules for instant application

$$\frac{\Gamma; C \vdash `a\ \mathsf{inst}(-) \quad \Gamma; C \vdash x : \forall `\alpha \preceq `b.T \quad \Gamma; C \vdash `a \preceq `b}{\Gamma; C \vdash x`a : T} \tag{3.91}$$

$$\frac{\Gamma; C \vdash `a\ \mathsf{inst}(+) \quad \Gamma; C \vdash x : \forall `\alpha \succeq `b.T \quad \Gamma; C \vdash `b \preceq `a}{\Gamma; C \vdash x`a : T} \tag{3.92}$$

$$\frac{\Gamma; C \vdash `a\ \mathsf{inst}(0) \quad \Gamma; C \vdash x : \forall `\alpha \in [`b, `c].T \quad \Gamma; C \vdash `a \in [`b, `c]}{\Gamma; C \vdash x`a : T} \tag{3.93}$$

As for term application, instant application has variable sets

$$\mathsf{v}(x`a) = \mathsf{v}(x) \cup \mathsf{v}(`a) \tag{3.94}$$

for $\mathsf{v} = \mathsf{fv}, \mathsf{cv}$, and reduction rules

$$\frac{x \to_R x'}{x`a \to_R x'`a} \qquad \frac{`a \to_R `a'}{x`a \to_R x`a'} \tag{3.95}$$

We also provide the following subtyping rules, taking into account the variance of the lifetime being quantified over

$$\frac{\Gamma; C \vdash x : \forall `\alpha \succeq `c.T \quad \Gamma; C \vdash `a \preceq `b}{\Gamma; C \vdash x`a / x`b} \qquad \frac{\Gamma; C \vdash x : \forall `\alpha \preceq `c.T \quad \Gamma; C \vdash `b \preceq `a}{\Gamma; C \vdash x`a / x`b} \tag{3.96}$$

3.4.3 Universes

We may now implement the hierarchy of *machine universes*, parametrized by instants and linearities

$$\frac{}{\vdash \mathcal{M}_i : \forall`\alpha \succeq -\infty, \Pi\ell : \mathit{Lin}.\mathcal{U}_{i+1}\ \ell} \qquad \frac{\Gamma; C \vdash A : \mathcal{M}_i`a\ \ell}{\Gamma \vdash A\ \mathsf{type}(\ell)} \tag{3.97}$$

Here, $\mathcal{M}_i`a\ \ell$ represents the universe of machine representable types with *denotations* of level $i$ which have linearity *at most* $\ell$ and live until *at least* the instant $`a$. Note that $\mathcal{M}_i'a\ell : \mathcal{U}_{i+1}$ is an intuitionistic type: this is because types themselves are not subject to ownership rules. We write

$$\mathcal{L}_i`a \equiv \mathcal{M}_i`a\ \mathsf{linear} \qquad \mathcal{A}_i`a \equiv \mathcal{M}_i`a\ \mathsf{affine} \qquad \mathcal{R}_i`a \equiv \mathcal{M}_i`a\ \mathsf{relevant} \qquad \mathcal{S}_i`a \equiv \mathcal{M}_i`a\ \mathsf{scalar} \tag{3.98}$$

$$\mathcal{M}_i\ell = \mathcal{M}_i\ \infty\ \ell, \qquad \mathcal{L}_i \equiv \mathcal{L}_i\ \infty \qquad \mathcal{A}_i \equiv \mathcal{A}_i\ \infty \qquad \mathcal{R}_i \equiv \mathcal{R}_i\ \infty \qquad \mathcal{S}_i \equiv \mathcal{S}_i\ \infty \tag{3.99}$$

and give subtyping rules

$$\frac{\Gamma; C \vdash `a \preceq `b \quad \Gamma; C \vdash \ell \preceq \ell'}{\Gamma; C \vdash \mathcal{M}_i`b\ \ell / \mathcal{M}_{i+j}`a\ \ell'} \qquad \frac{\Gamma; C \vdash `a\ \mathsf{inst} \quad \Gamma; C \vdash \ell : \mathsf{Lin}}{\Gamma; C \vdash \mathcal{M}_i`a\ \ell / \mathcal{U}_i} \tag{3.100}$$

In particular, we treat machine universes as subtypes of intuitionistic universes to allow the definition of families of machine-typed terms parametrized by intuitionistic constants. We now extend the rules in Equation 3.87 to make a universally quantified machine type itself a machine type (it is already an *intuitionistic* type since $\mathcal{M}_i`a\ \ell <: \mathcal{U}_i$) as follows

$$\frac{\Gamma, `\alpha \mathsf{R} R; C \vdash T : \mathsf{M}_i`b\ \ell}{\Gamma; C \vdash \forall`a\mathsf{R}R.T : \mathsf{M}_i`b\ \ell} \tag{3.101}$$

We define variable sets, for $\mathsf{v} = \mathsf{fv}, \mathsf{cv}$, as follows

$$\mathsf{v}(\mathcal{M}_i) = \varnothing \tag{3.102}$$

3.4.4 Variable Bindings

We introduce let-bindings with the following typing rule

$$\frac{\Gamma; x : A \vdash B \ \mathsf{type} \quad \Gamma; C \vdash a : A \quad \Gamma, x \leftarrow a : A; C' \vdash e : B \quad \Gamma, x \leftarrow a : A; ! \vdash e \ \mathsf{obsv}(x)}{\Gamma; \mathsf{bind}_\Gamma(\{(x : A, C, 0)\}, C') \vdash \mathsf{let}\ x = a\ \mathsf{in}\ e : B[a/x]} \tag{3.103}$$

and observation rules

$$\frac{\Gamma; ! \vdash a \ \mathsf{obsv}(y)}{\Gamma; ! \vdash \mathsf{let}\ x = a\ \mathsf{in}\ e \ \mathsf{obsv}(y)} \qquad \frac{\Gamma; ! \vdash a \ \mathsf{obsv}(y)}{\Gamma; ! \vdash \mathsf{let}\ x = a\ \mathsf{in}\ e \ \mathsf{obsv}(y)} \tag{3.104}$$

We give reduction rules

$$\frac{a \to_R a'}{\mathsf{let}\ x = a\ \mathsf{in}\ e \to_R \mathsf{let}\ x = a'\ \mathsf{in}\ e} \qquad \frac{e \to_R e'}{\mathsf{let}\ x = a\ \mathsf{in}\ e \to_R \mathsf{let}\ x = a\ \mathsf{in}\ e'} \tag{3.105}$$

and assign let-statements free variable set

$$\mathsf{fv}(\mathsf{let}\ x = a\ \mathsf{in}\ e) = \mathsf{fv}(a) \cup (\mathsf{fv}(e) \setminus \{x\}) \tag{3.106}$$

and constant variable set

$$\mathsf{cv}(\mathsf{let}\ x = a\ \mathsf{in}\ e) = \begin{cases} \mathsf{cv}(a) \cup \mathsf{cv}(e) & \text{if } x \notin \mathsf{cv}(e) \\ \mathsf{fv}(a) \cup (\mathsf{cv}(e) \setminus \{x\}) & \text{otherwise} \end{cases} \tag{3.107}$$

The expression $\mathsf{bind}_\Gamma(\{(x : A, C)\}, C')$ in Equation , in essence, introduces new opaque instants for the beginning/consumption/end of x (and, later, any components of x) while checking x itself is used in a way which respects linearity constraints. It then removes uses of x, and adds uses of the components of x from C, while constraining the newly added free instants to occur at appropriate times based off the dependencies between x and other variables. Formally, we use the following algorithm to compute $\mathsf{bind}_\Gamma(X, C)$, where X is a set of fully annotated variables with an optional flag $b \in \{0, 1\}$:

1. Initialize the result constraint set $C_R = (I_R, O_R, U_R, R_R) = C$.
2. **If**, for any $(x, C_x, b) \in X$,
    - $\mathsf{u}(x)$ occurs more than once in U_R, **or** $\mathsf{u}(x)$ and $\mathsf{p}(x)$ both occur in U_R **or** $\mathsf{u}(x)$ occurs in U_R and $b = 1$ **and**
    - we cannot deduce that $\Gamma, x \leftarrow: A; C_x \vdash x\ \mathsf{copy}$

   then return $C_R =!$ **else** remove any occurences of $\mathsf{u}(x)$ and $\mathsf{d}(x)$ from U_R
3. Collect every instant $\text{‘}a$ of the form $\mathsf{b}r$, $\mathsf{e}r$, $\mathsf{c}r$ with $X \cap \mathsf{fv}(r) \neq \varnothing$ in I into a set I_X
4. **For** each instant $\text{‘}a$ in I_X, replace $\text{‘}a$ in I_R with an opaque instant $\text{‘}a_o(u_a)$, written $\text{‘}a_o(1) = \text{‘}\mathsf{b}r(1)$, $\text{‘}i_o(0) = \text{‘}\mathsf{e}r(0)$, or $\text{‘}i_o(1) = \text{‘}\mathsf{c}r(1)$ for instants $\mathsf{b}r$, $\mathsf{e}r$, $\mathsf{c}r$ respectively, for convenience (but note that $\mathsf{fv}(\text{‘}i_o) = \varnothing$ for all such instants; names like $\text{‘}\mathsf{b}r$ are just a convenience, and, in particular, there is no actual dependency on r! We call this trick *skolemization.*). Note that beginnings and consumptions become separated, but ends *do not.*

5. **For** each instant $`i$ in $I_X$, **for** each instant $`j$ in I, **if** $\Gamma; C \vdash i \preceq j$, add $`i_o \preceq `j$ to I_R. **if** $\Gamma; C \vdash j \preceq i$, add $`j \preceq `i_o$ to I_R.

6. **For** $(x : A, C_x) \in X$,

   (a) Get the annotation $A : U$ from the fully annotated term $x : A$.

   (b) **If** $U = \mathcal{M}_i `a\ \ell$, **then** add $`\mathsf{e}x \preceq `a$ to I_R.

   (c) **For** all $\mathsf{u}(r), \mathsf{p}(r) \in C_x$,

      - **If** $\Gamma; C \vdash r\ \mathsf{copy}$, **then** add $\mathsf{b}r \preceq `\mathsf{b}x$ and $`\mathsf{b}x \preceq \mathsf{c}r$ to I_R (see Figure 3.5b)

      - **Else** add $\mathsf{c}r \preceq `\mathsf{b}x$ and $`\mathsf{b}x \preceq \mathsf{e}r$ to I_R (see Figure 3.5a)

   (d) **For** $\mathsf{d}(r) \in C_x$, add $\mathsf{b}r \preceq `\mathsf{b}x$ and $`\mathsf{b}x \preceq \mathsf{c}r$ to I_R (see Figure 3.5b)

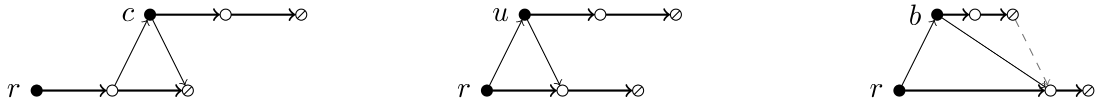
   (e) Catenate C_x with C_R

7. **Return** C_R

More than a syntactic convenience, let-statements allow us to convert a term with judgement $a : A$ to a term with judgement $x \leftarrow: A$, and hence, in particular, allow us to use the lifetime construction operators b, c, e. More importantly, while *denotationally* the let operator is transparent (see Equation 3.125 in Section 3.5.1), there is in general *no* rule requiring that

$$\mathsf{let}\ x = a\ \mathsf{in}\ e \to e \tag{3.108}$$

This is important, as we view the let operator as introducing a new resource on execution, taking ownership of whatever resources are owned by the expression a, and allowing it to be used within the scope e. The instant $\mathsf{b}x$ refers to this moment. The resource is then freed at some point between the end of the let-statement and the end of the current stack frame;



(a) $\mathsf{c}(r) \prec \mathsf{b}(c) \prec \mathsf{e}(r)$ (b) $\mathsf{b}(r) \prec \mathsf{b}(u) \prec \mathsf{c}(r)$ (c) $\mathsf{b}(r) \prec \mathsf{e}(b) \prec \mathsf{c}(r)$

Figure 3.5: Common patterns in constraint sets representing consumption, use, and borrowing respectively. Here, c consumes r, u uses r and b borrows r. The patterns here are constructed by the algorithm presented in Section 3.4.4. As usual, beginnings are solid black dots, consumptions white dots, and ends crossed-out dots. Note that Figure 3.5c is actually implied by a combination of Figure 3.5b and the lifetime bounds added in step 6b.

for such a resource x, $\mathsf{c}x$ refers to this moment, while $\mathsf{e}(x)$ refers to the last possible such moment. After the bind-operation, these instants are converted to opaque instants, leaving most of the constraints to be checked in the borrowck phase.

3.4.5 Borrow Checking

We now introduce the *borrow checking* algorithm; this will form the core of `isotope`'s type system. In particular, given a context Γ, we introduce a function $\mathsf{borrowck}_\Gamma(X, C)$ which, given a set of constraints C, and a set of variable-constraint pairs $X = \{(x : A, C_x, b)\}$, returns the constraints on a function with the variables in X returning a result having constraints C and borrowing flag b.

We begin by defining an instant constraint set's *maximal constraint set* as follows:

Definition 21. *An instant constraint set I is* maximal *with respect to $\Gamma; C$ if*

$$\forall `a, `b \in I_{\Gamma;C}, \Gamma; C \vdash `a \preceq `b \iff `a \equiv `b \lor (`a, `b) \in I^+ \tag{3.109}$$

where $I_{\Gamma;C}$ is the set of instants appearing in Γ or C, along with $\pm\infty$.

We provide the algorithm $\mathsf{maxconstrain}(\Gamma, C)$ for computing a maximal constraint set given $\Gamma; C$:

1. **If** $C =!$, **return** !

2. Initialize a result constraint set $C_R = C$, and a stack $\mathcal{S} = []$.

3. **For** each $(`a, `b) \in C$, push $`a$ and $`b$.

4. **For** each $`\alpha \mathsf{R} R \in \Gamma$, push $`\alpha$, and

    - **If** $\alpha \preceq `b$ or $\alpha \succeq `b$, push $`b$

    - **If** $\alpha \in [`b, `c]$, push $`b$ and $`c$.

5. **While** $\mathcal{S}$ is nonempty, pop $`a$, then

    - **If** $`a \equiv `b \wedge `c$, insert edges $(`a, `b)$, $(`a, `c)$, and push $`b, `c$

    - **If** $`a \equiv `b \vee `c$, insert edges $(`b, `a)$, $(`c, `a)$, and push $`b, `c$

    - **If** $`a \equiv \mathsf{is}`b$, insert edge $(`b, `a)$ and push $`b$.

    - **If** $`a \equiv \mathsf{ip}`b$, insert edge $(`a, `b)$ and push $`a$.

    - **If** $`a \equiv \mathsf{b}r$, insert edges $(`a, \mathsf{c}r)$, $(`a, \mathsf{e}r)$

    - **If** $`a \equiv \mathsf{c}r$, insert edges $(\mathsf{b}r, `a)$ and $(`a, \mathsf{e}r)$

    - **If** $`a \equiv \mathsf{e}r$, insert edges $(\mathsf{b}r, `a)$, $(\mathsf{e}r, `a)$

    - **If** $`a \equiv `\alpha$,

        - **If** $\alpha \preceq `b \in \Gamma$, insert edge $(`a, `b)$, and push $`b$.

        - **If** $\alpha \succeq `b \in \Gamma$, insert edge $(`b, `a)$, and push $`b$.

        - **If** $\alpha \in [`b, `c] \in \Gamma$, insert edges $(`b, `a)$ and $(`a, `c)$ and push $`b$ and $`c$.

    - **If** $`a \equiv `x$, do nothing.

6. For every node $`a$ in the resulting graph $I$, add edges $(`a, \infty)$ and $(-\infty, `a)$ to I.

7. Return I.

We define *consistent* constraint sets as those which do not identify separated instants (i.e., unique opaque instants, $\pm\infty$, beginnings and consumptions). Formally:

Definition 22 (Separated Instants)**.** Separated instants *are defined as:*

- *Separated opaque instants* $`x(1)$
- *Infinities* $\pm\infty$
- *Beginnings* $\mathsf{b}r$ *and consumptions* $\mathsf{c}r$

Definition 23. *An instant constraint set I is* locally consistent *if, for all separated instants $`i, `j$,*

$$(`i, `j) \in I^{=} \implies `i \equiv `j, \qquad (\infty, `i) \notin I^{+}, \qquad (`i, -\infty) \notin I^{+} \tag{3.110}$$

Definition 24. *A constraint set C is* consistent in Γ *if, for all separated instants $`i, `j$, we have*

$$`i \equiv `j \qquad \vee \qquad \Gamma; C \not\vdash `i = `j \tag{3.111}$$

Unfortunately, this definition is quite difficult to check, so we provide an easier one:

Definition 25. *A constraint set C is* weakly consistent in Γ *if, for all separated instants $`i, `j$ in $I_{\Gamma;C}$, we have*

$$`i \equiv `j \qquad \vee \qquad \Gamma; C \not\vdash `i = `j \tag{3.112}$$

Claim 6. *If $\hat{I} \supseteq I$ is maximal in Γ, then $C = (I, O, U, R)$ is consistent in Γ if $\hat{I}$ is locally consistent.*

Proof. Assume $\hat{I} \supseteq I$ is maximal in Γ and locally consistent. Then, for all $`a, `b \in I$, we have $`a, `b \in \hat{I}$.

Assume that $\Gamma; C \vdash `a = `b$ for separable instants $`a, `b \in C \cup \{\pm\infty\}$; we hence have $\Gamma; C \vdash `a \preceq `b$ and $\Gamma; C \vdash `b \preceq `a$ Since $\hat{I}$ is maximal in Γ, we have either $`a \equiv `b$ or $(`a, `b), (`b, `a) \in \hat{I}^+$; this would imply by definition that $\hat{I}$ is inconsistent, yielding a contradiction. Therefore, by definition, C is consistent in Γ. □

Conjecture 1. *If C is weakly consistent in Γ, then it is consistent in Γ*

We then define a subroutine $\mathsf{localck}(I)$, which checks an instant constraint set I for local consistency, as follows:

1. **For** each cycle c in I, treated as a graph, **if** c contains more than one separated instant, **return false**.
2. **return true**.

Using this, we then define a subroutine $\mathsf{cleanup}_\Gamma(C)$ which checks a constraint set for consistency (assuming Conjecture 1) and then returns it with floating instants (defined as opaque instants having only constraints involving other opaque instants) removed, as follows:

1. If $C = !$, return !, otherwise let $C = (I, O, U, R)$
2. Compute $\hat{I} = \mathsf{maxconstrain}(\Gamma, C)$
3. **If** $\mathsf{localck}(\hat{I})$ is false, **then return** !
4. **For** each connected component C in $\hat{I}$ (taken as an undirected graph), **if** all elements in C are opaque instants (i.e., of the form $`x(u)$ for $u \in \{0, 1\}$), remove them from $\hat{I}$.

Finally, we may define the borrow checking algorithm $\mathsf{borrowck}_\Gamma(X, C)$ as follows:

1. Bind the variables in X to produce constraint set $C' = \mathsf{bind}_\Gamma(X, C)$.

2. **return** $\mathsf{cleanup}_\Gamma(C')$

3.5 Machine Terms

In this section, we introduce *machine terms*, that is, terms subject to ownership typing. We begin by introducing the *denotation operator*, which provides an *internal* denotational semantics for `isotope` by converting any `isotope` term to a term in `isotope`'s intuitionistic fragment. We then cover functions, inductive datatypes, pattern matching, fixpoints, and builtin types such as arrays.

3.5.1 The Term and Denotation Operators

In Krishnaswami et al. [12], the universes of linear and intuitionistic types are connected by an *adjunction* of functors F, G taking intuitionistic types to linear types and vice versa. In this same spirit, we introduce the *term operators* (T, t) and *denotation operator* d to take intuitionistic types to machine types and vice versa. Specifically, given an intuitionistic term $a : A$, we define the machine term $\mathsf{t}a : \mathsf{T}A$ to represent a *computationally irrelevant* (with respect to machine terms) term of type A which can be used to carry intuitionistic data in machine types, via the following rules

$$\frac{\Gamma; ! \vdash A : \mathcal{U}_i}{\Gamma; C \vdash \mathsf{T}A : \mathcal{S}_i} \qquad \frac{\Gamma; C \vdash a : A}{\Gamma; C \vdash \mathsf{t}a : \mathsf{T}A} \tag{3.113}$$

Similarly, given a machine term $a : A$, we can consider it's "meaning," or *denotation*, as a mathematical value (i.e., an intuitionistic term) to be given by the *denotation operator* $\mathsf{d}a : \mathsf{d}A$, via the following rules

$$\frac{\Gamma; ! \vdash A : \mathcal{U}_i \ \mathsf{linear}}{\Gamma; C \vdash \mathsf{d}A : \mathcal{U}_i} \qquad \frac{\Gamma; C \vdash a : A}{\Gamma; C \vdash \mathsf{d}a : \mathsf{d}A} \tag{3.114}$$

These terms have free variable sets

$$\mathsf{fv}(\mathsf{d}a) = \mathsf{fv}(\mathsf{t}a) = \mathsf{fv}(a), \qquad \mathsf{fv}(\mathsf{T}A) = \mathsf{fv}(A) \tag{3.115}$$

On the other hand, both the t and d operators remove constancy constraints; that is,

$$\mathsf{cv}(\mathsf{d}a) = \mathsf{cv}(\mathsf{t}a) = \mathsf{cv}(\mathsf{T}A) = \varnothing \tag{3.116}$$

We provide reduction rules

$$\frac{a \to_R a'}{\mathsf{d}a \to_R \mathsf{d}a'} \qquad \frac{a \to_R a'}{\mathsf{t}a \to_R \mathsf{t}a'} \qquad \frac{A \to_R A'}{\mathsf{T}A \to_R \mathsf{T}A'} \tag{3.117}$$

We define the values of d via the *contextual* relation $\delta \subset \mathsf{i}$. For a computationally irrelevant term $\mathsf{t}a$, it's "meaning" $\mathsf{dt}a$ can most naturally be taken to be the meaning of the original term a. This yields rules

$$\frac{}{(\mathsf{dt}a, \mathsf{d}a) \in \delta} \qquad \frac{}{(\mathsf{dT}A, \mathsf{d}A) \in \delta} \tag{3.118}$$

On the other hand, for a purely intuitionistic term $t : T$, we should have $\mathsf{d}t = t$ and $\mathsf{d}T = T$; in particular, this gives us rules for typing univeres

$$\frac{}{(\mathsf{d}\mathcal{U}_i, \mathcal{U}_i) \in \delta} \qquad \frac{}{(\mathsf{d}\mathcal{M}_i`a\ \ell, \mathcal{U}_i) \in \delta} \tag{3.119}$$

In general, there are no rules for the d operator's behavior on variable bindings: $\mathsf{d}x$ is just $\mathsf{d}x$. However, we may extend the substitution operator as follows: for a term s, $s[a/\mathsf{d}x]$ substitutes a for every occurence of $\mathsf{d}x$ *in the δ-normal form of s*, and is undefined if x occurs otherwise. We make the following, for now unproven, claim

Conjecture 2. *For every well-typed term s and every variable x, $(\mathsf{d}s)[a/\mathsf{d}x]$ is well-defined.*

We use this operation to define rules for λ and Π as follows: as the operation d distributes over (term) application and abstraction, we have rules

$$\frac{}{(\mathsf{d}(st), (\mathsf{d}s)(\mathsf{d}t)) \in \delta}\ , \qquad \frac{}{(\mathsf{d}(\lambda x.s), \lambda x.\mathsf{d}s[x/\mathsf{d}x]) \in \delta} \tag{3.120}$$

which requires that the operation d distributes over Π as follows

$$\frac{}{(\mathsf{d}(\Pi x : A.B), \Pi x : \mathsf{d}A.\mathsf{d}B[x/\mathsf{d}x]) \in \delta} \tag{3.121}$$

As d produces intuitionistic terms, it has the effect of ignoring instants and hence instant parameters, giving rules

$$\frac{}{(\mathsf{d}(x`a), \mathsf{d}x) \in \delta} \qquad \frac{}{(\mathsf{d}(\hat{\lambda}`\alpha\mathsf{R}R.x), \mathsf{d}x) \in \delta} \qquad \frac{}{(\mathsf{d}(\forall`\alpha\mathsf{R}R.T), \mathsf{d}T) \in \delta} \tag{3.122}$$

To enable the manipulations of terms behind the operator t, we introduce relation $\tau \subset \mathsf{i}$. We introduce computation rule

$$\frac{}{((\mathsf{t}a)(\mathsf{t}b), \mathsf{t}(ab)) \in \tau} \tag{3.123}$$

This requires us to have T distribute over dependent function types as follows

$$\frac{}{(\mathsf{T}(\Pi x : A.B), (\Pi x : \mathsf{T}A.\mathsf{T}B)) \in \tau} \tag{3.124}$$

In spirit, this rule combined with Equation 3.118 makes d and (T, t) into a sort of *adjunction* between intuitionistic terms and machine terms, but a proper categorical semantics is out of the scope of this thesis.

Furthermore, we add rules for the particular case of the denotation of bound variables, in particular within let-bindings, as follows

$$\frac{}{\Gamma, x \leftarrow a : A; C \vdash (\mathsf{d}x, \mathsf{d}a) \in \delta} \qquad \frac{}{(\mathsf{d}(\mathsf{let}\ x = a\ \mathsf{in}\ e), (\mathsf{d}e)[\mathsf{d}a/\mathsf{d}x]) \in \delta} \tag{3.125}$$

3.5.2 Layout

As discussed in Section 2.5, to compile a type to a low-level language, we need to know it's *layout*, i.e., it's size and alignment, to be able to allocate enough space for it on the stack and to pass it to/from functions. Dynamically sized types such as variable-length-arrays, on the other hand, must be handled behind pointers. We introduce operators sizeof and $\mathsf{alignof}$ as follows:

$$\frac{\Gamma; ! \vdash A : \mathcal{M}_i\text{‘}a\ \ell}{\Gamma; C \vdash \mathsf{sizeof}\ A : \mathbb{N}} \qquad \frac{\Gamma; ! \vdash A : \mathcal{M}_i\text{‘}a\ \ell}{\Gamma; C \vdash \mathsf{alignof}\ A : \mathbb{N}} \tag{3.126}$$

We define the value of these operators via the relation $\sigma \subset \mathsf{i}$. We have not really introduced any machine types beyond $\mathsf{T}A$ yet, so we only have one σ-rule to give so far:

$$\frac{\Gamma; ! \vdash A \;\mathsf{type}}{\Gamma; C \vdash (\mathsf{sizeof}\ \mathsf{T}A, \mathsf{Just}\ 0) \in \sigma} \qquad \frac{\Gamma; ! \vdash A \;\mathsf{type}}{\Gamma; C \vdash (\mathsf{alignof}\ \mathsf{T}A, 0) \in \sigma} \tag{3.127}$$

In other words, squashed types are ZSTs. We say a type A is *sized* with respect to S, written $\mathsf{sized}_S(A)$, if $\mathsf{sizeof}\ A$ and $\mathsf{alignof}\ A$ i-normalize to terms only containing variables defined in Γ; i.e.

$$\mathsf{rfv}(\mathsf{sizeof}\ A) \cup \mathsf{rfv}(\mathsf{alignof}\ A) \subseteq S \tag{3.128}$$

3.5.3 Functions

We're now ready to define the core language construct of `isotope`: machine function types. The function types we provide here are represented by static function pointers; in particular, they are unboxed, and we do *not* support closures. We provide formation rule

$$\frac{\begin{array}{cc}(\mathsf{sym}(\Gamma), (x_j : \mathsf{d}A_j)_{j<i}; ! \vdash A_i : \mathsf{M}`a_i\ \ell_i)_i & (\mathsf{sized}_\Gamma(A_i))_i \\ \Gamma, (x_i, \mathsf{d}A_i)_i; ! \vdash R : \mathsf{M}`a_i\ \ell_i & \mathsf{sized}_\Gamma(R)\end{array}}{\Gamma; C \vdash \mathsf{Fn}((x_i : A_i)_i) \to R : \mathcal{S}_i} \tag{3.129}$$

In brief, Equation 3.129 allows us to construct a machine function type (which is always scalar, since it corresponds to a static function pointer) given argument types A_i and result type R, with the constraint that each A_i and R must be sized with respect to the variables in Γ, i.e., cannot be dynamic with respect to any of the arguments x_i. Otherwise, it is possible for types to depend on open non-intuitionistic terms, though we are restricted to depending on these terms *denotation* (hence, $\mathsf{d}A_i$ in the rule given). This is in contrast to Krishnaswami et al. [12], which disallows type dependency on open linear terms is disallowed entirely. Note also the use of sym; this implies that function types may depend on instants invariantly or covariantly, but not contravariantly. The given type theory has little support for contravariant dependencies, but this could change in the future with some extensions, hence why they were left in.

The free variable set of a function type is defined to be the expected value

$$\mathsf{fv}(\mathsf{Fn}((x_i : A_i)_i) \to R) = \mathsf{fv}(R) \setminus \{x_i\} \tag{3.130}$$

However, the *constant* variable set definition takes into account the fact that, for a function type to be valid, the result and argument types must be sized. Consequently, we pin any variables the size and alignment of A_i and R depend on to be constants, as follows:

$$\mathsf{cv}(\mathsf{Fn}((x_i : A_i)_i) \to R) = \mathsf{rfv}(\mathsf{sizeof}\ R) \cup \mathsf{rfv}(\mathsf{alignof}\ R) \cup \bigcup_i \mathsf{rfv}(\mathsf{sizeof}\ A_i) \cup \mathsf{rfv}(\mathsf{alignof}\ A_i) \tag{3.131}$$

We give the expected denotation rule

$$\frac{}{(\mathsf{d}(\mathsf{Fn}(x_i : A_i) \to R), \Pi x_1 : \mathsf{d}A_1. \ldots \Pi x_n : \mathsf{d}A_n.\mathsf{d}R[x_i/\mathsf{d}x_i]) \in \delta} \tag{3.132}$$

We may now give the following *introduction rule* for Fn-types:

$$\frac{\Gamma, (x_i \leftarrow: A_i)_i; C, (\mathsf{u}(x_i))_i \vdash r : R[\mathsf{d}x_i/x_i]_i \quad \begin{array}{c} x_i \notin \mathsf{cv}(r : R[\mathsf{d}x_i/x_i]_i) \\ (\Gamma, (x_i \leftarrow: A_i)_i; ! \vdash r\ \mathsf{obsv}(x_j))_j \end{array}}{\Gamma; \mathsf{borrowck}(\{(x_i : A_i, \varnothing)_i\}, C) \vdash \mathsf{fn}((x_i : A_i)_i) \mapsto r : \mathsf{Fn}((x_i : A_i)_i) \to R} \tag{3.133}$$

Note, in particular, the requirement that none of the function's variables x_i appear in the constant variable set of the function's result r; this ensures that no constant values (such as other functions) depend on any of the function's variables. We also require that r observe all the function's arguments; hence, we can assume a function call observes all it's arguments. We define a function's free variable set as expected

$$\mathsf{fv}(\mathsf{fn}((x_i : A_i)_i) \mapsto r) = \mathsf{fv}(r) \setminus \{x_i\} \tag{3.134}$$

On the other hand, since we require functions themselves to be constant, we define a function's constant-variable set to include any free variables in the result (minus the function's arguments)

$$\mathsf{cv}(\mathsf{fn}((x_i : A_i)_i) \mapsto r : F) = \mathsf{cv}(F) \cup (\mathsf{fv}(r) \setminus \{x_i\}) \tag{3.135}$$

We now provide a typing rule for function application, which works as expected

$$\frac{\Gamma; C \vdash f : \mathsf{Fn}(x_i : A_i) \to R \quad (\Gamma; C_i \vdash a_i \leftarrow: A_i([\mathsf{d}a_j/x_j])_{j<i})_i}{\Gamma; C, (C_i)_i, \mathsf{u}(a_i)_i \vdash f\ (a_i)_i : R([\mathsf{d}a_i/x_i])_i} \tag{3.136}$$

Since the function's result must observe all it's arguments by Equation 3.133, we have simple observation rule

$$\frac{\Gamma; ! \vdash f : \mathsf{Fn}(x_i : A_i) \to R \quad \exists j, \Gamma; ! \vdash a_j\ \mathsf{obsv}(x)}{\Gamma; C \vdash f\ (a_i)_i\ \mathsf{obsv(x)}} \tag{3.137}$$

In particular, as a_i observes itself (by Equation 3.39), we have $f\ (a_i)_i\ \mathsf{obsv}(a_i)$. Note, however, that, as for **let**-statements, there is *no* reduction rule implying

$$(\mathsf{fn}(x_i : A_i) \mapsto r)\ (a_i)_i \to r[a_i/x_i]_i \tag{3.138}$$

However, on the *denotation level*, we provide rule

$$\frac{}{(\mathsf{d}(\mathsf{fn}(x_i : A_i) \mapsto r), \lambda x_1. \ldots \lambda x_n.\mathsf{d}r) \in \delta} \tag{3.139}$$

Combined with Equation 3.120, this gives the expected result that

$$(\mathsf{d}(((\mathsf{fn}(x_i : A_i) \mapsto r)\ (a_i)_i)), \mathsf{d}r[\mathsf{d}a_i/\mathsf{d}x_i]_i) \in \delta \tag{3.140}$$

3.5.4 Fixpoints

We now define the notion of a *machine fixpoint* as follows:

Definition 26 (Machine Fixpoint)**.** *A* machine fixpoint definition *is an expression of the form* $\mathsf{Phi}([f_j : F_j = D_j])$ *where each* f_i *is a variable, each* F_i *is an* Fn*-type, and each* D_i *is a term. We treat this as a recursive definition of each function* f_i *in terms of the symbols* $\{f_j\}$*; in particular, we define*

$$\mathsf{dPhi}([f_j : F_j = D_j]) = \mathsf{Fix}(f_j : \mathsf{d}F_j = \mathsf{d}D_j[f_\ell/\mathsf{d}f_\ell]_\ell) \tag{3.141}$$

We say a machine fixpoint definition $\mathcal{F} = \mathsf{Phi}([f_j : F_j = D_j])$ *is* well-formed in context $\Gamma; C$ *if:*

- $\Gamma, (f_j : F_j)_j; C \vdash D_i : F_i$
- $\mathsf{d}\mathcal{F}$ *is well-formed in* $\Gamma; C$*, and in particular passes the termination check.*

For a well-formed machine fixpoint definition $\mathcal{F} = \mathsf{Phi}([f_j : F_j = D_j])$, we provide typing rule

$$\frac{(\Gamma, (f_j : F_j)_j; C_i \vdash D_i : F_i)_i}{\Gamma; (C_i)_i \vdash \mathcal{F} :: f_k : D_k} \tag{3.142}$$

and denotation

$$\frac{}{(\mathsf{d}(\mathcal{F} :: f_i), (\mathsf{d}\mathcal{F}) :: f_i) \in \delta} \tag{3.143}$$

as well as free-variable and constant sets

$$\mathsf{fv}(\mathcal{F} :: f_i) = \bigcup_i \mathsf{fv}(D_j) \setminus \{f_k\}_k, \qquad \mathsf{cv}(\mathcal{F} :: f_i) = \bigcup_j \mathsf{cv}(D_j) \setminus \{f_k\}_k \tag{3.144}$$

We furthermore provide reduction rules

$$\frac{\Gamma, (f_j : F_j)_j; C \vdash D_i \to_R D_i'}{\Gamma; C \vdash \mathsf{Phi}([f_j : F_j = D_j]) :: f_k \to_R \mathsf{Phi}([f_j : F_j = D_j'])} \text{ where } j \neq i \implies D_j' = D_j \tag{3.145}$$

$$\frac{\Gamma, (f_j : F_j)_j; C \vdash F_i \to_R F_i'}{\Gamma; C \vdash \mathsf{Phi}([f_j : F_j = D_j]) :: f_k \to_R \mathsf{Phi}([f_j : F_j' = D_j])} \text{ where } j \neq i \implies F_j' = F_j \tag{3.146}$$

3.6 Data Structures

We may now describe the final component of the `isotope` language, namely, it's data structures. We begin by defining a set of scalar integers of different widths, as well as array and pointer types, with the latter subdivided into *borrowed pointers* (references) and *owned pointers* (boxes). We do not yet provide rules for unboxed owned pointers (i.e., `&mut T`). We then proceed to give rules for casting types between layouts. Finally, we define a framework of machine inductive types as a generalization of Rust's `enum`, with pattern matching on such types a generalization of Rust's `match`.

3.6.1 Integers

We begin by defining a collection of primitive scalar types, $\mathsf{u1}, \mathsf{u8}, \mathsf{u16}, \mathsf{u32}, \mathsf{u64}$ representing 1, 8, 16, 32, and 64-bit integers respectively. For simplicity, we will assume pointers are 64-bits wide; in reality, a separate usize type is provided. We provide typing rules

$$\frac{n \in \{1, 8, 16, 32, 64\}}{\vdash \mathsf{u}n : \mathcal{S}_1} \qquad \frac{\Gamma \vdash m : \mathbb{N}}{\Gamma \vdash m\mathsf{u}n : \mathsf{u}n} \tag{3.147}$$

where $m\mathsf{u}n$ represents an n-bit integer literal, e.g. $74\mathsf{u8}$. We support hexadecimal, octal, and binary notations with the usual definitions. As a simplifying assumption, we define the size and alignment of these types as usual on x86-64 Linux, namely

$$\begin{gathered}
\frac{}{(\mathsf{sizeof}\ \mathsf{u1}, 1) \in \sigma} \qquad \frac{}{(\mathsf{alignof}\ \mathsf{u1}, 1) \in \sigma} \qquad \frac{}{(\mathsf{sizeof}\ \mathsf{u8}, 1) \in \sigma} \qquad \frac{}{(\mathsf{alignof}\ \mathsf{u8}, 1) \in \sigma} \\
\frac{}{(\mathsf{sizeof}\ \mathsf{u16}, 2) \in \sigma} \qquad \frac{}{(\mathsf{alignof}\ \mathsf{u16}, 2) \in \sigma} \qquad \frac{}{(\mathsf{sizeof}\ \mathsf{u32}, 4) \in \sigma} \qquad \frac{}{(\mathsf{alignof}\ \mathsf{u32}, 4) \in \sigma} \\
\frac{}{(\mathsf{sizeof}\ \mathsf{u64}, 8) \in \sigma} \qquad \frac{}{(\mathsf{alignof}\ \mathsf{u64}, 8) \in \sigma}
\end{gathered} \tag{3.148}$$

We introduce denotations

$$\frac{}{(\mathsf{d}(\mathsf{u}n), \mathsf{B}_n) \in \delta} \qquad \frac{}{(\mathsf{d}(m\mathsf{u}n), \mathsf{trunc}_n m) \in \delta} \tag{3.149}$$

Similarly, we introduce the usual collection of basic arithmetic and logical operations, e.g., n-bit integer addition $\mathsf{iadd}_n : \mathsf{Fn}(\mathsf{u}n, \mathsf{u}n) \to \mathsf{u}n$, with the expected denotations, e.g. $\mathsf{d}(\mathsf{iadd}_n) = \mathsf{add}_n$. We also provide cast operations $\mathsf{zext}_{n,m}$ and $\mathsf{sext}_{n,m}$ from $\mathsf{u}n$ to $\mathsf{u}m$ which provide zero-extended and sign-extended integer casts respectively.

3.6.2 References

We now introduce our first actual ownership type: the *(borrowed) reference type* $\&‘a\ T$, representing a value of type T which is valid to *read* or *observe* (but not use!) up to time $‘a$, with formation rule

$$\frac{\Gamma; C \vdash T : \mathcal{M}_i‘a\ \ell}{\Gamma; C \vdash \&‘a\ T : \mathcal{S}_i‘a} \tag{3.150}$$

This has introduction rule

$$\frac{\Gamma; C \vdash x \leftarrow: T}{\Gamma; C \vdash \&x : \&\mathsf{c}x\ T} \tag{3.151}$$

and elimination rule

$$\frac{\Gamma; C \vdash r \leftarrow: \&‘a\ T \quad \Gamma, x \leftarrow: T; C' \vdash e : B}{\Gamma; \mathsf{bind}_\Gamma(\{(x : T, (C, \mathsf{r}r, \mathsf{c}x \preceq ‘a), 1)\}, C') \vdash \mathsf{let}\ x = {}^*r\ \mathsf{in}\ e : B} \tag{3.152}$$

In short, this lets us use a reference $r : \&‘a\ T$ as a value of type T so long as such uses are:

- Before $‘a$
- After r is created
- Do not consume T, *unless* T is scalar (i.e. $T\ \mathsf{type}(\mathsf{scalar})$).

In particular, then, we can use a reference to a scalar almost like a value of that scalar, while references to linear types can really only be used to make other references (e.g., by *projection operations*, which take $\&T \to \&U$; if U is scalar, we can then use that). References, being represented by simple pointers, always have the same alignment and size, namely

$$\frac{}{(\mathsf{sizeof}\ \&‘aT, \mathsf{sizeof}\ \mathsf{u64}) \in \sigma} \qquad \frac{}{(\mathsf{alignof}\ \&‘aT, \mathsf{alignof}\ \mathsf{u64}) \in \sigma} \tag{3.153}$$

Whether a value is referenced or not is transparent to it's denotation, i.e.

$$\frac{}{(\mathsf{d}(\&`aT), \mathsf{d}T) \in \delta} \qquad \frac{}{(\mathsf{d}(\&t), \mathsf{d}t) \in \delta} \tag{3.154}$$

3.6.3 Boxing

We now introduce a type of *owned references*, or *boxes*, with the following formation rules

$$\frac{\Gamma; C \vdash T : \mathcal{M}_i`a\ \ell}{\Gamma; C \vdash \mathsf{Box}\ T : \mathcal{L}_i`a} \tag{3.155}$$

This is our first bona-fide linear type: boxes *must* be unboxed or forgotten in a valid `isotope` program. We can create a box by "*boxing*" a value, as follows

$$\frac{\Gamma; C \vdash t \leftarrow: T}{\Gamma; C, u(t) \vdash \mathsf{box}\ t : \mathsf{Box}\ T} \qquad \frac{\Gamma; ! \vdash t\ \mathsf{obsv}(x)}{\Gamma; C \vdash \mathsf{box}\ t\ \mathsf{obsv}(x)} \tag{3.156}$$

and destroy a box by "*unboxing*" it

$$\frac{\Gamma; C \vdash t \leftarrow: \mathsf{Box}\ T}{\Gamma; C, u(t) \vdash \mathsf{unbox}\ t : T} \qquad \frac{\Gamma; ! \vdash t\ \mathsf{obsv}(x)}{\Gamma; C \vdash \mathsf{unbox}\ t\ \mathsf{obsv}(x)} \tag{3.157}$$

or by leaking the underlying memory and *forgetting* it as follows:

$$\frac{\Gamma; C \vdash t \leftarrow: \mathsf{Box}\ T \quad \Gamma; C \vdash T : \mathcal{M}_i`a\ \ell}{\Gamma; C, u(t) \vdash \mathsf{forget}\ t : \&`a\ T} \qquad \frac{}{\Gamma; C \vdash \mathsf{forget}\ t\ \mathsf{obsv}(t)} \tag{3.158}$$

Note that, unlike unbox, forget does *not* observe anything it's argument observes (though it observes it's argument), as there is no guarantee that it will ever be used. Sometimes, we want to access the inside of a box without destroying it; to do so, we can get a pointer to the inside of a box via the deref operator as follows:

$$\frac{\Gamma; C \vdash r : \&`a\ \mathsf{Box}\ T}{\Gamma; C \vdash \mathsf{deref}\ r : \&`a\ T} \tag{3.159}$$

As with references, all boxes fit in a pointer, i.e.,

$$\frac{}{(\mathsf{sizeof}\ \ \mathsf{Box}\ T, \mathsf{sizeof}\ \mathsf{u64}) \in \sigma} \qquad \frac{}{(\mathsf{alignof}\ \mathsf{Box}\ T, \mathsf{alignof}\ \mathsf{u64}) \in \sigma} \tag{3.160}$$

and boxes are denotationally transparent, i.e.,

$$\frac{}{(\mathsf{d}(\mathsf{Box}\ T), \mathsf{d}T) \in \delta} \tag{3.161}$$

$$\frac{}{(\mathsf{d}(\mathsf{box}\ t), \mathsf{d}t) \in \delta} \qquad \frac{}{(\mathsf{d}(\mathsf{unbox}\ t), \mathsf{d}t) \in \delta} \qquad \frac{}{(\mathsf{d}(\mathsf{forget}\ t), \mathsf{d}t) \in \delta} \qquad \frac{}{(\mathsf{d}(\mathsf{deref}\ t), \mathsf{d}t) \in \delta} \tag{3.162}$$

3.6.4 Inductive Data

We can now define the `isotope` analog to Rust's `enum` types, Data-declarations, as follows

Definition 27 (Data Declaration)**.** *A* data declaration *is an expression of the form* $\mathcal{D} = \mathsf{Data}([I_j : A_j] := [c_{jk} : C_{jk}])$ *where each* A_j *is a* machine arity of sort s_j *and each* C_{jk} *is a* machine constructor for I_j*. We define the* denotation *of a data declaration as follows:*

$$\mathsf{dData}([I_j : A_j] := [c_{jk} : C_{jk}]) \equiv \mathsf{Ind}([I_j : \mathsf{d}A_j] := [c_{jk} : \mathsf{d}C_{jk}/[I_\ell/\mathsf{d}I_\ell]_\ell]) \tag{3.163}$$

We say a data declaration $\mathcal{D} \equiv \mathsf{Data}([I_j : A_j] := [c_{jk} : C_{jk}])$ *is* well-formed *in* $\Gamma; C$*, written* $\Gamma; C \vdash \mathcal{D}\ \mathsf{ok}$*, if*

- *A_j and C_{jk} are well-formed in $\Gamma; C$*

- *For each field F_{ijk} of each C_{jk}, we have* [6]

  $$\Gamma; C \vdash F_{ijk} : s_j \tag{3.164}$$

- *For each field F_{ijk} of each C_{jk}, we have*

  $$\Gamma; C \vdash \mathsf{sized}_\Gamma(F_{ijk}) \tag{3.165}$$

  In particular, the size of any field F_{ijk} must not depend on any of the defined inductive types I_j. Note this does not forbid recursive types in general, as, e.g., $\mathsf{Box}\ I_j$ *would be permitted since it's size and alignment are independent of I_j.*

- $\mathsf{d}\mathcal{D}$ *is well-formed (as an inductive definition)* [7]

We provide typing rules

$$\frac{\Gamma; C \vdash \mathcal{D}\ \mathsf{ok}}{\Gamma; C \vdash \mathcal{D} :: I_j : A_j[\mathcal{D} :: I_k / I_k]_k} \qquad \frac{\Gamma; C \vdash \mathcal{D}\ \mathsf{ok}}{\Gamma; C \vdash \mathcal{D} :: c_{jk} : C_{jk}[\mathcal{D} :: I_k / I_k]_k} \tag{3.166}$$

and denotations

$$\frac{}{(\mathsf{d}(\mathcal{D} :: I_j), (\mathsf{d}\mathcal{D}) :: I_j) \in \delta} \qquad \frac{}{(\mathsf{d}(\mathcal{D} :: c_{jk}), (\mathsf{d}\mathcal{D}) :: c_{jk}) \in \delta} \tag{3.167}$$

---

[6] This condition *implies* the condition from Definition 18 for inductive types, namely that $\Gamma; C \vdash C_{jk} : s_j$, but is strictly stronger in the following sense: the type constructor Fn always discards linearity and lifetime information, i.e., produces a type of type $\mathcal{S}_i$. Hence, requiring the fields to be of type s_j is a stronger requirement.

[7] This condition in essence delegates checking types of constructor for strict positivity to the purely inductive fragment of `isotope`.

In general, we will often omit assumptions of the form $\mathcal{D}$ ok for clarity, as for inductive types.

A *machine arity* of sort s is simply the equivalent of an arity of sort s for machine universe sorts $s = \mathcal{M}_i`a\ \ell$; formally:

Definition 28 (Machine Arity)**.** *A type T is a* machine arity of sort $\mathcal{M}_i`a\ \ell$ *if*

- $T = \mathcal{M}_i`a\ \ell$
- $T = \Pi x : U.V$ *where U is an arity of sort $\mathcal{M}_i`a\ \ell$*

A type T is an arity *if there exists a universe $\mathcal{U}_i$ such that T is an arity of sort $\mathcal{U}_i$.*

In particular, $\mathbb{N} \to \mathcal{S}_3$ is a machine arity, but neither of $\mathbb{N} \to \mathcal{U}_5$ (since the target is not a *machine* universe) or $\mathcal{S}_3 \to \mathbb{N}$ (since the target is not a universe at all) are arities. Indeed, we have

Claim 7. *If A is a machine arity of sort $\mathcal{M}_i`a\ \ell$, then $\mathsf{d}A$ is an arity of sort $\mathcal{U}_i$.*

Proof. Let A be a machine arity of sort $\mathcal{M}_i`a\ \ell$. We proceed by induction: assume that for all subterms $T$ of $A$, if $A$ is a machine arity of sort $\mathcal{M}_i`a\ \ell$, then $\mathsf{d}V$ is an arity of sort $\mathcal{U}_i$. By definition, we have either that

- $A = \mathcal{M}_i`a\ \ell$, in which case $\mathsf{d}A = \mathcal{U}_i$ is trivially an arity of sort $\mathcal{U}_i$.
- $A = \Pi x : U.V$. By induction, $\mathsf{d}V$ is an arity of sort $\mathcal{U}_i$, and hence by definition $\mathsf{d}A = \Pi x : \mathsf{d}U.\mathsf{d}V$ is an arity of sort $\mathcal{U}_i$ since $\mathsf{d}U : \mathcal{U}_i$

□

Similarly, a *machine constructor* for I_j is the machine universe equivalent of a type of constructor for I_j.

Definition 29 (Machine Constructor)**.** *A type T is a* machine constructor for I*, where I is a variable, if*

- $T \equiv I\ t_1\ ...\ t_n$*. In this case, T is said to have no fields.*
- $T \equiv \mathsf{Fn}(x_1 : F_1, ..., x_n : F_n) \to I\ t_1\ ...\ t_n$*. In this case the types $F_1, ..., F_n$ are called the* fields *of T.*

In particular, $\mathsf{Fn}(x : \mathsf{u32}, y : \mathsf{u32}) \to I\ a\ b$ is a machine type of constructor for I, but neither of $\Pi n : \mathbb{N}.\mathsf{Fn}(a : [\mathsf{u32}; n]) \to I\ n$ (since it is not an Fn-type) or $\mathsf{Fn}(x : \mathsf{u32}, y : \mathsf{u32}) \to \mathsf{u32}$ (since the target is not I) are types of constructor for I. Analogously to arities, we claim

Claim 8. *If A is a machine constructor for I_j, then $\mathsf{d}A[I/\mathsf{d}I]$ is a type of constructor for I.*

Proof. Let A be a machine constructor for I_j. Then by definition,

- $A \equiv I\ t_1\ ...\ t_n$, in which case

  $$\begin{aligned}\mathsf{d}A[I/\mathsf{d}I] &= (\mathsf{d}I\ t_1\ ...\ t_n)[I/\mathsf{d}I] = ((\mathsf{d}I)\ (\mathsf{d}t_1)\ ...\ (\mathsf{d}t_n))[I/\mathsf{d}I] \\ &= (\mathsf{d}I)[I/\mathsf{d}I]\ (\mathsf{d}t_1)[I/\mathsf{d}I]\ ...\ (\mathsf{d}t_n)[I/\mathsf{d}I] = I\ (\mathsf{d}t_1)[I/\mathsf{d}I]\ ...\ (\mathsf{d}t_n)[I/\mathsf{d}I] \quad (3.168)\end{aligned}$$

  which is by definition a type of constructor.

- $A \equiv \mathsf{Fn}(x_1 : F_1, ..., x_n : F_n) \to I\ t_1\ ...\ t_n$, in which case

  $$\begin{aligned}\mathsf{d}A[I/\mathsf{dI}] &= \mathsf{d}(\mathsf{Fn}(x_1 : F_1, ..., x_n : F_n) \to I\ t_1\ ...\ t_n)[I/\mathsf{dI}] \\ &= (\Pi x_1 : \mathsf{d}F_1. \ ... \ .\Pi x_n : \mathsf{d}F_n.\mathsf{d}I\ (\mathsf{d}t_1)\ ...\ (\mathsf{d}t_n))[I/\mathsf{d}I] \\ &= \Pi x_1 : \mathsf{d}F_1[I/\mathsf{d}I]. \ ... \ .\Pi x_n : \mathsf{d}F_n[I/\mathsf{d}I].I\ (\mathsf{d}t_1)[I/\mathsf{d}I]\ ...\ (\mathsf{d}t_n)[I/\mathsf{d}I] \quad (3.169)\end{aligned}$$

which is a type of constructor since $I\ (\mathsf{d}t_1)[I/\mathsf{d}I]\ ...\ (\mathsf{d}t_n)[I/\mathsf{d}I]$ is.

$\square$

We will lay out `isotope` data-declarations in memory like `Rust` enums: a tag indicating which constructor is in use, followed by the fields of that constructor (in order). In particular, therefore, we define where F_{ijk} are the fields of C_{jk},

$$\frac{}{(\mathsf{sizeof}\ (\mathsf{Data}([I_j : A_j] := [c_{jk} : C_{jk}]) :: I_\ell), \mathsf{enumsize}([[F_{ijk}]_i]_k)) \in \delta} \tag{3.170}$$

$$\frac{}{(\mathsf{alignof}\ (\mathcal{D} :: I_\ell), A) \in \delta} \tag{3.171}$$

where

$$A = \max\{\mathsf{tagsize}, (\mathsf{alignof} F_{i\ell k})_{ik}\}, \qquad \mathsf{tagsize} = \lceil \log_2 N/8 \rceil \tag{3.172}$$

where N is the number of constructors $c_{j1}, ..., c_{jN}$ for I_j and $\mathsf{enumsize}([[F_{ijk}]_i]_k))$ is the size required to store I_j as an `enum` in the C layout, e.g.

$$\mathsf{enumsize}([[F_{ijk}]_i]_k)) = \max_k(\mathsf{structsize}([(\mathsf{tagsize}, \mathsf{tagsize}), (\mathsf{sizeof}\ F_{ijk}, \mathsf{alignof}\ F_{ijk})_i], A)) \tag{3.173}$$

where $\mathsf{structsize}([(s_i, a_i)]_i, a)$ is the size of a struct with ordered fields having size/alignment (s_i, a_i) with alignment a in the C ABI *with ZSTs* (i.e., allowing types of size 0, as in Rust, but unlike, e.g., C++). We omit this definition for simplicity, as it has no bearing on the theory.

3.6.5 Pattern Matching

We may now introduce the match-statement, which serves as the machine-type analog of the case-statement, with the following typing rule

$$\frac{\begin{array}{c}\Gamma; C \vdash i \leftarrow: \mathcal{I} :: I_j \\ \Gamma; ! \vdash F : \mathsf{d}\mathcal{I} :: I_j \to \mathcal{M}_i`a\ \ell \quad (\Gamma, (p_n : P_{kn})_n; C_k \vdash \beta_k : F(\mathsf{d}(c_k\ (p_n)_n)))_k\end{array}}{\Gamma; C, \mathsf{p2u}\left(i, \bigcup_k (\mathsf{borrowck}_\Gamma(\{(p_n, \mathsf{p}(i), 0)_n\}, C_k))_k\right) \vdash \mathcal{I} :: \mathsf{match}_{I_j}\ i\ \{(c_k(p_n)_n \mapsto \beta_k,)_k\} : F(i)} \tag{3.174}$$

where $\mathsf{p2u}(i, C)$ is defined as follows:

- Remove all occurences of $\mathsf{p}(i)$ from C, and **if** any such occurences were found, add *one* occurence of $\mathsf{u}(i)$.

Breaking this down, it states that

- Given a machine type family $F : \mathsf{d}\mathcal{I} :: I_j \to \mathcal{M}_i`a\ \ell$ parametrized by the *denotation* of $\mathcal{I} :: I_j$;
- Given a variable $i \leftarrow: \mathcal{I} :: I_j$ under constraints C
- Given branches $c_k(p_n)_n \mapsto \beta_k,$, where each β_k is of type $F(\mathsf{d}(c_k\ (p_n)_n)))$ under constraints C_k; if C'_k is the result of borrow checking C_k as if it was a function (but, *unlike* static functions, branches do *not* make their dependencies constant) with parameters p_n.
- We may type the match statement $\mathcal{I} :: \mathsf{match}_{I_j}\ i\ \{(c_k(p_n)_n \mapsto \beta_k,)_k\}$ as $F(i)$ under constraints $C, \mathsf{p2u}\,(i, \bigcup_k (C'_k))$; i.e., marking i as fully used if any branch partially uses i, while otherwise taking the union of the usages, dependencies, and constraints of each individual branch.

We introduce an observation rule

$$\frac{(\Gamma, (p_n : P_{kn})_n; ! \vdash \beta_k\ \mathsf{obsv}(x))_k}{\Gamma; C \vdash \mathcal{I} :: \mathsf{match}_{I_j}\ i\ \{(c_k(p_n)_n \mapsto \beta_k,)_k\}\ \mathsf{obsv}(x)} \tag{3.175}$$

which, in essence, states that “a match statement observes something if all it’s branches do so,” and

$$\frac{((\Gamma, (p_n : P_{kn})_n; ! \vdash \beta_k\ \mathsf{obsv}(p_\ell))_k)_\ell \quad \Gamma; ! \vdash i\ \mathsf{obsv}(x)}{\Gamma; C \vdash \mathcal{I} :: \mathsf{match}_{I_j}\ i\ \{(c_k(p_n)_n \mapsto \beta_k,)_k\}\ \mathsf{obsv}(x)} \tag{3.176}$$

i.e., “a match statement observes something if it’s argument i does so and all it’s branches observe all components of i visible to that branch, and therefore, implictly, observe i.”

We give match statements denotations in terms of case statements, as expected:

$$\frac{}{\left(\mathsf{d}\left(\mathcal{I} :: \mathsf{match}_{I_j}\ F\ i\ \{(c_k(p_n)_n \mapsto \beta_k,)_k\} : F(i)\right), (\mathsf{d}\mathcal{I}) :: \mathsf{case}_{I_j}\{(c_k(\mathsf{d}p_n)_n) \mapsto \beta_k,)_k\}\right) \in \delta} \tag{3.177}$$

3.6.6 Casting

Sometimes, we wish to control the size and alignment of types without resorting to boxing (which has a high runtime overhead, as it requires memory allocations) or references (which require introducing the complexity overhead of instant parameters, and also potentially the runtime overhead of “pointer chasing”). To remedy this problem, we introduce *casting.* We begin by giving a type formation rule

$$\frac{\Gamma; ! \vdash s : \mathbb{N} \quad \Gamma; ! \vdash a : \mathbb{N} \quad \Gamma; C \vdash T : \mathcal{M}_i\text{‘}a\ \ell}{\Gamma; C \vdash \mathsf{cast}\ T\ s\ a : \mathcal{M}_i\text{‘}a\ \ell} \tag{3.178}$$

This type represents the "values of T which can fit in an allocation with size s and alignment a." This is represented by reduction rules

$$\frac{}{(\mathsf{sizeof}(\mathsf{cast}\ T\ s\ a), s) \in \sigma} \qquad \frac{}{(\mathsf{alignof}(\mathsf{cast}\ T\ s\ a), a) \in \sigma} \tag{3.179}$$

In particular, if $s = \mathsf{sizeof}\ T$ and $a = \mathsf{alignof}\ T$, then this type can be identified with T; we represent this by introducing reduction rule

$$\frac{}{(\mathsf{cast}\ T\ (\mathsf{sizeof}\ T)\ (\mathsf{alignof}\ T), T) \in \sigma} \tag{3.180}$$

Similarly, casts "overwrite" each other as follows:

$$\frac{}{(\mathsf{cast}\ (\mathsf{cast}\ T\ s\ a)\ s'\ a'\ T), \mathsf{cast}\ T\ s'\ a') \in \sigma} \tag{3.181}$$

To construct terms of cast type, we introduce the as expression, with the following syntax:

$$\frac{\begin{array}{c} \Gamma; ! \vdash s : \mathbb{N} \quad \Gamma; ! \vdash p : \mathsf{Le}\ (\mathsf{sizeof}\ T)\ s \\ \Gamma; C \vdash x : A \quad \Gamma; ! \vdash a : \mathbb{N} \quad \Gamma; ! \vdash q : \mathsf{Le}\ (\mathsf{alignof}\ T)\ a \end{array}}{\Gamma; C \vdash x\ \mathsf{as}\ s\ a\ p\ q : \mathsf{cast}\ T\ (\mathsf{sizeof}\ T)\ (\mathsf{alignof}\ T)} \tag{3.182}$$

We will often omit the proof terms p, q when sufficiently trivial (e.g., casting between constant sizes). We also extend the deref operator to cast references:

$$\frac{\Gamma; C \vdash c : \&`a\ \mathsf{cast}\ T\ s\ a}{\Gamma; C \vdash \mathsf{deref}\ c : \&`a\ T} \tag{3.183}$$

As with boxing, casting is transparent to denotation:

$$\frac{}{(\mathsf{d}(\mathsf{cast}\ T\ s\ a), \mathsf{d}T) \in \delta} \qquad \frac{}{(\mathsf{d}(x\ \mathsf{as}\ s\ a\ p\ q), \mathsf{d}x) \in \delta\ \ (\mathsf{d}(x\ \mathsf{as}\ s\ a\ p\ q), \mathsf{d}x) \in \delta} \tag{3.184}$$

3.6.7 Arrays

Given a machine type $A : \mathcal{M}_i\text{‘}a\ell$ and an integer n, we introduce *arrays* of length n $[A; n]$ with the following formation rule

$$\frac{\Gamma; C \vdash A : \mathcal{M}_i\text{‘}a\ell \quad \Gamma; ! \vdash n : \mathbb{N}}{\Gamma; C \vdash [A; n] : \mathcal{M}_i\text{‘}a\ell} \tag{3.185}$$

and introduction rule

$$\frac{(\Gamma; C_i \vdash a_i : A)_{i=1..n}}{\Gamma; (C_i)_i \vdash [(a_i)_i] : [A; n]} \tag{3.186}$$

Arrays are laid out as in C, consequently, we have

$$\frac{}{(\mathsf{sizeof}\ [T; n], n \cdot \mathsf{sizeof}\ T) \in \sigma} \qquad \frac{}{(\mathsf{alignof}\ [T; n], \mathsf{alignof}\ T) \in \sigma} \tag{3.187}$$

We will treat an array as a function from $\{0, ..., n-1\}$ to A, consequently, we introduce denotation

$$\frac{}{(\mathsf{d}[A; n], \mathsf{F}_n \to A) \in \delta} \qquad \frac{}{(\mathsf{d}[(a_i)_i], \mathsf{switch}_A\ (d\ a_i)_i) \in \delta} \tag{3.188}$$

We introduce an indexing operation:

$$\frac{\Gamma; C_a \vdash a : \&[A; n] \quad \Gamma; C_i \vdash i : \mathsf{u64} \quad \Gamma; ! \vdash p : \mathsf{Lt}\ (\mathsf{d}i)\ n}{\Gamma; C_a, C_i \vdash \mathsf{index}_p\ a\ i : \&A} \tag{3.189}$$

We do not really support arrays of linear types at the moment, as there's no way to observe them beyond using them completely. Similarly, even affine array support is somewhat lacklustre, as using one component uses the entire array. These are actually both drawbacks of the Rust type system itself, which uses `unsafe` library functions to fill in the gaps. I began work on an extension of the project to rectify this, but did not complete it on time.

Chapter 4

Compilation

In Chapter 3 we described the `isotope` language at a high level and gave a preliminary specification for it's type system and features. In this chapter, our goal is to give an algorithm for compiling *realizable* `isotope` machine terms, as defined in Section 3.5, to a bare-metal program (i.e., one requiring only minimal runtime support). Rather than directly output assembly or an intermediate representation, as discussed in Section 2.4, we will instead describe an algorithm targeting the C programming language for simplicity of exposition to avoid irrelevant implementation details.

In Section 4.1, we give a description of the process of compiling a closed `isotope` function; we begin by recalling the definition of *temporal dependence graphs* from Chapter 3 and how we can construct one from the constraint set C of a term $\Gamma; C \vdash a : A$. We then show how to define and actually compile a function at a high level by traversing it's graph and compiling it's subterms.

In Section 4.2, we then proceed to describe in detail the code that "node lowering" step described previously in the algorithm from 4.1 generates for each kind of machine term, along with some representation details. This completes the algorithm from 4.1, giving a complete pipeline from fully annotated `isotope` term to pseudo-C program; with some additional theoretically irrelevant implementation details handled, this should allow implementing a compiler for `isotope` terms.

4.1 Function Compilation

The goal of this Section is to define how to construct a fully-annotated `isotope` function $f \equiv \mathsf{fn}((x_i : A_i)_i) \to r$, assuming that, for every (non-instant) free variable $v \in \mathsf{fv}(r)$, we've compiled v to a constant C value c_v into a cache C. In particular, assume we are given derivation

$$\frac{\Gamma, (x_i \leftarrow: A_i)_i; C, (\mathsf{u}(x_i))_i \vdash r : R[\mathsf{d}x_i/x_i]_i \quad \begin{array}{c} x_i \notin \mathsf{cv}(r : R[\mathsf{d}x_i/x_i]_i) \\ (\Gamma, (x_i \leftarrow: A_i)_i; ! \vdash r\ \mathsf{obsv}(x_j))_j \end{array}}{\Gamma; \mathsf{borrowck}(\{(x_i : A_i, \varnothing)_i\}, C) \vdash \mathsf{fn}((x_i : A_i)_i) \mapsto r : \mathsf{Fn}((x_i : A_i)_i) \to R} \tag{4.1}$$

with $\mathsf{borrowck}(\{(x_i : A_i, \varnothing, 0)_i\}, C) \neq !$ We compute the *temporal dependence graph* of f to be given by

$$\hat{I} = \mathsf{maxconstrain}(\Gamma, \mathsf{bind}(\{(x_i, \varnothing, 0)_i\}, C)) \tag{4.2}$$

Since $\mathsf{borrowck} \neq !$, we note that $\hat{I}$ is consistent, and hence in particular never identifies two opaque beginnings $`bx$ (where an opaque beginning is the result of applying). We may hence define a strict partial order $\preceq$ on variables where

$$x \preceq_f y \iff (`\mathsf{b}x, `\mathsf{b}y) \in \hat{I}^+ \tag{4.3}$$

Note that we assume variable names are unique throughout the definition of r, α-converting as necessary to achieve this. Hence, to compile a function, we proceed as follows, given name N

1. Generate a C function with prototype

   $$\mathtt{R}\ N\ (\mathtt{x}_1 : \mathtt{A}_1, ..., \mathtt{x}_n : \mathtt{A}_n) \tag{4.4}$$

   where, given a sized machine type T, `T` is an array of bytes with size $\mathsf{sizeof}\ T$ and alignment $\mathsf{alignof}\ T$. We assume for simplicity that types are just automatically cast to each other if of compatible size and alignments, viewing them as "bags of bytes" rather than structured values.

2. Compile $r :: R[\mathsf{d}x_i/x_i]_i$ as `r` in f with cache $C \cup \{x_i \mapsto \mathtt{x}_i\}_i$

3. Generate the statement `return r;`

4. Close the function scope; the generated function N is the desired result.

In the next section, we define how to compile a fully annotated term r as `r` in f. Note that, throughout this algorithm, we rename variables as appropriate to avoid collisions.

4.2 Term Compilation

In this section, we define term compilation recursively for each kind of machine term. If term compilation is not defined for a particular kind of term, then compilation fails and we say that term is *unrealizable* (at least in the current implementation!)

4.2.1 Variables

To compile a variable $x : A$ as `r` in f with cache C, **if** $x \mapsto c_x \in C$ for some c_x, generate the statement

$$\texttt{A r = } c_x\texttt{;} \tag{4.5}$$

else fail.

4.2.2 Let Statements

To compile a let-statement let $x = a : A$ in $e : B$ as `r` in f with cache C,

1. Compile a as `x` in f with cache C.
2. Compile e as `r` in f with cache $C \cup \{x \mapsto \texttt{x}\}$.

4.2.3 Primitives

To compile a literal $\ell : L$ as `r`, generate the statement

$$\texttt{L r = } \ell\texttt{;} \tag{4.6}$$

(e.g., for 64`u8` : `u8`, generate `char r = 64;`). Similarly, to compile an arithmetic operation, e.g. add_{64}, to `r`, just define a nested function, e.g.

$$\texttt{uint64\_t r(s uint64\_t, t uint64\_t) \{ return s + t; \}} \tag{4.7}$$

4.2.4 Function Calls and Fixpoints

To compile a function call $f'\ t_1\ ...\ t_n : A$ to `r` in f with cache C,

1. Compile f' to `f` in f with cache C, and compile each t_i to $\mathtt{t}_i$ in f with cache C, such that the order they are compiled in respects $\preceq_f$.
2. Generate `A r = f(t_1,...,t_n)`

To compile a fixpoint $\mathsf{Fix}([f_j : F_j = D_j]) :: f_j$;

1. For each member of the fixpoint f_k, if already defined, let it be the value `f_k` (renaming as necessary to avoid collisions, e.g. via hash consing), otherwise
   (a) For every value of the fixpoint which does not have one yet, generate a prototype

   $$\mathtt{R}_k\ \mathtt{f}_k\ ((\mathtt{A}_{ik})_i) \tag{4.8}$$

   for $F_k = \mathsf{Fn}((A_{ik})_i) \to R_k$

   (b) Generate the function D_k with cache $C \cup \{f_k \mapsto \mathtt{f}_k\}$
2. Generate statement `void* r = f_k;`

4.2.5 References

To compile a reference $\&x$ as `r` in f with cache C,

1. Compile x as `x` in f with cache C
2. Generate `void* r = &x;`

4.2.6 Boxes

To compile a box $\mathsf{box}\,x : \mathsf{Box}\,T$ as `r` in f with cache C,

1. Compile x as `x` in f with cache C
2. Generate `void* r = malloc(sizeof(T));`
3. Generate `*r = x;`

Similarly, to compile an unboxing $\mathsf{unbox}\, x : T$ as `r` in f with cache C,

1. Compile x as `x` in f with cache C
2. Generate `T r = *(T*)x;`
3. Generate `free(x);`

whereas, to compile a forget-operation $\mathsf{forget}\, x$ as `r` in f with cache C, simply compile x, as forgetting is a no-op.

4.2.7 Casting

To compile a cast x add $s\ a\ p\ q$ as `r` in f with cache C, simply compile x, as casting is implicit.

4.2.8 Constructors and Pattern Matching

Given a data declaration $\mathcal{D} = \mathsf{Data}([I_j : A_j] := [c_{jk} : C_{jk}])$, define it's *associated struct* for I_j to be given by

$$\texttt{struct}\ \texttt{I}_j\ \{\texttt{tag} : \texttt{Tag}_{N_j},\ \texttt{union}\ \{(\texttt{struct}\ \{(\mathsf{a}_i : \mathsf{P}_{ijk})_i\})_k \mathsf{c}_{jk};\};\} \tag{4.9}$$

where A_{ijk} are the parameters of C_{jk}, N_j is the number of constructors c_{jk} for I_j and $\texttt{Tag}_N$ is an integer type holding values up to at least N. We may hence compile a constructor c_{jk}

to `r` by defining anonymous function

$$\mathtt{I}_j\,\mathtt{r((a\_i:\ \ P\_ijk)\_i)}\{\mathtt{return}\{\mathtt{tag} = k, \mathsf{c}_{jk} = \{(\mathsf{a}_i = \mathsf{a}_i)_i\}\};\} \tag{4.10}$$

We may then define a match-statement $\mathcal{D} :: \mathsf{match}_{I_j}\ i\ \{(c_{jk} a_1, ..., a_{n_j} \mapsto \beta_k)_k\}$ as `r` in f with cache C as follows, where R is the result type of the match statement:

1. Generate the statement `R r;`
2. Compile the term i as `i` in f with cache C
3. Begin to generate a `switch`-statement as follows:

$$\mathtt{switch}\ (\mathtt{i.tag})\{ \tag{4.11}$$

4. **For** each k,
    (a) Generate a branch `case` k :.
    (b) Compile the term β_k in f with cache $C \cup \{a_i \mapsto \mathsf{i.c}_{jk}.\mathsf{a}_i\}$ as `b`
    (c) Generate the statements `r = b; break;`
5. End the `switch` statement.

Chapter 5

Implementation

In this chapter, we give an overview of the two partial implementations of `isotope` we developed over the course of this project, giving a brief description of the basic architecture, design choices, and achievements of both. In Section 5.1, we give a brief overview of both implementations. Finally, in Section 5.2, we give a more detailed description of some of the features and design decisions of each implementation, as well as the contrasts between them and the evolution from the old to the new design.

5.1 `isotope` Implementations

We provide a partial implementation of a type checker and interpreter for a variation of the `isotope` language, as well as an implementation of a fragment of an earlier version of the type theory before major changes were made midway in the project, totaling 17,000 lines and 8,000 lines of Rust source respectively. The current implementation supports the majority of the Calculus of Constructions, though support for inductive types remains unfinished; however, only the older codebase has a working implementation of the borrowck algorithm.

As the type system described in this thesis is undecidable, we have had to make significant adaptations in the theory so as to make it implementable efficiently in code. In particular, the implementation and it's relationship to the system described in this paper are *unverified*, since the rules themselves are still a work in progress (with many extracted by simplifying the rules encoded in the `isotope` source).

Both our implementations of `isotope` are primarily designed to be API-oriented; that is, we define an API surface for programmatically constructing and reasoning about both fully-annotated and partially-annotated terms. Both versions of the API supported type checking (in the fragments of the language each supported), (basic) inference and reduction, as well as pattern matching on term ASTs (represented in memory somewhat more abstractly as graphs). Both implementations also included documentation (with every function receiving at least a line of description) generated using the Rust package manager's, `cargo`, builtin documentation functionality. We also provide a suite of numerous unit tests, integration tests, and doctests, which can be run using `cargo`'s inbuilt testing functionality.

We also provide a parser (written using the `nom` parser combinator library [6]) and read-evaluate-print-loop (REPL) program for each implementation, which uses the implementation's API as a library. This serves as an integration test of the API, an example for using advanced API functionality such as type inference (API consumers should usually be building fully annotated-terms, since this is more efficient), and as a debugging utility. The current version of `isotope` also provides a prettyprinting utility, written using a Rust implementation, `pretty` [21], of Wadler-style prettyprinting [19]. Again, however, this is mostly intended as a debugging tool, and therefore the syntax is not very pretty or concise.

Since the first implementation of `isotope` was flawed, due to an over-complicated implementation of constraint sets and due to the fact that it used variable names in lieu of de Bruijn indexes; hence, we eventually decided to start fresh with a second implementation, which unfortunately we were not able to complete in time. However, as `isotope` was specifically intended as an exploration of the design space, I believe this outcome to be at least somewhat successful. Hence, we will attempt to go over the lessons learned in drafting both designs, as well as some other approaches we considered during the design process in sections 5.2.

5.2 Features and Design

In this section, we give a brief overview of the features and design of each implementation of `isotope`, while discussing lessons learned along the way. Even though neither implementation is complete, both have contributed immensely to the abstract algorithms described in Chapters 3 and 4, and we believe the new implementation is only about a month or two from completion. We will attempt to describe both the old constraint set design and the incomplete new design in this section.

5.2.1 Equality

One of the most challenging features to implement in either API was equality checking between terms; while every well-typed term in the CoIC (and hence, hopefully, `isotope`) has a normal form [17, 2], simply normalizing every term and checking them for syntactic equality is quite inefficient, especially if normal forms had to be recomputed. The old version of `isotope` did basically this, but stored a lazily initialized, reference-counted pointer to the normal form of a term along with every term; hence, normalization would only have to occur once per term.

This still didn't work very well, however, as the old version of `isotope` also used variable names rather than de-Bruijn indexes, which meant that quotienting terms under α-conversion remained challenging, often requiring a full substitution for every equality check.

The second version of `isotope` rectified this by introducing the notion of an *equality context* Ξ, which, in essence, exposed a method to check whether two terms were definitely equal, definitely disequal, or potentially either, *given a notion of equality.* Terms were then compared recursively as follows:

1. Given expressions a, b;
2. Check if a, b are definitely equal or disequal in the context Ξ, if so, return which
3. Otherwise, if a, b are the same kind of expression, perform a recursive comparison of each component of the expression.

We provided functionality for explicitly extending a context Ξ with equality/disequality relations between normalized terms by building modified a disjoint set forest of terms (indexed into a set via $\mathcal{O}(1)$ hashing, as described in 5.2.3), as well as a wrapper interface for a context Ξ which would:

1. Given expressions a, b;
2. Check if a, b were definitely equal or disequal in Ξ, if so, return the result
3. Normalize a, b according to some configuration, cache the results in Ξ, and return any deductions made about equality

The current API allows users to pass an equality context Ξ as part of the typing context for all type checking and inference operations; by default, they can simply use the null context (which requires syntactic equality) or a fully normalizing context (replicating the behaviour of the old implementation). This allows API users the freedom to define quite complex custom notions of equality to be used in type checking (e.g., theoretically, extensionally using the results of an SMT solver), though we only tested such functionality on somewhat artificial notions of equality. More importantly, however, it allowed very efficient handling of normalization and equality checking, as the API user can explicitly control when and how often normalization occurs without worrying about correctness by using the "safe" portion of our API, which only allows defining subrelations of the judgemental equality relation.

Equality contexts which perform reduction may also set a maximum number of operations to perform as well as a maximum reduction depth, and may even define a custom reduction strategy; they may also be passed program annotations denoting, e.g., maximum recursion depth for or a particular property of a certain term. This design decision was originally meant to support nonterminating terms, inspired by Zombie [16].

5.2.2 Constraints and Borrowing

The original `isotope` implementation inferred constraint sets for terms automatically, which were stored in a pointer in the term data structure. In particular, this allowed us to forego assigning machine terms unique names, as the constraint set was computed directly and optimally from the structure of the term. Since the constraint sets for different terms often had extensive data sharing between them, we required that a context was used to construct terms, which would perform hash-consing on constraint sets, which were implemented using array-mapped hash tries. A function was implemented which borrow-checked constraint sets within a given scope; on construction of a machine function, the result of the borrow checking function was set as the term's constraint set, as in the formalism in Chapter 3. A bug in the trie implementation we used to implement constraint sets led to us using a temporary, highly inefficient implementation using regular hash maps, and eventually, the issues this led us to decide to rewrite the codebase altogether.

The new version of `isotope` instead required all machine terms to be tagged with a unique ID (intuitionistic terms were identified by virtue of being untagged), with IDs combined with de-Bruijn indexes for scope depth (for example, a term of the form $\lambda x.\mathsf{fn}(x : A) \mapsto r$ would have all subterms of r be assigned IDs with a de-Bruijn index of 2, since they are within the scope of both the machine function and lambda function). This additional scope information was to be used to generate anonymous, unboxed closure types, as were available in Rust. Constraint sets were to be implemented *externally* as part of the typing context (rather than as intrinsic parts of individual terms), and were to have a separate inference algorithm, as in the current formalism in Chapter 3; unfortunately, we did not get this far in the implementation.

5.2.3 Term Representation

In both implementations, we represent terms as a Rust `enum` behind an atomically reference-counted pointer; in particular, both implementations provide an entirely thread-safe API interface (but not, as of now, parallelized). The majority of the API surface for terms is exposed in both implementations via a `Value` interface implemented both by the terms themselves and the variants of the term `enum`, with most `Value` methods on terms being simple delegations to the `enum` variants (with, in some cases, e.g. normalization for the old implementation, some caching).

Both implementations require that term variants store a 64-bit "hash code," computed by hashing the data contained in each term; the hash of a term is then defined to be the hash of it's hash code. This allows us to hash terms in $\mathcal{O}(1)$ time, with hashcodes themselves precomputed in $\mathcal{O}(1)$ time at term constructions. This is important, as both implementations make extensive use of hash-tables of terms, which would otherwise be inefficient due to the potential for terms to be very large and spread out throughout memory (due to the extensive use of reference-counted pointers).

The presence of hash-codes allows a way to quickly test terms for *disequality*, as well as a very high-probability way to check for *equality*, both of which we take advantage of in our implementation of equality and typing contexts. Unfortunately, we cannot use hash-codes to perform a *certain* equality check. We experimented with global hash-consing during the design phase of the first implementation, but decided against it due to the unwieldiness and inefficiency of the global state and synchronization (due to thread-safety) required. However, when constructing terms, we do allow API users to pass a *local* hash-consing context to the constructor to maximize sharing; oftentimes, such as in the case of the parser, a large portion of an application can share a single thread-local context and hence produce terms with a high level of sharing. We take advantage of this by, when checking terms for equality, always first checking for pointer equality; this recovers most of the advantages of global hash-consing in cases with high sharing without most of the overhead.

Chapter 6

Conclusion and Further Work

In this chapter, we briefly go over some of the limitations of our current approach in Section 6.1 along with potential directions for further work in Section 6.2.

6.1 Limitations of Current Approach

Our current approach to integrating ownership types and dependent types has numerous limitations. In particular, we have no support for:

- Expressions which *partially* use other expressions, e.g., the Rust expression `x.y`, which uses only the component `y` of the `struct x`. Similarly, we have no support for expressions which perform *partial borrows* of other expressions.
- Expressions which make partial use of an array, map the elements of an array, or write to an array
- "Mutable references," which we may represent as linearly typed immutable terms.

All of these, however, can most likely be solved with some minor tweaks to the type theory and/or the addition of some new rules. A much more serious defect, in my opinion, is the unwieldiness of the "naming" system of judgements like $x \leftarrow: A$ used in the typing rules for `isotope`. As stated in Chapter 3, this is in reality a bit of a hack, as the real implementation, as described in Chapter 5, works by allowing expressions to be given unique tags, obviating the need for cumbersome constructions based off variable names. Another issue is that, variable names aside, there are simply a lot of non-orthogonal typing and reduction rules which make reasoning inductively about the type system a pain. We hope to address both these issues in future work by considering alternative formalisms based on proof nets, rather than standard deductive type theory.

6.2 Further Work

6.2.1 `isotope` Interpreter and Compiler

One of the most important directions for further work is, of course, to complete the implementation of the `isotope` interpreter and type-checker. While the current implementation of `isotope` is written in Rust, it may also make sense to attempt to implement a model `isotope` in a dependently typed language such as Agda so as to be able to formally prove some of it's properties in a machine-checked manner. Similarly, a compiler based off the algorithm in Chapter 4 has yet to be written, and there are many interesting design decisions, e.g., the treatment of tail call optimization, or JIT compilation of interpreted terms as an optimization, that we have not yet explored.

6.2.2 Metaprogramming and Semantics

Originally, `isotope` was supposed to have had metaprogramming functionality, in the form of the ability to perform induction on machine universes and terms. We also wished to internalize the treatment of instants so as to be able to reason about them from inside the type system. We did not proceed in this direction, as we wished to keep the type theory simple, but it could provide some interesting directions for further work.

As a particular, we defined the notion of a *optimization* as follows

Definition 30. *We define an* optimization *to be a relation $\mathcal{P} \subseteq \Lambda \times \Lambda$ from the set of `isotope` terms to itself such that*

$$\forall (s, t) \in \mathcal{P}, \mathsf{d}s = \mathsf{d}t \tag{6.1}$$

We say such a $\mathcal{P}$ is a resource preserving optimization *if*

$$\forall (s, t) \in \mathcal{P}, \mathsf{usage}(s) \preceq \mathsf{usage}(t) \tag{6.2}$$

Given a way to internally operate on terms in Λ, it could be possible to write verified optimizations for `isotope` *in `isotope`*, which I believe would be quite an interesting development for compiler toolchains in general.

6.2.3 Heap Model, Effects, and Locations

One of the goals of `isotope` was to allow reasoning about and verifying (an analog to) `unsafe` code *within* a Rust-like language (i.e., without requiring external tools). To do this, we need a proper model of pointers, the heap, effects, and locations which can be compiled efficiently but is also not too difficult to reason about mathematically. There are many possible ways one could go about this (e.g., Hoare type theory), and exploring this design space could be provide a fascinating source of further research directions.

6.2.4 Nontermination and Termination Checking

The original formulation of `isotope` was meant to include a (hopefully) consistent treatment of non-terminating terms via a non-termination monad. The handling of equality for non-terminating terms was originally based off a strategy similar to that found in the paper Zombie [16]. Exploring this design further could be an interesting avenue for further research, especially as it interacts with potential systems of effects and heap modification.

Bibliography

[1] Andreas Abel. *foetus – Termination Checker for Simple Functional Programs*. 1998.

[2] *Agda 2.6.0.1 Language Reference*. URL: `https://agda.readthedocs.io/en/v2.6.0.1/language/index.html#language-index`.

[3] Nick Benton. “A Mixed Linear and Non-Linear Logic: Proofs, Terms and Models”. In: Sept. 1994, pp. 121–135. ISBN: 978-3-540-60017-6. DOI: `10.1007/BFb0022251`.

[4] Jean-Philippe Bernardy et al. “Linear Haskell: practical linearity in a higher-order polymorphic language”. In: *arXiv e-prints*, arXiv:1710.09756 (Oct. 2017), arXiv:1710.09756. arXiv: `1710.09756 [cs.PL]`.

[5] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013. ISBN: 0262026651.

[6] Geoffroy Couprie. *nom*. `https://github.com/Geal/nom`. 2021.

[7] ISO. *ISO/IEC 9899:201x: Programming languages — C*. 2010, p. 683.

[8] Simon Peyton Jones, Norman Ramsey, and Fermin Reig. “C—-: A Portable Assembly Language that Supports Garbage Collection”. In: *Principles and Practice of Declarative Programming*. Ed. by Gopalan Nadathur. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 1–28. ISBN: 978-3-540-48164-5.

[9] Ralf Jung et al. “RustBelt: Securing the Foundations of the Rust Programming Language”. In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017). DOI: `10.1145/3158154`. URL: `https://doi.org/10.1145/3158154`.

[10] Ralf Jung et al. “Stacked Borrows: An Aliasing Model for Rust”. In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: `10.1145/3371109`. URL: `https://doi.org/10.1145/3371109`.

[11] Andrew D. Ker. *Lambda Calculus and Types*. 2020.

[12] Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. “Integrating Linear and Dependent Types”. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’15. Mumbai, India: Association for Computing Machinery, 2015, pp. 17–30. ISBN: 9781450333009. DOI: `10.1145/2676726.2676969`. URL: `https://doi.org/10.1145/2676726.2676969`.

[13] Nicholas D. Matsakis and Felix S. Klock. “The Rust Language”. In: *Ada Lett.* 34.3 (Oct. 2014), pp. 103–104. ISSN: 1094-3641. DOI: `10.1145/2692956.2663188`. URL: `https://doi.org/10.1145/2692956.2663188`.

[14] U. Norell. “Towards a practical programming language based on dependent type theory”. In: 2007.

[15] Dennis M. Ritchie. *The C Programming Language*. 1988.

[16] Vilhelm Sjöberg and Stephanie Weirich. “Programming up to Congruence”. In: *SIGPLAN Not.* 50.1 (Jan. 2015), pp. 369–382. ISSN: 0362-1340. DOI: `10.1145/2775051.2676974`. URL: `https://doi.org/10.1145/2775051.2676974`.

[17] The Coq Development Team. *The Coq Proof Assistant*. Version 8.13. Jan. 2021. DOI: 10.5281/zenodo.4501022. URL: https://doi.org/10.5281/zenodo.4501022.

[18] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: https://homotopytypetheory.org/book, 2013.

[19] P. Wadler and J. Kilmer. "A prettier printer". In: 2002.

[20] Philip Wadler. "Linear Types Can Change the World!" In: *PROGRAMMING CONCEPTS AND METHODS*. North, 1990.

[21] Markus Westerlind et al. *nom*. https://crates.io/crates/pretty. Version 0.10. 2020.