A Framework for Semiring-Annotated Type Systems

PhD Thesis

James Wood

Mathematically Structured Programming

Computer and Information Sciences

University of Strathclyde, Glasgow

October 8, 2023

This thesis is the result of the author's original research. It has been composed by the author and has not been previously submitted for examination which has led to the award of a degree.

Abstract

The use of proof assistants as a tool for programming language theorists is becoming ever more practical and widespread. There is a range of satisfactory implementations of simply typed calculi in proof assistants based on dependent type theory.

In this thesis, I extend an account of Simply Typed λ-calculus so as to be able to represent and reason about calculi whose variables have restricted usage patterns. Examples of such calculi include a logic with an S4 $\Box$-modality, in which certain variables cannot be used "inside" a box ($\Box$); and Linear Logic, in which linear variables have to be used exactly once. While there are existing implementations of some of these calculi in proof assistants, many of these implementations share little with the best presentations of simply typed calculi without variable usage restrictions, and thus end up being poorly understood or suboptimal in facilitating mechanised reasoning.

Concretely, the main result of this thesis is a framework for representing and reasoning about a wide range of calculi with restricted variable usage. All of these calculi support novel simultaneous renaming and substitution operations. Furthermore, I provide several other examples of generic and specific programs facilitated by the framework. All of this work is implemented in the proof assistant Agda.

Contents

List of Figures

Acknowledgements

My first thanks go to my supervisor, Bob Atkey. The combination of his own knowledge and his understanding of the lacks in my knowledge has let me have a positive and fulfilling experience as a PhD student. The wider Mathematically Structured Programming group at Strathclyde has also taught me much, well beyond what appears in this thesis. The group has also often been the source of fun and excitement in research, which I see as crucial to progress.

Special thanks go to Guillaume Allais, Jan de Muijnck-Hughes, and James McKinna, all of whom have read all or part of this thesis in draft form and whose comments have lead to great improvements in the final document. Similar thanks go to the anonymous reviewers of the papers that have lead to and formed part of this thesis.

The programming language research community as a whole has helped me get to where I am today. Interactions at conferences and other meetings have brought me plenty of new perspectives. Additionally, starting even before my PhD studies, the formerly Twitter, now Fediverse, community of researchers and practitioners have been crucial in getting me interested in programming languages and various more specific topics thereof.

I thank my family, particularly for our time together in the COVID-19 lockdown/work-from-home period. The pandemic caused difficulties for us all, but a positive approach got us through it and let us take advantage of what we could. More recently, I thank Ayaka, who has listened to and supported me over the last year of write-up. Above all, she has been very patient, and with my thesis-writing now ending, I hope that we can live our lives more fully together.

I have made many friends outside academia during my studies while living in Glas-

gow and, latterly, Edinburgh. Particularly, those of the Strathclyde Board Games Society and Japanese language learning groups have given me much needed fun times in evenings and weekends, as well as deeper friendships. I owe much of my good mood and motivation to carry on to these people.

The work described in this thesis would not have happened without the Agda programming language/proof assistant and both the contents and style developed in its standard library. I thank all contributors to these tools, as well as everyone who has offered help on how to use them. Similarly, the preparation of this thesis and related papers was much aided by TEX, LATEX, and the various packages needed to practically produce beautiful mathematical documents.

This document is adapted from the template by Jethro Browell (`https://www.overleaf.com/latex/templates/thesis-template-for-university-of-strathclyde/nfnrnmjqyxqg`), which was licensed under CC BY 4.0.

Chapter 0. Acknowledgements

Chapter 1

Introduction

Programming language design has a history stretching back almost to the inception of electronic computing. Programming language *theory* has arguably an even longer history, with parts of mathematical logic developed in the early 20th century forming the vocabulary of programming language theory as we know it today. Programming language theory studies mathematical models of programming languages to help us better understand programs in existing languages, and to inform the design of new language features, new languages, and new programming paradigms.

Within programming language research, type theory is a methodology for classifying program behaviour. The simplest and most common type systems classify programs by what values they may return. For example, in many — perhaps most — widely used programming languages, the compiler will keep track of the types of expressions and their subexpressions, and give an error or warning to the programmer whenever there is a mismatch between the expected type and the actual type. For example, the body of a C function with return type `char` is a program that, if it returns a value, will return a character. If that function has a parameter of type `int`, then we can compose it with a function with return type `int` to build up a larger program. We expect type systems to stop us from running programs with arguments of the wrong types. This holds of both static and dynamic type systems — with a static type system, the compiler will refuse to compile our code if we pass a `char` value to a function expecting an `int`, while with a dynamic type system, our program will do a check before running the

function to make sure that we have passed it an `int`. The abstraction produced in such simple type systems is the idea that a function will take arguments in accordance with its parameter types and produce a result in accordance with its result type, and furthermore, as readers, we do not have to inspect the function's implementation to see that these properties are true.

A more recent trend in type theory is to use types to describe other parts of a program's behaviour than what range of values it may return. For example, Java's *checked exceptions* can be seen as part of the static type system, in which programs are classified by which exceptions they may throw. More generally, *effect systems* [Lucassen and Gifford, 1988] classify programs based on all of the effects they may have — such as reading input and writing to files, and also internally defined effects, such as non-determinism and using an accumulator. In a type system which tracks effects (an *effect system*), programs by default are pure (i.e. have no effects), with effects being opt-in. Pure programs generally enjoy good properties, making them easy to reason about and easy for an optimising compiler to optimise.

In this thesis, I consider the dual of effect systems — *coeffect systems* — as introduced by Petricek et al. [2014]. In a coeffect system, we are interested not in what extra behaviour a program may exhibit (as with effects), but rather what extra abilities the context of a program may provide. Analogously to the case of effect systems, we typically restrict our coeffect-free programs to be "more pure" than usual. A standard example is to restrict to *linear* programs, in which each variable in the context is used exactly once. The ability to duplicate and discard variables is then seen as a coeffect, which can be tracked by a coeffect system. Restricting to linear programs may seem like an arbitrary restriction at first, but the expectation of linearity arises naturally in applications such as file-handling, session-typed communication, and approaches to mutable memory. I introduce linearity and its applications more fully in chapter 3.

The work of this thesis relies upon type theory in two distinct ways. Firstly, as I have introduced above, the main objects of study in this thesis are programming languages with interesting type systems. Secondly, type theory provides the basis of the proof assistant Agda I use to implement the aforementioned programming languages

and operations upon them. I will now introduce the idea of proof assistants.

A *proof assistant*, also known as an *interactive theorem prover*, is a piece of software that allows for the encoding of mathematical definitions, theorems, constructions, and proofs, and furthermore check that such encoded proofs are correct and that such encoded constructions are well formed. To truly be interactive, i.e. to actually assist, a proof assistant will usually have a user interface which can read partial proofs, display information about what more proof needs to be given, and provide actions that will help complete the proof.

Proof assistants have seen increasing use in programming language research in recent years. The most obvious reason why working in a proof assistant is seen as beneficial is that it ensures correctness. If the proof assistant accurately implements a suitable mathematical foundation, then any theorem proved in the proof assistant is guaranteed to be a true theorem of that foundation. These guarantees of correctness are particularly important when working with combinatorially complex mathematical objects, proofs about which often require the consideration of a large number of cases. Programming language syntaxes are often such complex objects, motivating the use of proof assistants when studying programming languages.

A second reason to use proof assistants is for the assistance they provide when exploring a mathematical theory. When we make a new definition, we may want to test how it works in a special case, or what constructions it allows us to perform. In a proof assistant, the assistance tools give us immediate feedback as to what moves are and aren't allowed. For example, if we define a complex type system, a proof assistant will let us interactively build typing derivations, making clear any side conditions and types of subderivations as we go. Also, as I do later in this thesis, a proof assistant allows us to build a very general theory, and practically use that theory directly in more specific cases without losing rigour.

Thirdly, analogously to how a strong static type system can give us more confidence when refactoring a program, the constant checking of proofs in a proof assistant gives us the confidence to change definitions and lemmas knowing that we will be guided towards the parts of our theory that need to be correspondingly changed. This can help

if we are developing a new programming language with a changing specification.

Finally, many proof assistants — including Agda [Agda Development Team, 2023], which I use in this thesis — double as programming languages themselves. This means that we can write programs and prove properties of them using the same tool. Also, many theorems proven in such a proof assistant have computational content. For example, if we prove a normalisation theorem for a programming language, this will typically yield a (verified) normalisation algorithm for it, which we can really run on a computer. As such, a development in a proof assistant can provide a reference implementation of a programming language, or even — as with Idris 2 [Brady, 2021], Lean 4 [de Moura and Ullrich, 2021], and Cedille [Guneratne et al., 2016] — the actual implementation of a programming language.

1.1 Outline of the thesis

This thesis proceeds as follows. The next two chapters, chapters 2 and 3, are introductory in nature, and cover two largely independent strands of prior work. In chapter 2, I introduce existing methods of representing and reasoning about type systems in proof assistants based on dependent type theory. I start from well established representations of well scoped and well typed terms, and develop these towards a recent approach to environment-based semantics given by Allais et al. [2021]. In chapter 3, I discuss the challenges faced when one extends a treatment of a simple type system, such as that given in chapter 2, to modal and linear type systems. We see that modal and linear type systems apparently violate some of the nice properties of the simply typed λ-calculus we required in chapter 2. I present a solution for intuitionistic S4 modal logic, but leave a solution for linear logic to the following chapters.

In the following two chapters, chapters 4 and 5, I present a calculus $\lambda\mathcal{R}$ parametrised by a partially ordered semiring of *usage annotations*. In chapter 4, I define the calculus, give some possible extensions, and show that it subsumes intuitionistic S4 modal logic and Intuitionistic Linear Logic. In chapter 5, I show that $\lambda\mathcal{R}$ enjoys generalised versions of the nice properties required in chapter 2, and I proceed to give novel definitions of simultaneous substitutions and their action on $\lambda\mathcal{R}$ terms. These two chapters are

adapted from the work of Wood and Atkey [2021].

The remaining three main chapters, chapters 6 to 8, adapt the syntactic and semantic framework of Allais et al. [2021], as presented at the end of chapter 2, to semiring-annotated calculi. Chapters 6 and 7 generalise the work on $\lambda\mathcal{R}$ presented in chapters 4 and 5, respectively. Chapter 6 shows how to formally describe the syntax of an arbitrary semiring-annotated calculus, following the constructions used in chapter 4. Chapter 7 then provides the generic environment-based semantic traversal on such syntaxes, providing renaming and substitution as per chapter 5 for all syntaxes as special cases of the generic traversal. Chapter 8 then gives further example uses of the generic traversal.

Finally, I conclude with chapter 9, which discusses the achievements of this thesis and openings for future work.

1.2 Naming and notation conventions

I assume familiarity with the Curry-Howard correspondence [Howard, 1980] throughout this thesis. I make no distinction between logics and type theories, and use terminology from each interchangeably. Each following bullet point lists a collection of synonyms.

- assumption, hypothesis, variable
- proposition, formula, type
- connective, type former
- derivation, proof, term
- derivable (formula), inhabited (type)

I carry out mechanised constructions and proofs in the proof assistant and programming language Agda [Agda Development Team, 2023]. Agda is based on Martin-Löf's intensional dependent type theory, so I similarly present non-mechanised constructions and proofs assuming a foundation given by dependent type theory, in a style inspired by the HoTT Book [Univalent Foundations Program, 2013]. I give a fuller introduction to Agda in section 2.1.

This thesis is written in colour, but should be readable without. Agda code has syntax highlighting, and various pieces of notation related to usage annotations are coloured green for emphasis. As a test, the word *colour* in the first sentence of this paragraph should appear with the first two letters red, the next two green, and the final two blue, with the shades of red, green, and blue used for Agda code. Additionally, the word *green* in the second sentence is written in the shade of green used for usage annotations.

Chapter 2

Mechanisation of simple types

In this chapter, I review and justify the family of approaches usually used to represent simple type systems inside dependently typed proof assistants. These approaches were first presented by Altenkirch and Reus [1999], who showed a way of representing well *scoped* terms in a language with polymorphic recursion, and extended the representation to well *typed* terms in a language with dependent types. The representation of terms relies on indexing on both a context — giving the types of all the free variables — and a type for the term itself. A basic operation on terms is *simultaneous substitution*, which replaces each variable in the context by a term in another context.

This chapter almost entirely reviews prior work. First, in section 2.1, I give an introduction to Agda, the proof assistant I work in throughout this thesis. Then, in section 2.2, I use a presentation of the dependently typed encoding of the simply typed λ-calculus from Altenkirch and Reus [1999] to set notational conventions. Section 2.3 presents the unpublished work of McBride [2005], successively deriving simultaneous renaming and simultaneous substitution for the terms defined in section 2.2. The rest of the chapter generalises the shared core of renaming and substitution in two dimensions: in section 2.4 following Allais et al. [2017] to cover semantic traversals, and in section 2.5 following Allais et al. [2021] to cover a whole range of simply typed syntaxes with binding, rather than just a specific syntax. Finally, I review some related work in section 2.6.

2.1 Agda primer

I use the proof assistant and programming language Agda throughout this thesis, with Agda code being used particularly in this chapter and chapters 6 to 8. As such, it is important for the reader to be able to read basic Agda syntax in order to benefit from the parts of the exposition that reside in code listings. The syntax of Agda is broadly similar to that of Haskell [Marlow, 2010], and relatively close to that of Standard ML, OCaml, and Coq version 8's Gallina sublanguage [Coq Team, 2023, Leroy et al., 2022, Milner et al., 1997]. I will assume that the reader is able to read basic Haskell code, and spend most time explaining differences thereof.

2.1.1 Lexical structure

Agda is extremely liberal in its set of allowed names. There is just a single lexical class (unlike in Haskell, where, for example, constructors start with a capital letter and definitions start with a lowercase letter), and names can be any string of Unicode characters except whitespace and special characters `.;{}()@"`, apart from those strings reserved as keywords or literals. Therefore, we can introduce names like `0x-+-λ→` to stand for any kind of identifiable thing. With such free-form names, ample spacing is required between identifiers. For example, while `0 ≤ 1` is a possible expression containing three identifiers, `0≤1` is a single valid identifier. Only the special characters may appear next to names without being separated by whitespace.

A character with unique behaviour in Agda's syntax is the underscore (`_`). Within a name, an underscore signifies that the name will function as a mixfix operator, allowing for an argument in the position of the underscore. For example, the full name of the ≤ operator used in the previous paragraph is `_≤_`, signifying that it can take an argument to its left and its right. We can also introduce closed operators, like `[_]`, which can take an argument between the square brackets (e.g. `[ 1 ]`, with spaces still being important). Mixfix operators can be partially applied by leaving underscores in the name in the application. For example, `_≤ 1` could be the predicate asserting that a number is less than or equal to 1.

On its own, an underscore has a completely different meaning, which can depend on context. In patterns, an underscore has the same meaning as it has in Haskell and ML — it holds the place of a pattern variable, but does not name that variable. In expressions, an underscore stands for an unspecified subterm which will be solved by unification [Abel and Pientka, 2011, Miller, 1992]. The solving of unspecified terms is canonical and respects $\beta\eta$-equality, unlike in Coq.

Spacing is important particularly important when dealing with underscores. For example, $\_\leq\ \_$ (with a space after the $\leq$ but not before) standing for the predicate asserting that a number is less than or equal to some unspecified number.

Like Haskell, Agda's syntax is indentation-sensitive. The distinctions conveyed by indentation are largely obvious or intuitive to human readers (for example, allowing for line-continuation or delineating nested modules), so I will not discuss them explicitly here.

2.1.2 Functions, Π-types

Simple function types take the form $A \to B$, coinciding with Haskell's syntax. Also as in Haskell and ML, the function arrow nests to the right. However, Agda has a termination checker ensuring that all definable functions are total, so many Haskell functions do not have a corresponding Agda function.

The key feature distinguishing Agda from Haskell is the presence of arbitrary dependent types, including dependent function types (Π-types). The basic syntax for Π-types is $(x : A) \to B$, where variable x can occur free in expression B. However, there are several syntactic conveniences I use throughout the code listings. For one, iterated Π-types can be abbreviated so that $(x : A) \to (y : B) \to C$ is written just $(x : A)\ (y : B) \to C$, omitting the first arrow. For another, prefixing an arrow with the $\forall$ symbol allows us to omit domain types. For example, $\forall\ x \to B$ is equivalent to $(x : \_) \to B$. Notice that this is a very different type to $x \to B$, which is a non-dependent function type equivalent to $(\_ : x) \to B$. When writing $\forall\ x \to B$, we assume that the occurrence of x in B tells us what type x should have (i.e. there is enough information to solve the underscore in $(x : \_) \to B$).

Just like in Haskell, functions in Agda can be introduced via λ-abstractions and clausal definitions, and are applied by juxtaposition. Agda also includes *extended* λ-abstractions, introduced via equivalent syntaxes λ **where** $x \to M$ and λ { $x \to M$ }, which allow for pattern-matching on the variable x (or all of the variables, if there are multiple variables).

Agda allows for arbitrary function arguments to be marked as *implicit* by replacing the round brackets in the type by curly braces. For example, if we have $f : \{x : A\} \to B$, then the argument to f is implicit. Being implicit means that an occurrence of f is treated as if it has been applied to an underscore, giving the expression f the type $B[\_/x]$ (i.e. B with _ substituted in for x; the substitution syntax is not part of Agda syntax). An implicit argument can also be given explicitly in two ways. The first of a series of implicit arguments can be given by surrounding the argument in curly braces, and any other implicit arguments in the series can be given by including the name of the argument. For example, $f\,\{\_\}$, $f\,\{x = \_\}$, and just f are all equivalent expressions, and the underscore can be filled in in either of the first two expressions to provide an actual value for the implicit argument. Implicit arguments are usually left out of λ-abstractions and clausal definitions, but can be bound to names and pattern-matched on using the same syntax as in expressions.

There are a few other places in the syntax using single curly braces, all of which have meanings related to implicit arguments. I also make a small amount of use of double curly braces ({{ and }}), which denote arguments which are to be solved by instance resolution. Instance resolution is very similar to Haskell's typeclass resolution — finding non-canonical solutions based on the instances in scope.

Agda uses Π-types where in Haskell we would use polymorphism. For example, we can define an identity function as below. The definition relies on quantifying over terms of type Set, i.e. (small) types. This definition also gives an example of defining a function with an implicit argument (X), which can typically be inferred from either the argument type or the return type, so can be omitted.

```
id₀ : {X : Set} → X → X
id₀ x = x
```

An unfortunate feature of the definition id_0 is that we cannot apply it to the expression Set, because Set contains only small types, and itself is a large type. We can work around these size issues using *universe level polymorphism* [Bezem et al., 2022], as in the following definition.

```
id : {ℓ : Level} {X : Set ℓ} → X → X
id x = x
```

Universe levels start at 0ℓ, with Set being an alias for Set 0ℓ (and also Set_0). Larger levels can be produced with the successor operator suc, and we can take the least upper bound of two levels using the operator _⊔_.

2.1.3 Data types

Agda's **data**-declarations are similar in scope to Haskell's, with the addition of indexing by terms of arbitrary type. **data**-declarations give us indexed inductive sum-of-product types.

All **data**-declarations use GADT syntax. The body of a declaration comprises a list of constructor names paired with their types. Where two constructors have the same type, they may be written on the same line with their names separated by whitespace, as I do with the two constructors of Bool below. Bool has two constructors — true and false — both of which have type Bool. ℕ also has two constructors, where zero has type ℕ and suc is inductive, with type ℕ → ℕ.

```
data Bool : Set where
  true false : Bool
```

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ
```

Bool and ℕ are both types, and indeed small types, as we can see by the fact that they are annotated to have type Set. We can also use **data**-declarations to define type *families* in various ways. The simplest is to add *parameters*, as in the type family List below. Parameters always appear to the left of the colon of the first line of the **data**-declaration, and are constant throughout the **data**-declaration. Variables to the left of

the colon can appear in the body of the **data**-declaration without further quantification.

```
data List (X : Set) : Set where
  [] : List X
  _::_ : X → List X → List X
```

Slightly more flexible than parameters are *Protestant indices*[1]. Protestant indices also appear to the left of the colon, and also must appear unmodified in the *return type* of all of the constructors. However, they may take different values in inductive appearances of the type family in the argument types of constructors. Protestant indices give a generalisation of polymorphic recursion to indices of arbitrary type [Henglein, 1993, Mycroft, 1984].

I give two examples of type families with Protestant indices. The first, NestedList is standard from the polymorphic recursion literature. It is worth noting at this point that Agda permits overloading of constructors, which are disambiguated by the type family they are being used to construct. This overloading allows List and NestedList to have constructors with the same names without confusion. The second example, ScopedTerm is a data structure representing well scoped untyped λ-calculus terms. The Protestant index s describes the number of variables in scope, which increases by 1 when we introduce a λ-abstraction. I will introduce Fin, a type family with a specified natural number of inhabitants, in the next set of examples. As a syntactic note, in the type of the app constructor, I use the two variable names M and N separated by whitespace to name two arguments with the same type.

```
data NestedList (X : Set) : Set where
  [] : NestedList X
  _::_ : X → NestedList (List X) → NestedList X

data ScopedTerm (s : ℕ) : Set where
  var : Fin s → ScopedTerm s
```

[1]The terminology of Protestant/Catholic indices is due to Peter Hancock. The mnemonic is that Catholics believe in transubstantiation, which is seen as analogous to the instantiation of Catholic indices with expressions that occurs during dependent pattern matching.

```
  lam : ScopedTerm (suc s) → ScopedTerm s
  app : (M N : ScopedTerm s) → ScopedTerm s
```

The most general way to make a type family is to introduce a *Catholic index*. The types of Catholic indices are specified to the right of the colon, and can be instantiated arbitrarily throughout the **data**-declaration. Catholic indices are not in scope for the body of the **data**-declaration, so the values filling them may need to be quantified over in each constructor. When this quantification is over a large type, like Set, the type family being defined will itself need to be large, e.g. inhabiting Set_1. This is a major reason for not defining types like List and NestedList using Catholic indices.

I give two examples of type families with Catholic indices. The first is the Fin family, as used in ScopedTerm above. By inspection of the return types of the constructors, there is no way to produce a canonical inhabitant of Fin zero. For Fin (suc n), we can potentially use either of the constructors. Either we use zero to get a canonical inhabitant, or if we can make a number with a smaller bound (i.e. an inhabitant of Fin n), we can use suc to produce a larger number.

```
data Fin : ℕ → Set where
  zero : ∀ {n} → Fin (suc n)
  suc : ∀ {n} → Fin n → Fin (suc n)
```

The second example of a type family with Catholic indices is more general in nature. Below I define *propositional equality*, written _≡_. It has two parameters and one Catholic index (though the standard library version of propositional equality I use throughout this thesis has an extra level parameter for the sake of universe level polymorphism). The constructor refl constructs an inhabitant of $M \equiv N$ only when N is definitionally equal to M (because terms are considered "the same" to the type checker exactly when they are definitionally equal). Notice that refl does not quantify over x because x is already in scope as a parameter.

```
data _≡_ {X : Set} (x : X) : X → Set where
  refl : x ≡ x
```

It is through type families like _≡_ that we can state and prove mathematical theorems in Agda. In the following subsection, I show how to use such indexed type families.

2.1.4 Clausal definitions

Clausal definitions of functions in Agda look very similar to their equivalents in Haskell. However, definitions in Agda regularly make use of *dependent pattern matching*, which is our primary way of using indexed data types. Recursive definitions are also conservatively checked for termination.

I will explain the salient aspects of clausal function definitions via two examples. The first, unimaginatively named lemma, shows a simple case where pattern matching modifies the context through unification of Catholic indices. The second, named elim-Fin-zero, gives an example of proper dependent pattern matching.

In the following definition lemma, we want to chase equations in order to prove that x is propositionally equal to z. We start with the following incomplete definition, where the expression { }0 marks an interaction point, or *hole*, in the program, to which we can apply interactive commands to complete the program.

```
lemma : ∀ {A : Set} {x y z : A} → x ≡ y → z ≡ y → x ≡ z
lemma p q = { }0
```

As a first step, I choose to match on the variable $p : x \equiv y$. The only applicable pattern is refl. Doing this match has the effect of unifying y — which is taking the position of the Catholic index of _≡_ — with x — which is the value of the index specified in the type of refl. Local variables act as unification variables, so the unification succeeds with most general unifier $[x := x, y := x]$. Therefore, the type of q becomes $z \equiv x$.

```
lemma : ∀ {A : Set} {x y z : A} → x ≡ y → z ≡ y → x ≡ z
lemma refl q = { }0
```

The next step is to match on q. This similarly unifies z and x, making the conclusion

type $z \equiv z$. Finally, this conclusion type is in the image of the refl constructor, so we may fill the hole with refl.

```
lemma : ∀ {A : Set} {x y z : A} → x ≡ y → z ≡ y → x ≡ z
lemma refl refl = refl
```

Full *dependent* pattern matching, as described by McBride and McKinna [2004], is when the unification of indices described above takes account of constructors. In particular, the constructors of a data type satisfy the "no confusion" property — constructors are injective and mutually disjoint. Where we encounter disjoint constructors during unification, we may dismiss the corresponding case as impossible. Consider the following example (elim-Fin-zero). We start with an argument i : Fin zero, and consider which constructors could possibly construct such a value. However, as noted earlier, both constructors of Fin target successor values of the index, from which zero is disjoint. Therefore, both cases are impossible. The notation when all cases are impossible is to place empty round brackets () in the place of the impossible argument, and to not provide a right-hand side to the clause.

```
elim-Fin-zero : ∀ {A : Set} → Fin zero → A
elim-Fin-zero ()
```

As an example of the injectivity of constructors, the obvious example is to internalise the proof of injectivity for a given constructor, as I do in suc-injective. We start with an argument p : suc $m \equiv$ suc n and match on it. This time, we do have a possible pattern — refl — but working out how to change the context relies on unifying suc m with suc n. We are justified in doing this, with most general unifier $[m := m, n := m]$, because suc is injective (with respect to propositional equality). If the checker for dependent pattern matching did not know that suc was injective — for example, if it were instead a defined function — then the unification would fail. This leads to the intuition that constructors and variables are well behaved with respect to dependent pattern matching, while other expressions are not.

```
suc-injective : ∀ {m n : ℕ} → suc m ≡ suc n → m ≡ n
suc-injective refl = refl
```

Ordinarily, each clause of a definition gives rise to a *definitional* equation between its left-hand side and right-hand side. In intensional type theory, as implemented by Agda, definitional and propositional equality are contrasted to each other. Definitional equality corresponds to a decidable fragment of the natural equational theory of the type theory. As such, definitional equality is an entirely metatheoretic notion, and we can neither assume nor prove directly definitional equations within the language. Definitional equality is sometimes also called *judgemental equality*, because it forms a judgement which plays a part in the rules of the type theory. As well as from the clauses of definitions, we also get definitional equations from β-reductions of λ-abstractions and η-laws of functions and records. Because the type checker treats definitionally equal terms equivalently, we are able to refactor up to definitional equality without changing any downstream code.

On the other hand, propositional equality is a notion internal to the language, as we have seen by defining propositional equality (`_≡_`) and proving things about it (`lemma`). Propositional equality is sometimes known as *typal equality* or *mathematical equality*. The latter name comes from the fact that propositional equality is the closest notion to what mathematicians usually call *equality*, because, for example, it allows us to prove things like $m + n \equiv n + m$ for all natural numbers m and n. Propositional equality satisfies Leibniz' law, meaning that an inhabitant of a type A can be coerced into an inhabitant of any type propositionally equal to A. However, this cast requires marking in the code, so is less convenient to use than definitional equality.

Definitional equality between two terms implies their propositional equality, because exactly when two terms are definitionally equal, the type checker is happy to accept `refl` as a proof. This relationship between the two is simple, but can still be deceptive. For example, consider the notion of injectivity with respect to definitional and propositional equality. A function f is injective (with respect to some notion of equality $\approx$) when, for all x and y, we have $f\,x \approx f\,y \to x \approx y$. Because $\approx$ appears both covariantly and contravariantly in this definition, we have implications in neither direction between definitional injectivity and propositional injectivity. Indeed, we can find examples of all four possibilities: constructors are injective in both senses; type formers, like `Fin` and

List, are definitionally injective but not propositionally injective; $\lambda\ (n : \mathbb{N}) \to n + n$ can be proven to be propositionally injective, but is not definitionally injective because $\_+\_$ is not injective; and nearly everything else is not injective in either sense.

Because the notions of definitional and propositional injectivity are incomparable, so too are the corresponding unification procedures. Propositional unification (using only the injectivity of constructors) is used during dependent pattern matching, while solving of implicit arguments and underscores in expressions is done by definitional unification.

2.1.5 Records, Σ-types

While Agda provides built-in basic Π-types, with special syntax described in section 2.1.2, it does not do the same for Σ-types. Instead, the default way to get the functionality of Σ-types is to declare record types, similarly to how we get sums via **data**-declarations. However, the standard library does provide Σ-types, via record types, using the following declaration.

```
record Σ {a b} (A : Set a) (B : A → Set b) : Set (a ⊔ b) where
  constructor _,_
  field
    proj₁ : A
    proj₂ : B proj₁
```

As does the standard library, I will begin to use universe level polymorphism in these example definitions. Here, a and b are the levels of the two projections. The level of the record type must be at least the level of the type of each field, and in this case, the smallest such level is $a \sqcup b$. As for the main points of interest in this **record**-declaration, it contains two fields. The first, proj_1, simply has type A. The second, proj_2, then has a type dependent on the value of the first field. Additionally, we give this record type a named constructor $\_,\_$. Any record type can also be constructed using the more verbose syntax **record** { proj_1 = { }1 ; proj_2 = { }2 }.

The standard library provides various notations for Σ-types, useful in various situations. In this thesis, I use $\Sigma[\ x \in A\ ]\ B\ x$ and $\exists \setminus\ x \to B\ x$ as equivalent notations for

Σ A B. Indeed, the η-contracted form can be used with ∃, as in ∃ B (\ is an alternative notation for λ, as in Haskell). Σ also specialises to non-dependent products, as given by the infix operator _×_. This is achieved by setting the parameter B to be a constant type family. The resulting operator _×_, as well as the non-dependent function type, behave better than their dependent counterparts with respect to unification because they allow us to remain in the first order fragment of higher order unification.

There are two main ways of using the fields of a record. The first is to put the projections into scope using **open** Σ, and then to use the field names to project out of arbitrary terms of Σ-type. This is what I will always do when using the Σ-type family. Within this paradigm, there are two further notational choices. Either, we can use the field names as functions, so that z = proj$_1$ z , proj$_2$ z, or we can use postfix projections via the space-dot notation, as in z = z .proj$_1$, z .proj$_2$. I tend to prefer the latter, also using it occasionally in ordinary mathematical notation (without the space). Both notations can also be used on the left-hand side of a clausal definition as *copatterns*. Copatterns let us think of records as being function-like, with the fields of a record type being the possible arguments we can pass to such a function.

The second way of using the fields of a record requires a motivating example. Consider the below definition of the type of semigroups at universe level ℓ. A semigroup has a carrier set, a binary operation on that set, and an associativity law for that binary operation.

```
record Semigroup ℓ : Set (suc ℓ) where
  infix 5 _•_
  field
    Carrier : Set ℓ
    _•_ : Carrier → Carrier → Carrier
    assoc : ∀ x y z → (x • y) • z ≡ x • (y • z)
```

In order to use the fields of Semigroup in the intended way, we do not open them into global scope. Doing so would mean that, for example, _•_ would take three arguments: the semigroup and its two intended arguments. Instead, we get to the point where we have a semigroup G in scope and use **open** Semigroup G to put into scope

the components *of* G. Then, the name Carrier in scope will refer to the carrier set of G, the name $\_\bullet\_$ will refer to the binary operator (which really takes two arguments), et cetera. Doing this gives the impression of working "inside" G, which is the way I typically work with algebraic sets with structure.

By η-equality, two inhabitants of a record type are definitionally equal exactly when they agree definitionally on all fields. This often makes record types much more convenient to work with than the corresponding single-constructor data types, which do not enjoy any η-laws. Notably, all inhabitants of the record type $\top$ with no fields are definitionally equal.

Along with Σ and $\top$, there are two more general-purpose record types I need to cover which take advantage of two special features of **record**-declarations (and also **data**-declarations, but I use **record**-declarations for the convenience reason given in the previous paragraph). The first feature is that the universe level of a record type has a lower bound (the level of each field) but no upper bound. Therefore, we can introduce the following declaration Lift, which takes a type A at level a and produces an equivalent type at a potentially higher level $a \sqcup \ell$. This type former is useful in situations which require a type at a specific level, such as when constructing a type using a function.

```
record Lift {a} ℓ (A : Set a) : Set (a ⊔ ℓ) where
  constructor lift
  field lower : A
```

The other interesting property we get from **record**-declarations is that the resulting type family is definitionally injective in its parameters. Therefore, record types behave well in the form of unification that solves implicit arguments. We can use this property to take any type family F and produce an equivalent family Wrap F which is definitionally injective.

```
record Wrap {a ℓ} {A : Set a} (F : A → Set ℓ) (x : A) : Set ℓ where
  constructor mk
  field get : F x
```

As an example, if we have a variable f : Wrap F y and pass it to a function with

a type of the form $\forall\ \{x\} \to \mathsf{Wrap}\ \mathsf{F}\ x \to \_$, Agda will successfully unify the type of f with the expected type of the argument, setting $[x := y]$. However, without the Wrap, we would need to unify F y with F x, which would fail if F were not injective, because there may be multiple acceptable values of x up to definitional equality.

The version of Wrap found in Agda's standard library is significantly more complicated to allow for type families with arbitrarily many arguments in a convenient syntax, using the n-ary functions of Allais [2019]. The version in the standard library is the one I use in this thesis. In fact, both versions of the Wrap type family are the first pieces of novel work to be presented in this thesis.

2.1.6 Colours

I use the "Conor colours" option for Agda syntax highlighting. This set of colours is inspired by Conor McBride's syntax highlighting for Epigram 2. The colour given to a name is determined by the type of declaration that name is bound to. The main colours are blue for types and type families, red for constructors of data types and fields of records, green for definitions which may unfold/compute, and *purple* for local variables.

Separately, I use green in many places for usage annotations in traditional typeset mathematical notation. This usage of green contrasts only with ordinary black text.

2.2 Term representation

The main use I have for Agda in this thesis is to represent and reason about programming languages. In particular, I am interested in representing *core languages* or *core calculi*. Given a standard multi-pass compiler, the core language can be identified as the first stage after which there can be no compile-time errors. In other words, (some) terms of the core language arise from taking the source code for a term, parsing it, scope-checking it, type-checking it, and doing whatever other static checks are done to it. Core language terms can also arise via optimisations on other core language terms, or could not actually appear in the compilation of any real program, but nonetheless

be potentially handled by the optimiser and backend. The core language can also be thought of as a representation of the *meaningful* part of a programming language, excluding (meaningless) erroneous programs.

In most compilers, the core language is represented as an annotated abstract syntax tree. By construction, the abstract syntax tree representation makes syntactically ill formed programs unrepresentable. When representing a core language in a dependently typed language, including Agda, we can take this idea further to make ill scoped, ill typed, and otherwise meaningless programs unrepresentable. Thus, we can truly *define* a core language, and moreover reason mathematically about it.

As is standard in the study of functional programming languages, I will take the core languages I consider to be variants of the simply typed λ-calculus [Barendregt, 1993, Church, 1940]. As part of the Curry-Howard correspondence [Howard, 1980], terms of a typed λ-calculus correspond to derivations in a natural deduction system, and therefore we can take inspiration in terms of definitions and methodology from logic when we come to mechanise core calculi in Agda. We could mechanise Gentzen's original definition of a natural deduction system directly, but this definition is quite complicated. In particular, if we want to give derivations an inductive definition, the use of the discharge mechanism means that we actually need an inductive-inductive type — derivations, particularly those using $\to$-introduction, can involve references to assumptions within their subderivations. An inductive-inductive definition of derivations would complicate our programs and proofs about natural deduction derivations, so I choose an alternative representation.

Indeed, most authors since Gentzen, whether mechanising their work or not, have opted to replace discharge of assumptions by explicit *contexts* and a variable rule. Contexts can be justified as a way to keep track of undischarged assumptions. In particular, we only produce derivations in the presence of a known collection of *free variables* specified by the context. In other words, derivations are *indexed* over their free variables and their types. When using an assumption within a derivation, we must say which free variable it corresponds to. Free variables are introduced by *variable-binding* rules, like $\to$-introduction. Figure 2.1 gives an example of the same derivation written in

Gentzen's style and in the explicit context style.

Explicit contexts can be seen as a mechanism for encoding a natural deduction system as a sequent calculus. However, the natural deduction character of the system is maintained by ensuring that the resultant sequent calculus is really an encoding of a natural deduction system. Concretely, this means that rules can only interact with the context in restricted ways:

- There is a designated *variable rule*, stating that any variable in the context can serve as a derivation of its type.
- Non-variable rules may only require subterms with *extended* contexts, i.e., subterms in which new variables have been bound. Non-variable rules are parametric in the existing free variables.

Having chosen to use explicit contexts, the mechanisation must have a chosen representation of contexts as a data structure. While the notation in figure 2.1 uses names f and x for variables, I opt for a nameless representation. In a nameless representation, variables are identified by their position in the context, rather than by a name. The absence of names means that α-equivalence is just on-the-nose equality, and also that we never have to reason about freshness of names. Agda does not have support for nominal techniques [Gabbay and Pitts, 2002], which may have made names a better option.

Most mechanisations choose contexts to be an inductive list of object-language types. However, I instead choose a functional, tree-shaped representation, as shown with the type Ctx. The type LTree is the inductive type generated by leaves ([-]) and nullary (ε) & binary (_<+>_) nodes, and serves as a generalised "length" of the context. The tree shape makes concatenation definitionally injective, so that in cases where multiple new variables are bound in a subterm (for example, $\otimes$-elimination), Agda's unification-based solving will be more able to infer which variables have just been bound. Within a given t : LTree, we can define the positions of t using Ptr. A *pointer* (Ptr) into a tree picks out a leaf ([-]) following a path of lefts ($\swarrow$) and rights ($\searrow$) at any binary nodes encountered.

$$\dfrac{\dfrac{[A \to A \to B]^f \quad [A]^x}{A \to B}\,{\to}\text{-E} \quad [A]^x}{\dfrac{B}{A \to B}\,{\to}\text{-I}^x}\,{\to}\text{-E}}{\dfrac{}{(A \to A \to B) \to (A \to B)}\,{\to}\text{-I}^f}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f : A \to A \to B, x : A \vdash A \to A \to B}\,\text{VAR}^f \quad \dfrac{}{f : A \to A \to B, x : A \vdash A}\,\text{VAR}^x}{f : A \to A \to B, x : A \vdash B}\,{\to}\text{-E} \quad \dfrac{}{f : A \to A \to B, x : A \vdash A}\,\text{VAR}^x}{f : A \to A \to B, x : A \vdash B}\,{\to}\text{-E}}{f : A \to A \to B \vdash A \to B}\,{\to}\text{-I}^x}{\vdash (A \to A \to B) \to (A \to B)}\,{\to}\text{-I}^f$$

Figure 2.1: A proof in Gentzen's natural deduction syntax, and a proof using explicit contexts (contexts coloured red)

```
data LTree : Set where
  [-] : LTree
  ε : LTree
  _<+>_ : (s t : LTree) → LTree

data Ptr : LTree → Set where
  here : Ptr [-]
  ↙ : ∀ {s t} → Ptr s → Ptr (s <+> t)
  ↘ : ∀ {s t} → Ptr t → Ptr (s <+> t)
```

I use Ptr to form a type family of LTree-indexed vectors Vector. These vectors serve as the data structure containing the types, and in later chapters usage information, of contexts. The advantages of the functional vector representation will not become clear until later chapters — particularly the example in section 8.1, where I make use of the ease of look-up and the η-law of functions. However, I claim for now that there is little to no disadvantage in the functional vector representation — in particular, we have no need for function extensionality principles because we never talk about equality of contexts. For example, instead of using an equality of contexts to coerce a term, we can use renaming.

```
Vector : Set a → LTree → Set a
Vector A s = Ptr s → A
```

The basic operations for building vectors from parts are [_] to create a singleton vector, [] to create an empty vector, and _++_ to append two vectors.

The shape of the context is usually not worth indexing over in term representation, so I hide this index using the record type Ctx. I also instantiate the element type of these vectors to Ty, the type of object-language types.

```
record Ctx : Set where
  constructor ctx
  field
```

```
    {shape} : LTree
    ty-ctx : Vector Ty shape
  open Ctx public
```

The three operators for building vectors lift to operations on Ctx by suffixing ^{c} — giving us $[\_]^c$, $[]^c$, and $\_{+}{+}^c\_$.

Our first data structure involving contexts is that of intrinsically typed variables. A variable of type $\Gamma \ni A$ is given by a path idx to a type in Γ, together with a proof tyq that this type is equal to A.

```
  record _∋_ (Γ : Ctx) (A : Ty) : Set where
    constructor el
    field
      idx : Ptr (Γ .shape)
      tyq : Γ .ty-ctx idx ≡ A
  open _∋_ public
```

Variables embed into terms via the var constructor of the family $\_\vdash\_$ of intrinsically simply typed terms. The only other syntactic forms we consider for now are the eliminator and constructor of function types $\_`\!\!\to\!\!\_$ — app and lam. Application app takes two subterms of the appropriate types, while the subterm of λ-abstraction lam is in an extended context $\Gamma\ {+}{+}^c\ [\ A\ ]^c$ — Γ concatenated with a singleton context containing the type A.

```
  data _⊢_ (Γ : Ctx) : Ty → Set where
    var : ∀ {A} → Γ ∋ A → Γ ⊢ A
    app : ∀ {A B} → Γ ⊢ A `→ B → Γ ⊢ A → Γ ⊢ B
    lam : ∀ {A B} → Γ ++ᶜ [ A ]ᶜ ⊢ B → Γ ⊢ A `→ B
```

Using this encoding, the Church numeral for 2 appears as follows. In standard notation, this would be $\lambda f.\ \lambda x.\ f\ (f\ x)$. To refer to f in the main body of the expression, we skip one binder (using ↙) and pick the next one (using ↘) and pick its only bound

variable (using here). To refer to x, we do not skip its binder, instead picking it and its only bound variable.

```
two : []ᶜ ⊢ (ι ‘→ ι) ‘→ (ι ‘→ ι)
two = lam (lam
  (app (var (el (↙ (↘ here)) refl))
    (app (var (el (↙ (↘ here)) refl)) (var (el (↘ here) refl)))))
```

2.3 Renaming and substitution

A basic operation on any syntax with variables is *substitution* — the replacement of variables in a term by terms with the same type as the variables. In a sense, this is the defining operation of variables — a variable is a placeholder for a term, or equivalently in logic, a hypothesis is a placeholder for an arbitrary proof. In a type theory or logic, terms can bind variables, and we will typically have operational semantics rules combining a term binding a variable with a term that is to be substituted into the place of that variable, like the β-rule for λ-calculus functions.

While substitution has this extra role in a lot of the syntaxes with binding we care about, variable-binding also significantly complicates the substitution operation. Substitution acts on the free variables of a term, replacing them by terms, but binders mean that some subterms have *more* free variables than our original term. This causes different challenges for different representations of terms. For example, with named variables and shadowing, naïvely defined substitution could fall foul of variable capture. In our approach, based on De Bruijn indices, the difficulty is that an index i outside a binder of n variables corresponds to an index $n + i$ inside the binder. Therefore, when substituting under a binder, we must first increment any free variables contained in terms we are substituting in, which is a form of *renaming*. Renaming replaces each free variable by another free variable, and is a special case of substitution. We must, however, define renaming before substitution, so as to avoid the definition of substitution being circular. Renaming avoids a similar circularity because when renaming goes under a binder, we only have to increment each variable being renamed in, rather than each

variable *in each term* being substituted in.

In this section, I formally implement simultaneous renaming and substitution for the terms defined in the previous section. Simultaneous substitution turns out to have a simple definition, which generalises into other algorithms over terms with binders. The section concludes with a unified implementation of renaming and substitution, leaving further generalisation to the next section.

2.3.1 Simultaneous renaming and simultaneous substitution

A simultaneous renaming from Γ to Δ is a type-preserving map from variables in Δ to *variables* in Γ, while a simultaneous substitution is a map into *terms* in Γ. While simultaneous substitution gives us a notion of one context being *derivable* from another, simultaneous renaming gives a similar notion of derivability restricted to structural rules.

In the derivation below, we assume the existence of a derivation of $B, C \to C \vdash C$, and by the admissibility of substitution we thus have a derivation of $A \to B, A \vdash C$. Intuitively, the context $A \to B, A$ derives the context $B, C \to C$, so anything derived from $B, C \to C$ can also be derived from $A \to B, A$. We see formally that $A \to B, A$ derives $B, C \to C$ by deriving each element of the latter from the former — hence the first two premises of the Subst rule below, deriving B and $C \to C$ from $A \to B, A$.

$$\dfrac{\begin{array}{c}\Pi\\ A \to B, A \vdash B\end{array} \qquad \dfrac{\dfrac{}{A \to B, A, C \vdash C}\,\textsc{Var}}{A \to B, A \vdash C \to C}\,{\to}\text{-I} \qquad B, C \to C \vdash C}{A \to B, A \vdash C}\,\textsc{Subst}$$

$$\text{where } \Pi := \dfrac{\dfrac{}{A \to B, A \vdash A \to B}\,\textsc{Var} \qquad \dfrac{}{A \to B, A \vdash A}\,\textsc{Var}}{A \to B, A \vdash B}\,{\to}\text{-E}$$

2.3.2 Proofs of admissibility of renaming and substitution

A renaming from Γ to Δ is a map from variables in Δ to variables in Γ, represented in Agda as follows.

```
Ren : (Γ Δ : Ctx) → Set
Ren Γ Δ = ∀ {A} → Δ ∋ A → Γ ∋ A
```

The action of a renaming ρ on terms is given by ren ρ, with ren defined below. The idea of simultaneous renaming is to preserve the structure of the term, but replace all of the variables from Δ by variables from Γ, with the mapping given by the renaming ρ.

```
ren : ∀ {Γ Δ A} → Ren Γ Δ → Δ ⊢ A → Γ ⊢ A
ren ρ (var x) = var (ρ x)
ren ρ (app M N) = app (ren ρ M) (ren ρ N)
ren ρ (lam M) = lam (ren (bindRen ρ) M)
```

The var case is where the action of the renaming happens: the variable x from Δ is mapped to the variable $\rho\ x$ from Γ. In the app case, we have terms $M : \Delta \vdash A \mathbin{`\!\!\rightarrow} B$ and $N : \Delta \vdash A$. We may apply ren ρ recursively to both M and N to change their contexts from Δ to Γ, and the app constructor then produces the desired term in Γ. Finally, in the lam case, we get a term $M : \Delta \mathbin{++^c} [\ A\ ]^c \vdash B$ and, after introducing a lam on the right, are in need of a term of type $\Gamma \mathbin{++^c} [\ A\ ]^c \vdash B$. To recursively apply ren to M, we must thus extend the renaming ρ : Ren Γ Δ with the newly bound variable. For this, we need an auxiliary function bindRen such that bindRen ρ : Ren ($\Gamma \mathbin{++^c} [\ A\ ]^c$) ($\Delta \mathbin{++^c} [\ A\ ]^c$). This new renaming will act like ρ for variables in Δ, and map the new variable of type A to the corresponding new variable in $\Gamma \mathbin{++^c} [\ A\ ]^c$.

```
bindRen : ∀ {Γ Δ Θ} → Ren Γ Δ → Ren (Γ ++ᶜ Θ) (Δ ++ᶜ Θ)
bindRen ρ (el (↙ i) q) = ↙ᵛ (ρ (el i q))
bindRen ρ (el (↘ i) q) = el (↘ i) q
```

The bindRen given here has a slightly generalised type, where instead of binding just a single variable of type A, we could bind a whole context Θ of new variables. The first case of bindRen is for old variables from Δ, where we apply ρ to get a variable in Γ, and then use $\swarrow^v$ to embed that variable into $\Gamma \mathbin{++^c} \Theta$. The second case is for new variables from Θ, which embed straight into $\Gamma \mathbin{++^c} \Theta$.

Meanwhile, a substitution from Γ to Δ is an inhabitant of Sub Γ Δ, as defined below. This definition is identical to the definition of Ren, except that it gives us *terms* in Γ rather than variables.

```
Sub : (Γ Δ : Ctx) → Set
Sub Γ Δ = ∀ {A} → Δ ∋ A → Γ ⊢ A
```

The sub function below gives the action of a substitution. Similarly to renaming, we want to preserve the structure of the term, except now variables in the original term are replaced by *terms* in the new context.

```
sub : ∀ {Γ Δ A} → Sub Γ Δ → Δ ⊢ A → Γ ⊢ A
sub ρ (var x) = ρ x
sub ρ (app M N) = app (sub ρ M) (sub ρ N)
sub ρ (lam M) = lam (sub (bindSub ρ) M)
```

Given that this time, ρ is a substitution rather than a renaming, ρ x is a term, and is sufficient in the var case. The app case again deals with the subterms recursively and then recombines them with app. In the lam case, we again have a mismatch if we want to apply sub recursively to the subterm M with an extra free variable. We have ρ : Sub Γ Δ but need a substitution of type Sub (Γ ++^{c} [A]^c) (Δ ++^{c} [A]^c), so we introduce the auxiliary definition bindSub.

```
bindSub : ∀ {Γ Δ Θ} → Sub Γ Δ → Sub (Γ ++ᶜ Θ) (Δ ++ᶜ Θ)
bindSub ρ (el (↙ i) q) = ↙ᵗ (ρ (el i q))
bindSub ρ (el (↘ i) q) = var (el (↘ i) q)
```

For the old variables in the first case, we have ρ to turn them into terms in Γ. Turning a term in Γ into a term in Γ ++^{c} Θ requires a form of weakening we have not yet proved, so I write ↙^{t} in analogy with ↙^{v}, and prove it below. In the second case, we want to substitute the new variable by the *term* referring to this new variable in Γ ++^{c} Θ.

The final piece to define substitution is to define the function that weakens a term by some newly bound variables Δ. For this, we use the action of renaming, which we

have fully defined already, and in particular rename each variable in the term from a variable in Γ to a variable in $\Gamma +\!\!+^c \Delta$.

```
↙ᵗ : ∀ {Γ Δ A} → Γ ⊢ A → Γ ++ᶜ Δ ⊢ A
↙ᵗ = ren ↙ᵛ
```

With this, the action of substitution is defined, and depends on the action of renaming.

2.3.3 Syntactic kits

As observed by Benton et al. [2012], McBride [2005], the statements of simultaneous renaming and simultaneous substitution are very similar, with substitution being the generalisation that allows replacement of variables by terms rather than just other variables. Following McBride [2005], I will introduce a type family Env of *environments*, and redefine Ren and Sub as environments of variables and terms, respectively.

```
Env : (K : Ctx → Ty → Set) (Γ Δ : Ctx) → Set
Env K Γ Δ = ∀ {A} → Δ ∋ A → K Γ A


Ren = Env _∋_
Sub = Env _⊢_
```

The processes I described for constructing proofs of the admissibility of renaming and substitution were also similar. Indeed, when we line up the resulting functions, ren and sub, and their auxiliaries, bindRen and bindSub, we notice only three key differences:

- In the first cases of bindRen and bindSub, we do $\swarrow^v$ and $\swarrow^t$, respectively, based on whether we are weakening a variable or a term.
- In the second case of bindSub, we do an extra wrapping of the new variable by var, so as to make it a term to go in the substitution.
- In the var case of ren, we have var ($\rho\ x$) rather than just $\rho\ x$, because the renaming ρ gives us a variable rather than a term.

We may abstract over these three differences using the record Kit. As in Env, we think of K as being either $\_\ni\_$ or $\_\vdash\_$. The fields of Kit are given in the same order as the points of difference above. Wherever the difference was presence or absence of var, we will be able to fill that field with either var or the identity function id.

```
record Kit (K : Ctx → Ty → Set) : Set where
  constructor kit
  field
    ↙ᵏ : ∀ {Γ Δ A} → K Γ A → K (Γ ++ᶜ Δ) A
    vr : ∀ {Γ A} → Γ ∋ A → K Γ A
    tm : ∀ {Γ A} → K Γ A → Γ ⊢ A
```

The field $\swarrow^k$ can be seen as a property of the judgement form K, saying that it supports a form of weakening. We use vr when adding a newly bound variable to an environment, and use tm when we do a lookup from an environment and want to get a term out. Given a Kit K, we can write the syntactic traversal function trav and its auxiliary bindEnv, in the model of ren, sub, and their auxiliaries.

```
trav : ∀ {Γ Δ A} → Env K Γ Δ → Δ ⊢ A → Γ ⊢ A
trav ρ (var x) = tm (ρ x)
trav ρ (app M N) = app (trav ρ M) (trav ρ N)
trav ρ (lam M) = lam (trav (bindEnv ρ) M)

bindEnv : ∀ {Γ Δ Θ} → Env K Γ Δ → Env K (Γ ++ᶜ Θ) (Δ ++ᶜ Θ)
bindEnv ρ (el (↙ i) q) = ↙ᵏ (ρ (el i q))
bindEnv ρ (el (↘ i) q) = vr (el (↘ i) q)
```

Concrete kits can be given for variables and terms either by inspecting ren and sub or by following the types. Notice that the kit for terms requires the admissibility of renaming so as to achieve weakening of a substitution by newly bound variables. Fortunately, this can be the ren defined below in terms of trav, so we can keep trav as the only syntactic traversal we have to write.

```
∋-kit : Kit _∋_
∋-kit = kit ↙ᵛ id var

ren = trav ∋-kit
```

```
⊢-kit : Kit _⊢_
⊢-kit = kit (ren ↙ᵛ) var id

sub = trav ⊢-kit
```

2.4 Generic semantics

The traversal trav from the last section is generic in the sense that $\mathcal{V}$, the type of entries in an environment, can be instantiated to many different things. However, in practice we only use $\ni$ and $\vdash$, giving us renaming and substitution, respectively. This is because trav only targets terms, and does so by keeping term constructors intact and replacing only the variables by things from the environment. This makes substitution the most general possible traversal.

If we want to capture a broader range of traversals, including not just syntactic but also *semantic* operations, we must be able to target things other than terms, and act in an interesting way on term constructors. Allais et al. [2017] show how to do this generalisation of trav to a function sem with the following type, where $\mathcal{C}$ is the type family we are targeting. In this section, I discuss the assumptions needed to implement such a function.

```
sem : ∀ {Γ Δ A} → Env 𝒱 Γ Δ → Δ ⊢ A → 𝒞 Γ A
```

Following the implementation of trav, we see that $\mathcal{C}$ will need to support a semantic counterpart of each syntactic form (var, app, and lam). With syntactic kits, we already asked for the field tm to interpret $\mathcal{V}$-values as terms. We rename tm to ⟦var⟧ to reflect its role in the semantic traversal sem. Now, we will also ask for fields to replace the right-hand side applications of the other term constructors. For application, we can stick with the obvious thing: we should be able to combine a semantic function and its semantic argument to get the semantic result.

```
⟦var⟧ : ∀ {Γ A} → 𝒱 Γ A → 𝒞 Γ A
⟦app⟧ : ∀ {Γ A B} → 𝒞 Γ (A ‘→ B) → 𝒞 Γ A → 𝒞 Γ B
```

However, we want to treat binding constructs specially, particularly because there are semantics with no notion of binding. We instead provide a function from values to computations that works in any *extension* of the current context. Keeping $\swarrow^k$ as before, we get the following semantic replacement for Kit.

```
record Semantics (𝒱 𝒞 : Ctx → Ty → Set) : Set where
  constructor kit
  field
    ↙ᵏ : ∀ {Γ Δ A} → 𝒱 Γ A → 𝒱 (Γ ++ᶜ Δ) A
    ⟦var⟧ : ∀ {Γ A} → 𝒱 Γ A → 𝒞 Γ A
    ⟦app⟧ : ∀ {Γ A B} → 𝒞 Γ (A ‘→ B) → 𝒞 Γ A → 𝒞 Γ B
    ⟦lam⟧ : ∀ {Γ A B} →
      (∀ {Δ} → 𝒱 (Γ ++ᶜ Δ) A → 𝒞 (Γ ++ᶜ Δ) B) → 𝒞 Γ (A ‘→ B)
```

With the aim of abstracting away from explicit contexts, bringing us closer to natural deduction, we can use some new notation to rephrase these requirements. We will work in Ctx → Set rather than Set. One of the basic connectives in this setting is the *pointwise* arrow $\_\dot{\to}\_$, which acts in Ctx → Set like the non-dependent arrow does in Set. Another basic component is the $\forall[\_]$ notation, which embeds Ctx → Set into Set by using an implicit Π-type to quantify over *all* contexts. Finally, at this stage, I introduce a modality ◯ encapsulating the pattern of considering arbitrary *extensions* of a context. To facilitate working in this point-free setting, I give infix versions of the families $\mathcal{V}$ and $\mathcal{C}$ (respectively $\_\mathcal{V}\vDash\_$ and $\_\mathcal{C}\vDash\_$). The principal use of these aliases is to fill the right argument with a type (occurring explicitly), and leave the left argument as _, i.e., a context given through the point-free machinery.

```
◯ : (Ctx → Set) → (Ctx → Set)
◯ T Γ = ∀ {Δ} → T (Γ ++ᶜ Δ)
```

```
record Semantics (𝒱 𝒞 : Ctx → Ty → Set) : Set where
  constructor kit
  infix 20 _𝒱⊨_ _𝒞⊨_; private _𝒱⊨_ = 𝒱; _𝒞⊨_ = 𝒞
```

field

```
↙ᵏ : ∀ {A} → ∀[ _𝒱⊨ A ⇢ ◯ (_𝒱⊨ A) ]
⟦var⟧ : ∀ {A} → ∀[ _𝒱⊨ A ⇢ _𝒞⊨ A ]
⟦app⟧ : ∀ {A B} → ∀[ _𝒞⊨ (A ‘→ B) ⇢ _𝒞⊨ A ⇢ _𝒞⊨ B ]
⟦lam⟧ : ∀ {A B} → ∀[ ◯ (_𝒱⊨ A ⇢ _𝒞⊨ B) ⇢ _𝒞⊨ (A ‘→ B) ]
```

To illustrate this definition, I will discuss a syntactic traversal — renaming — and a semantic traversal — a standard Set semantics.

For the renaming semantics, as with the renaming kit, we specify that environments hold variables (_∋_) and show that variables satisfy the required form of weakening ($\swarrow^v$). Meanwhile, whereas all syntactic kits target terms (_⊢_), with a semantic traversal we must specify the target. The fields ⟦var⟧ and ⟦app⟧ follow straightforwardly, with variables embedding into terms and a pair of terms of the right types giving an application term in the same context, via the relevant constructors. For the ⟦lam⟧ case, we are given b : ◯ (_∋ A ⇢ _⊢ B) Γ, and after producing a lam, are left needing a term in Γ ++^{c} [A]^c ⊢ B. That the type of b is wrapped in ◯ gives us the ability to use b in the extended context Γ ++^{c} [A]^c. In particular, we point at the new variable to yield the desired term in the same context.

```
RenSem : Semantics _∋_ _⊢_
RenSem .↙ᵏ v = ↙ᵛ v
RenSem .⟦var⟧ = var
RenSem .⟦app⟧ = app
RenSem .⟦lam⟧ b = lam (b (↘ᵛ here⁰))

ren = sem RenSem
```

To produce a Set semantics, we shift from targeting terms to targeting the interpretation of terms. In particular, ⟦ Γ ⊢ A ⟧ is the type of functions from the interpretation of Γ to the interpretation of A. The interpretation of a type is defined as usual, by recursion on the structure of the type. The interpretation of a context is the interpretation for each of its types. We still have environments storing variables, which delays

the interpretation of variables to the ⟦var⟧ case and allows newly bound variables to be referred to directly as variables, rather than fetching them up-front from an environment of interpretations. In the ⟦app⟧ case, we have $m : ⟦\ \Gamma\ ⟧\mathsf{Ctx} \to (⟦\ A\ ⟧\mathsf{Ty} \to ⟦\ B\ ⟧\mathsf{Ty})$ and $n : ⟦\ \Gamma\ ⟧\mathsf{Ctx} \to ⟦\ A\ ⟧\mathsf{Ty}$, and combine them in the usual way by distributing the interpretation of the context $\gamma : ⟦\ \Gamma\ ⟧\mathsf{Ctx}$. The ⟦lam⟧ case involves the same placement of the new variable into the environment as in RenSem. Finally, we get the function set from terms to their interpretations by passing in the identity ∋-environment id.

```
⟦_⟧Ty : Ty → Set
⟦ ι ⟧Ty = ℕ
⟦ A ‘→ B ⟧Ty = ⟦ A ⟧Ty → ⟦ B ⟧Ty
⟦_⟧Ctx : Ctx → Set
⟦ Γ ⟧Ctx = Lift_n ⟦_⟧Ty (Γ .ty-ctx)
⟦_⊢_⟧ : Ctx → Ty → Set
⟦ Γ ⊢ A ⟧ = ⟦ Γ ⟧Ctx → ⟦ A ⟧Ty

SetSem : Semantics _∋_ ⟦_⊢_⟧
SetSem .↙^k v = ↙^v v
SetSem .⟦var⟧ (el i refl) γ = γ .get i
SetSem .⟦app⟧ m n γ = (m γ) (n γ)
SetSem .⟦lam⟧ b γ x = b (↘^v here^v) (γ ++_n [ x ]_n)

set : ∀ {Γ A} → Γ ⊢ A → ⟦ Γ ⊢ A ⟧
set = sem SetSem id
```

The definition of Semantics above essentially enforces that the term being traversed and the result of the traversal share the same binding structure. Concretely, lam is the only case where we can bind new variables, and at that point we must do exactly one binding. This is fine for renaming and substitution, which preserve the binding structure, and also for a standard denotational semantics, which is sufficiently abstracted from binding. However, if we want to do other syntactic translations — for example, converting from a syntax with n-ary functions to a syntax with only unary Curried functions — it would be useful to allow more choices when going under a binder. To this

end, I replace the one-step binding modality ◯ by the all-possible-renamings modality □. □ $T\ \Gamma$ states that T holds not only at Γ, but also at any context Γ^+ containing Γ (including $\Gamma +\!\!+^c \Delta$ for any Δ).

```
□ : (Ctx → Set) → (Ctx → Set)
□ T Γ = ∀ {Γ⁺} → Ren Γ⁺ Γ → T Γ⁺
```

As well as the flexibility in binding structure, the □ modality allows us to use the somewhat more well behaved and standard relation of renaming, rather than strict context extension. The resulting definition of Semantics is as follows, and is simply a version of the previous definition where ◯ has been replaced by □. It will become apparent when we implement the traversal sem why the first field also changes to include a □.

```
record Semantics (𝒱 𝒞 : Ctx → Ty → Set) : Set where
  infix 20 _𝒱⊨_ _𝒞⊨_; private _𝒱⊨_ = 𝒱; _𝒞⊨_ = 𝒞
  field
    ren^𝒱 : ∀ {A} → ∀[ _𝒱⊨ A ·→ □ (_𝒱⊨ A) ]
    ⟦var⟧ : ∀ {A} → ∀[ _𝒱⊨ A ·→ _𝒞⊨ A ]
    ⟦app⟧ : ∀ {A B} → ∀[ _𝒞⊨ (A ‘→ B) ·→ _𝒞⊨ A ·→ _𝒞⊨ B ]
    ⟦lam⟧ : ∀ {A B} → ∀[ □ (_𝒱⊨ A ·→ _𝒞⊨ B) ·→ _𝒞⊨ (A ‘→ B) ]
```

Writing a □-based semantics is very similar to writing a ◯-based semantics, so I will only give one further example. I generalise the renaming example to derive a semantic traversal from any syntactic traversal. We need a slightly modified definition of Kit to provide *renaming* of $\mathcal{V}$-values, rather than just extension.

```
record Kit (𝒱 : Ctx → Ty → Set) : Set where
  constructor kit
  infix 20 _𝒱⊨_; private _𝒱⊨_ = 𝒱
  field
    ren^𝒱 : ∀ {A} → ∀[ _𝒱⊨ A ·→ □ (_𝒱⊨ A) ]
    vr : ∀ {A} → ∀[ _∋ A ·→ _𝒱⊨ A ]
```

```
    tm : ∀ {A} → ∀[ _𝒱⊨ A ⇢ _⊢ A ]
  open Kit
```

The interesting feature of the corresponding Semantics is that we now pass b the renaming $\swarrow^v$, projecting the original context Γ out of the lam-extended context $\Gamma +\!\!+^c [\ A\ ]^c$.

```
  kit→sem : ∀ {𝒱} → Kit 𝒱 → Semantics 𝒱 _⊢_
  kit→sem K .ren^𝒱 = K .ren^𝒱
  kit→sem K .⟦var⟧ = K .tm
  kit→sem K .⟦app⟧ = app
  kit→sem K .⟦lam⟧ b = lam (b ↙^v (K .vr (↘^v here^v)))
```

With Semantics fixed, I also give the corresponding implementation of sem. Like trav from section 2.3.3, we need a bindEnv function, but I have updated this to deal with renamings and the $\Box$-operator, and also to fit better with concepts important for generic *syntax* (as in section 2.5). bindEnv now takes the old environment ρ : $[\ \mathcal{V}\ ]\ \Gamma \Rightarrow^e \Delta$, a renaming r : Ren Γ^+ Γ, and an environment σ in the renamed context Γ^+ targeting the context extension Δr. In the lam case of sem, the ⟦lam⟧ field gives us exactly the renaming to pass to bindEnv, while the extension environment σ is made as a singleton environment containing just the A-value v.

```
  sem : ∀ {Γ Δ A} → Env 𝒱 Γ Δ → Δ ⊢ A → Γ 𝒞⊨ A
  sem ρ (var x) = ⟦var⟧ (ρ x)
  sem ρ (app M N) = ⟦app⟧ (sem ρ M) (sem ρ N)
  sem ρ (lam M) = ⟦lam⟧ λ r v →
    sem (bindEnv ρ r (λ { (el i refl) → v })) M

  bindEnv : ∀ {Δ Δr} →
    ∀[ [ 𝒱 ]_⇒^e Δ ⇢ □ ([ 𝒱 ]_⇒^e Δr ⇢ [ 𝒱 ]_⇒^e (Δ ++^c Δr)) ]
  bindEnv ρ r σ (el (↙ i) q) = ren^𝒱 (ρ (el i q)) r
  bindEnv ρ r σ (el (↘ i) q) = σ (el i q)
```

2.5 Generic syntax

We have seen in previous sections a method for defining well typed terms, providing them with the basic operations of renaming and substitution, and defining type-preserving semantic traversals over those terms. However, the Agda code we have seen only deals with one specific kind of terms — simply typed λ-calculus with a base type and function types. The aim of this section is to write some code to which we can pass a *description* or *signature* of a syntax and have it produce all of the same machinery.

The description of a syntax will closely resemble the logical rules Gentzen gave for natural deduction systems NJ and NK, but we give them a revised interpretation. Where Gentzen intended his rules to be applied schematically, and hypothetical proofs to be handled via *discharge* of hypotheses, we will take the rules formally to produce a system with explicit contexts and a variable rule. However, knowing that this resulting system came from such a description means that we can derive variable-handling features, such as substitution, in a generic way.

I will present a scheme based on the work of Allais et al. [2021] such that figure 2.2 is interpreted as the type system we studied in the previous sections (simply typed λ-calculus with a base type and function types). Remember that, while these look like inference rules, I am treating them entirely formally, collected together into a *syntax description*. The information presented in figure 2.2 is essentially all of the information needed for the type system sans any details about variables. In particular, notice:

- Contexts, in particular the context of a rule's conclusion, which is shared in all premises in the resulting type system, are elided. The only part of any context I record is the newly bound variables in premises, such as the variable bound by a λ-abstraction.

- There is no explicit variable rule. It is understood that any $x : A$ in the context of the resulting type system can be used to yield a term with type A.

Such a scheme commits us to a certain approach to variable binding and context management, but does not commit us to anything about the meaning of types. For example, we do not declare that APP and LAM are "elimination" and "introduction"

$$\frac{\vdash A \to B \quad \vdash A}{\vdash B}\ \textsc{app} \qquad\qquad \frac{A \vdash B}{\vdash A \to B}\ \textsc{lam}$$

Figure 2.2: An example syntax description

$$\begin{array}{rr} \text{Premises} & ps, qs ::= \Delta \vdash A \mid \ \mid ps \quad qs \\ \text{Rule} & r, s ::= \dfrac{ps}{\vdash A} \end{array}$$

Figure 2.3: The grammar of typing rules

forms for the function type former. This limits our generic results to matters of syntax and variables, but provides a platform upon which a future semantic scheme could rest.

A syntax description is a set of fully instantiated rules. In our running example, this set contains a APP-rule and a LAM-rule for each pair of types A and B.

To construct the syntax given by a description, we keep var as before, and have another constructor con for all of the logical rules. con takes a rule r with premises $\Delta_1 \vdash A_1; \ldots; \Delta_n \vdash A_n$ and conclusion A, and the remainder of its type is as follows.

$$\mathsf{con}\ r : \forall \Gamma.\ (\Gamma, \Delta_1 \vdash A_1) \times \cdots \times (\Gamma, \Delta_n \vdash A_n) \to \Gamma \vdash A$$

Note that, in this type, $\vdash$ is the type family of terms we are inductively constructing, as opposed to the description syntax found in the premises.

In our generic version of Semantics, we keep the ren^$\mathcal{V}$ and ⟦var⟧ fields as before, and replace ⟦app⟧ and ⟦lam⟧ by a ⟦con⟧ field as follows.

$$[\![\mathsf{con}]\!]\ r : \forall\Big[\ \Box\Big(\overset{\mathcal{V}}{\Longrightarrow} \Delta_1 \dot{\to} \overset{\mathcal{C}}{\models} A_1\Big) \dot{\times} \cdots \dot{\times} \Box\Big(\overset{\mathcal{V}}{\Longrightarrow} \Delta_n \dot{\to} \overset{\mathcal{C}}{\models} A_n\Big) \dot{\to} \overset{\mathcal{C}}{\models} A\ \Big]$$

I use $\overset{\mathcal{C}}{\models} A$ for the Agda notation _$\mathcal{C}\vDash$ A, while $\overset{\mathcal{V}}{\Longrightarrow} \Delta$ stands for the type family of environments $\lambda\ \Gamma \to$ Env $\mathcal{V}\ \Gamma\ \Delta$. Environments appear in this definition simply as a way to write a product of $\mathcal{V}$-values — one value for each element of Δ. We could make a special case of premises which do not bind any variables, as did Allais et al. [2021], eliding the $\Box$ and empty environment, but I choose not to for uniformity and simplicity

of presentation.

To generate the expressions involving ellipses, I give an interpretation of the formal rule descriptions. The interpretation is parametrised on some $, \_ ⟦ \_ ⊢ \_ ⟧ : \mathsf{Ctx} → \mathsf{OpenFam}$, where, in $, \Delta ⟦ \Gamma ⊢ A ⟧$, the context Δ stands for the newly bound variables of a premise, Γ is the context as it was below the rule's horizontal line, and A is the type of the premise. In the con constructor for terms, the parameter is $\Gamma, \Delta ⊢ A$, and in the ⟦con⟧ field for semantics, the parameter is $\Box\Big(\overset{\mathcal{V}}{\Longrightarrow} \Delta \dot{\to} \overset{\mathcal{C}}{\models} A \Big)\, \Gamma$.

A single premise with newly bound variables is interpreted by shuffling the parts into the right place, while multiple premises are interpreted as pointwise products of the individual premises (giving the ellipses above).

```
⟦_⟧p : Premises → Ctx → Set
⟦ ⟨ Δ ‘⊢ A ⟩ ⟧p Γ = , Δ ⟦ Γ ⊢ A ⟧
⟦ ‘i ⟧p = i
⟦ p ‘×̇ q ⟧p = ⟦ p ⟧p ×̇ ⟦ q ⟧p
```

A rule, with all its parameters instantiated, targets a specific type A', which we check to match the desired type A. Finally, a whole System comprises a set L of rule labels, and $rs : L → \mathsf{Rule}$. The interpretation of these data is to pick a rule label l, and then take the interpretation of the rule $rs\ l$. For the sake of defining terms as a least fixed point, it is important to note that the interpretation of a syntax description is strictly positive in the parameter $, \_ ⟦ \_ ⊢ \_ ⟧$.

```
⟦_⟧r : Rule → OpenFam
⟦ ps ‘⇾ A′ ⟧r Γ A = A′ ≡ A × ⟦ ps ⟧p Γ

⟦_⟧s : System → OpenFam
⟦ L ▷ rs ⟧s Γ A = Σ[ l ∈ L ] ⟦ rs l ⟧r Γ A
```

The interpretation of a system description as a single layer of syntax is functorial, supporting the map-s function when the parameter $, \_ ⟦ \_ ⊢ \_ ⟧$ is given as an extra argument named X or Y (which are both fixed as parameters of map-s, together with Γ and Δ).

```
map-s : (s : System) →
  (∀ {Θ A} → X Θ Γ A → Y Θ Δ A) →
  (∀ {A} → ⟦ s ⟧s X Γ A → ⟦ s ⟧s Y Δ A)
```

The implementation of map-s is straightforward, so I do not list it here. We preserve the shape of the syntactic layer, applying the function to each X we find (wherever the description contains $\langle\ \Delta\ `\vdash\ A\ \rangle$).

This map-s will be used in the generic syntax version of sem to recursively apply sem to all subterms. However, a major distinction between generic syntax and the specific syntax of previous sections is that the subterms found by map-s are not recognised by Agda's termination checker as *structurally smaller* than the original term. Therefore, a naïvely written sem will fail Agda's termination check.

To make sem pass the termination check, we have four major options:

1. Assert sem to be terminating, bypassing the termination check.
2. Use Agda's *sized types* [Abel, 2010] to remember that the subterms are smaller.
3. Avoid sized types, and index terms over some user-defined type (for example, natural numbers or ordinal notations) which is structurally smaller at subterms.
4. Inline a new, instantiated version of map-s wherever it is used.

Each of these approaches has drawbacks. Approach 1 is clearly unsafe, in the sense that the fundamental lemma sem is not being completely checked for type-correctness. Approach 2 is also unsafe, because Agda's sized type implementation is known to make the system inconsistent [Abel, 2015]. Meanwhile, approach 3 is safe, but entails a lot of manually extracted and supplied extra arguments, which I think would distract from the presentation and make the resulting code harder to use. Finally, approach 4 is safe, but limits code reuse (both of the function map-s itself and any lemmas we may prove about it). I choose to follow Allais et al. [2021] in using sized types, justified by the idea that Agda may eventually have a sound implementation of sized types, at which point I would want my code to be as easy to update for that new version of Agda as

possible. Fiore and Szamozvancev [2022] use approach 4, and in fact have only one use of (their equivalent of) map-s in their development.

Using sized types, my type family of System-generic terms is as below. Scope is a name for the transformation of an OpenFam into a Ctx → OpenFam which appends the extra context to the existing context before applying the original OpenFam (in this case, producing something like $\Gamma, \Delta \vdash A$ from $\vdash$). The Agda builtin ↑ produces a bigger size from an existing size sz, giving us here that the size of a term is 1 bigger than the size of all of its immediate subterms. Agda's elaborator and termination checker have special support for sizes, so we do not have to worry much about them from this point on.

```
data [_,_]_⊢_ (d : System) : Size → OpenFam where
  ‘var : ∀ {sz} → ∀[ _∋_                          ⇾ [ d , (↑ sz) ]_⊢_ ]
  ‘con : ∀ {sz} → ∀[ ⟦ d ⟧s (Scope ([ d , sz ]_⊢_)) ⇾ [ d , (↑ sz) ]_⊢_ ]
```

Corresponding to the generic ‘con constructor for terms, we have a generic field ⟦con⟧ in the updated Semantics record. In place of where one might expect Scope $\mathcal{C}$, we instead have Kripke $\mathcal{V}$ $\mathcal{C}$, with Kripke defined below. The form of Kripke follows from the shape we saw in the type of the ⟦lam⟧ field we saw in section 2.4, where I use an environment targeting Δ as a way to say "a value for each type in Δ".

```
Kripke : (𝒱 𝒞 : OpenFam) → Ctx → OpenFam
Kripke 𝒱 𝒞 Δ Γ A = □ ([ 𝒱 ]_⇒ᵉ Δ ⇾ [ 𝒞 ]_⊨ A) Γ


record Semantics (d : System) (𝒱 𝒞 : OpenFam) : Set where
  infix 20 _𝒱⊨_ _𝒞⊨_; private _𝒱⊨_ = 𝒱; _𝒞⊨_ = 𝒞
  field
    ren^𝒱 : ∀ {A} → ∀[ _𝒱⊨ A ⇾ □ (_𝒱⊨ A) ]
    ⟦var⟧ : ∀[ 𝒱 ⇾ 𝒞 ]
    ⟦con⟧ : ∀[ ⟦ d ⟧s (Kripke 𝒱 𝒞) ⇾ 𝒞 ]
```

Finally, we get a generic semantic traversal as follows. The function bindEnv is unchanged from section 2.4, as it never mentions the syntax. The type of the traversal

sem is also basically unchanged — we just need to account for arbitrary term sizes (sz), which will get smaller when recursing on subterms. I have chosen, as did Allais et al. [2021], to define sem mutually with a function body, which is like a counterpart to sem dealing with newly bound variables. Note that the mutual recursion is not essential — for example, body could simply be inlined. The ‘var case of sem is as before. The ‘con case, if viewed appropriately, is a direct generalisation of the lam case from earlier. We recursively apply sem to all immediate subterms contained in M (as found by map-s), with an environment updated to reflect the newly bound variables in each premise of the rule that was applied.

```
sem : ∀ {Γ Δ} → [ 𝒱 ] Γ ⇒ᵉ Δ → ∀ {sz A} →
  [ d , sz ] Δ ⊢ A → Γ 𝒞⊨ A
body : ∀ {Γ Δ} → [ 𝒱 ] Γ ⇒ᵉ Δ → ∀ {sz Θ A} →
  Scope [ d , sz ]_⊢_ Θ Δ A → Kripke 𝒱 𝒞 Θ Γ A

sem ρ (‘var v) = ⟦var⟧ (ρ v)
sem ρ (‘con M) = ⟦con⟧ (map-s d (body ρ) M)

body ρ M r σ = sem (bindEnv ρ r σ) M
```

2.6 Related work

There is a vast literature on formalisations of syntaxes with binding, which I cannot possibly do justice to in a reasonably sized thesis chapter. Instead, I limit myself to comparisons of the Allais et al. [2021] method I follow in this thesis to just its closest related work.

2.6.1 Autosubst

Schäfer et al. [2015] present the system *Autosubst*, which provides various tools for working with syntaxes with binding in the Coq proof assistant. Autosubst is based on similar ideas to those Allais et al. use: De Bruijn-indexed terms with a distinguished variable rule and notion of binding, acted upon by simultaneous renaming and substitution.

The simplest differences are essentially matters of choosing the encoding that best fits the proof assistant being used. Coq users tend to prefer using unindexed types and propositions indexed over them — in this case, a type of unscoped and untyped terms plus a "well typed" predicate — whereas Agda users prefer to work with only well formed data (well scoped and well typed terms). The latter approach more readily allows us to show generically that substitution preserves scoping and typing, but the former approach, conversely, allows for bespoke proofs of such facts. For example, one theorem of Schäfer et al. [2015] is type preservation for CC_ω, a dependent type system we cannot express using the machinery of Allais et al. [2021]. In principle, one could use Allais et al.'s machinery as the basis of a similar bespoke proof, but as far as I am aware, this has not been tried.

Another main difference is that Autosubst is presented to the user largely as a black-box implementation of substitution and related lemmas, in contrast to Allais et al.'s work exposing the Semantics bundle to the user, and having substitution be just one instance. Allais et al. [2017] and Allais et al. [2021] provide many examples of traversals over syntax using the same generic environment management as used by substitution. However, the focus on substitution in Autosubst has meant that reasoning about substitutions has been given more developed support. For example, the library provides a tactic `autosubst` which automates many equational proofs involving substitutions based on the σ-calculus of Abadi et al. [1991].

An interesting feature of Autosubst is *heterogeneous substitution*. The motivation for heterogeneous substitution is to handle systems like system F, where types and terms are syntactically distinct, but both feature binding and require a substitution operation. Furthermore, binding and substitution of types also affects the syntax of terms, thanks to Λ terms. Allais et al. provide no direct equivalent to heterogeneous substitution, and it is unclear how well their work can handle polymorphic calculi.

Kaiser et al. [2018] propose some modifications to Autosubst which, as far as I can tell, have not yet been incorporated, although all of the case studies in the paper are mechanised in Coq. In the paper, they adapt and extend the work of Allais et al. on generic semantic traversals to cover a variant of system F. They use the term *multivari-*

ate traversal for the generalisation of heterogeneous substitution to semantic traversals. It appears that this work could be followed through to produce syntax descriptions covering polymorphic calculi, which would provide a route for this work to be incorporated into the Autosubst library.

2.6.2 Second order abstract syntax

Marcelo Fiore and various collaborators have a long line of work aiming for a categorical account of variable-binding [Fiore et al., 1999, Fiore, 2008, Fiore and Hamana, 2013, Fiore and Hur, 2010, Fiore and Mahmoud, 2010]. A recent, particularly relevant paper from this line of work is that of Fiore and Szamozvancev [2022], which mechanises some of this work to obtain a framework similar in scope to the work of Allais et al. [2021]. However, there are several differences between the resulting mechanised frameworks, of methodological, mathematical, and technical nature.

Though Allais et al. [2021] did not state their results in categorical terms, it is still useful to infer what the category-theoretic statement would have been, and compare it to the statement actually given by Fiore et al. [1999] and Fiore and Szamozvancev [2022]. When we do this, we see that while Fiore et al. deal with the category of contexts under renaming, and presheaves on that category. This means that every model must be shown to respect renaming before touching the framework of Fiore and Szamozvancev. Meanwhile, Allais et al. make use of the interplay between the discrete category of contexts and the category of contexts under renaming. For example, the models (Semantics) of Allais et al. [2021] require only that the family of semantic values $\mathcal{V}$ be shown to respect renaming, while the fact that the result family $\mathcal{C}$ respects renaming follows as a corollary of the traversal function trav. Fiore and Szamozvancev make no distinction between $\mathcal{V}$ and $\mathcal{C}$, essentially only having $\mathcal{C}$, but requiring it to respect renaming before getting the morphism to that model from the initial model (the syntax). In particular, this interplay allows us to derive renaming for terms in the way we saw in section 2.5 — by making $\mathcal{V}$ the family of variables, which trivially respects renaming.

Perhaps the relative complexity of the categorical account of the work of Allais et al. [2021] is why the authors decided not to state it in such terms. However, it is also likely

that Allais et al. developed their work in quite a different style to how Fiore et al. did, despite arriving at similar theories. The work of Allais et al. [2021] is designed first and foremost to facilitate programming language mechanisations, and thus pays a lot of attention to syntax and traversals of it, making sure that the results compute well in Agda. On the other hand, Fiore et al. started from theoretical investigations about the category of models of theories with binding, and only applied their work to the mechanisation of programming languages in the much later work of Fiore and Szamozvancev [2022].

In terms of the underlying theory, both works extend multi-sorted universal algebrawith variable-binding. However, the two extensions are subtly different. Universal algebra already has a notion of variable, which supports renaming and substitution, and which allows a given term to be evaluated when the free variables of that term are assigned semantic values. Allais et al. reuse this notion of variable, and allow binding of such variables in terms. On the other hand, Fiore and Szamozvancev recast the existing variables as *metavariables*, and introduce a new notion of (bindable) variables separately into the syntax. The resulting metavariables can then stand for arbitrary *open* terms, and thus each one remembers its context and requires an explicit substitution whenever it is used.

Fiore and Szamozvancev use metavariables to form descriptions of relations over terms, like an equational theory over the simply typed λ-calculus. Terms have, as well as their context of variables, a context of metavariables, and terms can contain a metavariable wherever they could contain a subterm. There is then an operation of metavariable substitution, which substitutes terms in the place of metavariables. Metavariable substitution is used to instantiate the rules of described relations/theories. In contrast, Allais et al. make do with the variables of the metalanguage (i.e. Agda variables).

More concrete work by McLaughlin et al. [2018], building on the approach to semantics of Allais et al. [2017] and section 2.4, also deals with syntactic contexts into which terms can be substituted (as part of developing notions of contextual equivalence). However, this work makes use of several different notions of syntactic context,

rather than just the one given naturally by metavariables in the framework of Fiore and Szamozvancev. This suggests that more research is needed about the various roles of metavariables before any particular approach is standardised.

Technologically, Fiore and Szamozvancev provide an external domain-specific language for syntax descriptions. From such a description, a Python program generates some boilerplate Agda code providing the types, the algebraic signature, the well typed terms, and a proof that the terms are the initial model. Using code generation like this resembles parts of the Autosubst plugin, as opposed to the purely internal, generic programming-based solution given by Allais et al. and in this thesis.

Fiore and Szamozvancev [2022, p. 19] avoid sized types by defining their equivalent of the functorial mapping map-s specialised to sem, with these two functions being mutually recursive. This is a standard solution, where code reuse is traded away in favour of satisfying the termination checker and avoiding sized types.

Despite all of these differences, I chose to base the work of this thesis on that of Allais et al. [2021] for largely circumstantial reasons. I only became aware of the work of Fiore and Szamozvancev [2022] when it was published, by which time I had already completed the bulk of the work presented in this thesis. Also, one of the authors of the former paper is my PhD supervisor, and the other authors too were working nearby, so it was easy to discuss their work quickly and informally. I think it is clear what it would mean to adapt the later work of this thesis to a framework in the style of Fiore and Szamozvancev rather than Allais et al., but it may not be obvious how usage restrictions (for ordinary variables) and metavariables should interact.

2.6.3 Substitution-based semantics

A feature shared by the frameworks of Allais et al. [2021] and Fiore et al. [1999] (and the rest of the work described in section 2.6.2) is that they are both often concerned about how renamings act upon semantics/models. The basic currency of both systems is presheaves on the category of contexts under renaming, and when the user wants to produce a semantics, they must show that the desired semantics forms such a presheaf — amounting to showing that the relevant type family respects renamings (in the direction

that may introduce new, unused variables to the context).

In contrast, the work of Hirschowitz et al. [2022] is based around the category of contexts under *substitution* or, more precisely, the morphisms given by semantic environments of whatever semantics is being defined. The focus on substitution makes this work simpler than renaming-based frameworks, because we get to avoid talking about renaming, whereas we always ultimately need to talk about substitution. However, what is a gain for internal simplicity is a loss for usability. For the syntax, starting from substitution means that the framework provides no help in *proving* the admissibility of substitution, because the syntax is not considered a model *until* we can show that it admits substitution. Similarly, for semantic models, we have to prove that each model we consider respects semantic substitution, which is a stronger requirement than respecting renaming.

Additionally, Hirschowitz et al. [2022] gain further simplicity of definitions by not keeping track of contexts. For example, untyped substitutions on a set (as opposed to scope-indexed family) X are functions from $\mathbb{N}$ to X. Similarly, the simply typed families introduced in section 6 of the aforementioned paper are indexed on their type, but not their context. Ignoring scopes/contexts lets one talk about monads (as in *De Bruijn monads*) rather than a more complex notion like relative monads [Altenkirch et al., 2015]. However, it also means losing potentially useful and important information.

2.6.4 Nominal techniques

There have been essentially two approaches when it comes to incorporating nominal techniques into proof assistants. The first is to develop a new foundational theory and develop a new proof assistant on top of it. The second is to take an existing proof assistant and provide a library, possibly based on unsafe features.

The best-developed nominal foundational theory at the time of writing is *Fraenkel-Mostowski (FM) set theory*, as introduced by Gabbay [2001] and Gabbay and Pitts [2002]. Gabbay's thesis presents FM set theory, and then uses it as the basis for a proof assistant Isabelle/FM and programming language FreshML. FM is a variant of ZF set theory containing an infinite set $\mathbb{A}$ of *atoms* or *names*, and the *equivariance axiom*,

stating that every FM proposition is invariant under consistent permutation of $\mathbb{A}$. In this setting, one can define the quantifier $\mathcal{N}a.\ \phi$, read as introducing a *new* or *fresh* name a to be used in the proposition ϕ. As a material set theory, FM is unusual in that it refutes the axiom of choice. In terms of applications to proof assistants, this means that FM is incompatible with much existing Isabelle-based work, including anything using the Hilbert ε-operator, which is used liberally in Isabelle/HOL. For example, if the syntax of a programming language is formalised in FM, then there is no formal connection to the kind of foundations in which interesting denotational semantics have likely been formalised.

Related to nominal set theories is the recent work of Pitts [2023] about theories of *locally nameless sets*. This behaves quite similarly to nominal theories, in that free variables have names, and renaming is given by primitive operations. In fact, Pitts [2023] shows that every locally nameless set is a nominal set.

The other approach — implementing a nominal library within an existing proof assistant — is exemplified by the Nominal Isabelle library of Urban [2008]. This work is based around having a countably infinite discrete type `name` (which could be a type of natural numbers or strings, for example) which is treated abstractly, and then defining liftings of permutations on `name` to permutations on any other types involving `name`s. Because these liftings are defined explicitly within Isabelle/HOL, the Nominal Isabelle setup amounts to an explicit model of a nominal logic within Isabelle/HOL. This setup resembles what Allais et al. [2021] did within Agda, and similarly what I do later in this thesis — creating a library within an existing proof assistant. The main downside, compared to working internally to a nominal set theory, is that the external constructions of Nominal Isabelle are more complicated, and involve keeping track of more properties that do not come as part of the theory the proof assistant understands.

There has also been an effort to replicate the Nominal Isabelle work in Coq by Aydemir et al. [2006]. However, this work appears to have been abandoned before publication, and the approach is unclear from the work-in-progress paper. Nominal libraries have also been made in general-purpose programming languages, like in Haskell with the nom package [Gabbay, 2020].

From a broader perspective, I claim that nominal techniques solve a subtly different problem to that solved by the likes of Autosubst, the work of Fiore, and the techniques discussed in detail in this chapter. The names found in nominal techniques are more general than (names of) variables in formal systems. This means that nominal techniques have been applicable to problems outside standard programming language theory, such as in the representation of graphs and in topology [Gabbay and Gabbay, 2017]. However, in certain ways, this is arguably not a good fit for type systems. In a basic nominal formalisation of a syntax with binding, there is no real distinction between the context of a term and the collection of that term's free variables. This, for example, leaves no natural place to put the types of variables, except as either a partial function from the set of names to types, or from the computed set of free variables to types. The discrepancy becomes even more apparent in substructural systems, as we see later, where we want tight control over the context, as opposed to the scope.

2.6.5 Logical frameworks

Logical frameworks based on higher order abstract syntax are another approach which has been used to mechanise the metatheory of logics and programming languages. Primary among logical frameworks in recent study is the Edinburgh Logical Framework [Harper et al., 1993], also known as *LF*. The underlying theory of LF is a dependent type theory λ^{Π} featuring *weak* function spaces. Weak function spaces differs from the usual strong function spaces in that functions in a weak function space cannot inspect their arguments — only place them whole in their result. Therefore, variables of the logical framework can be used as if they were the variables of the object language.

Below I give two example constructors of a type family tm. These declarations are similar to the rule descriptions I gave in figure 2.2 for their lack of explicit contexts. In this example, Π is the type former for dependent weak function spaces, with $\to$ being the non-dependent specialisation of Π. The quoted arrow $`\to$ is a type former of the object language, as in the previous examples in this chapter.

$$\text{lam} : \Pi A, B.\ (\text{tm}\, A \to \text{tm}\, B) \to \text{tm}\, (A \mathbin{`\!\to} B)$$
$$\text{app} : \Pi A, B.\ \text{tm}\, (A \mathbin{`\!\to} B) \to \text{tm}\, A \to \text{tm}\, B$$

Like several other approaches I have described in this chapter, logical frameworks aim to manage variables, variable binding, and contexts in a generic way, giving the user only the choice of what logical rules they want to add to a basic calculus. In fact, logical frameworks go further than any other approach I have described, never giving the user reified access to the context of object language terms, and giving immediate access to (single) substitution simply as application of weak functions. LF also natively supports more complex calculi than the simply typed λ-calculus — for example allowing us to implement and reason about System F.

The main drawback of logical frameworks is that making one requires making a new proof assistant, typically with no compatibility with existing proof assistants and their libraries of useful mathematical definitions and proofs. Particularly, if you want to consider calculi not (conveniently) expressible in λ^{Π} (as I do, with substructural calculi), then you need to make a new logical framework which is probably also incompatible with existing logical frameworks. As such, no single logical framework acts as a convenient and natural foundation upon which to build a broad range of mathematics including substructural logics and their metatheory.

Chapter 3

Linearity and modality

The techniques of chapter 2 were developed to handle idealised core calculi, like the simply typed λ-calculus and its many variants. Such core calculi can be seen as simplified models of real programming languages and/or their intermediate representations. A potential programming language feature — required for many desired semantics but only relatively recently appearing in programming languages — that is not handled by the techniques of chapter 2 is restrictions on how variables can be used. In the simple type systems covered in chapter 2, the only restriction on the use of a variable is that it is used as an expression of the appropriate type. However, we could also consider having somehow "guarded" subexpressions in which some variables bound outside cannot occur, or systems in which the use of a variable in one part of an expression precludes its use in another part of that expression (the variable having been "used up"). I will introduce and motivate some such systems in this chapter, and argue that they are impossible to capture directly in the framework presented in section 2.5.

Languages and systems implementing features inspired by linear logic include Rust [Matsakis and Klock, 2014, Rust team, 2023], Granule [Orchard et al., 2019], ATS [Xi, 2004, Zhu and Xi, 2005], an experimental Haskell extension [Bernardy et al., 2017], and various implementations of session types [Hüttel et al., 2016]. Additionally, forms of linearity have been used in theoretical work, for example to bound computational complexity [Girard et al., 1992, Hofmann, 2003] or to write programs which support incremental updates to computations [Ehrhard, 2018, Ehrhard and Regnier,

2003]. As for modality, Ivašković et al. [2020] show that common data-flow analyses for imperative languages can be recast as modal type systems, though such analyses have generally been developed in an ad hoc fashion. Such a recasting may be useful to understand the metatheoretic properties of programs passing such static analyses. More user-facing implementations of modal type systems include the system for stack allocation in OCaml [Dolan and White, 2022] and the approach to programmer-annotated erasure found in Agda and Idris [Atkey, 2018, Brady, 2021].

In this chapter, I look at two standard type-theoretic features that motivate considering calculi going beyond simple type theories as delineated in chapter 2 or equivalently by Allais et al. [2021]. The first feature is a $\Box$-modality — specifically, the $\Box$-modality of intuitionistic S4, which I discuss in section 3.1. The second feature is linearity, which I discuss in section 3.2. In particular, I discuss possible syntaxes for each, with a mind to being able to apply the techniques described in chapter 2. After having introduced these topics, I finish this chapter with a survey of prior work on representing such type theories in proof assistants in section 3.3.

3.1 Intuitionistic S4 modal logic

Modal logics, and in particular their modalities, are usually presented in philosophical and mathematical logic in an axiomatic style. For example, the common modal logic S4 may be presented by taking a presentation of classical logic, adding a unary $\Box$ ("box") operator to the formulas, keeping the logical rules as-are, and adding all of the axioms and rules listed in figure 3.1. Note that, unlike elsewhere in this thesis, the empty context in the necessitation rule N is a proper restriction: While the axioms hold in any context, thanks to weakening, and ordinary logical rules hold in any context, the necessitation rule only holds in the empty context. We may apply weakening to the conclusion of N to embed such a derivation into a larger derivation, but we may not use hypotheses from the outside in such a subderivation.

In this thesis, I am working with intuitionistic logics, so I start with a base of intuitionistic logic, and study intuitionistic S4 (IS4). The axiomatic presentation can be

$$\frac{\vdash A}{\vdash \Box A}\ \mathrm{N} \qquad \frac{}{\vdash \Box(A \to B) \to (\Box A \to \Box B)}\ \mathrm{K} \qquad \frac{}{\vdash \Box A \to A}\ \mathrm{T} \qquad \frac{}{\vdash \Box A \to \Box(\Box A)}\ 4$$

Figure 3.1: The axioms and rules required in a traditional presentation of S4

introduced on top of various presentations of intuitionistic/classical logic — including natural deduction, sequent calculi, Hilbert systems, and abstractly taking the set of intuitionistic/classical tautologies. For concreteness and rigour, I will assume a natural deduction system NJ allowing for explicit weakening.

The axiomatic presentation is convenient for talking about semantics, with different choices of axioms corresponding fairly directly to natural restrictions on models. However, the syntax is quite poorly behaved, at least for programming language applications. As mentioned earlier, the necessitation rule makes requirements of its context, which makes it incompatible with the methods discussed in chapter 2. Essentially, we are required to make necessitation a special case in renaming, substitution, and all the other traversals. Additionally, axioms require us to modify the notion of canonical form [Prawitz, 1965, p. 79], and redo any proofs that rely on all closed terms of function type being λ-abstractions.

The methodology I use to produce a nicer syntax is that of Pfenning and Davies [1999]: capturing the desired modality in the judgemental structure of the syntax. To start, note the following two facts about a "promotion" rule similar to one mentioned by Benton et al. [1993].

Proposition 3.1.1. When $\Gamma = B_1, \ldots, B_n$, let $\Box\Gamma$ abbreviate $\Box B_1, \ldots, \Box B_n$. Then, in NJ + N + K + 4, the following principle is admissible.

$$\frac{\Box\Gamma \vdash A}{\Box\Gamma \vdash \Box A}\ \textsc{Promotion}$$

Proof. By induction on n. When $n = 0$, Promotion becomes just the necessitation rule N. For $n = |\Gamma| + 1$, we have the following derivation, where $4'$ and K' abbreviate the obvious combinations of 4 and K, respectively, with $\to$-E. I silently apply weakening

throughout the derivation for the sake of concision.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Box\Gamma, \Box B \vdash A}{\Box\Gamma \vdash \Box B \to A}\,{\to}\text{-I}}{\Box\Gamma \vdash \Box(\Box B \to A)}\,\text{IH} \qquad \dfrac{\dfrac{}{\Box B \vdash \Box B}\,\textsc{Var}}{\Box B \vdash \Box\Box B}\,4'}{\Box\Gamma, \Box B \vdash \Box A}\,\text{K}'$$

□

Proposition 3.1.2. In NJ + Promotion + T, we can derive the original axioms and rule of S4: N, K, T, and 4

Proof. The rule N and axiom T are immediate. For axioms K and 4, we have the following derivations, where T′ abbreviates the obvious combination of T and →-E.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Box(A \to B) \vdash A \to B}\,\text{T}' \qquad \dfrac{}{\Box A \vdash A}\,\text{T}'}{\Box(A \to B), \Box A \vdash B}\,{\to}\text{-E}}{\Box(A \to B), \Box A \vdash \Box B}\,\textsc{Promotion}}{\Box(A \to B) \vdash \Box A \to \Box B}\,{\to}\text{-I}}{\vdash \Box(A \to B) \to \Box A \to \Box B}\,{\to}\text{-I}$$

$$\dfrac{\dfrac{\dfrac{}{\Box A \vdash \Box A}\,\textsc{Var}}{\Box A \vdash \Box\Box A}\,\textsc{Promotion}}{\vdash \Box A \to \Box\Box A}\,{\to}\text{-I}$$

□

Propositions 3.1.1 and 3.1.2 suggest that a Promotion-like rule to introduce the □ operator and a T-like rule to eliminate the □ operator may be a possible approach to capture S4 in a natural deduction style. However, we do not want to add such a rule to our system because it is not stable under substitution. For example, let $\sigma : \Box A \wedge \Box B \overset{\vdash}{\Longrightarrow} \Box A, \Box B$ be the substitution whose two components are given by left projection and right projection, respectively. The substitution principle says that we should be able to turn any derivation of $\Box A, \Box B \vdash \Box C$ into a derivation of $\Box A \wedge \Box B \vdash \Box C$. However, if we concluded $\Box A, \Box B \vdash \Box C$ by Promotion, then it becomes unclear how to conclude $\Box A \wedge \Box B \vdash \Box C$. In particular, Promotion does not apply because the formula $\Box A \wedge \Box B$ is headed by $\wedge$, rather than □.

Instead of constraining the types of all of the items in the context, the solution given by Pfenning and Davies [1999] is to diversify the shapes of contexts, and constrain those

$$\frac{\Gamma_v \vdash A\ true}{\Gamma \vdash \Box A\ true}\ \Box\text{-I} \qquad \frac{\Gamma \vdash \Box A\ true \qquad \Gamma, A\ valid \vdash B\ true}{\Gamma \vdash B\ true}\ \Box\text{-E} \qquad \frac{\Gamma \ni (A\ valid)}{\Gamma \vdash A\ true}\ \textsc{vVar}$$

Figure 3.2: The new rules of the Pfenning and Davies presentation of IS4.

shapes when giving our necessitation rule. In this case, we are interested in hypotheses being $\Box$-like, so we introduce a judgement form $A\ valid$, contrasting with the judgement we had previously written just A, but will now write $A\ true$. We construct the system so that $A\ valid$ is equivalent to $\Box A\ true$. Because the choice of *valid* or *true* is part of the structure of the context, rather than being part of the type, we are able to define substitution between contexts in a way that treats valid and true entries differently — restricting substitution so that it is compatible with an adapted version of Promotion.

Let us take the natural deduction presentation of the simply typed λ-calculus given in section 2.2. Where that system has sequents of the form $A_1, \ldots, A_n \vdash B$, we instead now write $A_1\ true, \ldots, A_n\ true \vdash B\ true$. We may also have hypothetical occurrences of $A\ valid$, but all logical rules will target judgements of the form $A\ true$. Hypotheses of the form $A\ valid$ arise from the elimination rule for the $\Box$ operator, $\Box$-E. The only special thing about valid hypotheses is that, once they are bound, they are preserved by the $\Box$-I rule, which is our recasting of Promotion as the introduction rule for $\Box$. Otherwise, like true hypotheses, an $A\ valid$ hypothesis can be used to obtain $A\ true$, via the vVar rule, as justified by axiom 4.

The three new rules are listed in figure 3.2. Let Γ_v be the context Γ but with all true hypotheses removed, leaving only valid hypotheses. Because I am working with a De Bruijn index-style presentation, weakening needs to be admissible. Therefore, the $\Box$-I rule needs to remove its true hypotheses, rather than only applying in a context in which there are no true hypotheses.

Whereas Pfenning and Davies [1999] prove single substitution for both true and valid hypotheses (as well as *possible* hypotheses, and for possible judgements, in their system dealing with a possibility modality $\Diamond$) separately, I build on the simultaneous substitution procedure given in section 2.3.3 to give a unified simultaneous substitution procedure. Recall that the basic definition was that of $\vDash$-*environment*, defined in the

intuitionistic case by $\Gamma \overset{\vDash}{\Longrightarrow} \Delta \coloneqq \Pi A.\ \Delta \ni A \to \Gamma \vDash A$. Essentially the same formula deals with true hypotheses, but for valid hypotheses, we need to think of something new. We want to be able to instantiate $\vDash$ to $\vdash$ so as to derive substitution, but $\Gamma \vdash A\ \mathit{valid}$ is not a valid sequent in our system. Instead, I consider $\Gamma \vdash \Box A\ \mathit{true}$, and in particular the canonical way of deriving such a sequent: the $\Box$-I rule. $\Box$-I says that we can get $\Gamma \vdash \Box A\ \mathit{true}$ from $\Gamma_v \vdash A\ \mathit{true}$. I take the form of latter sequent to replace $\Gamma \vDash A\ \mathit{valid}$, giving the following definition.

Definition 3.1.3. A *(modal)* $\vDash$*-environment* from Γ to Δ is given as follows.

$$\Gamma \overset{\vDash}{\Longrightarrow} \Delta \coloneqq \Pi A.\ (\Delta \ni A\ \mathit{true} \to \Gamma \vDash A) \times (\Delta \ni A\ \mathit{valid} \to \Gamma_v \vDash A)$$

With this definition, I reproduce the kit-based machinery from section 2.3.3. However, I make crucial use of the *stability under renaming* property — rather than *stability under context extension* — to deal with not just context extensions, but also discarding of true hypotheses by the $\Box$-I rule. To state this property, I firstly need to be precise about what a renaming is.

Definition 3.1.4. The set of variables in context Γ of type A, notated $\Gamma \ni A$, is the disjoint union of the sets $\Gamma \ni A\ \mathit{true}$ and $\Gamma \ni A\ \mathit{valid}$ — respectively, the true variables and the valid variables in Γ of type A.

The set of *renamings* from Γ to Δ is $\Gamma \overset{\ni}{\Longrightarrow} \Delta$.

Definition 3.1.4 allows us to state a unified variable rule: For each $x : \Gamma \ni A$, we have a corresponding term $x : \Gamma \vdash A\ \mathit{true}$. However, with the new definition of both environments and variables, we must check that they interact in the desired way.

Lemma 3.1.5 (lookup)**.** If $\vDash$ respects renaming (i.e., we have a function $\mathrm{ren}^{\vDash} : \Gamma \overset{\ni}{\Longrightarrow} \Delta \to \Delta \vDash A \to \Gamma \vDash A$ for each Γ, Δ, and A), then from an environment $\rho : \Gamma \overset{\vDash}{\Longrightarrow} \Delta$ and a variable $x : \Delta \ni A$, we get a value $\rho(x) : \Gamma \vDash A$.

Proof. I proceed by cases on whether x is a true or valid variable. In the true case, the result is straightforward. In the valid case, definition 3.1.3 gives us a value in $\Gamma_v \vDash A$.

We need a renaming of type $\Gamma \overset{\vDash}{\Longrightarrow} \Gamma_v$ to get the desired value in $\Gamma \vDash A$. The renaming is the one that, conceptually, discards the true hypotheses. □

We then need to prove that environments are preserved by everything we do to contexts in a derivation. One thing we do, like in the simply typed case, is to bind new variables, and this is handled by lemma 3.1.6. The other thing, which is new for modal logic, is to discard the true hypotheses, as in the □-I rule, which is handled by lemma 3.1.7.

Lemma 3.1.6 (bindEnv)**.** If we have $\mathrm{vr} : \Gamma \ni A \to \Gamma \vDash A$ for all Γ and A, and $\mathrm{ren}^{\vDash} : \Gamma \overset{\ni}{\Longrightarrow} \Delta \to \Delta \vDash A \to \Gamma \vDash A$ for all Γ, Δ, and A, then from an environment $\rho : \Gamma \overset{\vDash}{\Longrightarrow} \Delta$ we can get an environment of type $\Gamma, \Theta \overset{\vDash}{\Longrightarrow} \Delta, \Theta$ for all Γ, Δ, and Θ.

Proof. I consider separately the part of the environment we are constructing that deals with true hypotheses and the part that deals with valid hypotheses. These cases correspond to the two factors in the expression in definition 3.1.3.

The case for true hypotheses is essentially the same as in section 2.3.3. We take cases on whether the $x : \Delta, \Theta \ni A$ *true* is in Δ or Θ. In the Δ case, we apply ρ and then rename via $\mathrm{ren}^{\vDash}$ using the obvious renaming of type $\Gamma, \Theta \overset{\ni}{\Longrightarrow} \Gamma$. In the Θ case, we embed our $z : \Theta \ni A$ *true* as $\searrow z : \Gamma, \Theta \ni A$ *true*, and use vr to get the required value.

The case for valid hypotheses is similar. We take the same cases, with the Δ case differing only in which part of ρ we need to use (the valid part, rather than the true part). In the Θ case, we have $z : \Theta \ni A$ *valid*. The ${}_v$ operation keeps all valid hypotheses, so we have $z_v : \Theta_v \ni A$ *valid*. Using the fact that $(\Gamma, \Theta)_v = \Gamma_v, \Theta_v$, we have $\searrow z_v : (\Gamma, \Theta)_v \ni A$ *valid*, and we use vr to get the required value. □

Lemma 3.1.7. From an environment $\rho : \Gamma \overset{\vDash}{\Longrightarrow} \Delta$, we can get an environment $\rho_v : \Gamma_v \overset{\vDash}{\Longrightarrow} \Delta_v$.

Proof. Suppose $x : \Delta_v \ni A$. Because Δ_v contains only and all of the valid hypotheses from Δ, we actually have $x' : \Delta \ni A$ *valid*. Applying the second part of ρ on x' gives us a value in $\Gamma_v \vDash A$. We actually wanted a value in $(\Gamma_v)_v \vDash A$, but ${}_v$ is idempotent, so we already have this. □

Then, we use all of the previous features to prove the key theorem.

Theorem 3.1.8 (trav)**.** For any notion of value $\vDash$ which is stable under renaming, admits variables (via some vr as in lemma 3.1.6), and embeds into terms (via some $\mathrm{tm} : \Gamma \vDash A \to \Gamma \vdash A\ \mathit{true}$ for all Γ and A), an environment $\rho : \Gamma \overset{\vDash}{\Longrightarrow} \Delta$ and a term $M : \Delta \vdash A\ \mathit{true}$ yield a term $M[\rho] : \Gamma \vdash A\ \mathit{true}$.

Proof. I proceed by induction on the term M. Here I consider only variables and the two rules governing the $\Box$ connective. It is easy to adapt this procedure to any of the types from simply typed λ-calculus.

I consider variables being given by the unified variable rule proposed in the paragraph before lemma 3.1.5. Given this, from a variable $x : \Delta \ni A$, lemma 3.1.5 gives us a value $\rho(x) : \Gamma \vDash A$, which tm turns into a term of the desired form.

If the term was made from the $\Box$-I rule, we use that rule to produce the resulting term, and need to get from a term $M : \Delta_v \vdash A\ \mathit{true}$ to a term in $\Gamma_v \vdash A\ \mathit{true}$. It is enough to use the induction hypothesis, updating the environment using lemma 3.1.7.

If the term was made from the $\Box$-E rule, we again use the same rule to construct the output, but have to get from a term $M : \Delta \vdash \Box A\ \mathit{true}$ to a term in $\Gamma \vdash \Box A\ \mathit{true}$, and from a term $N : \Delta, A\ \mathit{valid} \vdash B\ \mathit{true}$ to a term in $\Gamma, A\ \mathit{valid} \vdash B\ \mathit{true}$. The former follows from a straightforward use of the induction hypothesis, while the latter needs lemma 3.1.6 before applying the induction hypothesis. $\Box$

To get our desired corollaries of simultaneous renaming and substitution, we first have to show that variables respect renaming. In the pure intuitionistic case of chapter 2, this was trivial because renamings were precisely functions between sets of variables. In the modal case, we have to be careful about the distinction between true and valid hypotheses, and how the context is restricted in the valid case.

Lemma 3.1.9 (variables respect renaming)**.** Given a renaming $\rho : \Gamma \overset{\ni}{\Longrightarrow} \Delta$ and a variable $x : \Delta \ni A$, we get a variable in $\Gamma \ni A$.

Proof. Note that we cannot use lemma 3.1.5 with $\vDash$ instantiated to $\ni$ because it would circularly require that variables respect renaming. However, the procedure in this case is similar to that of lemma 3.1.5.

We take cases on whether x is true or valid, and the result in the true case comes immediately by applying the true part of ρ. For $x : \Delta \ni A$ *valid*, ρ gives us a variable in $\Gamma_v \ni A$. This variable must be valid ($\Gamma_v \ni A$ *valid*), and from there it is easy to get a variable in the larger context Γ. □

Corollary 3.1.10 (renaming)**.** Given a renaming $\rho : \Gamma \overset{\ni}{\Longrightarrow} \Delta$ and a term $M : \Delta \vdash A$ *true*, we get a term in $\Gamma \vdash A$ *true*.

Proof. We use theorem 3.1.8, with $\vDash$ being $\ni$, vr being the identity function, tm being the unified variable rule, and renaming of $\ni$ being given by lemma 3.1.9. □

Corollary 3.1.11 (substitution)**.** Given a substitution $\rho : \Gamma \overset{\vdash}{\Longrightarrow} \Delta$ and a term $M : \Delta \vdash A$ *true*, we get a term in $\Gamma \vdash A$ *true*.

Proof. We use theorem 3.1.8, with $\vDash$ being $- \vdash -$ *true*, vr being the unified variable rule, tm being the identity function, and renaming being given by corollary 3.1.10. □

With these basic syntactic lemmas proved in a manner largely following the proofs of chapter 2, it is plausible that this approach could be extended to handle generic semantics and generic syntax, following sections 2.4 and 2.5, respectively. However, I hold off from developing this until phrasing it more generally in chapter 6.

3.2 Intuitionistic Linear Logic

Where modal logics can be seen as additions to an underlying classical or intuitionistic logic, linear logic is a much more radical change. Formally, many modal logics, including (I)S4, are conservative extensions of the underlying classical or intuitionistic logic, meaning that statements not mentioning the modal operators $\Box$ and $\Diamond$ are provable in modal logic if and only if they are provable in the underlying non-modal logic. In linear logic, however, we a priori severely restrict provability of basic formulae, and then use modalities to recover the strength of classical or intuitionistic logic.

In this thesis, I consider intuitionistic linear logic (ILL). ILL can be understood as a logic of resources. Whereas in classical logic we read a proposition A to implicitly mean "A is true", we may read a proposition A of linear logic to implicitly mean "I have an

A". A sequent $A \vdash B$ can then be understood as saying "if I have an A, I can give it up so as to have a B", and the corresponding implication $A \multimap B$ can be understood as saying "I have a *mechanism* for having a B if I have and am willing to give up an A".

The resource interpretation of intuitionistic linear logic is often explained by analogy with a vending machine. Suppose we have a vending machine which takes pound coins and sells bottles of water for £1 and chocolate bars for £2. We can represent the latter two facts by the judgements $£1 \vdash \text{water}$ and $£1, £1 \vdash \text{bar}$, respectively. Notice that having two £1 coins is very different from having just one £1 coin. We can represent having compound objects using $\otimes$-products (tensor-products), so that having a £1 coin together with a chocolate bar is represented as $£1 \otimes \text{bar}$, and the mechanism for getting a chocolate bar from the vending machine is described by $(£1 \otimes £1) \multimap \text{bar}$. The entire protocol of one transaction with the vending machine is represented as $(£1 \multimap \text{water}) \,\&\, ((£1 \otimes £1) \multimap \text{bar})$, where the &-product (with-product) represents giving the user a choice of which of the two conjuncts to have. Finally, that we can do multiple transactions with the vending machine is represented as $!((£1 \multimap \text{water}) \,\&\, ((£1 \otimes £1) \multimap \text{bar}))$, where the !-modality (read "bang" or "of course") represents having an arbitrary number of copies of its argument.

Such reasoning is useful in computer science because it allows us to distinguish what we had before some event with what we got after it. In both intuitionistic and classical logic, if we had a £1 coin and a way to turn a £1 coin into a bottle of water, we'd be able to have both the £1 coin and the bottle of water at once. Instead, linear logic acknowledges that the £1 coin was spent, so at one time we had just the £1 coin, and at a later time we had just the bottle of water. In real applications, such reasoning is useful in, for example, dealing with mutable state [Makwana and Krishnaswami, 2019] — in which, when we mutate the state, we forget the old value and have only the new value — and producing a session type system [Wadler, 2012] — in which the protocol progresses as messages are sent.

In this section, I will carefully introduce intuitionistic linear logic so as to make the intuitive readings more precise.

3.2.1 The multiplicative-additive fragment

The multiplicative-additive fragment of linear logic (MALL) is the fragment where all hypotheses are linear (must be used exactly once). I will extend MALL with the *exponential* modality in section 3.2.2. MALL is unable to embed intuitionistic or classical logic, as MALL is unable to reflect any of the discarding or duplication that can be done in proofs using weakening or contraction.

In short, the syntax of intuitionistic MALL can be described as intuitionistic logic with the structural rules of *weakening* and *contraction* removed. However, without the presence of weakening and contraction, we have to be more careful about the rules we state, so as not to accidentally admit weakening and contraction. The lack of these structural rules also allows us to observe a new phenomenon: the distinction (at the level of provability) between *additive* and *multiplicative* formulations of existing connectives (in particular, the conjunction connective).

I present MALL in figure 3.3 in a sequent calculus style, as it was presented by Girard [1987].

To encode what it means to use a hypothesis *exactly once*, we first need to decide what counts as a use. The simplest case is that the identity sequent counts as a single use of its sole hypothesis, and conversely does not count as a use of any other hypotheses. For sequential proofs, created by the Cut rule, if we have a proof of A using Γ, and a proof of B using Δ and A, then we have a proof of B transitively using Δ and Γ. The exchange rule Exch says that use is invariant under permutation.

For the logical connectives, we have genuine choices as to what it means to use them. Two cases — disjunction ($\oplus$) and (linear) implication ($\multimap$) — are somewhat intuitive from intuitionistic logic. A canonical proof of a disjunction is a tag and a proof of one of the two disjuncts. This suggests that a proof of a disjunction only uses the same hypotheses as the proof of the disjunct we actually choose, with the other disjunct being irrelevant. Correspondingly, when we use a disjunction hypothesis, we will only actually use one of the cases, so each branch should use the same hypotheses. For implication, use is sequential like with the Cut rule, and its left rule is more or less the only choice that allows use of the surrounding hypotheses.

$$A, B, C ::= X \mid I \mid A \otimes B \mid A \multimap B \mid 0 \mid A \oplus B \mid \top \mid A \,\&\, B$$
$$\Gamma, \Delta, \Theta ::= \cdot \mid \Gamma, A$$

$$\frac{}{A \vdash A}\ \textsc{Id} \qquad \frac{\Gamma \vdash A \quad \Delta, A \vdash B}{\Gamma, \Delta \vdash B}\ \textsc{Cut} \qquad \frac{\Gamma, B, A, \Delta \vdash C}{\Gamma, A, B, \Delta \vdash C}\ \textsc{Exch}$$

$$\frac{\Gamma \vdash C}{\Gamma, I \vdash C}\ I\text{-L} \qquad \frac{}{\cdot \vdash I}\ I\text{-R} \qquad \frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C}\ \otimes\text{-L}$$

$$\frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B}\ \otimes\text{-R} \qquad \frac{\Gamma \vdash A \quad \Delta, B \vdash C}{\Gamma, \Delta, A \multimap B \vdash C}\ \multimap\text{-L}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}\ \multimap\text{-R} \qquad \frac{}{\Gamma, 0 \vdash C}\ 0\text{-L} \qquad \text{(no 0-R)}$$

$$\frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \oplus B \vdash C}\ \oplus\text{-L} \qquad \frac{\Gamma \vdash A_i}{\Gamma \vdash A_0 \oplus A_1}\ \oplus\text{-R}_i \qquad \text{(no } \top\text{-L)}$$

$$\frac{}{\Gamma \vdash \top}\ \top\text{-R} \qquad \frac{\Gamma, A_i \vdash C}{\Gamma, A_0 \,\&\, A_1 \vdash C}\ \&\text{-L}_i \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \,\&\, B}\ \&\text{-R}$$

Figure 3.3: Multiplicative-additive fragment of linear logic

For conjunction, there are two choices: Either the conjuncts *together* use all of the hypotheses, or each of the conjuncts *individually* uses all of the hypotheses. The former choice gives us the tensor-product ($\otimes$), and the latter choice gives us the with-product (&). These products are equivalent up to provability in logics with weakening and contraction, but distinct in linear logic.

Implication ($\multimap$) and the tensor-product ($\otimes$, I) comprise the *multiplicative* fragment, while disjunction ($\oplus$, 0) and the with-product (&, $\top$) comprise the *additive* fragment. Categorically, the additive fragment corresponds to products and coproducts, while the multiplicative fragment corresponds to multicategorical tensor products and closure.

3.2.2 The !-modality

Figure 3.4 shows the rules we can add to MALL to get the full sequent calculus for intuitionistic linear logic (ILL). In ILL, $!A$ is defined to be a proposition whose occurrences as antecedents can be deleted (Weakening) and duplicated (Contraction), from which

$$A, B, C ::= \cdots \mid !A$$

$$\frac{!\Gamma \vdash A}{!\Gamma \vdash !A}\ \textsc{Promotion} \qquad \frac{\Gamma, A \vdash B}{\Gamma, !A \vdash B}\ \textsc{Dereliction}$$

$$\frac{\Gamma \vdash B}{\Gamma, !A \vdash B}\ \textsc{Weakening} \qquad \frac{\Gamma, !A, !A \vdash B}{\Gamma, !A \vdash B}\ \textsc{Contraction}$$

Figure 3.4: The sequent calculus rules for the !-modality

we can extract A (Dereliction), and which we can form from a conclusion A only when all antecedents are of the form $!X$ for some proposition X (Promotion). In short, $!A$ can be seen as an intuitionistic version of A, supporting all of the structural rules of LJ, and only being able to be formed when it does not depend on anything linear.

The Dereliction rule is comparable to the T axiom we saw in section 3.1. The Promotion rule is exactly like the Promotion rule we saw in section 3.1, and for the same reasons we want to avoid having such a rule in a natural deduction system for intuitionistic linear logic. Note that Promotion is not problematic in the sequent calculus, at least insofar as it maintains Cut-elimination.

Additional to the problems with Promotion in a natural deduction system, defining the !-modality as in figure 3.4 has the odd feature that, unlike all of the other connectives of ILL, ! is not characterised by a universal property. This can be seen by the fact that taking the rules for ! and replacing each occurrence of ! by a fresh connective $!'$ produces a logically distinct connective. One cannot produce any derivation of $!'A \vdash !A$ because Promotion does not apply when there are antecedents not of the form $!X$. This lack of characterisation also holds of the rule and axioms we first gave to the $\Box$-modality in figure 3.1, and essentially any presentation that does not incorporate the modality into the judgemental structure of the calculus.

A noteworthy solution to the substitution problem of Promotion is given by Benton et al. [1993]. The solution they give is compatible with the form of sequents introduced above, but works purely in terms of right-rules like standard natural deduction calculi. I give their rules for the !-modality in figure 3.5. The Weakening and Contraction

$$\frac{\Delta_1 \vdash !A_1 \quad \cdots \quad \Delta_n \vdash !A_n \quad !A_1, \ldots, !A_n \vdash B}{\Delta_1, \ldots, \Delta_n \vdash !B}\ \textsc{Promotion} \qquad \frac{\Gamma \vdash !A}{\Gamma \vdash A}\ \textsc{Dereliction}$$

$$\frac{\Gamma \vdash !A \quad \Delta \vdash B}{\Gamma, \Delta \vdash B}\ \textsc{Weakening} \qquad \frac{\Gamma \vdash !A \quad \Delta, !A, !A \vdash B}{\Gamma, \Delta \vdash B}\ \textsc{Contraction}$$

Figure 3.5: The Benton et al. [1993] rules for the !-modality

rules are reminiscent of pattern-matching elimination principles, allowing us to form an inhabitant of the type of interest ($!A$) and then continue producing the originally desired result (B) in an updated context. However, somewhat unusually, we can choose which of the two "pattern-matching" principles to use, depending on whether we want to delete or duplicate the value of type $!A$. Another elimination-like rule is Dereliction, which straightforwardly lets us derive A from $!A$ similarly to the corresponding sequent calculus rule from figure 3.4, but on the right.

The Benton et al. Promotion rule is more formally complex, but can be understood as follows. We notice that the problem with the Promotion rule of fig:bang-seq when added to a natural deduction system is that it restricts the types of assumptions in the concluding sequent. In other words, the sequent calculus Promotion rule is acting like both a right-rule and, problematically, a left-rule, to some degree. Acting on the context in a natural deduction system risks making substitution inadmissible, as happens in this case. Instead, Benton et al. paraphrase this restriction on the context as the construction of a *new* context $!A_1, \ldots, !A_n$ for the primary subderivation. Heuristically, this new Promotion rule admits substitution because any substitution will pass into the relevant premises deriving $!A_1$ to $!A_n$. More abstractly, we can notice that the first n premises essentially form an explicit substitution from $\Delta_1, \ldots, \Delta_n$ to $!A_1, \ldots, !A_n$, so any further substitution applied to a Promotion-headed term is precomposed onto this explicit substitution [Abadi et al., 1991].

The Benton et al. Promotion rule is a clever solution to the substitution problem of (intuitionistic) linear logic, but the resulting system comes with its own problems. If we were to write programs in a language directly implementing the Benton et al. calculus, then the Promotion rule would be a pain point because it makes us rebind

all of the !-typed variables we need to new variables. Additionally, we have to be explicit everywhere about weakening and contraction, potentially making working with !-typed variables much more fiddly than the corresponding variables would be in a non-substructural programming language. These syntactic annoyances may be worked around in a realistic implementation by some new elaboration procedures, but we would prefer not to rely on such elaboration if there is an acceptable core calculus which more closely matches the programs we want to write. Therefore, I investigate such a system, Dual Intuitionistic Linear Logic, in the following subsection.

3.2.3 Dual Intuitionistic Linear Logic

Dual Intuitionistic Linear Logic (DILL) is a syntax for intuitionistic linear logic introduced by Barber [1996]. Its key feature is splitting assumptions into *linear* assumptions and *intuitionistic* assumptions — sometimes called the *dual context* approach. Intuitionistic assumptions behave like the variables of simply typed λ-calculus. In contrast, the linear assumptions behave as in the linear calculus we saw in section 3.2. For example, an intuitionistic assumption of A in an instance of the $\otimes$-introduction rule is automatically copied to both premises. This contrasts with an assumption of $!A$ in the purely linear calculus, which must first be contracted into two assumptions, with one going to each premise. Compared to the modal system we saw in section 3.1, linear assumptions correspond to true assumptions, and intuitionistic assumptions correspond to valid assumptions.

The new feature when dealing with linear logic, compared to modal logic, is that linear and intuitionistic assumptions satisfy different structural rules. In our De Bruijn index style, where every multi-premise rule does the maximal contraction and weakening occurs in the leafwardmost possible position, this means that all of the rules have to manage the substructurality of linear assumptions. To help manage linear assumptions, I introduce three pieces of notation.

Definition 3.2.1. I write Γ *int* to state that context Γ contains only intuitionistic assumptions, i.e., no linear assumptions.

Definition 3.2.2. I write $\Gamma + \Delta$ to stand for a context combining Γ and Δ. Specifically, the operation is only defined when Γ and Δ contain the same intuitionistic assumptions, and in that case the result contains the same intuitionistic assumptions. The linear assumptions of the result are the disjoint union of the linear assumptions of each of Γ and Δ.

Definition 3.2.3. I write $\Gamma \ni A$, like in section 3.1, to stand for the set of variables of type A we can get from Γ. Specifically, if Γ contains an assumption $x : A\ \mathit{int}$, then x gives rise to an inhabitant of $\Gamma \ni A$ if and only if $\Gamma\ \mathit{int}$; and if Γ contains an assumption $x : A\ \mathit{lin}$, then x gives rise to an inhabitant of $\Gamma \ni A$ if and only if x is the only linear assumption in Γ. These restrictions essentially encode the fact that we do not allow linear variables to be discarded via the use of another variable.

Barber [1996] uses the notational convenience of dividing linear and intuitionistic assumptions into two separate contexts, hence the name of the *dual context* approach. I choose not to do this so as to draw out the connection with the approach of Pfenning and Davies [1999], as well as the approach I will take in later chapters. I also think that it is more instructive to treat the context as a single unified thing, so that we can clearly see what operations on the context have to be preserved by environments (for use in renaming, substitution, and other traversals).

I list the complete rules of DILL in figure 3.6. Given definitions 3.2.1 to 3.2.3, the rules for multiplicative and additive connectives look not too dissimilar to those in figure 3.3. However, their treatment of intuitionistic assumptions (contraction and weakening automatically, as in non-substructural calculi) is new. Meanwhile, the rules for the !-modality are the same as those for the $\Box$-modality given in figure 3.2, except for the implicit structural rules applying to linear assumptions. For example, where the $\Box$-I rule discards true assumptions, the !-I rule cannot discard linear assumptions, but instead requires that there are no linear assumptions in the context when the rule is applied, so that no linear assumptions can be used in the subproof.

I will not give a direct substitution procedure for DILL like the one I gave for the modal system of section 3.1. Suffice to say, where environments for the modal calculus had to be preserved by binding of variables and pruning of all true variables, environ-

$$A, B, C ::= X \mid I \mid A \otimes B \mid A \multimap B \mid 0 \mid A \oplus B \mid \top \mid A \,\&\, B \mid !A$$
$$\Gamma, \Delta, \Theta ::= \cdot \mid \Gamma, A\ \mathit{lin} \mid \Gamma, A\ \mathit{int}$$
$$\mathcal{S} ::= \Gamma \vdash A\ \mathit{lin}$$

$$\frac{\Gamma \ni A}{\Gamma \vdash A\ \mathit{lin}}\ \textsc{Var} \qquad \frac{\Gamma \vdash I\ \mathit{lin} \qquad \Delta \vdash C\ \mathit{lin}}{\Gamma + \Delta \vdash C\ \mathit{lin}}\ I\text{-E} \qquad \frac{\Gamma\ \mathit{int}}{\Gamma \vdash I\ \mathit{lin}}\ I\text{-I}$$

$$\frac{\Gamma \vdash A \otimes B\ \mathit{lin} \qquad \Delta, A\ \mathit{lin}, B\ \mathit{lin} \vdash C\ \mathit{lin}}{\Gamma + \Delta \vdash C\ \mathit{lin}}\ I\text{-E} \qquad \frac{\Gamma \vdash A\ \mathit{lin} \qquad \Delta \vdash B\ \mathit{lin}}{\Gamma + \Delta \vdash A \otimes B\ \mathit{lin}}\ \otimes\text{-I}$$

$$\frac{\Gamma \vdash A \multimap B\ \mathit{lin} \qquad \Delta \vdash A\ \mathit{lin}}{\Gamma + \Delta \vdash B\ \mathit{lin}}\ \multimap\text{-E} \qquad \frac{\Gamma, A\ \mathit{lin} \vdash B\ \mathit{lin}}{\Gamma \vdash A \multimap B\ \mathit{lin}}\ \multimap\text{-I} \qquad \frac{\Gamma \vdash 0\ \mathit{lin}}{\Gamma + \Delta \vdash C\ \mathit{lin}}\ 0\text{-E}$$

$$\text{(no 0-I)} \qquad \frac{\Gamma \vdash A \oplus B\ \mathit{lin} \qquad \Delta, A\ \mathit{lin} \vdash C\ \mathit{lin} \qquad \Delta, B\ \mathit{lin} \vdash C\ \mathit{lin}}{\Gamma + \Delta \vdash C\ \mathit{lin}}\ \oplus\text{-E}$$

$$\frac{\Gamma \vdash A_i\ \mathit{lin}}{\Gamma \vdash A_0 \oplus A_1\ \mathit{lin}}\ \oplus\text{-I}_i \qquad \text{(no } \top\text{-E)} \qquad \frac{}{\Gamma \vdash \top\ \mathit{lin}}\ \top\text{-I} \qquad \frac{\Gamma \vdash A_0 \,\&\, A_1\ \mathit{lin}}{\Gamma \vdash A_i\ \mathit{lin}}\ \&\text{-E}_i$$

$$\frac{\Gamma \vdash A\ \mathit{lin} \qquad \Gamma \vdash B\ \mathit{lin}}{\Gamma \vdash A \,\&\, B\ \mathit{lin}}\ \&\text{-I} \qquad \frac{\Gamma \vdash !A\ \mathit{lin} \qquad \Delta, A\ \mathit{int} \vdash !C\ \mathit{lin}}{\Gamma + \Delta \vdash C\ \mathit{lin}}\ !\text{-E}$$

$$\frac{\Gamma \vdash A\ \mathit{lin} \qquad \Gamma\ \mathit{int}}{\Gamma \vdash !A\ \mathit{lin}}\ !\text{-I}$$

Figure 3.6: Dual Intuitionistic Linear Logic

ments for DILL have to be preserved by similar plus the operations of definitions 3.2.1 and 3.2.2. These definitions themselves are somewhat technical, and I do not believe that delving into the further technicalities of how they interact with environments will provide useful enough intuitions to justify including here. However, I will revisit the idea of environments for linear calculi in a more general semiring-annotated setting in chapter 5.

Finally, notice that the rules given in figure 3.6 characterise ! up to logical equivalence. The main feature ensuring this property is that each logical rule contains exactly one occurrence of a logical connective, meaning that each logical rule is only defining that connective in that place. For the full characterisation result, we also require local soundness and completeness of !, as discussed by Pfenning and Davies [1999] in the modal setting.

3.3 Mechanisations and systematisations of substructural logics

In this section, I give an overview of techniques which have been used in previous work to mechanise linear logic in proof assistants. Naïve approaches often struggle to represent concatenation of contexts in a way which is amenable to the way dependent type theory-based proof assistants work. Problems even arise when working rigorously on paper when trying to avoid an explicit exchange rule, such as how definition 3.2.2 is not a precise definition of a binary operator on lists.

3.3.1 Typing with leftovers

Typing with leftovers, introduced by Allais [2018], is a technique developed to specify an algorithm for linear type checking as a declarative type system. The idea is to consider an input context, a term, and an output context, where the input context contains all of the variables in scope, and the output context is the same minus any variables used by the term. Type-checking of adjacent subterms of, for example, an application of the $\otimes$-introduction rule, is done by threading the context through from the output of

$$\Gamma, \Delta, \Theta ::= \cdot \mid \Gamma, 1x : A \mid \Gamma, 0x : A$$
$$\mathcal{S} ::= \Gamma \vdash M : A \boxtimes \Delta$$

Figure 3.7: Typing with leftovers, context and sequent syntax

$$\frac{}{\Gamma, 1x : A \vdash x : A \boxtimes \Gamma, 0x : A}\ \textsc{Var} \qquad \frac{}{\Gamma \vdash * : I \boxtimes \Gamma}\ I\text{-I}$$

$$\frac{\Gamma \vdash M : I \boxtimes \Delta \qquad \Delta \vdash N : C \boxtimes \Theta}{\Gamma \vdash \text{let } * = M \text{ in } N : C \boxtimes \Theta}\ I\text{-E} \qquad \frac{\Gamma \vdash M : A \boxtimes \Delta \qquad \Delta \vdash N : B \boxtimes \Theta}{\Gamma \vdash (M, N) : A \otimes B \boxtimes \Theta}\ \otimes\text{-I}$$

$$\frac{\Gamma \vdash M : A \otimes B \boxtimes \Delta \qquad \Delta, 1x : A, 1y : B \vdash N : C \boxtimes \Theta, 0x : A, 0y : B}{\Gamma \vdash \text{let } (x, y) = M \text{ in } N : C \boxtimes \Theta}\ \otimes\text{-E}$$

$$\frac{\Gamma, 1x : A \vdash M : B \boxtimes \Delta, 0x : A}{\Gamma \vdash \lambda x.\ M : A \multimap B \boxtimes \Delta}\ \multimap\text{-I} \qquad \frac{\Gamma \vdash M : A \multimap B \boxtimes \Delta \qquad \Delta \vdash N : A \boxtimes \Theta}{\Gamma \vdash M\ N : B \boxtimes \Theta}\ \multimap\text{-E}$$

Figure 3.8: Typing with leftovers, multiplicative fragment

the first term to the input of the second. Bound variables are introduced in the input context of the term in which they are bound, and are expected to be absent in the output context of that term.

Figures 3.7 and 3.8 give rules in the typing-with-leftovers style for the multiplicative fragment of intuitionistic linear logic. Where Allais marks *fresh* and *stale* variables, I use the notation I will use starting in chapter 4, labelling such variables with 1 and 0, respectively. Intuitively, the number describes how many more times that variable is to be used.

The original paper extends the logic covered to binary additives — & and $\oplus$ — with rules that check that terms agree on output contexts, as seen in figure 3.9. However, it is less clear how to handle nullary additives — $\top$ and 0 — as we would have 0 (rather than 2) potential candidates for the output context. At some level, this problem is unavoidable in a system modelling linearity checking because any checking strategy will expose the ambiguity in sequents like $1x : A \vdash (\langle\rangle, \langle\rangle) : \top \otimes \top$ of whether the variable x was consumed in the left half or the right half. Such an example is also considered in related work on proof search for linear logics, such as the work of Winikoff and Harland

$$\frac{\Gamma \vdash M : A \boxtimes \Delta \qquad \Gamma \vdash N : B \boxtimes \Delta}{\Gamma \vdash \{M, N\} : A \,\&\, B \boxtimes \Delta}\ \&\text{-I}$$

$$\frac{\Gamma \vdash M : A \oplus B \boxtimes \Delta \qquad \Delta, 1x : A \vdash N : C \boxtimes \Theta, 0x : A \qquad \Delta, 1y : B \vdash O : C \boxtimes \Theta, 0y : B}{\Gamma \vdash \text{case } M \text{ of } \{x.\ N; y.\ O\} : C \boxtimes \Theta}\ \oplus\text{-E}$$

Figure 3.9: Typing with leftovers, a selection of the additive rules

$$\frac{}{\Gamma, \omega x : A \vdash x : A \boxtimes \Gamma, \omega x : A}\ \textsc{IVar} \qquad \frac{0\gamma, 0\delta, \omega\theta \vdash M : A \boxtimes 0\gamma, 0\delta, \omega\theta}{0\gamma, 1\delta, \omega\theta \vdash [M] : !A \boxtimes 0\gamma, 1\delta, \omega\theta}\ !\text{-I}$$

$$\frac{\Gamma \vdash M : !A \boxtimes \Delta \qquad \Delta, \omega x : A \vdash N : C \boxtimes \Theta, \omega x : A}{\Gamma \vdash \text{let } [x] = M \text{ in } N : C \boxtimes \Theta}\ !\text{-E}$$

Figure 3.10: Typing with leftovers, a possible way to capture !

[1994, p. 11] and Cervesato et al. [2000, p. 150]. It is not immediately clear whether the different solutions proposed by these papers will apply to Allais' and my settings, given that they both act on a set of formulas restricted to facilitate proof search. The solutions also add significant, seemingly somewhat ad hoc, structure to the syntax of sequents, with no semantic justification (rather being justified by making their respective proof search algorithms efficient).

The original paper also does not show how to capture the exponential modality !. The solution given by both Winikoff and Harland [1994] and Cervesato et al. [2000] is, as in DILL, to distinguish between linear variables and intuitionistic variables. This gives rules like those of figure 3.10. The important invariant is that linear and intuitionistic variables stay distinct, so any intuitionistic variable in the input context (annotated by ω) must be intuitionistic in the output context.

However, this adaptation of the DILL style does not obviously generalise to semiring annotations. Even for the multiplicative fragment, we seem to be working against the direction of addition, instead using a subtraction operation whenever we use a variable. An algebraic presentation of this fragment is given by Zalakain and Dardha [2021], in terms of a partial functional cancellative addition operation. For exponentials, though,

and particularly the !-introduction rule, what I have done seems ad-hoc, not based on any pointwise algebraic operation.

Also, the unusual form of sequents can cause some problems when working with a typing with leftovers system. For example, a traditional intuitionistic linear logic sequent $\Gamma \vdash M : A$ corresponds to many different typing with leftovers sequents, including:

- $1\Gamma \vdash M : A \boxtimes 0\Gamma$
- $1\Gamma, 1x : B \vdash M : A \boxtimes 0\Gamma, 1x : B$
- $1\Gamma, 0x : B \vdash M : A \boxtimes 0\Gamma, 0x : B$

Generally, any variable not used in the term can be added to both the input and the output context with the same annotation. Many of these variations are likely to appear in various typing derivations, depending on what terms surround a given subterm. In particular, many different variations can appear for the same term in different subterms of the same derivation, even when the scope of the two occurrences is the same. This means that if we want to implement substitution, which involves putting a term into an unknown surrounding, we have to navigate these different forms via the *framing property*.

The unusual form of sequents also somewhat obscures any attempt to interpret the terms of a typing-with-leftovers system. Though the $\boxtimes$ notation suggests a semantics into symmetric monoidal closed categories where terms are morphisms from one iterated tensor product (the input context) to another (the type and output context), the syntax is incomplete for this semantics because we cannot produce anything interesting in the output context.

Another piece of related work using a typing-with-leftovers style is that of Polakow [2015]. There, Polakow encodes a linear embedded domain-specific language inside Haskell using typeclass constraints.

3.3.2 Yalla

Laurent [2018] presents a collection of linear logics in a uniform style, and various proofs

relating them. The logics share varying amounts of definitions and theorems — for example, the main linear logic is parametrised on whether to include mix rules and whether to restrict exchange to cyclic permutations, whereas systems like the Lambek calculus (with no exchange) and polarised linear logic are defined separately from scratch.

The style used in Yalla is to realise sequents as lists of formulas. The active formula tends to be forced to be the first formula in such a list, with an explicit exchange rule being used to move such formulas into place. Laurent [2018] points out that using multisets, as do some less formal presentations of linear logic, is insufficient at distinguishing certain proofs involving repeated assumptions or conclusions. For example, we expect there to be two distinct derivations of $A \otimes A \vdash A \otimes A$ (up to the appropriate equational theory): the one that keeps the pair in the same order and the one that flips the order. But if the $\otimes$-L rule unpacks the formula $A \otimes A$ into the *multiset* $\{A, A\}$, then we forget the order of the input pair.

Despite making sure to distinguish distinct proofs, the Yalla library does not define any equational theories of proofs, so does not prove any results relying on these distinctions. While the complication of defining an equational theory in the presence of an exchange rule is probably largely inevitable, having the exchange rule introduces redundancy such that many equivalent proofs are not equal as data structures. The mechanisations presented in chapter 2 all sought to reduce this kind of redundancy for intuitionistic logic, so that the only non-trivial equations in the equational theory are β- and η-rules (i.e. computationally interesting rules). I will restore this property of the representation in chapter 4.

The relevance of Yalla to the work in this thesis is limited by the fact that Yalla is based entirely on sequent calculi, whereas I am considering only natural deduction calculi.

3.3.3 Co-De-Bruijn syntax

McBride [2018] presents a mechanisation of the simply typed λ-calculus in what he calls *co-De-Bruijn* style. He notes that syntax based on De Bruijn indices, as presented in chapter 2, finds a canonical way to place contractions and weakenings by eagerly

placing contractions wherever they could be needed (i.e., whenever a rule has multiple premises) and leaving weakening as late as possible (i.e., at the variable rule and at rules with no premises). Co-De-Bruijn syntax, by contrast, finds a canonical way to place contractions and weakenings by doing the reverse: weakening as early as possible (i.e., as soon as the variable is bound) and contracting only where necessary.

Such a scheme straightforwardly adapts to multiplicative linear logic by modifying the data structures presented by McBride [2018] to disallow all contraction and weakening, as presented by Rouvoet et al. [2020]. With the additive rules, however, we get cases where variables appear multiple times syntactically in a term but are still considered linear by the type system, the simplest example being $x : A \vdash \langle x, x \rangle : A \,\&\, A$. Such rules are perhaps a new kind compared to what McBride considers, but it seems likely that just copying the same context to all the premises would not be too hard to accommodate. Meanwhile, we may consider implementing the !-modality as in DILL, with intuitionistic variables handled using the regular co-De-Bruijn machinery from the paper. In summary, the co-De-Bruijn approach is promising for capturing linearity, but has not been thoroughly investigated.

The other relevant contribution from Rouvoet et al. [2020] is to recast the context-splitting relations via connectives inspired by bunched logic [O'Hearn and Pym, 1999]. I adapt this method in section 4.3, but with a different notion of context-splitting.

3.3.4 Fitch-style modalities

An alternative to the approach of Pfenning and Davies [1999] to adapt modal logics (and particularly IS4) to natural deduction is using Fitch-style calculi. Fitch-style calculi, as codified and studied by Borghuis [1994], are distinguished by allowing for contexts containing *locks*, written 🔒, with the variable rule being restricted so that only variables not behind locks are immediately accessible.

Below I give the main rules of Fitch-style IK (intuitionistic logic K). Other normal modal logics are obtained by strengthening the $\Box$-elimination rule to remove varying numbers of locks. For example, we can add axiom T by allowing for 0 or 1 locks to be removed, axiom 4 by allowing 1 or more locks to be removed, or both axioms together

by allowing any number of locks to be removed from the right-hand end of the context (together with any variables to the right of a removed lock). The $\Box$-I rule stays the same, and forms part of an adjunction of the form "$🔒 \dashv \Box$".

$$\frac{🔒 \notin \Gamma'}{\Gamma, A, \Gamma' \vdash A}\ \textsc{Var} \qquad \frac{\Gamma, 🔒 \vdash A}{\Gamma \vdash \Box A}\ \Box\text{-I} \qquad \frac{🔒 \notin \Gamma' \qquad \Gamma \vdash \Box A}{\Gamma, 🔒, \Gamma' \vdash A}\ \Box\text{-E}_K$$

A syntactic advantage of Fitch-style calculi over the calculus introduced by Pfenning and Davies [1999] is that Fitch-style calculi support a projection-style eliminator for $\Box$, which makes it easier to use than the pattern-matching eliminator of Pfenning and Davies. A disadvantage is that 🔒 cannot be understood as a kind of hypothetical judgement like the rest of the context, so many of the heuristics we relied on in chapter 2 and section 3.1 fail. In fact, the addition of locks represents a large change to the judgemental structure of the calculus, apparently requiring a complete overhaul of the basic metatheory.

Valliappan et al. [2022] have completed a significant mechanisation of the metatheory of a Fitch-style calculus in Agda. This work shows that, despite the change in the structure of the metatheory, Fitch-style calculi are amenable to mechanised proofs.

I am not aware of any work discussing linear Fitch-style calculi.

3.3.5 Systematisations of substructural logics

Several pieces of prior work have aimed to give general accounts of a range of substructural calculi, in a similar vein to existing accounts of aspects of non-substructural calculi. I review some of these, particularly as a way to give a comparison to the adaptation of the methods of chapter 2 that I spend the rest of this thesis on.

Linear Logical Framework In section 2.6.5, I discussed logical frameworks based on the λ^{Π}-calculus, and their use in the mechanisation of non-substructural programming languages. Cervesato and Pfenning [2002] extend λ^{Π} to a calculus $\lambda^{\Pi \multimap \& \top}$, and use that to create a logical framework supporting linear logics: the Linear Logical Framework (LLF). The Π type former still forms intuitionistic weak dependent function spaces,

while the new $\multimap$ forms linear weak non-dependent function spaces. They handle the distinction between intuitionistic and linear variables thus introduced in the same style as DILL, with the argument of an intuitionistic application having the same intuitionistic restriction as we saw in DILL's !-I rule.

The additive connectives & and $\top$ provide a way to state rules whose premises respectively share and arbitrarily consume linear variables, like in the rules for additive connectives in linear logic. I will revisit this method of stating typing rules in terms of *sharing* and *separating* (as given by right-nested sequences of $\multimap$s) conjunctions of premises in section 4.3, though in section 4.3.2 I argue that there is a closer connection to bunched logics than linear logics.

The main focus of the original paper is to represent mutation in an ML-like language via state updates mediated by $\multimap$, though they also mention having mechanised some metatheory of linear calculi. Some example programs are currently available at `https://www.cs.cmu.edu/~iliano/projects/LLF/index.html`.

Encoding linearity in LF Crary [2010] gives a method of encoding linearity constraints in a conventional, non-substructural, logical framework. He implements this approach in the LF-based proof assistant Twelf [Pfenning and Schürmann, 1999]. He uses a predicate `linear : (term -> term) -> type`, where `term` is a type of preterms, and thus `term -> term` is (thanks to the weak function space of LF) the type of preterms with one free variable. The predicate `linear` then says that that free variable is used linearly in its term, which is defined inductively on the structure of preterms. The `linear` predicate is used by the typing relation wherever variables are bound. The development handles all of intuitionistic linear logic, with the !-modality treated with a DILL-style distinction between linear and intuitionistic variables. Intuitionistic variables are implemented simply by not checking for linearity in the !-E rule. Crary [2010, §4] also shows how to adapt this technique to a PD-style presentation of IS4.

As an example, let us look at the typing and linearity rules for the binary tensor product. Typing is given by a relation `of : term -> tp -> type`, where `tp` is the type of object-level types. Each syntactic form has a typing rule and potentially several

linearity rules, understood disjunctively as logic programming clauses. The rules for the introduction form are listed below. The typing relation is just like it would be for pairs in the simply typed λ-calculus: $(M, N) : A \otimes B$ if $M : A$ and $N : B$. The linearity rules are symmetric, so I will just consider `linear/tpair1`. It says that $x \vdash (M[x], N)$ is linear if $x \vdash M[x]$ is linear. This rule is subtle in that not applying `x` to `N` implies that `x` is fresh (and therefore unused) in `N`. Where $\otimes$-pairs have two linearity rules, the I-unit, and also the introduction form for !, have no linearity rules, meaning that no linear variables can be used in or by them.

```
of/tpair : of (tpair M N) (tensor A B) <- of M A <- of N B.
linear/tpair1 : linear ([x] tpair (M x) N) <- linear ([x] M x).
linear/tpair2 : linear ([x] tpair M (N x)) <- linear ([x] N x).
```

Meanwhile, the rules for the eliminator are somewhat more involved, thanks to the bound variables. First, the typing rule shows how `of` relies on `linear`, checking each bound variable for linearity. Because we have two bound variables, we need to check that the term `N` is linear in both. We do this by checking that, for all `y`, `N` is linear in `x`, and that for all `x`, `N` is linear in `y`. The linearity rules have the same choice and careful management of free variables as they did for the introduction form. In addition, in the rule `linear/lett2`, the bound variables in `N` have to be universally quantified while we check for linearity in the free variable `z`.

```
of/lett : of (lett M ([x] [y] N x y)) C
  <- of M (tensor A B) <- ({x} of x A -> {y} of y B -> of (N x y) C)
  <- ({y} linear ([x] N x y)) <- ({x} linear ([y] N x y))
linear/lett1 : linear ([z] lett (M z) ([x] [y] N x y))
  <- linear ([z] M z)
linear/lett2 : linear ([z] lett M ([x] [y] N z x y))
  <- ({x} {y} linear ([z] N z x y))
```

Crary [2010] goes on to extend this encoding with intuitionistic dependent Π-types, producing an object theory comparable to the $\lambda^{\Pi \multimap \& \top}$ metatheory developed

by Cervesato and Pfenning [2002]. If one wants to mix linearity and dependency following the methodology of Atkey [2018], then it is crucial that linear variables are still free in subterms from which they have been discarded. At first sight, it appears that Crary's encoding violates this because of its use of "does not appear free" to mean "is not used" in many linearity rules. However, one could imagine introducing an `unused` predicate similar to `linear` in order to handle unused free variables, at the cost of a few extra rules and a somewhat heavier encoding (scaling with the *size* of the term, rather than the depth). Indeed, one could imagine parametrising the `linear` predicate so as to encode the semiring-annotated systems I discuss in chapter 4.

The approach of Crary [2010] removes the objection to the work of Cervesato and Pfenning [2002] that each new substructural discipline would need a new proof assistant by encoding linearity in a standard intuitionistic logical framework. However, the encoding makes linear variables second-class compared to intuitionistic variables. While intuitionistic variables are just there thanks to the metatheory, linear variables must essentially be explicitly quantified.

Licata-Shulman-Riley Licata et al. [2017] describe a framework for specifying and working with a wide range of substructural logics. I discuss exactly what this range is in section 4.6, in relation to the calculi I describe in the rest of this thesis. For now, it suffices to say that this framework is specified in enough detail that it should be possible to mechanise it directly, but I am not aware of anyone having done so. Restall [1999] describes a similar approach.

Tanaka-Power The work of Fiore et al. [1999], which I discussed in section 2.6.2, has been extended to substructural syntaxes by Tanaka and Power [2006]. This work gives a mechanism for turning a description of contexts and their structural rules (expressed as a pseudo-monad on the 2-category of categories) into a framework for defining substructural syntaxes, and more generally a category of algebras of which the syntax is the initial object. As examples, they give the untyped λ-calculus, an untyped multiplicative linear logic, and a bunched logic. These examples show a broad range of substructural disciplines they support — comparable to the work of Licata et al. [2017] (which I dis-

cuss in section 4.6), and more than I discuss in this thesis. However, they also show two of the limitations of their approach. Firstly, this work provides no way to track types, though it should be possible to incorporate types at the expense of complicating the categorical constructions they use. Secondly, it appears to be impossible to encode the syntactic forms used for the additive connectives (i.e., the Cartesian product and coproduct) of linear logic. Essentially, subterms can only be combined into a larger term in the same ways as how contexts can be appended together. For example, in a bunched logic, contexts can be combined through both sharing conjunction (the Cartesian product) and separating conjunction (a monoidal product). Correspondingly, the syntax descriptions of Tanaka and Power allow for the syntax of sharing pairs and separating pairs. However, in the case of linear logic, contexts can only be combined via a monoidal product, so we only get separating pairs (tensor-products) and not sharing pairs (with-products).

Semiring-annotated systems There have been several appearances in the literature of calculi in which variables are annotated with some algebraic usage information which is reflected in the types of the calculus. Such calculi appear in the work of Abadi et al. [1999] and Reed and Pierce [2010], where annotations are used to control information flow and sensitivity to perturbations, respectively. Following these disparate calculi, the work of Brunel et al. [2014], Ghica and Smith [2014], and Petricek et al. [2014] sought to unify these calculi using (variations on) partially ordered semirings as the source of annotations. Each of these papers also provided further examples of calculi with semiring usage annotations. I discuss calculi based on partially ordered semiring annotations further in the following chapter, as it provides the basis for the rest of this thesis.

Chapter 4

Usage restriction via semirings

The methods described in chapter 2 for the simply typed λ-calculus make crucial use of *weakening* — the fact that if we have $\Gamma \vdash A$, then we also have $\Gamma, \Delta \vdash A$. We use this property to update environments as we take them under binders. However, as we saw in chapter 3, there are interesting calculi in which general weakening does not hold. As such, one of the aims of this chapter will be to find a form of weakening applicable to variables of any type, while essentially retaining linearity (as opposed to affineness).

This chapter proceeds as follows. In section 4.1, I give an intuitive introduction to semiring annotations on variables, based on replicating features of DILL (introduced in section 3.2.3). Then, section 4.2 formalises the ideas of section 4.1 into a calculus $\lambda\mathcal{R}$. This calculus has appeared in my previous work [Wood and Atkey, 2021], and can be seen as a simply typed version of Atkey's dependently typed calculus QTT [Atkey, 2018]. Given this new calculus $\lambda\mathcal{R}$, the first goal is to apply the techniques of chapter 2 to it, yielding a simultaneous substitution operation. To do this, I use section 4.3 to introduce notation that allows us to restate the typing rules of $\lambda\mathcal{R}$ to not mention contexts explicitly, as was the style in chapter 2. This new notation — the *bunched connectives* — is versatile at defining simply typed usage-aware syntaxes, and I give further non-$\lambda\mathcal{R}$ examples in section 4.4. Finally, I justify connections to linear logic and modal logic in section 4.5, where I translate $\lambda\mathcal{R}$ terms to and from DILL [Barber, 1996] and the modal calculus of Pfenning and Davies [1999].

This chapter and the following chapter re-present and expand the work of Wood and

Atkey [2021]. For the thesis version, I have dropped mention of *skew* semirings, which allows the algebraic components to be more robust and better abstracted. In particular, in chapter 5, I talk about linear maps rather than matrices, and define environments in terms of usage-checked variables rather than raw well typed variables.

4.1 Motivation for semiring annotations

The question of defining calculi which do not semantically admit weakening and contraction but also do not rely on variables going out of scope is directly addressed by McBride [2016]. The first technique suggested is to, instead of removing variables from the context of certain subterms, add an annotation to free variables saying whether or not they are to be used. I use an annotation 0 on variables that are not to be used, and an annotation 1 on variables that are to be used. This convention lets us transcribe the usual $\otimes$-introduction rule (below left) as a rule with usage annotations (below right). In the notation on the right, I let $\Gamma = \mathcal{P}\gamma$ and $\Delta = \mathcal{Q}\delta$, where Γ is a whole context comprising a *usage context* $\mathcal{P}$ and a *typing context* γ. A usage context is a list of usage annotations, so $\mathcal{P} = r_1, \ldots, r_m$ and a typing context is a list of types, so $\gamma = A_1, \ldots, A_m$. When combined, the usage context and the typing context will be of the same length. Explicit contexts will usually be written with usage annotations and types interspersed, as $r_1 A_1, \ldots, r_m A_m$. I use $r\gamma$ to abbreviate $rx_1, \ldots, rx_m$.

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B} \quad \rightsquigarrow \quad \frac{1\gamma, 0\delta \vdash A \qquad 0\gamma, 1\delta \vdash B}{1\gamma, 1\delta \vdash A \otimes B}$$

The eventual target of all these 0-annotated variables is the variable rule, which I transcribe as follows. The 1 shows us that we can use the variable thus annotated, while the 0s let us discard all of the other variables in γ.

$$\frac{}{A \vdash A} \quad \rightsquigarrow \quad \frac{}{0\gamma, 1A \vdash A}$$

The use of 0 gives us the property that variables never go out of scope in subterms; rather, we lose the ability to use certain variables, but retain the ability to refer to them metatheoretically. Additionally, we recover a form of weakening: if $\Gamma \vdash A$, then

also $\Gamma, 0\delta \vdash A$, because the resulting term indeed uses no variables from δ. I prove the admissibility of weakening for terms will come in section 5.3.

If we follow the DILL style of variable management explained in section 3.2.3, there are not just the two states *to be used* (1) and *not to be used* (0), but also *usable unrestrictedly*. If we assign unrestricted (or *intuitionistic*) variables an annotation ω, we can make the following transcription of the DILL $\otimes$-introduction rule.

$$\frac{\Theta; \Gamma \vdash A \qquad \Theta; \Delta \vdash B}{\Theta; \Gamma, \Delta \vdash A \otimes B} \quad \rightsquigarrow \quad \frac{\omega\theta, 1\gamma, 0\delta \vdash A \qquad \omega\theta, 0\gamma, 1\delta \vdash B}{\omega\theta, 1\gamma, 1\delta \vdash A \otimes B}$$

To conceptualise the criteria on the usage annotations involved in this rule, I introduce an additive structure over usage annotations. The rule stated above relies on the facts that $1 + 0 = 1$, $0 + 1 = 1$, and $\omega + \omega = \omega$. Addition lifts pointwise to vectors of usage annotations (the green capital calligraphic $\mathcal{P}$, $\mathcal{Q}$, and $\mathcal{R}$). A beneficial side-effect of the fact that $0 + 0 = 0$ is that the rule on the right below is actually more general, and accepts 0-annotated variables in its conclusion, which is essential for weakening to be admissible.

$$\frac{\omega\theta, 1\gamma, 0\delta \vdash A \qquad \omega\theta, 0\gamma, 1\delta \vdash B}{\omega\theta, 1\gamma, 1\delta \vdash A \otimes B} \quad \rightsquigarrow \quad \frac{\mathcal{R} = \mathcal{P} + \mathcal{Q} \qquad \mathcal{P}\gamma \vdash A \qquad \mathcal{Q}\gamma \vdash B}{\mathcal{R}\gamma \vdash A \otimes B}$$

Some other transcriptions from DILL to the usage annotation style are as follows. I unify the variable rules (the one for linear variables and the one for intuitionistic variables) by introducing a coercibility ordering $\leq$ on usage annotations. We have $\omega \leq 1$ because an intuitionistic variable can fill the demand of a linear variable by dereliction. We also have $\omega \leq 0$, because intuitionistic variables can be weakened away like 0-annotated variables. This ordering information is shown in the diagram $\begin{smallmatrix} 0 & & 1 \\ & \omega & \end{smallmatrix}$. All together, this means that at the (only) variable rule, the variable being used must have annotation less than or equal to 1, and every other variable must have annotation less than or equal to 0. I write this requirement as $\mathcal{R} \leq \langle x|$, where $\langle x|$ is the *basis vector* at position x.

$$\frac{}{\Theta; A \vdash A} \quad \rightsquigarrow \quad \frac{}{\omega\theta, 1A, 0\delta \vdash A} \quad \rightsquigarrow \quad \frac{\mathcal{R} \leq \langle x| \qquad \gamma_x = A}{\mathcal{R}\gamma \vdash A}$$

$$\frac{}{\Theta, A; \cdot \vdash A} \quad \rightsquigarrow \quad \frac{}{\omega\theta, \omega A, 0\delta \vdash A} \quad \rightsquigarrow \quad \frac{\mathcal{R} \leq \langle x| \qquad \gamma_x = A}{\mathcal{R}\gamma \vdash A}$$

The final interesting rule form to cover is found in DILL's !-introduction rule. DILL's !-introduction can be though of as an ω-ary counterpart to $\otimes$-introduction, though with the same premise each time rather than ω-many premises. This explains why only ω- and 0-annotated variables can appear in the conclusion of !-introduction, and also justifies the choice below of multiplication (vector scaling) as the algebraic operation controlling the !-modality.

$$\frac{\Theta; \cdot \vdash A}{\Theta; \cdot \vdash !A} \quad \rightsquigarrow \quad \frac{\omega\theta, 0\delta \vdash A}{\omega\theta, 0\delta \vdash !A} \quad \rightsquigarrow \quad \frac{\mathcal{R} \leq \omega\mathcal{P} \qquad \mathcal{P}\gamma \vdash A}{\mathcal{R}\gamma \vdash !_\omega A}$$

In summary, the structure we have required of the set of usage annotations is that they have addition (for $\otimes$-introduction and similar rules), multiplication (for !-introduction), a 1 (for a variable being used), a 0 (for a variable not being used), and an ordering (allowing for subsumption of usage restrictions). Together, these form a *partially ordered semiring* (posemiring), the laws of which are both supported by examples and necessary for the syntax to be well behaved.

Definition 4.1.1. A *semiring* is a monoid in the multicategory of commutative monoids and multilinear maps. Unpacked, this means that we have a set $\mathscr{R}$ together with elements 0 and 1 and binary operators $+$ and $\cdot$ (with $\cdot$ usually written as juxtaposition) such that the following hold for all $x, y, z \in \mathscr{R}$.

- $0 + x = x$; $x + 0 = x$; $(x + y) + z = x + (y + z)$; $x + y = y + x$
- $1x = x$; $x1 = x$; $(xy)z = x(yz)$
- $0x = 0$; $x0 = 0$; $(x + y)z = xz + yz$; $x(y + z) = xy + xz$

Definition 4.1.2. A *posemiring* is a semiring in the category of partially ordered sets (posets). Unpacked, this means that we have a semiring (in the category of sets)

$$A, B, C ::= I \mid A \otimes B \mid A \multimap B \mid \top \mid A \,\&\, B \mid 0 \mid A \oplus B \mid !_r A$$

Figure 4.1: The types of $\lambda\mathcal{R}$

$(\mathscr{R}, 0, +, 1, \cdot)$ such that $\mathscr{R}$ has a partial order (written $\leq$) and addition and multiplication are monotonic with respect to $\leq$ in both arguments.

For concreteness, I collect together the definition of the $\{0, 1, \omega\}$ posemiring I have been using so far.

Example 4.1.3. The $\{0, 1, \omega\}$ *posemiring*, also known as the *linearity posemiring*, has the operations given as follows, with $0 \coloneqq 0$ and $1 \coloneqq 1$:

$+$	0	1	ω
0	0	1	ω
1	1	ω	ω
ω	ω	ω	ω

$*$	0	1	ω
0	0	0	0
1	0	1	ω
ω	0	ω	ω

4.2 A usage-annotated calculus $\lambda\mathcal{R}$

In this section, I introduce the syntax of the type theory $\lambda\mathcal{R}$, which makes use of posemiring usage annotations to express the usage restrictions found in DILL and other calculi. For the rest of this thesis, particularly chapters 4 and 5, $\lambda\mathcal{R}$ will serve as both a prototypical usage-constrained syntax and a target of semantic analyses.

The calculus $\lambda\mathcal{R}$ is similar in spirit to intuitionistic linear logic (ILL), which we saw in chapter 3. The types of $\lambda\mathcal{R}$, listed in figure 4.1, are almost identical to those of ILL, differing only in the exponential modality ! (read "bang"). In particular, I include distinguished tensor- and with-product types ($\otimes$, &) and their units (I, $\top$), function types ($\multimap$), additive sum types and their unit ($\oplus$, 0), and the graded modality $!_r$. The idea of $!_r$ is to internalise an annotation of r on a variable in the context.

I will not cover any operational semantics or equational theory of $\lambda\mathcal{R}$ in this thesis. I will discuss a denotational semantics inspired by that of Abel and Bernardy [2020] in section 8.3.

The following features are of note.

$$\frac{\gamma \ni x : A \qquad \mathcal{P} \leq \langle x|}{\mathcal{P}\gamma \vdash A}\ \textsc{Var}$$

$$\frac{\mathcal{P} \leq 0}{\mathcal{P}\gamma \vdash I}\ I\text{-I} \qquad \frac{\mathcal{R} \leq \mathcal{P} + \mathcal{Q} \qquad \mathcal{P}\gamma \vdash I \qquad \mathcal{Q}\gamma \vdash C}{\mathcal{R}\gamma \vdash C}\ I\text{-E}$$

$$\frac{\mathcal{R} \leq \mathcal{P} + \mathcal{Q} \qquad \begin{array}{l}\mathcal{P}\gamma \vdash A \\ \mathcal{Q}\gamma \vdash B\end{array}}{\mathcal{R}\gamma \vdash A \otimes B}\ \otimes\text{-I} \qquad \frac{\begin{array}{l}\mathcal{R} \leq \mathcal{P} + \mathcal{Q} \\ \mathcal{P}\gamma \vdash A \otimes B\end{array} \qquad \mathcal{Q}\gamma, 1A, 1B \vdash C}{\mathcal{R}\gamma \vdash C}\ \otimes\text{-E}$$

$$\frac{\mathcal{R}\gamma, 1A \vdash B}{\mathcal{R}\gamma \vdash A \multimap B}\ \multimap\text{-I} \qquad \frac{\mathcal{R} \leq \mathcal{P} + \mathcal{Q} \qquad \mathcal{P}\gamma \vdash A \multimap B \qquad \mathcal{Q}\gamma \vdash A}{\mathcal{R}\gamma \vdash B}\ \multimap\text{-E}$$

$$\frac{}{\mathcal{R}\gamma \vdash \top}\ \top\text{-I} \qquad (\text{no } \top\text{-E})$$

$$\frac{\mathcal{R}\gamma \vdash A \qquad \mathcal{R}\gamma \vdash B}{\mathcal{R}\gamma \vdash A \,\&\, B}\ \&\text{-I} \qquad \frac{\mathcal{R}\gamma \vdash A_0 \,\&\, A_1}{\mathcal{R}\gamma \vdash A_i}\ \&\text{-E}_i,\ \textsc{for}\ i \in \{0, 1\}$$

$$(\text{no } 0\text{-I}) \qquad \frac{\mathcal{R} \leq \mathcal{P} + \mathcal{Q} \qquad \mathcal{P}\gamma \vdash 0}{\mathcal{R}\gamma \vdash C}\ 0\text{-E}$$

$$\frac{\mathcal{R}\gamma \vdash A_i}{\mathcal{R}\gamma \vdash A_0 \oplus A_1}\ \oplus\text{-I}_i,\ \textsc{for}\ i \in \{0, 1\} \qquad \frac{\begin{array}{ll}\mathcal{R} \leq \mathcal{P} + \mathcal{Q} & \mathcal{Q}\gamma, 1A \vdash C \\ \mathcal{P}\gamma \vdash A \oplus B & \mathcal{Q}\gamma, 1B \vdash C\end{array}}{\mathcal{R}\gamma \vdash C}\ \oplus\text{-E}$$

$$\frac{\mathcal{R} \leq r\mathcal{P} \qquad \mathcal{P}\gamma \vdash A}{\mathcal{R}\gamma \vdash\, !rA}\ !\text{-I} \qquad \frac{\mathcal{R} \leq \mathcal{P} + \mathcal{Q} \qquad \mathcal{P}\gamma \vdash\, !rA \qquad \mathcal{Q}\gamma, rA \vdash C}{\mathcal{R}\gamma \vdash C}\ !\text{-E}$$

Figure 4.2: $\lambda\mathcal{R}$

Subusaging Several typing rules contain constraints of the form $\mathcal{P} \leq \mathcal{Q}$, for certain usage vectors $\mathcal{P}$ and $\mathcal{Q}$. We saw subusaging in the introduction to this chapter in the specific case of $\mathscr{R}$ being formed from the poset $\{0 > \omega < 1\}$. This allowed variables annotated ω ("unrestricted") to be both weakened/discarded (because $\omega \leq 0$) and dereliicted/used (because $\omega \leq 1$). Subsumption of usage annotations is essential to nearly all interesting choices of $\mathscr{R}$. However, in the toy example of exact usage counting using the set $\mathbb{N}$ of annotations, we set the order to be just equality as a matter of simplicity.

For usage annotations r and s, the inequality $r \leq s$ states that an assumption with annotation r can be used wherever an assumption with annotation s is required. A mnemonic is that r is less specific than s. The principle is reflected by the admissible subusaging rule, where the order has been lifted from annotations to usage contexts. The subusaging rule is a simple corollary of renaming, as given in chapter 5.

$$\frac{\mathcal{P} \leq \mathcal{Q} \qquad \mathcal{Q}\gamma \vdash A}{\mathcal{P}\gamma \vdash A}\ \textsc{Subuse}$$

Note that my subusaging order is reversed relative to the similar ordering found in previous work — particularly Orchard et al. [2019] and related work from those authors. I find the order I use preferable because it matches the standard for simultaneous renaming and substitution (we will see later that a $\leq$-relationship between contexts gives rise to a renaming), and all of these make rules like Subuse interpretable as generalised notions of composition between orderings/renamings/substitutions and terms.

Tensor- and with-products Like intuitionistic linear logic (ILL), $\lambda\mathcal{R}$ distinguishes tensor-products ($A \otimes B$) from with-products ($A \,\&\, B$). Whereas in ILL, rules like $\otimes$-introduction involve splitting the assumptions between the two subterms, in $\lambda\mathcal{R}$, this splitting is done by choosing usage annotations for the premises which add up to the usage annotations of the conclusion. For example, we can derive $\vdash A \otimes B \multimap B \otimes A$ as follows. Notice that the assumption $A \otimes B$ is still present in all subderivations, even after it has been "used up". The only thing that stops us using the assumption again is that, for a general choice of $\mathscr{R}$, we do not have $0 \leq 1$ or $1 \leq 1 + 1$.

$$\nabla \coloneqq \dfrac{\dfrac{}{(0\ 1\ 1) \leq (0\ 0\ 1) + (0\ 1\ 0)} \quad \dfrac{\dfrac{}{(0\ 0\ 1) \leq (0\ 0\ 1)}}{0(A \otimes B), 0A, 1B \vdash B}\ \textsc{Var} \quad \dfrac{\dfrac{}{(0\ 1\ 0) \leq (0\ 1\ 0)}}{0(A \otimes B), 1A, 0B \vdash A}\ \textsc{Var}}{0(A \otimes B), 1A, 1B \vdash B \otimes A}\ \otimes\text{-I}$$

$$\dfrac{\dfrac{}{(1) \leq (1) + (0)} \quad \dfrac{\dfrac{}{(1) \leq (1)}}{1(A \otimes B) \vdash A \otimes B}\ \textsc{Var} \quad \begin{matrix}\nabla \\ 0(A \otimes B), 1A, 1B \vdash B \otimes A\end{matrix}}{\dfrac{1(A \otimes B) \vdash B \otimes A}{\vdash A \otimes B \multimap B \otimes A}\ \multimap\text{-I}}\ \otimes\text{-E}$$

Example 4.2.1. Let $A \circ\!\!-\!\!\circ B$ abbreviate $(A \multimap B) \mathbin{\&} (B \multimap A)$. Then the following judgements hold for any partially ordered semiring. Derivations are left as an exercise to the reader.

- $\vdash A \oplus A \multimap A$
- $\vdash A \multimap A \mathbin{\&} A$
- $\vdash A \oplus 0 \circ\!\!-\!\!\circ A$
- $\vdash A \otimes 0 \circ\!\!-\!\!\circ 0$
- $\vdash\ !1A \circ\!\!-\!\!\circ A$
- If $r \leq s$, then $\vdash\ !rA \multimap\ !sA$

Example 4.2.2. Let $\mathscr{R} \coloneqq (\mathbb{N}, =, 0, +, 1, \times)$, that is, specialise to the posemiring made of the usual semiring of natural numbers with ordering given by equality. Under this discipline, the usage constraints enforce a form of exact usage counting. The following judgements then hold. Derivations are left as an exercise to the reader.

- $\vdash\ !2A \multimap A \otimes A$
- $\vdash\ !5A \multimap\ !2A \otimes\ !3A$

4.2.1 Other posemirings

Now that we have seen the role of usage annotations in $\lambda\mathcal{R}$, I will give more examples of posemirings for tracking interesting usage patterns.

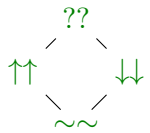
Example 4.2.3. The singleton set gives rise to a posemiring in a unique way. When the usage annotations of $\lambda\mathcal{R}$ are taken from this trivial posemiring, we recover a version of intuitionistic simply typed λ-calculus featuring redundant connectives $\otimes$ (equivalent to & in the pure setting) and $!*$ (where $!*A \simeq A$).

Example 4.2.4. The *monotonicity* posemiring is defined over the set of symbols $\{??, \uparrow\uparrow, \downarrow\downarrow, \sim\sim\}$. The idea is that each symbol represents the possible *variance* of an input (free variable) with respect to some partial ordering on a semantic domain of elements. $\uparrow\uparrow$ represents covariance (if that input goes up, the output goes up), $\downarrow\downarrow$ represents contravariance (if that input goes *down*, the output goes up), $\sim\sim$ gives no guarantees (if that input remains constant, the output (trivially) goes up), and $??$ says that that input is irrelevant (whatever changes are made to that input, the output (trivially) goes up).

I take $0 \coloneqq ??$, $1 \coloneqq \uparrow\uparrow$, and define the following operations:

$+$	$??$	$\uparrow\uparrow$	$\downarrow\downarrow$	$\sim\sim$
$??$	$??$	$\uparrow\uparrow$	$\downarrow\downarrow$	$\sim\sim$
$\uparrow\uparrow$	$\uparrow\uparrow$	$\uparrow\uparrow$	$\sim\sim$	$\sim\sim$
$\downarrow\downarrow$	$\downarrow\downarrow$	$\sim\sim$	$\downarrow\downarrow$	$\sim\sim$
$\sim\sim$	$\sim\sim$	$\sim\sim$	$\sim\sim$	$\sim\sim$

$*$	$??$	$\uparrow\uparrow$	$\downarrow\downarrow$	$\sim\sim$
$??$	$??$	$??$	$??$	$??$
$\uparrow\uparrow$	$??$	$\uparrow\uparrow$	$\downarrow\downarrow$	$\sim\sim$
$\downarrow\downarrow$	$??$	$\downarrow\downarrow$	$\uparrow\uparrow$	$\sim\sim$
$\sim\sim$	$??$	$\sim\sim$	$\sim\sim$	$\sim\sim$



Addition represents an intersection of guarantees. For example, if a variable is used covariantly in one subterm and contravariantly in another, we can only make the trivial guarantee represented by $\sim\sim$. Multiplication is mainly interesting for multiplication by $\downarrow\downarrow$, which flips the variance on any other annotation. As such, $!\downarrow\downarrow A$ represents a contravariant A. The flipping (involutive) behaviour of $\downarrow\downarrow$ lets us notice that x is covariant in a term like $-(-x)$, where $-$ is a constant of type $!\downarrow\downarrow\mathbb{Z} \multimap \mathbb{Z}$.

A similar, but distinct, collection of modalities for monotonicity is given by Arntzenius [2019].

Example 4.2.5. The *sensitivity* posemiring [Reed and Pierce, 2010] is given by $(\mathbb{R}^+, \geq, 0, +, 1, \times)$, where $\mathbb{R}^+$ is the non-negative real numbers extended with ∞ (distances), and the rest of the structure comes from the standard operations on real numbers (except that $0 \times \infty = \infty \times 0 = 0$). Note that the order is reversed, making ∞ coercible to any other annotation and anything coercible to 0.

The intended semantics for these annotations is to interpret types as metric spaces and terms as Lipschitz-continuous maps. That is to say, each type A comes with a notion of distance d_A, and a map $f : A \to B$ satisfies

$$\exists r \in \mathbb{R}^+. \forall x, y.\ d_B(f(x), f(y)) \leq r d_A(x, y).$$

I internalise this constant r so that derivations of $rA \vdash B$ are interpreted as r-Lipschitz-continuous functions. If $r = \infty$, then the Lipschitz condition is trivially satisfied, meaning that ∞-annotated variables can be used without constraint. If $r = 0$, then f has to be constant, and the corresponding syntactic constraint is that a 0-annotated variable cannot be used.

Addition forbids general contraction, which would otherwise allow arbitrary finite blow-up of the effect of any non-0-annotated variable. However, the ordering, with 0 at the top, means that we have general weakening, so the resulting sensitivity calculus has an affine flavour.

Such a system has been applied to sensitivity analysis, a component of differential privacy, by Reed and Pierce [2010].

4.3 Bunched connectives

The typing rules of $\lambda\mathcal{R}$ presented in figure 4.2 contain a lot of detail and repeated patterns. For example, nearly half of the rules include the premise $\mathcal{R} \leq \mathcal{P} + \mathcal{Q}$. Also, the presence of usage annotations, which are often different in different parts of a rule, means that we keep repeating the context. Explicit contexts go against the style we established in chapter 2, which is based around being parametric in the context, so that substitution is agnostic to the details of typing rules.

To encapsulate the repeated patterns and facilitate an implicit context style, I introduce the *bunched connectives* for premises. These are inspired by bunched logic [O'Hearn and Pym, 1999], and will not only be used for stating the syntax, but will be used as an abstraction of common patterns in the development of the metatheory. The idea is to generalise the space between premises from Gentzen's natural deduction to allow for any linear combination of usage annotations. Among other things, this generalisation will allow us to distinguish between &-introduction and ⊗-introduction by a choice of connective: either *sharing* or *separating* conjunction. These connectives are defined in figure 4.3 in Agda notation.

The bunched connectives are parametrised over two sets and three relations. For syntax, the set A will be Ctx, the type of contexts, and R will be Ann, the type of usage annotations (scalars). For the relations, the notation is meant to be suggestive, with $\Gamma \leq 0$ typically stating that all of the annotations in Γ are less than or equal to 0; $\Gamma \leq\!\!\lfloor\ \Delta\ +\ \Theta\ \rfloor$ typically stating that Γ, Δ, and Θ all agree on their types but the usage context of Γ is less than or equal to the sum of the usage contexts of Δ and Θ; and $\Gamma \leq\!\!\lfloor\ r\ *_l\ \Delta\ \rfloor$ typically stating that Γ and Δ agree on their types but have the evident scaling relationship with r on their usage annotations. I use the same symbols for the connectives both in Agda code and in otherwise standard mathematical/logical notation.

The first two connectives are those we've already seen for intuitionistic systems — $\dot{1}$ and $\dot{\times}$. The absence of premises is encoded by $\dot{1}$, while the space between premises sharing a context is encoded by $\dot{\times}$. As for implication, I temporarily avoid giving a $\dot{\to}$ connectives, instead fusing it together with $\forall[\ -\ ]$ to produce the *set* of context-preserving functions $T \multimap U$. When we interpret a typing rule as a constructor of an inductive definition, $\multimap$ interprets the horizontal line, reflecting the fact that the usage annotations we start off with in the premises are those of the conclusion, corresponding to a general principle of resource conservation. The prototypical rules that use $\dot{1}$ and $\dot{\times}$ are the introduction rules for $\top$ and &, respectively.

Parameters:

```
{A R : Set} (_≤0 : A → Set) (_≤⌊_+_⌋ : A → A → A → Set)
(_≤⌊_ *ₗ_⌋ : A → R → A → Set)
```

Connectives:

```
i : A → Set
i _ = ⊤

_⇾_ : (T U : A → Set) → Set
T ⇾ U = ∀ {x} → (T x → U x)

_×̇_ : (T U : A → Set) → A → Set
(T ×̇ U) x = T x × U x

record I* (x : A) : Set where
  constructor I*⟨_⟩
  field
    split : x ≤0

record _*_ (T U : A → Set) (x : A) : Set where
  constructor _*⟨_⟩_
  field
    {y z} : A
    T-prf : T y
    split : x ≤⌊ y + z ⌋
    U-prf : U z

record _-*_ (T U : A → Set) (x : A) : Set where
  constructor lam*
  field
    app* : ∀ {y z} (split : z ≤⌊ x + y ⌋) (T-prf : T y) → U z

record _·_ (r : R) (T : A → Set) (x : A) : Set where
  constructor ⟨_⟩·_
  field
    {z} : A
    split : x ≤⌊ r *ₗ z ⌋
    T-prf : T z
```

Figure 4.3: The bunched connectives

$$\frac{}{\mathcal{R}\Gamma \vdash \top} \quad \rightsquigarrow \quad \frac{\mathrm{i}}{\vdash \top}$$

$$\frac{\mathcal{R}\Gamma \vdash A \qquad \mathcal{R}\Gamma \vdash B}{\mathcal{R}\Gamma \vdash A \,\&\, B} \quad \rightsquigarrow \quad \frac{\vdash A \quad \dot{\times} \quad \vdash B}{\vdash A \,\&\, B}$$

The rest of the bunched connectives — I^*, $*$, $\cdot$, and $-\!\!*$ — involve linear decompositions of the usage annotations. The three basic left semimodule operators — zero, addition, and left-scaling — each get a bunched connective — I^*, $*$, and $r\cdot$, respectively. The prototypical typing rules for each of these three connectives are the introduction rules for I, $\otimes$, and $!r$, respectively.

$$\frac{\mathcal{R} \leq 0}{\mathcal{R}\Gamma \vdash I} \quad \rightsquigarrow \quad \frac{I^*}{\vdash I}$$

$$\frac{\mathcal{P}\Gamma \vdash A \qquad \mathcal{Q}\Gamma \vdash B \qquad \mathcal{R} \leq \mathcal{P} + \mathcal{Q}}{\mathcal{R}\Gamma \vdash A \otimes B} \quad \rightsquigarrow \quad \frac{\vdash A \quad * \quad \vdash B}{\vdash A \otimes B}$$

$$\frac{\mathcal{P}\Gamma \vdash A \qquad \mathcal{R} \leq r\mathcal{P}}{\mathcal{R}\Gamma \vdash\, !rA} \quad \rightsquigarrow \quad \frac{r \cdot (\vdash A)}{\vdash\, !rA}$$

4.3.1 $\lambda\mathcal{R}$ stated using bunched connectives

The full system $\lambda\mathcal{R}$ is stated in terms of bunched connectives in figure 4.4. The bunched connectives also yield a reasonably concise definition of the Agda data type of $\lambda\mathcal{R}$ derivations, as seen in figure 4.5.

4.3.2 Connection with bunched logic

While we have seen a connection between the bunched connectives and the connectives of $\lambda\mathcal{R}$, the two should not be confused. In particular, the bunched connectives obey different laws to what we would expect from linear logic. For example, it would make sense to define a bunched connective $\dot{+}$, defined analogously to $\dot{\times}$. This $\dot{+}$ could be

$$\frac{\exists\, A}{\vdash A}\ \textsc{Var}$$

$$\frac{I^*}{\vdash I}\ I\text{-I} \qquad \frac{\vdash I \quad * \quad \vdash C}{\vdash C}\ I\text{-E}$$

$$\frac{\vdash A \quad * \quad \vdash B}{\vdash A \otimes B}\ \otimes\text{-I} \qquad \frac{\vdash A \otimes B \quad * \quad 1A, 1B \vdash C}{\vdash C}\ \otimes\text{-E}$$

$$\frac{1A \vdash B}{\vdash A \multimap B}\ \multimap\text{-I} \qquad \frac{\vdash A \multimap B \quad * \quad \vdash A}{\vdash B}\ \multimap\text{-E}$$

$$\frac{\dot{1}}{\vdash \top}\ \top\text{-I} \qquad \text{(no } \top\text{-E)}$$

$$\frac{\vdash A \quad \dot{\times} \quad \vdash B}{\vdash A \,\&\, B}\ \&\text{-I} \qquad \frac{\vdash A_0 \,\&\, A_1}{\vdash A_i}\ \&\text{-E}_i,\ \textsc{for}\ i \in \{0,1\}$$

$$\text{(no 0-I)} \qquad \frac{\vdash 0 \quad * \quad \dot{1}}{\vdash C}\ 0\text{-E}$$

$$\frac{\vdash A_i}{\vdash A_0 \oplus A_1}\ \oplus\text{-I}_i,\ \textsc{for}\ i \in \{0,1\} \qquad \frac{\vdash A \oplus B \quad * \quad (1A \vdash C \quad \dot{\times} \quad 1B \vdash C)}{\vdash C}\ \oplus\text{-E}$$

$$\frac{r \cdot (\vdash A)}{\vdash\, !rA}\ !\text{-I} \qquad \frac{\vdash\, !rA \quad * \quad rA \vdash C}{\vdash C}\ !\text{-E}$$

Figure 4.4: $\lambda\mathcal{R}$ stated using bunched connectives

```
data Ty : Set where
  ι I ⊤ O : Ty
  _⊗_ _⊸_ _&_ _⊕_ : (A B : Ty) → Ty
  ! : Ann → Ty → Ty

Bind : Ctx → (Ctx → Set) → (Ctx → Set)
Bind Δ T Γ = T (Γ ++ᶜ Δ)

data _⊢_ : Ctx → Ty → Set where
  var  : _∋ A                                  → _⊢ A
  Ii   : I*                                    → _⊢ I
  Ie   : _⊢ I ∗ _⊢ C                           → _⊢ C
  ⊗i   : _⊢ A ∗ _⊢ B                           → _⊢ A ⊗ B
  ⊗e   : _⊢ A ⊗ B ∗ Bind ([ 1# • A ]ᶜ ++ᶜ [ 1# • B ]ᶜ) (_⊢ C)
         ⇾
         _⊢ C
  ⊸i   : Bind [ 1# • A ]ᶜ (_⊢ B)              → _⊢ A ⊸ B
  ⊸e   : _⊢ A ⊸ B ∗ _⊢ A                       → _⊢ B
  ⊤i   : ⊤                                     → _⊢ ⊤
  &i   : _⊢ A ×̇ _⊢ B                           → _⊢ A & B
  &e0  : _⊢ A & B                              → _⊢ A
  &e1  : _⊢ A & B                              → _⊢ B
  Oe   : _⊢ O ∗ i                              → _⊢ C
  ⊕i0  : _⊢ A                                  → _⊢ A ⊕ B
  ⊕i1  : _⊢ B                                  → _⊢ A ⊕ B
  ⊕e   : _⊢ A ⊕ B ∗ (Bind [ 1# • A ]ᶜ (_⊢ C) ×̇ Bind [ 1# • B ]ᶜ (_⊢ C))
         ⇾
         _⊢ C
  !i   : r · _⊢ A                              → _⊢ ! r A
  !e   : _⊢ ! r A ∗ Bind [ r • A ]ᶜ (_⊢ C)     → _⊢ C
```

Figure 4.5: $\lambda\mathcal{R}$ stated using bunched connectives in Agda

used to rephrase the introduction rules for $\oplus$. We then have maps both ways between $T \mathbin{\dot\times} (U \mathbin{\dot+} V)$ and $(T \mathbin{\dot\times} U) \mathbin{\dot+} (T \mathbin{\dot\times} V)$, reminiscent of the distributivity of additive connectives in bunched logic, whereas linear logic only has a map from $(A \& B) \oplus (A \& C)$ to $A \mathbin{\&} (B \oplus C)$, and not a map the other way. Looking at the interpretations, the connection with bunched logic makes sense. Instead of the partial commutative monoid (often representing heaps) found in standard semantics of bunched logic, we have a left semimodule of usage contexts, which we are similarly interested in splitting and sharing between various subterms.

From bunched logic, we would expect the Cartesian product $\mathbin{\dot\times}$ to have an internal hom. In the intuitionistic case, $\mathbin{\dot\to}$ filled this role. However, with usage contexts, it makes sense for open types to be presheaves over the partial order of contexts under pointwise $\leq$ of usage annotations. The family $T \mathbin{\dot\to} U$ does not satisfy the functoriality condition because of the contravariance in the domain T. Instead, as found in models of bunched logic, we would want a Kripke function space, like $\lambda\Gamma.\ \forall\Gamma' \leq \Gamma.\ T\,\Gamma' \to U\,\Gamma'$. However, I do not make use of such a connective.

The separating conjunction $*$ can be seen as a decategorified version of Day convolution [Day, 1970]. It also resembles the use of ternary frames in semantics of non-distributive logics [Restall, 1999, chapter 12].

4.3.3 Operations on bunched connectives

To manipulate terms and other open types defined using bunched connectives, we need the zero, addition, and multiplication relations to satisfy some laws. For example, to achieve a symmetry map $T * U \mathrel{-\!\!*} U * T$, we need addition to satisfy the commutativity law $\forall x, y, z : A.\ x \leq y + z \to x \leq z + y$.

For all uses of bunched connectives in this thesis, the carrier set A will form a partial order — for example, with contexts, the order is given by the pointwise order on the usage vectors. We then consider the category whose objects are posets and whose morphisms are relations $R : A \nrightarrow B$ satisfying the contravariant-covariant law $\forall x, x', y, y'.\ x' \leq x \to y \leq y' \to xRy \to x'Ry'$. This category can be given the usual monoidal product of relations, which is the pointwise product of posets on ob-

jects. Then, we will always expect zero and addition to together form a cocommutative comonoid in this category. With this structure, we can get the following equivalences and functions.

$$I^* * T \dashv\vdash T \qquad T * I^* \dashv\vdash T \qquad (T * U) * V \dashv\vdash T * (U * V) \qquad T * U \dashv\vdash U * T$$

$$(T \mathrel{-\!\!*} U) * T \multimap U \qquad I^* \mathrel{-\!\!*} T \dashv\vdash T \qquad (T * U) \mathrel{-\!\!*} V \dashv\vdash T \mathrel{-\!\!*} (U \mathrel{-\!\!*} V)$$

I do not use algebraic properties of multiplication in conjunction with manipulation of bunched connectives in this thesis, but we could expect scalar multiplication to add a comodule structure over cosemiring R to the cocommutative comonoid given by zero and addition.

4.4 Additions to and variations of $\lambda\mathcal{R}$

The bunched connectives give us a means by which to quickly design and experiment with new type systems. We will see in chapter 6 that all such syntaxes are well behaved to the extend that they all support the appropriate notion of substitution, which again expedites experimentation. In this section, I give some example syntactic features that can be covered in the bunched connective paradigm.

4.4.1 Alternative object-language connectives

I presented $\lambda\mathcal{R}$ as an essentially arbitrary collection of rules, with no broader characterisation than that it intuitively seems to capture some notion of "usage" and that it supports the semantics I give in section 8.3. Therefore, in this subsection I give some examples of variant syntaxes others, with different use cases, may be interested in. In particular, I give a variant of function types where usage annotations go "on the arrow", and a variant of tensor products with a stronger elimination rule.

We can produce an annotated function arrow connective by combining parts of the rules for the unannotated function arrow $\multimap$ with parts of the rules for the modal operator !.

$$\frac{rA \vdash B}{\vdash rA \multimap B}\ r\ \multimap\text{-I} \qquad\qquad \frac{\vdash rA \multimap B \quad * \quad r \cdot (\vdash A)}{\vdash B}\ r\ \multimap\text{-E}$$

We can also give an alternative elimination rule for $\otimes$-products, as found in Granule [Hughes et al., 2021, Orchard et al., 2019] and the work of Abel and Bernardy [2020] and Reed and Pierce [2010]. This modified rule allows us to, for example, have a variable $rx : A \otimes B$ for any r, and pattern-match it into $ry : A, rz : B$. With the standard $\lambda\mathcal{R}$ rule from figure 4.2, the newly bound variables are always given annotation 1, and we would only be able to do the match in the first place if r were coercible to (i.e. less than or equal to) 1. The alternative rule below is not compatible with linear logic, because it allows us to derive $!r(A \otimes B) \multimap !rA \otimes !rB$ parametric in r, A, and B.

$$\frac{r \cdot (\vdash A \otimes B) \quad * \quad rA, rB \vdash C}{\vdash C}\ \otimes\text{-E}'$$

4.4.2 Adding inductive types and recursion

Based on an intuitive understanding of "usage", recursion introduces a new phenomenon relative to the forms of programs we have seen so far: Terms can be used an unbounded number of times. For example, notice the following reduction in Agda.

```
foldr _+_ 0 (1 :: 2 :: 3 :: [])      ⇝
1 + foldr _+_ 0 (2 :: 3 :: [])       ⇝
1 + (2 + foldr _+_ 0 (3 :: []))      ⇝
1 + (2 + (3 + foldr _+_ 0 []))       ⇝
1 + (2 + (3 + 0))
```

The function `_+_` has been copied into 3 different places in the running of the program. This copying is despite no type telling us that `_+_` would be used 3 times (both `[1,2,3]` and `[2,3]` have type List $\mathbb{N}$, despite the corresponding folds using `_+_` a different number of times). As such, when checking an application of foldr, we need check that we can use its functional argument (`_+_` in this case) an arbitrary number

of times. If we were to fix $\mathscr{R}$ as the $\{0, 1, \omega\}$ posemiring, then wrapping the type of the functional argument in $!\omega$ would suffice. However, we want to remain generic in the choice of semiring.

The following additions to $\lambda\mathcal{R}$ support a broad class of inductive types. I define strictly positive functors syntactically, with the only notable restrictions being not being allowed to use the type variable X in the domain of a function type and within a !-type. I then add least fixed points of such strictly positive functors to the syntax of types.

$$
\begin{aligned}
U &::= A \multimap (-) \\
\odot &::= \otimes \mid \oplus \mid \& \\
F[X], G[X] &::= X \mid A \mid U(F[X]) \mid F[X] \odot G[X] \\
A &::= \cdots \mid \mu X.\ F[X]
\end{aligned}
$$

Example 4.4.1. We may define $\mathrm{List}_A \coloneqq \mu X.\ I \oplus (A \otimes X)$.

In the typing rules, introduction of an inductive type is standard. For the elimination rule, we follow a similar pattern to other pattern-matching rules — $\oplus$-E, $\otimes$-E, and !-E — by splitting the context and typing the eliminand in one half ($\mathcal{P}$). We type the continuation in the other half, but because the continuation may be used multiple times, and in a modal context, we require that $\mathcal{Q}$ is preserved by all linear operations.

$$
\frac{\mathcal{R}\gamma \vdash F[\mu X.\ F[X]]}{\mathcal{R}\gamma \vdash \mu X.\ F[X]}\ \mu\text{-I}
$$

$$
\frac{\mathcal{R} \leq \mathcal{P} + \mathcal{Q} \qquad \mathcal{P}\gamma \vdash \mu X.\ F[X] \qquad \begin{array}{l} \mathcal{Q} \leq 0 \\ \mathcal{Q} \leq \mathcal{Q} + \mathcal{Q} \end{array} \qquad \mathcal{Q}\gamma, 1F[C] \vdash C}{\mathcal{R}\gamma \vdash C}\ \mu\text{-E}
$$

Example 4.4.2. For lists, we can derive the following introduction and elimination rules (with usage constraints in the application of μ-E in the FOLDR rule, relating $\mathcal{R}$ to $\mathcal{P}$ and $\mathcal{Q}$ and restricting $\mathcal{Q}$, omitted to save space).

$$\textsc{nil}\left[\dfrac{\dfrac{\mathcal{R} \leq 0}{\mathcal{R}\gamma \vdash I}\, I\text{-I}}{\dfrac{\mathcal{R}\gamma \vdash I \oplus (A \otimes \mathrm{List}_A)}{\mathcal{R}\gamma \vdash \mathrm{List}_A}\, \mu\text{-I}}\, \oplus\text{-I}_0\right] \quad \textsc{cons}\left[\dfrac{\dfrac{\dfrac{\mathcal{R} \leq \mathcal{P} + \mathcal{Q} \quad \mathcal{P}\gamma \vdash A \quad \mathcal{Q}\gamma \vdash \mathrm{List}_A}{\mathcal{R}\gamma \vdash A \otimes \mathrm{List}_A}\, \otimes\text{-I}}{\mathcal{R}\gamma \vdash I \oplus (A \otimes \mathrm{List}_A)}\, \oplus\text{-I}_1}{\mathcal{R}\gamma \vdash \mathrm{List}_A}\, \mu\text{-I}\right]$$

$$\textsc{foldr}\left[\dfrac{\mathcal{P}\gamma \vdash \mathrm{List}_A \qquad \dfrac{\dfrac{}{0\gamma, 1(I \oplus (A \otimes C)) \vdash I \oplus (A \otimes C)}\,\textsc{Var} \qquad \nabla^n \qquad \nabla^c}{\mathcal{Q}\gamma, 1(I \oplus (A \otimes C)) \vdash C}\, \oplus\text{-E}}{\mathcal{R}\gamma \vdash C}\, \mu\text{-E}\right]$$

$$\text{where } \nabla^n := \dfrac{\dfrac{\dfrac{}{0\gamma, 1I \vdash I}\,\textsc{Var} \qquad \dfrac{\mathcal{Q}\gamma \vdash C}{\mathcal{Q}\gamma, 0I \vdash C}\,\textsc{Wk}}{\mathcal{Q}\gamma, 1I \vdash C}\, I\text{-E}}{\mathcal{Q}\gamma, 0(I \oplus (A \otimes C)), 1I \vdash C}\,\textsc{Wk}$$

$$\text{and } \nabla^c := \dfrac{\dfrac{\dfrac{}{0\gamma, 1(A \otimes C) \vdash A \otimes C}\,\textsc{Var} \qquad \dfrac{\mathcal{Q}\gamma, 1A, 1C \vdash C}{\mathcal{Q}\gamma, 0(A \otimes C), 1A, 1C \vdash C}\,\textsc{Wk}}{\mathcal{Q}\gamma, 1(A \otimes C) \vdash C}\, \otimes\text{-E}}{\mathcal{Q}\gamma, 0(I \oplus (A \otimes C)), 1(A \otimes C) \vdash C}\,\textsc{Wk}$$

Following section 4.3, I want to turn the ad hoc constraints on $\mathcal{P}$, $\mathcal{Q}$, and $\mathcal{R}$ into the result of some premise connectives. To do this, I introduce a new connective $\square^{0+}$ defined below, along with the resulting implicit-context typing rules.

```
record □⁰⁺ (T : A → Set) (x : A) : Set where
  constructor □⟨_,_,_⟩_
  field
    {y} : A
    strengthen : x ≤ y
    split-0 : y ≤0
    split-+ : y ≤⌈ y + y ⌉
    T-prf : T y
```

$$\frac{\vdash F[\mu X.\ F[X]]}{\vdash \mu X.\ F[X]}\ \mu\text{-I} \qquad \frac{\vdash \mu X.\ F[X] \quad * \quad \square^{0+}(1F[C] \vdash C)}{\vdash C}\ \mu\text{-E}$$

Example 4.4.3. We can state the rules for lists derived in example 4.4.2 as follows.

$$\frac{I^*}{\vdash \mathrm{List}_A} \qquad \frac{\vdash A \quad * \quad \vdash \mathrm{List}_A}{\vdash \mathrm{List}_A} \qquad \frac{\vdash \mathrm{List}_A \quad * \quad \square^{0+}\big(\vdash C \quad \dot{\times} \quad 1A, 1C \vdash C\big)}{\vdash C}$$

Among the inductive types given by this μ operator are also &-*lists* (*with-lists*).

Example 4.4.4. Let us define &-*lists* as follows.

$$\mathrm{WithList}_A \coloneqq \mu X.\ \top \oplus (A \,\&\, X)$$

Unlike ordinary lists (which we may call $\otimes$-*lists*), &-lists support a *lookup* operation with the type below, where $\mathbb{N} \coloneqq \mu X.\ I \oplus X$.

$$\mathit{lookup} : \mathrm{WithList}_A \multimap \mathbb{N} \multimap \top \oplus A$$

The $\lambda\mathcal{R}$ term implementing *lookup* in terms of iterators (μ-E) is complex and tedious, so I do not show it here. Instead, I provide the following clausal specification, where nil$(-)$ and cons$(-)$ are the constructors of &-lists, zero() and succ() are the constructors of $\mathbb{N}$, inl$(-)$ and inr$(-)$ are the constructors of $\oplus$, $\langle\rangle$ constructs the inhabitant of $\top$, and postfix $.\pi_l$ and $.\pi_r$ are the projections out of a &-pair.

$$\begin{array}{llll} \mathit{lookup} & \mathrm{nil}(t) & i & \coloneqq \ \mathrm{inl}(\langle\rangle) \\ \mathit{lookup} & \mathrm{cons}(xxs) & \mathrm{zero}() & \coloneqq \ \mathrm{inr}(xxs.\pi_l) \\ \mathit{lookup} & \mathrm{cons}(xxs) & \mathrm{succ}(i) & \coloneqq \ \mathit{lookup}\,(xxs.\pi_r)\,i \end{array}$$

As suggested by the notation, $\square^{0+}$ may not be particularly canonical in terms of what operations the usage context is stable under. For example, we could also ask for $\forall r.\ \mathcal{Q} \leq r\mathcal{Q}$, i.e. that the usage context is stable under any scaling. This would allow us to include $!r(-)$ in our grammar of strictly positive functors. In certain semirings, this

could give us strange types, like, taking the monotonicity semiring, the type $\mu X.\ I \oplus (\iota \otimes !{\downarrow\downarrow}X)$, which is the type of lists over the base type ι where alternating elements are treated contravariantly and covariantly. This may motivate us to restrict strictly positive functors to allow only a subset of usage annotations. Separately, in some contexts we may only need to know that $\mathcal{Q} \leq 0$. For example, in the linearity semiring, being less than or equal to 0 happens to imply closure under addition, because the only annotations less than or equal to 0 are 0 and ω. Also, in semirings for relevant systems, all annotations are stable under addition (up to $\leq$ in the appropriate direction), so we need only check for being less than or equal to 0.

Similar modalities named $\Box$ appear in the work of Bizjak and Birkedal [2018] when they deal with *persistent predicates* in bunched logic, and in the work of Choudhury and Krishnaswami [2020] in a capability-aware calculus. Both of these papers, in common with this thesis, aim to pick out the *pure* or *safe* objects, i.e. those not depending on any external resources.

4.5 Representing existing linear and modal logics

A motivating reason to consider the system $\lambda\mathcal{R}$ is that instances of it correspond to previously studied systems. In this section, I present translations from $\lambda\mathcal{R}$ to Dual Intuitionistic Linear Logic [Barber, 1996] and the modal system of Pfenning and Davies [1999], and vice versa. These translations are not mechanised, as part of the reason for developing $\lambda\mathcal{R}$ was to avoid mechanising these systems directly. We cannot prove that the translations form an equivalence, because I have not written down an equational theory for $\lambda\mathcal{R}$, but I expect this to be easy enough to do.

4.5.1 Dual Intuitionistic Linear Logic

Dual Intuitionistic Linear Logic is a particular formulation of intuitionistic linear logic introduced by Barber [1996]. Its key feature, which simplifies the metatheory of linear logic, is the use of separate contexts for linear and intuitionistic free variables. Here I show that DILL is a fragment of the instantiation of $\lambda\mathcal{R}$ at the linearity semiring

$$\frac{}{\gamma, A; \cdot \vdash A}\ \textsc{Int-Ax} \qquad \frac{}{\gamma; A \vdash A}\ \textsc{Lin-Ax} \qquad \frac{}{\gamma; \cdot \vdash I}\ I\text{-I} \qquad \frac{\gamma; \delta_1 \vdash I \quad \gamma; \delta_2 \vdash A}{\gamma; \delta_1, \delta_2 \vdash A}\ I\text{-E}$$

$$\frac{\gamma; \delta_1 \vdash A \quad \gamma, \delta_2 \vdash B}{\gamma; \delta_1, \delta_2 \vdash A \otimes B}\ \otimes\text{-I} \qquad \frac{\gamma; \delta_1 \vdash A \otimes B \quad \gamma; \delta_2, A, B \vdash C}{\gamma; \delta_1, \delta_2 \vdash C}\ \otimes\text{-E}$$

$$\frac{\gamma; \delta, A \vdash B}{\gamma; \delta \vdash A \multimap B}\ \multimap\text{-I} \qquad \frac{\gamma; \delta_1 \vdash A \multimap B \quad \gamma; \delta_2 \vdash A}{\gamma; \delta_1, \delta_2 \vdash B}\ \multimap\text{-E} \qquad \frac{\gamma; \cdot \vdash A}{\gamma; \cdot \vdash\ !A}\ !\text{-I}$$

$$\frac{\gamma; \delta_1 \vdash\ !A \quad \gamma, A; \delta_2 \vdash B}{\gamma; \delta_1, \delta_2 \vdash B}\ !\text{-E} \qquad \frac{}{\gamma; \delta \vdash \top}\ \top\text{-I} \qquad \frac{\gamma; \delta \vdash A \quad \gamma, \delta \vdash B}{\gamma; \delta \vdash A \,\&\, B}\ \&\text{-I}$$

$$\frac{\gamma; \delta \vdash A_0 \,\&\, A_1}{\gamma; \delta \vdash A_i}\ \&\text{-E}_i \qquad \frac{\gamma; \delta_1 \vdash 0}{\gamma; \delta_1, \delta_2 \vdash A}\ 0\text{-E} \qquad \frac{\gamma; \delta \vdash A_i}{\gamma; \delta \vdash A_0 \oplus A_1}\ \oplus\text{-I}_i$$

$$\frac{\gamma; \delta_1 \vdash A \oplus B \quad \gamma; \delta_2, A \vdash C \quad \gamma; \delta_2, B \vdash C}{\gamma; \delta_1, \delta_2 \vdash C}\ \oplus\text{-E}$$

Figure 4.6: The rules of DILL, extended with additive connectives

$\{0, 1, \omega\}$.

The types of DILL are the same as the types of $\lambda\mathcal{R}$, except for the restriction of $!r$ to just $!\omega$. I will write the latter simply as ! when it appears in DILL. I add sums and with-products to the calculus of Barber [1996], with the obvious rules (stated fully in section 4.5.1). These additive type formers present no additional difficulty to the translation.

Proposition 4.5.1 (DILL $\to \lambda\mathcal{R}$)**.** Given a DILL derivation of $\gamma; \delta \vdash A$, we can produce a $\lambda\mathcal{R}_{01\omega}$ derivation of $\omega\gamma, 1\delta \vdash A$.

Proof. By induction on the derivation. We have $\omega \leq 0$, which allows us to discard intuitionistic variables at the var rules, and both $1 \leq 1$ and $\omega \leq 1$, which allow us to use both linear and intuitionistic variables.

Weakening is used when splitting linear variables between two premises. For example, $\otimes$-I in DILL is as follows.

$$\frac{\gamma; \delta_t \vdash t : A \quad \gamma; \delta_u \vdash u : B}{\gamma; \delta_t, \delta_u \vdash t \otimes u : A \otimes B}\ \otimes\text{-I}$$

$$
\begin{array}{rcl}
\text{DILL} \hookrightarrow \lambda\mathcal{R}_{01\omega} & & \\
Y & \mapsto & \iota_Y \\
I & \mapsto & I \\
A \otimes B & \mapsto & A \otimes B \\
A \multimap B & \mapsto & A \multimap B \\
!A & \mapsto & !\omega A \\
0 & \mapsto & 0 \\
A \oplus B & \mapsto & A \oplus B \\
\top & \mapsto & \top \\
A \,\&\, B & \mapsto & A \,\&\, B
\end{array}
\qquad\qquad
\begin{array}{rcl}
\text{PD} \hookrightarrow \lambda\mathcal{R}_{01\square} & & \\
Y & \mapsto & \iota_Y \\
\top & \mapsto & I \\
A \wedge B & \mapsto & A \,\&\, B \\
A \supset B & \mapsto & A \multimap B \\
\square A & \mapsto & !\square A \\
\bot & \mapsto & 0 \\
A \vee B & \mapsto & A \oplus B
\end{array}
$$

Figure 4.7: Embedding of DILL and PD types into $\lambda\mathcal{R}$

From this, our new derivation is as follows.

$$
\dfrac{\dfrac{\overset{ih_t}{\omega\gamma, 1\delta_t \vdash M_t : A}}{\omega\gamma, 1\delta_t, 0\delta_u \vdash M_t : A}\ \textsc{Weak} \qquad \dfrac{\dfrac{\overset{ih_u}{\omega\gamma, 1\delta_u \vdash M_u : A}}{\omega\gamma, 0\delta_t, 1\delta_u \vdash M_u : A}\ \textsc{Weak}}{\omega\gamma, 1\delta_t, 1\delta_u \vdash (M_t, M_u) : A \otimes B}\ \otimes\text{-I}
$$

□

When translating from $\lambda\mathcal{R}$ to DILL, we first coerce the $\lambda\mathcal{R}$ derivation to be in a form easily amenable to translation into DILL. An example of a $\lambda\mathcal{R}$ derivation with no direct translation into DILL is the following. In DILL terms, the intuitionistic variable of the conclusion becomes a linear variable in the premises. Such a move is admissible in DILL, but does not come naturally.

$$
\dfrac{\dfrac{}{1A : x \vdash A}\ \textsc{Var} \qquad \dfrac{}{1A : x \vdash A}\ \textsc{Var} \qquad \omega \leq 1 + 1}{\omega A : x \vdash A \otimes A}\ \otimes\text{-I}
$$

To avoid such situations, and therefore manipulations on DILL derivations, I show that all $\lambda\mathcal{R}_{01\omega}$ derivations can be made in *bottom-up* style. In bottom-up style, the algebraic facts we make use of are dictated by making most general choices based on the conclusions of rules. Bottom-up style corresponds to a (non-deterministic) form of *usage checking*, and the following lemma can be understood as saying that that form of usage checking is sufficiently general.

Definition 4.5.2. A derivation is said to be 01ω-*bottom-up* if only the following facts about addition and multiplication are used, and all proofs of inequalities not at leaves are by reflexivity (i.e, not using the facts that $\omega \leq 0$ and $\omega \leq 1$).

$+$	0	1	ω
0	0	1	-
1	1	-	-
ω	-	-	ω

$*$	0	1	ω
0	-	-	0
1	0	1	ω
ω	0	-	ω

Bottom-up style enforces that whenever we split a context into two (for example, in the rule $\otimes$-I) all unused variables in the conclusion stay unused in the premises, intuitionistic variables stay intuitionistic, and linear variables go either left or right. Multiplication is only used in the rule $!r$-I, at which point both the result and left argument are available. Here, the bottom-up style enforces that linear variables never appear in the premise of $!\omega$-I.

Lemma 4.5.3. Every $\lambda\mathcal{R}_{01\omega}$ derivation can be translated into a bottom-up $\lambda\mathcal{R}_{01\omega}$ derivation.

Proof. By induction on the shape of the derivation. When we come across a non-bottom-up use of addition, it must be that the corresponding variable in the conclusion has annotation ω. By subusaging, we can give this variable annotation ω in the premises too, before translating the subderivations to bottom-up style. A similar argument applies to uses of multiplication, remembering that both the left argument and result are fixed. □

Proposition 4.5.4 ($\lambda\mathcal{R} \to$ DILL)**.** Given a $\lambda\mathcal{R}_{01\omega}$ derivation of $\omega\gamma, 1\delta, 0\theta \vdash A$ which contains only types expressible in DILL, we can produce a DILL derivation of $\gamma; \delta \vdash A$.

Proof. By induction on the derivation having been translated to bottom-up form.

In the case of VAR, all of the unused variables have annotation greater than 0, i.e., 0 or ω. Those annotated 0 are absent from the DILL derivation, and those annotated ω are in the intuitionistic context. The used variable is annotated either 1 or ω. In the first case, we use LIN-AX, and in the second case, INT-AX.

All binding of variables in $\lambda\mathcal{R}$ maps directly onto DILL.

$$\frac{}{\gamma; \delta, A\ true \vdash A\ true}\ \text{HYP} \qquad \frac{}{\gamma, A\ valid; \delta \vdash A\ true}\ \text{HYP*} \qquad \frac{\gamma; \delta, A\ true \vdash B\ true}{\gamma; \delta \vdash A \supset B\ true}\ \supset\text{I}$$

$$\frac{\gamma; \delta \vdash A \supset B\ true \qquad \gamma; \delta \vdash A\ true}{\gamma; \delta \vdash B\ true}\ \supset\text{E} \qquad \frac{\gamma; \cdot \vdash A\ true}{\gamma; \delta \vdash \Box A\ true}\ \Box\text{I}$$

$$\frac{\gamma; \delta \vdash \Box A\ true \qquad \gamma, A\ valid; \delta \vdash B\ true}{\gamma; \delta \vdash B\ true}\ \Box\text{-E} \qquad \frac{}{\gamma; \delta \vdash \top\ true}\ \top\text{-I}$$

$$\frac{\gamma; \delta \vdash A\ true \qquad \gamma, \delta \vdash B\ true}{\gamma; \delta \vdash A \wedge B\ true}\ \wedge\text{-I} \qquad \frac{\gamma; \delta \vdash A_0 \wedge A_1\ true}{\gamma; \delta \vdash A_i\ true}\ \wedge\text{-E}_i \qquad \frac{\gamma; \delta \vdash \bot\ true}{\gamma; \delta \vdash A\ true}\ \bot\text{-E}$$

$$\frac{\gamma; \delta \vdash A_i\ true}{\gamma; \delta \vdash A_0 \vee A_1\ true}\ \vee\text{-I}_i$$

$$\frac{\gamma; \delta \vdash A \vee B\ true \qquad \gamma; \delta, A \vdash C\ true \qquad \gamma; \delta, B \vdash C\ true}{\gamma; \delta \vdash C\ true}\ \vee\text{-E}$$

Figure 4.8: The rules of PD, extended with several standard connectives

Because we translated to bottom-up form, additions, as seen in, for example, the $\otimes$-I rule, can be handled straightforwardly. Any intuitionistic variables in the conclusion correspond to intuitionistic variables in both premises. Any linear variables in the conclusion correspond to a linear variable in exactly one of the premises, and is absent in the other premise.

The only remaining rule is $!_r$-I, of which we only cover $!_\omega$-I (the other two targeting types not found in DILL). In this case, we know that every variable in the conclusion is annotated either 0 or ω, and every variable in the premise is annotated the same way. This corresponds exactly to the restrictions of DILL's !-I. □

4.5.2 Pfenning-Davies

The translation to and from the modal system of Pfenning and Davies [1999] (henceforth *PD*) is similar to the translation to and from DILL. I present my variant of PD, again adding some common connectives, in section 4.5.2 The main difference is the algebra at which $\lambda\mathcal{R}$ is instantiated.

Definition 4.5.5. Let $01\Box$ denote the following semiring on the partially ordered set $\{\Box \lhd 1 \lhd 0\}$.

- $0 := 0$.
- $+$ is the meet ($\wedge$) according to the subusaging order.
- $1 := 1$.
-

$*$	0	1	$\Box$
0	0	0	0
1	0	1	$\Box$
$\Box$	0	$\Box$	$\Box$

The 0 annotation plays only a formal role in this example. Meanwhile, 1 and $\Box$ correspond to the judgement forms *true* and *valid* from PD. Addition being the meet makes it idempotent. Furthermore, it gives us that $1 + \Box = \Box$ — if somewhere we require an assumption to be true, and elsewhere require it to be valid, then ultimately it must be valid (from which we can deduce that it is true). Multiplication is designed to make $!\Box$ act like PD's $\Box$. In particular, $\Box * \Box = \Box$ says that the valid assumptions are available before and after $!\Box$-I, whereas $\Box * 1 = \Box$ says that valid assumptions in the conclusion can be weakened to true assumptions in the premise. The latter fact does not appear in PD, and will be excluded from *bottom-up* derivations.

To keep my notation consistent with that of DILL, I swap the roles of γ and δ in PD compared to what they were in the original paper. Thus, my PD judgements are of the form $\gamma; \delta \vdash A\ \mathit{true}$, where γ contains valid assumptions and δ contains true assumptions.

Proposition 4.5.6 (PD $\to \lambda\mathcal{R}$)**.** Given a PD derivation of $\gamma; \delta \vdash t : A\ \mathit{true}$, we can produce a $\lambda\mathcal{R}_{01\Box}$ derivation of $\Box\gamma, 1\delta \vdash A$.

Proof. By induction on the PD derivation. Most PD rules have direct $\lambda\mathcal{R}$ counterparts, noting that variables of any annotation can be discarded and duplicated because we have both $r \leq 0$ and $r \leq r + r$ for all r.

Care must be taken with the $\Box$I rule. We have, from the induction hypothesis, a $\lambda\mathcal{R}$ derivation of $\Box\gamma \vdash A$. By $!\Box$-I, we have $\Box\gamma \vdash !\Box A$. To get the desired conclusion,

we must use Weak to get $\Box\gamma, 0\delta \vdash !\Box A$, and then Subuse on the variables we just introduced (noting that $1 \leq 0$) to get $\Box\gamma, 1\delta \vdash !\Box A$. □

For translating from $\lambda\mathcal{R}_{01\Box}$ to PD, I introduce a similar notion of *bottom-up* derivations as I did for DILL. Every $\lambda\mathcal{R}_{01\Box}$ derivation can be translated into bottom-up style, and then be directly translated into PD.

Definition 4.5.7. A derivation is said to be $01\Box$*-bottom-up* if only the following facts about addition and multiplication are used, and all proofs of inequalities not at leaves are by reflexivity.

$+$	0	1	$\Box$
0	0	-	-
1	-	1	-
$\Box$	-	-	$\Box$

$*$	0	1	$\Box$
0	-	-	0
1	0	1	$\Box$
$\Box$	0	-	$\Box$

Lemma 4.5.8. Every $\lambda\mathcal{R}_{01\Box}$ derivation can be translated into a bottom-up $\lambda\mathcal{R}_{01\Box}$ derivation.

Proof. By induction on the shape of the derivation. Given that addition is a meet, it is clear that any non-bottom-up uses of addition come from one of the arguments being greater than the result. Therefore, it is safe to make this argument smaller in the corresponding premise (via subusaging), before translating that subderivation. For multiplication, again, there is always a lesser value of the right argument that will take us from a non-bottom-up fact to a bottom-up fact with the same left argument and result. □

Additionally, PD has no direct equivalents to the tensor unit and tensor products. Therefore, I note that if the semiring of usage annotations satisfies some simple but strong criteria (as $01\Box$ does), then we can use the Cartesian unit and products in their stead.

Lemma 4.5.9. If $1 \leq 0$ and $1 \leq 1 + 1$, then I is interderivable with $\top$, and for all A and B, $A \otimes B$ is interderivable with $A \,\&\, B$.

Proof. The following derivations suffice. I omit standard inequalities to emphasise those which use this lemma's assumptions.

$$\frac{}{1I \vdash \top}\ \top\text{-I} \qquad \frac{\overline{(1) \leq (0)}}{1\top \vdash I}\ I\text{-I}$$

$$\frac{\dfrac{}{1(A \otimes B) \vdash A \otimes B}\ \textsc{Var} \qquad \dfrac{\dfrac{\overline{(0,1,1) \leq (0,1,0)}}{0(A \otimes B), 1A, 1B \vdash A}\ \textsc{Var} \qquad \dfrac{\overline{(0,1,1) \leq (0,0,1)}}{0(A \otimes B), 1A, 1B \vdash B}\ \textsc{Var}}{0(A \otimes B), 1A, 1B \vdash A \,\&\, B}\ \&\text{-I}}{1(A \otimes B) \vdash A \,\&\, B}\ \otimes\text{-E}$$

$$\frac{\dfrac{}{(1) \leq (1) + (1)} \qquad \dfrac{\dfrac{}{1(A \,\&\, B) \vdash A \,\&\, B}\ \textsc{Var}}{1(A \,\&\, B) \vdash A}\ \&\text{-E}_0 \qquad \dfrac{\dfrac{}{1(A \,\&\, B) \vdash A \,\&\, B}\ \textsc{Var}}{1(A \,\&\, B) \vdash B}\ \&\text{-E}_1}{1(A \,\&\, B) \vdash A \otimes B}\ \otimes\text{-I}$$

□

Remark 4.5.10. By multiplying through on both sides, $1 \leq 0$ is equivalent to $\forall x.\ x \leq 0$, and $1 \leq 1 + 1$ is equivalent to $\forall x.\ x \leq x + x$.

Finally, I give the translation itself.

Proposition 4.5.11 ($\lambda\mathcal{R} \to$ PD)**.** Given a $\lambda\mathcal{R}_{01\Box}$ derivation of $\Box\gamma, 1\delta, 0\theta \vdash M : A$ which does not contain types using $!0$ or $!1$, we can produce a PD derivation of $\gamma; \delta \vdash A$ *true*.

Proof. We translate away tensor products and tensor units using lemma 4.5.9, and translate the resulting derivation to bottom-up form. The proof proceeds by induction on the resulting derivation in the obvious way. □

As mentioned in section 4.4.1, the system of Abel and Bernardy [2020] is unable to embed PD in this way, as it would prove $\Box(A \wedge B) \to \Box A \wedge \Box B$, where PD and $\lambda\mathcal{R}$ do not. In fact, this example shows that, even when weakening and contraction are admissible, with- and tensor-products are distinct in their system in the presence of modalities.

4.6 Conclusion

In this chapter, I have presented the calculus $\lambda\mathcal{R}$, with particular focus on the posemiring usage annotations. Posemirings are sufficient for many use cases, as shown by the examples in this chapter. However, there are many more substructural disciplines in the literature which cannot be expressed using posemirings.

An important point to note is that $\lambda\mathcal{R}$ cannot be instantiated to become a bunched logic, in the sense of O'Hearn and Pym [1999]. This is despite the use of the bunched connectives in the *definition* of $\lambda\mathcal{R}$, and despite the formally tree-shaped contexts. However, the variable rule of $\lambda\mathcal{R}$ essentially treats the variables in the context independently, except for the individual checking of usage annotations on each variable. Therefore, we cannot talk about more interesting spatial relationships between variables, as required by bunched logic. This property also precludes us from capturing calculi without exchange (non-commutative logics), such as Lambek calculus [Lambek, 1958]. Fitch-style systems [Borghuis, 1994] probably also are precluded similarly.

The work I present here has some methodological similarities with the earlier work of Licata et al. [2017]. In that framework, one can provide a very precise *mode theory*, expressing which structural rules are available, and from it get a sequent calculus with cut-elimination. They encode many systems, including a non-associative logic (where context are trees obeying no structural rules) and a bunched logic, and the framework is probably expressive enough to encode all posemiring-based usage disciplines. However, aside from the structural rules, the sequent calculus is fixed — there are two connectives: F (internalising the left of the sequent, comparable to my $!_{r_1}(-) \otimes \cdots \otimes !_{r_n}(-)$) and U (internalising the whole sequent, comparable to my $!_{r_1}(-) \multimap \cdots \multimap !_{r_n}(-) \multimap (-)$), which have left and right rules, and then the only other rule is the variable rule. This restriction is what allows the cut elimination theorem to be even plausible, but means that other connectives, like additives and inductive types, and more exotic syntaxes, like those I will present in section 6.3, would have to be developed from scratch.

The distinction between sharing and separating conjunction of premises naturally falls out of the posemiring approach to usage restrictions. However, a similar distinction

also appears in other approaches to linearity, and perhaps in other substructural systems. Indeed, in the paper that inspired the bunched connectives [Rouvoet et al., 2020], linearity is enforced in a similar style as in Yalla [Laurent, 2018], using lists which can be split. It would be interesting to see future work on substructural systems work out the appropriate premise connectives and define their systems from there, rather than directly manipulating contexts. It may also be possible to see future work profitably abstracting over the posemiring annotations, requiring only that contexts form something like a commutative Rel-monoid supporting variable-binding and variable access.

Chapter 5

Renaming and substitution for $\lambda\mathcal{R}$

In chapter 4, I defined my calculus of interest $\lambda\mathcal{R}$. In this chapter, I develop the necessary syntactic metatheory for specifying and implementing the substitution operation. I follow the approach of section 2.3 using syntactic kits, but have to make significant changes to the underlying notion of *environment* before doing so. I give and informally motivate these changes to environments in section 5.1, and prove some properties of the new definition in section 5.2. Finally, I apply these new environments to the syntax of $\lambda\mathcal{R}$ in section 5.3 to derive renaming and substitution operators.

5.1 What are linear renaming and substitution?

In an effort to reuse the syntactic kits and traversals approach of section 2.3.3, I will derive the types of simultaneous renaming and simultaneous substitution from a generic type of *environments*. To get a type of environments suitable for the usage-aware setting, I first analyse intuitionistic environments (as introduced in section 2.3.3 definition Env), distilling the easy-to-use functional definition (definition 5.1.1) into a more basic recursive definition (definition 5.1.2). This recursive definition is easy to make usage-aware (definition 5.1.3), which gives a basis from which to derive the function-based definition I will take as primary (definition 5.1.7). The resulting definition makes explicit the role of algebraic linearity in the metatheory of semiring-annotated calculi.

Recalling from section 2.3, we have the following definition of environments for

simple types.

Definition 5.1.1 (Simple environment)**.** For $\mathcal{V} : \mathrm{Ctx} \to \mathrm{Ty} \to \mathrm{Set}$, a $\mathcal{V}$-*environment* between simply typed contexts Γ and Δ is a function, polymorphic in type A, from variables of type A in Δ to inhabitants of $\mathcal{V}\,\Gamma\,A$. We write the type of such environments as $\Gamma \overset{\mathcal{V}}{\Longrightarrow} \Delta$.

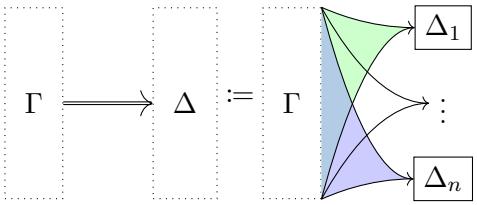
This definition is inadequate for $\lambda\mathcal{R}$. For example, suppose we have a term $(M \otimes N) : \mathcal{R}\gamma \vdash A \otimes B$ and a substitution $\sigma : \mathcal{R}\gamma \overset{\vdash}{\Longrightarrow} \mathcal{R}'\delta$. From the $\otimes$-I rule, we have $M : \mathcal{P}\gamma \vdash A$ and $N : \mathcal{Q}\gamma \vdash B$ for some $\mathcal{P}$ and $\mathcal{Q}$ such that $\mathcal{R} \leq \mathcal{P} + \mathcal{Q}$. We want to apply σ to the subterms M and N, but this is impossible because their contexts are not $\mathcal{R}\gamma$, and we have no way to adapt σ to these new contexts. Another instructive failure is the general non-existence of identity environments, like a renaming of type $1A, 1B \overset{\exists}{\Longrightarrow} 1A, 1B$. We do not have a variable of type $1A, 1B \ni A$ or, symmetrically, $1A, 1B \ni B$, because, in each case, there is one variable with annotation 1 which we have not actually used. This example suggests that the values of a usage-aware environment should be derived in *different* usage contexts, such as in $1A, 0B \ni A$.

To see why this definition of environment works for simply typed λ-calculus but not $\lambda\mathcal{R}$, let us look at an equivalent definition by recursion on the target context. This recursive definition (definition 5.1.2), and particularly the case where Δ is a concatenation, makes it clear how Γ is being copied for use in each $\mathcal{V}$-value. I take the equivalence of definition 5.1.1 and definition 5.1.2 as obvious, because any function from variables in Δ can be defunctionalised as a data structure with the same shape as Δ.

Definition 5.1.2 (Simple recursive environment)**.** A *recursive* $\mathcal{V}$-*environment* between simply typed contexts Γ and Δ is defined by cases on the shape of Δ (where $\Gamma \overset{\mathcal{V}}{\Longrightarrow}_R \Delta$ is the notation for the type of recursive environments for given $\mathcal{V}$, Γ, and Δ):

- There is an environment $\langle\rangle : \Gamma \overset{\mathcal{V}}{\Longrightarrow}_R \cdot$.
- For $\rho_l : \Gamma \overset{\mathcal{V}}{\Longrightarrow}_R \Delta_l$ and $\rho_r : \Gamma \overset{\mathcal{V}}{\Longrightarrow}_R \Delta_r$, we have an environment $\langle \rho_l, \rho_r \rangle : \Gamma \overset{\mathcal{V}}{\Longrightarrow}_R \Delta_l, \Delta_r$.
- For any value $v : \mathcal{V}\,\Gamma\,A$, we have an environment $\langle v \rangle : \Gamma \overset{\mathcal{V}}{\Longrightarrow}_R A$.

I picture the sharing of Γ in definition 5.1.2 in the diagram below. The converging arrows from Γ to each Δ_i represent the indices of values appearing in a simple environment.



To account for usage, we must replace the simple repetition of Γ by repetition of just the types γ and *redistribution* of the usage annotations $\mathcal{P}$. Fortunately, our three basic ways of sharing up usage vectors — zero, addition, and scaling — apply directly to the three possible shapes of the target context — empty, concatenation, and a usage-annotated singleton.

Definition 5.1.3 (Usage-annotated recursive environment)**.** A *recursive $\mathcal{V}$-environment* between annotated contexts Γ and Δ is defined by cases on the shape of Δ (where $\Gamma \overset{\mathcal{V}}{\Longrightarrow}_R \Delta$ is the notation for the type of recursive environments for given $\mathcal{V}$, Γ, and Δ):

- There is one environment $\langle\rangle : \mathcal{P}\gamma \overset{\mathcal{V}}{\Longrightarrow}_R \cdot$ whenever $\mathcal{P} \leq 0$.
- For $\rho_l : \mathcal{P}_l\gamma \overset{\mathcal{V}}{\Longrightarrow}_R \Delta_l$ and $\rho_r : \mathcal{P}_r\gamma \overset{\mathcal{V}}{\Longrightarrow}_R \Delta_r$, we have an environment $\langle\rho_l, \rho_r\rangle : \mathcal{P}\gamma \overset{\mathcal{V}}{\Longrightarrow}_R \Delta_l, \Delta_r$ whenever $\mathcal{P} \leq \mathcal{P}_l + \mathcal{P}_r$.
- For any value $v : \mathcal{V}\,\mathcal{P}'\gamma\,A$, we have an environment $\langle v\rangle : \mathcal{P}\gamma \overset{\mathcal{V}}{\Longrightarrow}_R rA$ whenever $\mathcal{P} \leq r\mathcal{P}'$.

Example 5.1.4. Take $\mathscr{R} = (\mathbb{N}, =, 0, +, 1, \times)$, with the equality order chosen to avoid any concerns around subsumption of annotations. Then, there is an intuitionistic recursive environment (substitution) as follows, where $y\,z$ is the application of y to z.

$$\langle\langle z\rangle, \langle y\,z\rangle\rangle : (x : A, y : B \to C, z : B) \overset{\vdash}{\Longrightarrow}_R (B, C)$$

There is also a usage-aware recursive environment

$$\langle\langle z\rangle, \langle y\,z\rangle\rangle : (0x : A, 2y : B \multimap C, 3z : B) \overset{\vdash}{\Longrightarrow}_R (1B, 2C).$$

The latter relies on the observations that $\begin{pmatrix} 0 & 2 & 3 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix} + \begin{pmatrix} 0 & 2 & 2 \end{pmatrix}$ and, on the right, that $\begin{pmatrix} 0 & 2 & 2 \end{pmatrix} = 2\begin{pmatrix} 0 & 1 & 1 \end{pmatrix}$. Then, we have $0x : A, 0y : B \multimap C, 1z : B \vdash z : B$ and $0x : A, 1y : B \multimap C, 1z : B \vdash y\, z : C$.

Example 5.1.5. Take $\mathscr{R} = (\mathbb{N}, =, 0, +, 1, \times)$. Then, there is an intuitionistic recursive environment (renaming) as follows, where $\langle -, -, - \rangle$ abbreviates $\langle -, \langle -, - \rangle \rangle$ (matching a similar abbreviation in the notation of contexts).

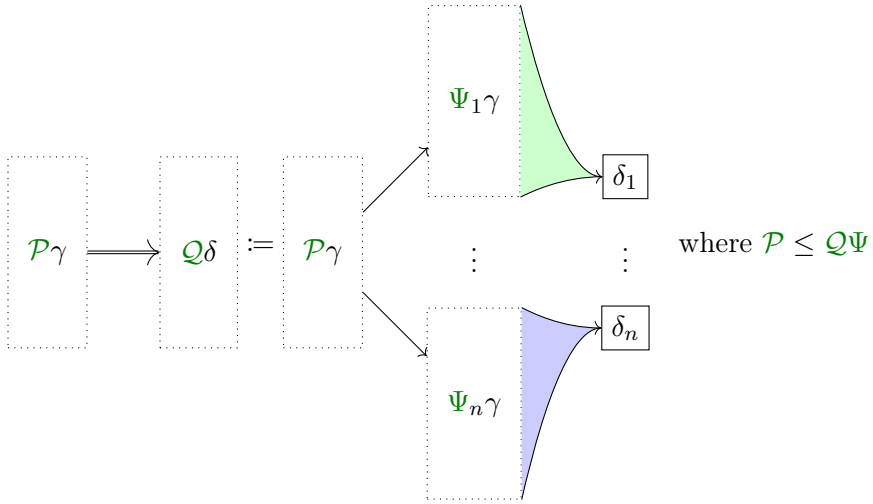
$$\langle \langle c \rangle, \langle a \rangle, \langle a \rangle \rangle : (a : A, b : B, c : C, d : D) \overset{\exists}{\Longrightarrow}_R (C, A, A).$$

There is also a usage-aware recursive environment

$$\langle \langle c \rangle, \langle a \rangle, \langle a \rangle \rangle : (6a : A, 0b : B, 1c : C, 0d : D) \overset{\exists}{\Longrightarrow}_R (1C, 2A, 4A).$$

Intuitively, this choice of usage annotations works because the 6 As on the left can be divided into the $2 + 4$ As on the right. Similarly, the 0 Bs and 0 Ds on the left can be discarded to yield none on the right. Note that these divisions are directed, so we cannot merge variables or introduce new variables on the right.

From example 5.1.4, we can see that the important usage vectors are the initial one $\begin{pmatrix} 0 & 2 & 3 \end{pmatrix}$ and the usage vectors at which terms are derived: $\begin{pmatrix} 0 & 0 & 1 \end{pmatrix}$ and $\begin{pmatrix} 0 & 1 & 1 \end{pmatrix}$. I will call the latter the *leaf vectors*. The intermediate vector $\begin{pmatrix} 0 & 2 & 2 \end{pmatrix}$ can be worked out from the leaf vector $\begin{pmatrix} 0 & 1 & 1 \end{pmatrix}$ and the scaling factor 2 found in the codomain context $1B, 2C$. Even when the ordering on annotations is given by a non-equivalence relation $\leq$, there is a canonical least choice for all of the intermediate vectors, together with a constraint that the entire linear combination of all the leaf vectors is less than or equal to the initial usage vector. In symbols, we may let Ψ be the collection of leaf vectors indexed by items in Δ, and state the constraint as $\mathcal{P} \leq \sum_{(x:rA)\in\Delta} r\Psi_x$. Seeing Ψ instead as a $|\Delta| \times |\Gamma|$ matrix, this constraint is $\mathcal{P} \leq \mathcal{Q}\Psi$, using vector-matrix multiplication. The resulting picture is below, showing $\mathcal{P}$ being split up into Ψ, and then each $\mathcal{V}$-value being constructed in a separate $\Psi_i\gamma$.



From this point, we can recover a functional-style definition of usage-aware environments. We choose our leaf vectors Ψ up-front, check the inequality, and then produce a value at each leaf vector.

Definition 5.1.6 (Usage-annotated environment (tentative))**.** A $\mathcal{V}$-*environment* between annotated contexts Γ and Δ (written $\mathcal{P}\gamma$ and $\mathcal{Q}\delta$, respectively, when convenient) is a matrix $\Psi : \mathscr{R}^{|\Delta|\times|\Gamma|}$ such that $\mathcal{P} \leq \mathcal{Q}\Psi$ and for each $(x : A) \in \delta$ we have a value of type $\mathcal{V}\,\Psi_x\gamma\,A$.

I find this definition somewhat fiddly because of its reliance on low-level concepts like non-usage-checked variables and rows of a matrix. We note that $\Psi_x = \langle x|\Psi$, from which point, requiring not just $\mathcal{V}\,\Psi_x\gamma\,A$ but rather $\mathcal{V}\,(\mathcal{Q}'\Psi)\gamma\,A$ for any $\mathcal{Q}' \leq \langle x|$ is a minor change (and equivalent if $\mathcal{V}$ respects subusaging, which is practically always the case). "An x such that $(x : A) \in \delta$ and $\mathcal{Q}' \leq \langle x|\Psi$" is exactly the definition of $\mathcal{Q}'\delta \ni A$. I further regularise this clause by asking for a $\mathcal{P}' \leq \mathcal{Q}'\Psi$ rather than $\mathcal{Q}'\Psi$ exactly, leaving us needing, for each $\mathcal{P}'$ and $\mathcal{Q}'$ related in the same way (Ψ) as $\mathcal{P}$ and $\mathcal{Q}$, a function from $\mathcal{Q}'\delta \ni A$ to $\mathcal{V}\,\mathcal{P}'\gamma\,A$. Finally, I choose to switch from matrices and matrix multiplication to linear maps and their actions, which are easier to work with. All of these changes yield my primary definition of an environment for usage-annotated calculi, which will be used for the rest of this chapter and in chapter 6.

Definition 5.1.7 (Usage-annotated environment)**.** A $\mathcal{V}$-*environment* between annotated contexts Γ and Δ (written $\mathcal{P}\gamma$ and $\mathcal{Q}\delta$, respectively, when convenient) is a linear

map $\Psi : \mathscr{R}^{|\Delta|} \to \mathscr{R}^{|\Gamma|}$ (written postfix) such that $\mathcal{P} \leq \mathcal{Q}\Psi$ and for each A, $\mathcal{P}'$, and $\mathcal{Q}'$ such that $\mathcal{P}' \leq \mathcal{Q}'\Psi$, a function from $\mathcal{Q}'\delta \ni A$ to $\mathcal{V}\,\mathcal{P}'\gamma\,A$.

Notation 5.1.8. When there are multiple environments in question and ρ is such an environment, I use the notation $\rho.\Psi$ to refer to Ψ. For example, $\mathcal{P} \leq \mathcal{Q}(\rho.\Psi)$. For the action on variables, I write $\rho(x)$, where $x : \mathcal{Q}'\delta \ni A$. The expression "$\rho(x)$" alone is ambiguous because of the slack in the usage context $\mathcal{P}'$ of the resulting value. Therefore, I will always make sure $\mathcal{P}'$ and $\mathcal{Q}'$ clear when using this notation.

The following simple lemma shows that usage-annotated environments are, in a sense, as good as simple environments on usage-checked variables. What usage-annotated environments give us beyond simple environments is the ability to accommodate linear decompositions, in a way I will make precise in the next section.

Lemma 5.1.9. We can use an environment $\rho : \Gamma \overset{\mathcal{V}}{\Longrightarrow} \Delta$ to map a usage-checked variable $x : \Delta \ni A$ to a value of type $\mathcal{V}\,\Gamma\,A$.

Proof. Let $\Gamma = \mathcal{P}\gamma$ and $\Delta = \mathcal{Q}\delta$. Set $\mathcal{P}' \coloneqq \mathcal{P}$ and $\mathcal{Q}' \coloneqq \mathcal{Q}$, then $\mathcal{P} \leq \mathcal{Q}\Psi$ by the constraint in ρ, so we can take the $\mathcal{V}$-value $\rho(x)$. □

5.2 Properties of linear environments

I settle on definition 5.1.7, and prove various properties about it.

Lemma 5.2.1. Given an environment $\rho : \mathcal{P}\gamma \overset{\mathcal{V}}{\Longrightarrow} \mathcal{Q}\delta$ and a $\mathcal{P}'$ and a $\mathcal{Q}'$ such that $\mathcal{P}' \leq \mathcal{Q}'(\rho.\Psi)$, there is also an environment of type $\mathcal{P}'\gamma \overset{\mathcal{V}}{\Longrightarrow} \mathcal{Q}'\delta$ with the same linear map and action on variables.

Proof. The only part of the definition of an environment dependent on $\mathcal{P}$ or $\mathcal{Q}$ is the constraint $\mathcal{P} \leq \mathcal{Q}\Psi$, which we are able to replace for $\mathcal{P}'$ and $\mathcal{Q}'$. □

When constructing an environment, we can do so by cases on the shape of the target context. We can create an environment into the empty context when all usage annotations on the source context are 0. We can create an environment into a concatenated context when we can additively split up the annotations of the source context

and produce environments into both halves from the split sources. We can create an environment into a singleton context when there is a context r times smaller than the source context in which we can produce a value of the appropriate type.

Lemma 5.2.2. We can define all of the following equivalences for any values of the free variables, assuming that $\mathcal{V}$ respects subusaging (i.e., $\mathcal{P}' \leq \mathcal{P} \to \mathcal{V}\,\mathcal{P}\gamma \multimap \mathcal{V}\,\mathcal{P}'\gamma$).

- $I^* \multimap\!\!\circ \left(- \overset{\mathcal{V}}{\Longrightarrow} \cdot\right)$
- $\left(- \overset{\mathcal{V}}{\Longrightarrow} \Delta_l\right) * \left(- \overset{\mathcal{V}}{\Longrightarrow} \Delta_r\right) \multimap\!\!\circ \left(- \overset{\mathcal{V}}{\Longrightarrow} \Delta_l, \Delta_r\right)$
- $r \cdot (\mathcal{V}\,(-)\,A) \multimap\!\!\circ \left(- \overset{\mathcal{V}}{\Longrightarrow} rA\right)$

Proof. There are 6 cases to check. Throughout, we write Γ as $\mathcal{P}\gamma$ and Δ as $\mathcal{Q}\delta$ when convenient.

$I^*(\multimap)$ Let Ψ be the unique linear map out of the zero space. By assumption and definition, $\mathcal{P} \leq 0 = \mathcal{Q}\Psi$. There are no variables to act upon.

$I^*(\multimap\!\!\circ)$ $\mathcal{Q}\Psi$ is an empty sum, so if $\mathcal{P} \leq \mathcal{Q}\Psi$ then $\mathcal{P} \leq 0$.

$*(\multimap)$ Let the given environments be $\rho_l : \mathcal{P}_l\gamma \overset{\mathcal{V}}{\Longrightarrow} \mathcal{Q}_l\delta$ and $\rho_r : \mathcal{P}_r\gamma \overset{\mathcal{V}}{\Longrightarrow} \mathcal{Q}_r\delta$, with $\mathcal{P} \leq \mathcal{P}_l + \mathcal{P}_r$. Define $\Psi := [\rho_l.\Psi, \rho_r.\Psi]$, using the coproduct structure of the concatenated vector space. We have $\mathcal{P} \leq \mathcal{P}_l + \mathcal{P}_r \leq \mathcal{Q}_l(\rho_l.\Psi) + \mathcal{Q}_r(\rho_r.\Psi) = \begin{pmatrix}\mathcal{Q}_l & \mathcal{Q}_r\end{pmatrix}\Psi$. To act on variables, we are given $\mathcal{P}' \leq \begin{pmatrix}\mathcal{Q}'_l & \mathcal{Q}'_r\end{pmatrix}\Psi$ and $\mathcal{Q}'_l\delta_l, \mathcal{Q}'_r\delta_r \ni A$. Without loss of generality, let us have $\mathcal{Q}'_l\delta_l \ni A$ and $\mathcal{Q}'_r \leq 0$. Thus, $\mathcal{P}' \leq \mathcal{Q}'_l(\rho_l.\Psi) + \mathcal{Q}'_r(\rho_r.\Psi) \leq \mathcal{Q}'_l(\rho_l.\Psi)$, and we can act on the variable using ρ_l.

$*(\multimap\!\!\circ)$ Let the unnamed context be Γ, also written $\mathcal{P}\gamma$. The linear map $\Psi : \mathscr{R}^{|\Delta_l|+|\Delta_r|} \to \mathscr{R}^{|\Gamma|}$ splits into $\Psi_l : \mathscr{R}^{|\Delta_l|} \to \mathscr{R}^{|\Gamma|} := \langle \mathrm{id}, 0\rangle; \Psi$ and $\Psi_r : \mathscr{R}^{|\Delta_r|} \to \mathscr{R}^{|\Gamma|} := \langle 0, \mathrm{id}\rangle; \Psi$, using the product structure of the concatenated vector space. Let $\mathcal{P}_l := \mathcal{Q}_l\Psi_l$ and $\mathcal{P}_r := \mathcal{Q}_r\Psi_r$, by definition satisfying the required constraints. For the action on variables, let us consider the left environment (with the right environment following symmetrically). We are given $\mathcal{P}'_l \leq \mathcal{Q}'_l\Psi_l$ and $\mathcal{Q}'_l\delta_l \ni A$. From these, we get $\mathcal{P}'_l \leq \mathcal{Q}'_l\Psi_l = \begin{pmatrix}\mathcal{Q}'_l & 0\end{pmatrix}\Psi$ and $\mathcal{Q}'_l\delta_l, 0\delta_r \ni A$. We can therefore act using the original environment.

$\cdot(\twoheadrightarrow)$ Let $\mathcal{P}$ and $\mathcal{P}'$ be such that $\mathcal{P} \leq r\mathcal{P}'$ and let $v : \mathcal{V}\,\mathcal{P}'\gamma\,A$. Let $\Psi : \mathscr{R} \to \mathscr{R}^{|\gamma|} \coloneqq r' \mapsto r'\mathcal{P}'$. By definition and the previous assumption, we have $\mathcal{P} \leq r\Psi$. When acting on a variable, we have $\mathcal{P}'' \leq r'\Psi$ and $r'A \ni A'$. The latter tells us that $A = A'$ and $r' \leq 1$. Thus, $\mathcal{P}'' \leq \mathcal{P}'$. Therefore, by subusaging, we may produce a value of type $\mathcal{V}\,\mathcal{P}'\gamma\,A$, which we can take to be v.

$\cdot(\twoheadleftarrow)$ Let us have an environment of type $\mathcal{P}\gamma \overset{\mathcal{V}}{\Longrightarrow} rA$. We want to use its action on variables to yield a value. To do this, we let $\mathcal{P}' \coloneqq 1\Psi$, and use this equation, together with the fact that we have a variable of type $1A \ni A$, to get a value of type $\mathcal{V}\,\mathcal{P}'\gamma\,A$. Furthermore, we derive $\mathcal{P} \leq r\Psi = r\mathcal{P}'$, as required.

□

We could, as in definition 5.1.3, use these three clauses to define what an environment is. However, such a definition appears to require creative induction hypotheses in the proving of simple lemmas, in contrast to the more direct proofs I achieve below using definition 5.1.7. To take a concrete example, consider how we may construct an "identity" environment of type $\Gamma \overset{\mathcal{V}}{\Longrightarrow} \Gamma$, as in lemma 5.2.7 below. If we try to directly proceed by induction on Γ, we get to the case where we are aiming to construct an environment of type $\mathcal{P}\gamma, \mathcal{Q}\delta \overset{\mathcal{V}}{\Longrightarrow} \mathcal{P}\gamma, \mathcal{Q}\delta$ by constructing environments of types $\mathcal{P}\gamma, 0\delta \overset{\mathcal{V}}{\Longrightarrow} \mathcal{P}\gamma$ and $0\gamma, \mathcal{Q}\delta \overset{\mathcal{V}}{\Longrightarrow} \mathcal{Q}\delta$. These are not identity environments, and thus do not come from the hypotheses of a simple induction. In contrast, using definition 5.1.7, in lemma 5.2.7 we are able to use the standard fact that there are identity linear maps, and on top of such a map worry only about the value assigned to each variable.

One of the primary test cases for environments is simultaneous substitution, which will look like the SUB rule below. Note that we have taken $\mathcal{V} \coloneqq \vdash$ — i.e. that the values yielded by the environment are terms, namely the terms to be substituted in for the free variables of the derivation of $\Delta \vdash A$.

$$\frac{\Gamma \overset{\vdash}{\Longrightarrow} \Delta \qquad \Delta \vdash A}{\Gamma \vdash A}\ \textsc{sub}$$

The admissibility of substitution will be by induction on the derivation of $\Delta \vdash A$, so we will need to be able to adapt any environment we are given to work with any

possible context of new premises yielded by the rules of figure 4.4. In the simply typed case, the only change to the context we encountered was the binding of new variables. With usage annotations, we furthermore have linear decompositions of the context, necessitating changes to the environment whenever usage annotations change.

There are three kinds of linear decompositions we have to deal with: zero, addition, and scaling; corresponding to bunched connectives I^*, $*$, and $r \cdot$, respectively. In each of these three cases, we have a simple preservation lemma, transforming an environment of type $\Gamma \overset{\mathcal{V}}{\Longrightarrow} \Delta$ and a decomposition of Δ into a decomposition of Γ and environments for all of the decomposed fragments of Γ and Δ.

Lemma 5.2.3 (environments preserve zero)**.** Given an environment $\rho : \mathcal{P}\gamma \overset{\mathcal{V}}{\Longrightarrow} \mathcal{Q}\delta$ such that $\mathcal{Q} \leq 0$, we also have that $\mathcal{P} \leq 0$.

Proof. $\mathcal{P} \leq \mathcal{Q}\Psi \leq 0\Psi = 0$, by environment compatibility from ρ and monotonicity and linearity of Ψ. □

Lemma 5.2.4 (environments preserve addition)**.** Given an environment $\rho : \mathcal{P}\gamma \overset{\mathcal{V}}{\Longrightarrow} \mathcal{Q}\delta$ such that $\mathcal{Q} \leq \mathcal{Q}_l + \mathcal{Q}_r$ for some $\mathcal{Q}_l$ and $\mathcal{Q}_r$, we also have $\mathcal{P}_l$ and $\mathcal{P}_r$ such that $\mathcal{P} \leq \mathcal{P}_l + \mathcal{P}_r$ and there are environments $\rho_l : \mathcal{P}_l\gamma \overset{\mathcal{V}}{\Longrightarrow} \mathcal{Q}_l\delta$ and $\rho_r : \mathcal{P}_r\gamma \overset{\mathcal{V}}{\Longrightarrow} \mathcal{Q}_r\delta$.

Proof. Let $\mathcal{P}_l \coloneqq \mathcal{Q}_l\Psi$ and $\mathcal{P}_r \coloneqq \mathcal{Q}_r\Psi$. Then, $\mathcal{P} \leq \mathcal{Q}\Psi \leq (\mathcal{Q}_l + \mathcal{Q}_r)\Psi = \mathcal{Q}_l\Psi + \mathcal{Q}_r\Psi = \mathcal{P}_l + \mathcal{P}_r$, satisfying the first condition. Because clearly $\mathcal{P}_l \leq \mathcal{Q}_l\Psi$ and $\mathcal{P}_r \leq \mathcal{Q}_r\Psi$, applying lemma 5.2.1 to ρ gives us the required new environments ρ_l and ρ_r. □

Lemma 5.2.5 (environments preserve scaling)**.** Given an environment $\rho : \mathcal{P}\gamma \overset{\mathcal{V}}{\Longrightarrow} \mathcal{Q}\delta$ such that $\mathcal{Q} \leq r\mathcal{Q}'$ for some $\mathcal{Q}'$, we also have a $\mathcal{P}'$ such that $\mathcal{P} \leq r\mathcal{P}'$ and there is an environment $\rho' : \mathcal{P}'\gamma \overset{\mathcal{V}}{\Longrightarrow} \mathcal{Q}'\delta$.

Proof. Let $\mathcal{P}' \coloneqq \mathcal{Q}'\Psi$. Then, $\mathcal{P} \leq \mathcal{Q}\Psi \leq (r\mathcal{Q}')\Psi = r(\mathcal{Q}'\Psi) = r\mathcal{P}'$, satisfying the first condition. Because clearly $\mathcal{P}' \leq \mathcal{Q}'\Psi$, applying lemma 5.2.1 to ρ gives us the required new environment ρ'. □

The final change environments need to preserve is the binding of new free variables. In section 2.3.3, we had the operation bindEnv for this purpose in the intuitionistic

setting. There, we relied on $\mathcal{V}$ supporting a map from $\ni$-variables and admitting weakening. In the usage-annotated setting, the former requirement is updated to having a map from usage-checked $\exists$-variables. As for the latter requirement, it turns out that we only need $\mathcal{V}$ to admit weakening by 0-annotated variables, which is much more reasonable than general weakening. Lemma 5.2.6 adapts bindEnv for the usage-annotated setting.

Lemma 5.2.6 (bindEnv)**.** Given functions $\swarrow^k : \forall \Gamma, \mathcal{R}, \theta.\ \mathcal{R} \leq 0 \to \mathcal{V}\,\Gamma \nrightarrow \mathcal{V}\,(\Gamma, \mathcal{R}\theta)$ and $\mathrm{vr} : \exists \nrightarrow \mathcal{V}$, we can turn an environment of type $\Gamma \overset{\mathcal{V}}{\Longrightarrow} \Delta$ into an environment of type $\Gamma, \Theta \overset{\mathcal{V}}{\Longrightarrow} \Delta, \Theta$ for any context Θ.

Proof. Let $\mathcal{P}\gamma \coloneqq \Gamma$, $\mathcal{Q}\delta \coloneqq \Delta$, and $\mathcal{R}\theta \coloneqq \Theta$. Let the new linear map $\Psi' : \mathscr{R}^{|\Delta|+|\Theta|} \to \mathscr{R}^{|\Gamma|+|\Theta|}$ be $\Psi \oplus I$. That is, in block matrix notation, $\begin{pmatrix} \Psi & 0 \\ 0 & I \end{pmatrix}$. Checking that this linear map fits, we have $\begin{pmatrix} \mathcal{P} & \mathcal{R} \end{pmatrix} \leq \begin{pmatrix} \mathcal{Q}\Psi & \mathcal{R}I \end{pmatrix} = \begin{pmatrix} \mathcal{Q} & \mathcal{R} \end{pmatrix} (\Psi \oplus I)$. For the action on variables, we are given vectors $\mathcal{P}'$, $\mathcal{R}'_{\mathcal{P}}$, $\mathcal{Q}'$, and $\mathcal{R}'_{\mathcal{Q}}$ such that $\begin{pmatrix} \mathcal{P}' & \mathcal{R}'_{\mathcal{P}} \end{pmatrix} \leq \begin{pmatrix} \mathcal{Q}' & \mathcal{R}'_{\mathcal{Q}} \end{pmatrix} (\Psi \oplus I)$ and we have a variable of type $\mathcal{Q}'\delta, \mathcal{R}'_{\mathcal{Q}}\theta \ni A$ for some type A. The constraint on the new vectors reduces to $\mathcal{P}' \leq \mathcal{Q}'\Psi$ and $\mathcal{R}'_{\mathcal{P}} \leq \mathcal{R}'_{\mathcal{Q}}$. From the variable we either have a variable x in δ with $\mathcal{Q}' \leq \langle x|$ and $\mathcal{R}'_{\mathcal{Q}} \leq 0$, or a variable y in θ with $\mathcal{Q}' \leq 0$ and $\mathcal{R}'_{\mathcal{Q}} \leq \langle y|$. In the former case, the action of the original environment on x gives us a $\mathcal{V}$-value in $\mathcal{P}'\gamma$, and the 0-weakening principle $\swarrow^k$, noting that $\mathcal{R}'_{\mathcal{P}} \leq \mathcal{R}'_{\mathcal{Q}} \leq 0$, gives us a $\mathcal{V}$-value in $\mathcal{P}'\gamma, \mathcal{R}'_{\mathcal{P}}\theta$. In the latter case, we have that $\begin{pmatrix} \mathcal{P}' & \mathcal{R}'_{\mathcal{P}} \end{pmatrix} \leq \begin{pmatrix} \mathcal{Q}'\Psi & \mathcal{R}'_{\mathcal{Q}} \end{pmatrix} \leq \begin{pmatrix} 0\Psi & \langle y| \end{pmatrix} = \begin{pmatrix} 0 & \langle y| \end{pmatrix} = \langle \searrow y|$, so y also serves as a usage-checked variable in $\mathcal{P}'\gamma, \mathcal{R}'_{\mathcal{P}}\theta$. From this usage-checked variable, we get a $\mathcal{V}$-value in the same context using vr. □

I put together the preceding pieces to give a syntactic traversal operation over $\lambda\mathcal{R}$ in the following section. For the rest of this section, I observe some more constructions purely on environments — in particular, composition of environments given certain assumptions about the families of values.

Following Altenkirch et al. [2015], we expect (intuitionistic) STλC syntax to form a relative monad over $\ni$ seen as a functor from the category of contexts under renaming

to the functor category $[\mathrm{Ty}, \mathrm{Set}]$, where Ty is the discrete category of STλC types. Notice that, given $F, G : [\mathrm{Ty}, \mathrm{Set}]$, a morphism from F to G is a function of type $F \multimap G$ (with naturality being trivial). Therefore, we expect a relative monad, given as a Kleisli triple, to have a unit $\eta_\Gamma : \Gamma \ni (-) \multimap \Gamma \vdash (-)$ given by the variable rule, and a Kleisli extension operator ${}^*{}_{\Gamma,\Delta} : (\Gamma \ni (-) \multimap \Delta \vdash (-)) \to (\Gamma \vdash (-) \multimap \Delta \vdash (-))$ given by substitution. Composition of substitutions falls out of this framework as Kleisli composition. However, in the usage-aware case, substitution needs not just a mapping of variables $f : \Gamma \ni (-) \multimap \Delta \vdash (-)$, but rather an environment $\rho : \Delta \overset{\vdash}{\Longrightarrow} \Gamma$, as we have already discussed. It therefore makes sense for our replacement for the Kleisli extension operator to similarly take an environment rather than a simple variable mapping.

Lemma 5.2.8 below amounts to deriving a modified notion of Kleisli composition from a modified Kleisli extension. Additionally, lemma 5.2.7 is required to turn a monadic unit into an identity environment. Both lemmas are stated in terms of general $\mathcal{U}/\mathcal{V}/\mathcal{W}$-environments, with some specific examples (e.g. for renaming and substitution) below them.

Lemma 5.2.7 (Identity environment)**.** Given a function

$$\mathrm{vr}_{\Gamma'} : \Gamma' \ni (-) \multimap \mathcal{V}\,\Gamma'$$

for any Γ we have an environment $\mathrm{id} : \Gamma \overset{\mathcal{V}}{\Longrightarrow} \Gamma$.

Proof. Let $\Gamma = \mathcal{P}\gamma$. Let Ψ be the identity map, which clearly satisfies $\mathcal{P} \leq \mathcal{P}\Psi$. When acting on a variable, the inequality $\mathcal{P}' \leq \mathcal{Q}'\Psi$ means that $\mathcal{P}' \leq \mathcal{Q}'$. We are given a variable of type $\mathcal{Q}'\gamma \ni A$, which we can coerce to a variable of type $\mathcal{P}'\gamma \ni A$, upon which we apply vr to get the required value of type $\mathcal{V}\,\mathcal{P}'\gamma\,A$. □

Lemma 5.2.8 (Composition of environments)**.** Given a function

$$\mathrm{lift}_{\Gamma',\Delta'} : \Gamma' \overset{\mathcal{U}}{\Longrightarrow} \Delta' \to \mathcal{V}\,\Delta' \multimap \mathcal{W}\,\Gamma'$$

we can compose environments $\rho : \Gamma \overset{\mathcal{U}}{\Longrightarrow} \Delta$ and $\sigma : \Delta \overset{\mathcal{V}}{\Longrightarrow} \Theta$ into an environment $\rho \gg \sigma : \Gamma \overset{\mathcal{W}}{\Longrightarrow} \Theta$.

Proof. Let $\Gamma = \mathcal{P}\gamma$, $\Delta = \mathcal{Q}\delta$, and $\Theta = \mathcal{R}\theta$. Take Ψ to be the composition $(\sigma.\Psi)(\rho.\Psi)$, noting that $\mathcal{P} \leq \mathcal{Q}(\rho.\Psi) \leq (\mathcal{R}(\sigma.\Psi))(\rho.\Psi) = \mathcal{R}\Psi$ thanks to the inequalities yielded by σ and ρ. When acting on a variable, we are given $\mathcal{P}' \leq \mathcal{R}'\Psi$ and a variable $v : \mathcal{R}'\theta \ni A$, and want a value of type $\mathcal{W}\,\mathcal{P}'\gamma\,A$. Let $\mathcal{Q}' \coloneqq \mathcal{R}'(\sigma.\Psi)$, with inequality $\mathcal{Q}' \leq \mathcal{R}'(\sigma.\Psi)$ giving us a value $\sigma(v) : \mathcal{V}\,\mathcal{Q}'\delta\,A$. We wish to apply lift to $\sigma(v)$ with $\Gamma' \coloneqq \mathcal{P}'\gamma$ and $\Delta' \coloneqq \mathcal{Q}'\delta$ to complete the construction of the $\mathcal{W}$-value. To do this, we need an environment of type $\mathcal{P}'\gamma \overset{\mathcal{U}}{\Longrightarrow} \mathcal{Q}'\delta$, which we can get from ρ using lemma 5.2.1, noting that $\mathcal{P}' \leq \mathcal{R}'(\sigma.\Psi)(\rho.\Psi) = \mathcal{Q}'(\rho.\Psi)$. □

We can derive the following corollaries as instances of environment composition.

Corollary 5.2.9 (Composition of renamings)**.** Given renamings $\rho : \Gamma \overset{\ni}{\Longrightarrow} \Delta$ and $\sigma : \Delta \overset{\ni}{\Longrightarrow} \Theta$, we can form their composite $\rho;\sigma : \Gamma \overset{\ni}{\Longrightarrow} \Theta$.

Proof. Take $\mathcal{U} = \mathcal{V} = \mathcal{W} = \ni$ in lemma 5.2.8. Then let $\mathrm{lift}\,\rho\,x \coloneqq \rho(x)$. □

Corollary 5.2.10 (Post-composition with a renaming)**.** Given an environment $\rho : \Gamma \overset{\mathcal{U}}{\Longrightarrow} \Delta$ and a renaming $\sigma : \Delta \overset{\ni}{\Longrightarrow} \Theta$, we can form their composite $\rho;\sigma : \Gamma \overset{\mathcal{U}}{\Longrightarrow} \Theta$.

Proof. As in corollary 5.2.9. □

Corollary 5.2.11 (Pointwise renaming of an environment)**.** If $\overset{\mathcal{V}}{\models}$ respects renaming, then so does $\overset{\mathcal{V}}{\Longrightarrow}$ (on the left).

Proof. Suppose we have $\rho : \Gamma \overset{\ni}{\Longrightarrow} \Delta$ and $\sigma : \Delta \overset{\mathcal{V}}{\Longrightarrow} \Theta$. We want to compose these via lemma 5.2.8 with $U = \ni$ and $\mathcal{V} = \mathcal{W}$. The function lift is given exactly by the fact that $\mathcal{V}$ respects renaming. □

Corollary 5.2.12 (Composition of substitutions)**.** Given substitutions $\rho : \Gamma \overset{\vdash}{\Longrightarrow} \Delta$ and $\sigma : \Delta \overset{\vdash}{\Longrightarrow} \Theta$, we can form their composite $\rho;\sigma : \Gamma \overset{\vdash}{\Longrightarrow} \Theta$.

Proof. Take $\mathcal{U} = \mathcal{V} = \mathcal{W} = \vdash$ in lemma 5.2.8. Then, lift is given by the action of a substitution on a term (see sub in the following section). □

Corollary 5.2.13 (Composing semantics with substitution)**.** If we have a semantics (in the sense of section 2.4 and section 7.4) from $\mathcal{U}$ to $\mathcal{W}$, then from an environment $\rho : \Gamma \overset{\mathcal{U}}{\Longrightarrow} \Delta$ and a substitution $\sigma : \Delta \overset{\vdash}{\Longrightarrow} \Theta$, we can form the composite $\rho;\sigma : \Gamma \overset{\mathcal{W}}{\Longrightarrow} \Theta$.

5.3 Substitution is admissible in $\lambda\mathcal{R}$

I can now show that, using the notion of *environment* derived in section 5.1, we can replicate the Agda proofs from section 2.3.3 in the usage-aware setting of $\lambda\mathcal{R}$. From section 5.2, we know that environments are preserved under all syntax-forming operations: zero, addition, scaling, and binding. What is left is to show how these properties are deployed, and also how to go on and prove the admissibility of simultaneous renaming, simultaneous substitution, and then single substitution.

There are a few notational changes necessary in the Agda code, compared to the typeset mathematics above. Usage vectors, elsewhere called $\mathcal{P}$, $\mathcal{Q}$, and $\mathcal{R}$ are rendered as P, Q, and R, respectively. Usage contexts and typing contexts are tied together with the ctx constructor, rather than simple juxtaposition. Environments, elsewhere notated $\Gamma \overset{\mathcal{V}}{\Longrightarrow} \Delta$, are rendered as $[\ \mathcal{V}\ ]\ \Gamma \Rightarrow^{e} \Delta$.

We start with a slightly modified definition of Kit. We saw in lemma 5.2.6 that in the usage-annotated context, we restrict weakening of $\mathcal{V}$-values to just 0-use variables. Meanwhile, the function vr, also seen in lemma 5.2.6, maps usage-checked variables to $\mathcal{V}$-values, and the function tm, used to coerce V-values yielded by the environment into terms, stays the same. I state weakening in a slightly different way than previously, so as to help unification against a known result type (avoiding the problem described by McBride [2012] as *green slime*). The type Weakening $\mathcal{V}$ can be read as saying that, for any context $\mathcal{P}\gamma$ of shape $s + t$, if the right of $\mathcal{P}$ is below 0, then a value in the left part of $\mathcal{P}\gamma$ weakens to a value in the whole of $\mathcal{P}\gamma$.

```
Weakening : ∀ {v} {A : Set} → (Ctx → A → Set v) → Set v
Weakening 𝒱 =
  ∀ {s t P} {γ : Vector Ty (s <+> t)} → P ∘ ↘ ≤* 0* →
  𝒱 (ctx (P ∘ ↙) (γ ∘ ↙)) ⊸ 𝒱 (ctx P γ)

record Kit (𝒱 : Ctx → Ty → Set) : Set where
  constructor kit
  field
    ↙ᵏ : Weakening 𝒱
```

```
vr : _∃_ ⇾ 𝒱
tm : 𝒱 ⇾ _⊢_
```

To demonstrate the important points succinctly, I cut $\lambda\mathcal{R}$ down to just the $!r$-fragment. The introduction rule and pattern-matching eliminator feature scaling, addition, and variable binding, missing out only on sharing (which is trivial) and zero (which is simpler than, and analogous to, addition). The resulting type of well typed terms is below.

```
Bind : Ctx → (Ctx → Set) → (Ctx → Set)
Bind Δ T Γ = T (Γ ++ᶜ Δ)

data _⊢_ : Ctx → Ty → Set where
  var : _∃_ ⇾ _⊢_
  !-I : ∀ {r A} → r · (_⊢ A) ⇾ _⊢ ‘! r A
  !-E : ∀ {r A C} → (_⊢ ‘! r A) ∗ Bind [ r • A ]ᶜ (_⊢ C) ⇾ _⊢ C
```

Given a `Kit` $\mathcal{V}$, lemma 5.2.6 gives a function with the following type.

```
bindEnv : ∀ {Γ Δ Θ} → [ 𝒱 ] Γ ⇒ᵉ Δ → [ 𝒱 ] Γ ++ᶜ Θ ⇒ᵉ Δ ++ᶜ Θ
```

Given `bindEnv` (lemma 5.2.6), `env-+` (lemma 5.2.4), and `env-*` (lemma 5.2.5), we can reproduce the syntactic traversal `trav`. With all these lemmas in place, writing `trav` becomes routine. When processing a rule, we work our way up through the premise connectives, applying `env-*` wherever we see a $\cdot^c$, `env-+` wherever we see a $*^c$, and `bindEnv` wherever we see a `Bind`. We then use whatever environments (with names beginning with ρ) and whatever usage vector splitting facts (with names beginning with sp) come out of this process to recursively traverse the subterms and recombine the results.

```
trav : ∀ {Γ Δ A} → [ 𝒱 ] Γ ⇒ᵉ Δ → Δ ⊢ A → Γ ⊢ A
trav ρ (var x) = tm (ρ .lookup (ρ .fit-here) x)
trav ρ (!-I (⟨ sp* ⟩·ᶜ M)) =
  let sp*′ , ρ′ = env-* (ρ , sp*) in
  !-I (⟨ sp*′ ⟩·ᶜ (trav ρ′ M))
```

```
trav ρ (!-E (M *ᶜ⟨ sp+ ⟩ N)) =
  let ρl ↘, sp+′ ,↙ ρr = env-+ (ρ , sp+) in
  !-E (trav ρl M *ᶜ⟨ sp+′ ⟩ trav (bindEnv ρr) N)
```

Instantiating the generic syntactic traversal trav to renaming looks just like it did in the intuitionistic case. I have consistently replaced intuitionistic variables by linear variables, so id and var still work to embed variables into variables and terms, respectively. Weakening for variables $\swarrow^v$ (not pictured) has been updated to note that, for $\mathcal{P} \leq \langle x|$ and $\mathcal{R} \leq 0$, we also have $\begin{pmatrix}\mathcal{P} & \mathcal{R}\end{pmatrix} \leq \langle\swarrow x|$.

```
∃-kit : Kit _∃_
∃-kit = kit ↙ᵛ id var

ren : ∀ {Γ Δ A} → [ _∃_ ] Γ ⇒ᵉ Δ → Δ ⊢ A → Γ ⊢ A
ren = trav ∃-kit
```

In the intuitionistic case, environments were just functions, so we passed the variable weakening function $\swarrow^v$ to the function ren to yield a term weakening function. However, a usage-aware environment is a function packed together with usage distribution data. As such, we must make an environment version of $\swarrow^v$. I start with a general lemma ↙^Env, stating that if $\mathcal{V}$ supports weakening, then so do $\mathcal{V}$-environments (in their domain context). This lemma then specialises to variables, with the identity renaming id^Env on the left part of the context and the proof $R0$ that the right part of the context is below 0 combining to give the desired weakening environment.

```
↙^Env : {𝒱 : Ctx → Ty → Set} → Weakening 𝒱 → Weakening [ 𝒱 ]_⇒ᵉ_
↙^Env ↙^𝒱 R0 ρ .Ψ = [ ρ .Ψ | 0ᴿ ]ᴿ
↙^Env ↙^𝒱 R0 ρ .fit-here = ρ .fit-here , ≤*→0* R0
↙^Env ↙^𝒱 R0 ρ .lookup (r , sp0) v = ↙^𝒱 (0*→≤* sp0) (ρ .lookup r v)

↙ᵛ-env :
  ∀ {s t P} {γ : Vector Ty (s <+> t)} → P ∘ ↘ ≤* 0* →
  [ _∃_ ] ctx P γ ⇒ᵉ ctx (P ∘ ↙) (γ ∘ ↙)
↙ᵛ-env R0 = ↙^Env ↙ᵛ R0 id^Env
```

This is what we need to instantiate trav for substitution. As a reminder, I also give the type of sub in rule form.

```
⊢-kit : Kit _⊢_
⊢-kit = kit (λ R0 → ren (↙ᵛ-env R0)) var id

sub : ∀ {Γ Δ A} → [ _⊢_ ] Γ ⇒ᵉ Δ → Δ ⊢ A → Γ ⊢ A
sub = trav ⊢-kit
```

$$\frac{\Gamma \overset{\vdash}{\Longrightarrow} \Delta \qquad \Delta \vdash B}{\Gamma \vdash B}\ \text{SUB}$$

Finally, the simultaneous substitution sub specialises to single substitution.

Corollary 5.3.1 (Single substitution)**.** The following equivalent rules are admissible.

$$\frac{\mathcal{R} \le r\mathcal{P} + \mathcal{Q} \qquad \mathcal{P}\gamma \vdash A \qquad \mathcal{Q}\gamma, rA \vdash B}{\mathcal{R}\gamma \vdash B} \qquad \frac{r \cdot (\vdash A) \quad * \quad rA \vdash B}{\vdash B}$$

Proof. It is enough to construct a substitution of type $\mathcal{R}\gamma \overset{\vdash}{\Longrightarrow} \mathcal{Q}\gamma, rA$. To do this, we use lemma 5.2.2 cases $*(\twoheadrightarrow)$ and $\cdot(\twoheadrightarrow)$ on inequalities $\mathcal{R} \le \mathcal{Q} + r\mathcal{P}$ and $r\mathcal{P} \le r\mathcal{P}$ respectively to leave us needing a substitution of type $\mathcal{Q}\gamma \overset{\vdash}{\Longrightarrow} \mathcal{Q}\gamma$ and a term of type $\mathcal{P}\gamma \vdash A$. For the substitution, we give the identity substitution (lemma 5.2.7), and we have the term as a hypothesis. □

5.4 Comparison with Petricek's substitution lemma

A similar substitution lemma to the one presented in this chapter appears in the PhD thesis of Petricek [2017, p. 138] under the name *multi-nary substitution*. In my notation, Petricek's substitution rule looks like the following, up to permutation of the contexts containing Γ. Note that if $\Delta = \mathcal{Q}\delta$, then $r\Delta$ denotes the context $(r\mathcal{Q})\delta$. This rule is essentially an iterated version of the standard linear single substitution principle, and is used by Petricek as a strengthened induction hypothesis required to derive single substitution.

$$\frac{\Delta_1 \vdash A_1 \qquad \cdots \qquad \Delta_n \vdash A_n \qquad \Gamma, r_1 A_1, \ldots, r_n A_n \vdash B}{\Gamma, r_1\Delta_1, \ldots, r_n\Delta_n \vdash B}$$

We can derive Petricek-style multi-nary substitution as a corollary of my simultaneous substitution, using reasoning similar to that of corollary 5.3.1.

Corollary 5.4.1. Petricek's multi-nary substitution, as stated above, is admissible in $\lambda\mathcal{R}$.

Proof. It is enough to provide a substitution of type

$$\Gamma, r_1\Delta_1, \ldots, r_n\Delta_n \overset{\vdash}{\Longrightarrow} \Gamma, r_1 A_1, \ldots, r_n A_n.$$

To do this, we use lemma 5.2.2 repeatedly, leaving us needing a substitution of type $\Gamma, 0\Delta_1, \ldots, 0\Delta_n \overset{\vdash}{\Longrightarrow} \Gamma$ and terms of types

$$\begin{gathered} 0\gamma, \Delta_1, 0\delta_2, \ldots, 0\delta_{n-1}, 0\delta_n \vdash A_1 \\ \vdots \\ 0\gamma, 0\delta_1, 0\delta_2, \ldots, 0\delta_{n-1}, \Delta_n \vdash A_n. \end{gathered}$$

The identity substitution and weakening by 0-annotated variables is enough to make these requirements line up with the given hypotheses. □

My substitution principle is stronger than Petricek's. Where Petricek requires that distinct variables be available for each hypothesis, I allow for separation of uses via addition of contexts. Below is a prototypical example.

Example 5.4.2. Let $\mathscr{R} \coloneqq (\mathbb{N}, =, 0, +, 1, \times)$, the exact usage-counting posemiring. Then, we can construct a substitution $\rho : 2A \overset{\vdash}{\Longrightarrow} 1A, 1A$, yielding a transformation of terms of the following form:

$$\frac{1A, 1A \vdash B}{2A \vdash B}.$$

To construct ρ, we use lemma 5.2.2 case $*(\multimap)$, using the fact that $2 \leq 1 + 1$. From there, two identity substitutions suffice. The action of ρ on terms is to merge the two variables into one. Note that a renaming, rather than a substitution, would also suffice.

Most notably, my (single) substitution principle more naturally fits the requirement we would have for the reduct of the β-rule for functions in $\lambda\mathcal{R}$, whereas Petricek's

substitution principle would need some additional transformation for it to fit properly. This comes from the fact that the $\lambda\mathcal{R}$ function application rule introduces an algebraic $(+)$ separation between its premises, whereas Petricek's substitution principle separates premises only via concatenation.

5.5 Conclusion

In this and the preceding chapter, I have developed a discipline for specifying the syntax of linear and modal type systems, and furthermore developing the syntactic metatheory of those type systems. All of these are based on semirings, and the linear algebra arising from considering a usage context full of semiring elements as a vector.

These developments can be seen in retrospect as a generalisation of the methods explained in chapter 2. In terms of premise connectives in the syntactic rules, we have generalised from just $\{\dot{1}, \dot{\times}\}$ to $\{\dot{1}, \dot{\times}, I^*, *, r \cdot, \square^{0+}\}$, maintaining our ability to keep the context implicit. Similarly to how rule premises can require separation of usage annotations, our new environments can require such a separation between their entries thanks to the linear map they now contain. I have generalised the key property of a kit from arbitrary weakening to weakening by 0-annotated variables, and using that have produced a substitution operation based on the same principles as that from section 2.3.

Having generalised all of the components — namely the contexts, the syntax, and the notion of environment — the type of the substitution operation looks the same as it did for intuitionistic STλC. Being able to maintain this uniformity is a key step towards generalising the rest of chapter 2 (i.e., sections 2.4 and 2.5), as I do in chapter 6.

Future work may want to extend the work of this chapter, in which case there are some unanswered questions. Principal among these, in my mind, is dealing with equivalence/equality of environments. We want to talk about equality of environments for two related purposes. The most immediate is that we want to develop the equational theory of renaming and substitution — for example, when we use lemma 5.2.8 to compose substitutions, we expect that composition to be associative and unital (with respect to lemma 5.2.7). These equations of substitutions should yield equations on terms in which such substitutions have been applied. Slightly more abstractly, I would like to

develop a theory of *quantitative multicategories*, in which multimorphisms have in their domain a list of objects paired with usage annotations. I would hope for $\lambda\mathcal{R}$ types and terms to give an example quantitative multicategory, analogously to how Lambek calculus gives an example of an ordinary multicategory and the simply typed λ-calculus gives an example of a Cartesian multicategory.

Intuitionistic environments $\rho, \sigma : \Gamma \xRightarrow{\mathcal{V}} \Delta$ are equal if and only if, for each type A and each variable $x : \Delta \ni A$, we have $\rho\, x = \sigma\, x$. This follows from what we expect of equality of functions (function extensionality). Usage-aware environments, on the other hand, are Σ-types — a way of dividing up the usage annotations of Γ, and then a function producing $\mathcal{V}$-values whose usage annotations come from that division. Equality of Σ-types is tricky — we need to equate the first components, then rewrite the types of the second components by this equation before equating them. In practice, the equations rewriting other equations build up so much that I have given up on a first effort to give a treatment of the equational theory of substitutions. Note that recursively defined environments (definition 5.1.3) are also Σ-types in cases where Δ is a concatenation of contexts, so that definition does not clearly help.

I hope that people working on substructural type systems in the future can take inspiration from the process laid out in section 5.1 when working out the appropriate notion of environment for their discipline. Particularly, definition 5.1.3 (the recursive definition) should serve as a specification, if not the actual implementation, when coming across a new substructural discipline. As for the progression to definition 5.1.7, this appears to arise from the fact that the quantitative usage information is a refinement of the intuitionistic De Bruijn index-based syntax.

Chapter 6

Generic usage-annotated syntax

In chapter 4, we saw how to use parametrisation over a partially ordered semiring to recreate a range of usage-aware calculi. However, $\lambda\mathcal{R}$, with its fixed set of type formers and syntactic forms, is a long way from capturing the full range of linear-like programming languages studied in the literature and required in practice.

In this chapter, I take the framework for typed syntaxes with binding developed by Allais et al. [2021] and apply the principles we discovered in chapter 4 to yield a framework for defining calculi with semiring-based usage restrictions on variables. Syntactically, I claim that this framework ranges over all finitary variable-based simply typed semiring-annotated calculi, with justification by comparison to the framework of Allais et al. [2021] and some novel examples in section 6.3.

The work in this chapter is fully mechanised in Agda, which allows me to be precise about the various levels of domain-specific languages which appear. Most of the code and text of this chapter, chapter 7, and the examples in sections 8.1 and 8.3 is adapted from Wood and Atkey [2022].

The aim of this chapter is to produce a domain-specific language of *syntax descriptions*. One may also use the terminology "universe" in place of "domain-specific language", alluding to the *universe pattern* common in dependently typed programming [Benke et al., 2003]. A syntax description is essentially a concise, high-level representation of a type system's syntactic rules. The information contained in a syntax description is comparable to what is written (informally) in figure 4.4. The key features

allowing these descriptions to capture semiring-annotated calculi are the distinction between sharing ($\dot{\times}$) and separating ($*$) conjunction of premises, modal scaling by a semiring element ($\cdot$), and the inclusion of semiring annotations on newly bound variables.

6.1 Descriptions of Systems

I introduce syntax descriptions in three layers: System, Rule, and Premises. A type System is made up of multiple Rules. Each Rule comprises a Premises and a conclusion type. We assume that there is a Ty : Set of types for the system in scope.

The Premise data type describes premises of rules, using the bunched connectives from figure 4.3. A single premise is introduced by the $\langle \_ \text{`}\vdash \_ \rangle$ constructor. This allows binding of additional variables Δ (with specified types and usage annotations) and the specification of a conclusion type A for this premise. The remaining constructors are descriptions for the bunched connectives.

```
data Premises : Set where
  ⟨_`⊢_⟩ : (Δ : Ctx) (A : Ty) → Premises
  `i : Premises;      _`×̇_ : (p q : Premises) → Premises
  `I* : Premises;     _`*_ : (p q : Premises) → Premises
  _`·_ : (r : Ann) (p : Premises) → Premises
```

A Rule is a pair of some Premises and a conclusion. I suggestively use a quoted version of the "universal entailment" arrow $\multimap$, the unquoted version of which interprets the horizontal line in a traditionally presented typing rule.

```
record Rule : Set where
  constructor _`⊸_
  field premises : Premises; conclusion : Ty
```

Finally, a System consists of a set of rule labels (i.e., constructor names), and for each label a description of the corresponding rule. We use $\triangleright$ as infix notation for systems to associate the label set with the rules.

```
record System : Set₁ where
  constructor _▷_
  field Label : Set; rules : (l : Label) → Rule
```

As an example, we transcribe a fragment of $\lambda\mathcal{R}$ (as defined in figure 4.2 and figure 4.4) into a description. We give the set of types of this system as a data type Ty (together with a base type ι). We assume that there is a posemiring Ann in scope for the annotations. There is one label for each instantiation of a logical rule, but the labels contain no further information about subterms or restrictions on the context. This will be provided when we associate labels with Rules in a System.

```
data Ty : Set where
  ι : Ty
  _⊸_ _⊕_ : (A B : Ty) → Ty
  ! : (r : Ann) (A : Ty) → Ty

data Hand : Set where ll rr : Hand
```

```
data 'λR : Set where
  '⊸I '⊸E : (A B : Ty) → 'λR
  '⊕I : (i : Hand) (A B : Ty) → 'λR
  '⊕E : (A B C : Ty) → 'λR
  '!I : (r : Ann) (A : Ty) → 'λR
  '!E : (r : Ann) (A C : Ty) → 'λR
```

To build a system, we associate with each label a rule:

```
λR : System
λR = 'λR ▷ λ where
  ('⊸I A B)     → ⟨ [ 1# • A ]ᶜ '⊢ B ⟩                          '→ (A ⊸ B)
  ('⊸E A B)     → (⟨ []ᶜ '⊢ A ⊸ B ⟩ '* ⟨ []ᶜ '⊢ A ⟩)             '→ B
  ('!I r A)     → (r '· ⟨ []ᶜ '⊢ A ⟩)                           '→ (! r A)
  ('!E r A C)   → (⟨ []ᶜ '⊢ ! r A ⟩ '* ⟨ [ r • A ]ᶜ '⊢ C ⟩)     '→ C
  ('⊕I ll A B)  → ⟨ []ᶜ '⊢ A ⟩                                  '→ (A ⊕ B)
  ('⊕I rr A B)  → ⟨ []ᶜ '⊢ B ⟩                                  '→ (A ⊕ B)
  ('⊕E A B C)   →
    ⟨ []ᶜ '⊢ A ⊕ B ⟩ '* (⟨ [ 1# • A ]ᶜ '⊢ C ⟩ '⅋ ⟨ [ 1# • B ]ᶜ '⊢ C ⟩) '→ C
```

Compared to figure 4.4, modulo the Agda notation, we can see that the fundamental

structure has been preserved: The rules match one-to-one, and the bunched premises are the same. A major difference is that we do not include a counterpart to the var rule in a System. Variables are common to all the systems representable in our framework.

6.2 Terms of a System

The next thing we want to do is to build terms in the described type system. The following definitions are useful for talking about types indexed over contexts, judgement forms, and judgement forms admitting newly bound variables, respectively.

```
OpenType : ∀ ℓ → Set (suc ℓ)
OpenType ℓ = Ctx → Set ℓ

OpenFam : ∀ ℓ → Set (suc ℓ)
OpenFam ℓ = Ctx → Ty → Set ℓ

ExtOpenFam : ∀ ℓ → Set (suc ℓ)
ExtOpenFam ℓ = Ctx → OpenFam ℓ
```

To specify the meaning of descriptions, we assume some X : ExtOpenFam _, over which we form one layer of syntax, using the function ⟦_⟧p that interprets Premises defined below. The first argument to X is the new variables bound by this layer of syntax, as exemplified in the first clause of ⟦_⟧p. The second argument is the context containing the variables being carried over from the previous layer. Notice that this is not, in general, the same as the context from the previous layer, because the usage annotations may have been changed by connectives like _‘*_ and _‘·_. The third argument is the type of subterm required.

The remainder of the clauses of ⟦_⟧p are given by the bunched connectives, as listed below.

```
⟦_⟧p : Premises → ExtOpenFam ℓ → OpenType ℓ
⟦ ⟨ Δ ‘⊢ A ⟩ ⟧p X Γ = X Δ Γ A
⟦ ‘i ⟧p X = i;      ⟦ p ‘×̇ q ⟧p X = ⟦ p ⟧p X ×̇ ⟦ q ⟧p X
```

```
⟦ ‘I* ⟧p X = I*;   ⟦ p ‘* q ⟧p X = ⟦ p ⟧p X * ⟦ q ⟧p X
⟦ r ‘· p ⟧p X = r · ⟦ p ⟧p X
```

The interpretation of a Rule checks that the rule targets the desired type and then interprets the rule's premises ps. Notice that the interpretation of the premises is independent of the conclusion of the rule, which accounts for the use of OpenType in ⟦_⟧p versus OpenFam in ⟦_⟧r.

```
⟦_⟧r : Rule → ExtOpenFam ℓ → OpenFam ℓ
⟦ ps ‘⇾ A′ ⟧r X Γ A = A′ ≡ A × ⟦ ps ⟧p X Γ
```

The interpretation of a System is to choose a rule label l from L and interpret the corresponding rule $rs\ l$ in the same context and for the same conclusion.

```
⟦_⟧s : System → ExtOpenFam ℓ → OpenFam ℓ
⟦ L ▷ rs ⟧s X Γ A = Σ[ l ∈ L ] ⟦ rs l ⟧r X Γ A
```

The most obvious way to make an X : ExtOpenFam _ is to use some existing OpenFam on an extended context. I define Scope to do this: take the new variables Δ, concatenate them onto the existing context Γ, and pass the extended context onto the judgement T.

```
Scope : ∀ {ℓ} → OpenFam ℓ → ExtOpenFam ℓ
Scope T Δ Γ A = T (Γ ++ᶜ Δ) A
```

I use Scope to deal with new variables in syntax. Terms resemble the free monad over a layer-of-syntax functor, though that picture is complicated by variable binding. A term is either a variable or a use of a logical rule together with terms for each of the required subterms. The Size argument sz is a use of Agda's sized types to record that subterms are smaller than the surrounding term for the termination checker.

```
data [_,_]_⊢_ (d : System) : Size → OpenFam 0ℓ where
  ‘var  : _∋_                          ⇾ [ d , ↑ sz ]_⊢_
  ‘con  : ⟦ d ⟧s (Scope [ d , sz ]_⊢_)  ⇾ [ d , ↑ sz ]_⊢_
```

Terms in this data type are difficult to write by hand, due to the need for proofs that the usage contexts are handled correctly. For example, the following term is needed to show that, in the $\{0, 1, \omega\}$ (linearity) posemiring of example 4.1.3, $!\omega$ forms a comonad. Pattern synonyms $\multimap$I, !E′, and !I′ stand for applications of ‘con, with the latter two taking explicit usage contexts and proofs. On concrete posemirings (as in this example), unification is particularly poor at inferring the usage contexts from the proofs because addition and multiplication are no longer (definitionally) injective. The function var# is a way of turning a statically known De Bruijn level and a usage proof into an application of ‘var. In the type, ∞ is the “infinite” size, which all sizes are less than. Effectively, writing ∞ allows us to ignore sizes when we are not doing recursion involving sizes.

```
cojoin-!ω : ∀ A → [ λR , ∞ ] []ᶜ ⊢ (! ω# A ⊸ ! ω# (! ω# A))
cojoin-!ω A =
  ⊸I (!E′ ([] ++ [ 1# ]) ([] ++ [ 0# ]) ([]ₙ ++ₙ [ ≤-refl ]ₙ)
    (var# 0 (([]ₙ ++ₙ [ ≤-refl ]ₙ) ++ₙ []ₙ))
    (!I′    (([] ++ [ 0# ]) ++ [ ω# ])
            (([]ₙ ++ₙ [ ≤-refl ]ₙ) ++ₙ [ ≤-refl ]ₙ)
      (!I′  ((([] ++ [ 0# ]) ++ [ ω# ]) ++ [])
            ((([]ₙ ++ₙ [ ≤-refl ]ₙ) ++ₙ [ ≤-refl ]ₙ) ++ₙ []ₙ)
        (var# 1
          (((([]ₙ ++ₙ [ ≤-refl ]ₙ) ++ₙ [ ω≤1 ]ₙ) ++ₙ []ₙ) ++ₙ []ₙ))))))
```

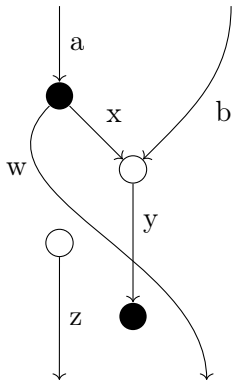
Writing terms like this is clearly unsustainable. We will see a way of automating the necessary proofs via a System-generic elaborator in section 8.1.

6.3 More example syntaxes

With the range of representable syntaxes now formalised, we can explore encoding techniques for syntaxes more exotic than STλC and $\lambda\mathcal{R}$. As well as the variations presented in section 4.4, we can represent a usage-annotated $\mu\tilde{\mu}$-calculus and a Linear/non Linear system.

6.3.1 An encoding of graphs

As a non-logical example of a syntax that can be encoded using linear syntax descriptions, I consider a language of directed acyclic hypergraphs, which can be used as string diagrams for symmetric monoidal categories. I want to represent hypergraphs like the following, made up of operators {  } and wires between them with no fan-in or fan-out. I have given the wires names (a, b, w, x, y, and z), which I use in a textual representation of the graph on the right. The textual representation is in A-normal form, where the two nullary operators have names beginning with `eta`, the two binary operators have names beginning with `mu`, and names having the suffix `white` or `black` based on the corresponding operator's colour.




```
a, b |-
let w, x := mu_black(a) in
let y := mu_white(x, b) in
let z := eta_white() in
let := eta_black(y) in
z, w
```

In the textual representation, I treat inputs a and b as free variables, while outputs z and w are listed on the final line. The rest of the expression is a series of `let`-expressions binding zero or more variables to the results of an operator applied to other variables. All of the variables are used linearly, avoiding any fan-out.

To turn this representation into a type system, I do the following. First, I fix the usage annotation semiring as the $\{0, 1, \omega\}$-semiring to achieve linearity. Then, I set Ty to be the following type. All of the variables get type wire, with a context full of wire-type variables corresponding to the inputs of the graph. Meanwhile, the whole expression has type bundle s, where s gives the shape of the outputs.

```
data Sort : Set where
  wire : Sort
  bundle : LTree → Sort
```

The type system itself is as follows. Each operator gets its own syntactic construct, with essentially the same naming conventions as for the textual representation above. Each of these constructs is an entire `let`-expression. There is also an ‘end construct, allowing us to finish and list the outputs.

To see how the encodings of the operations work, let us look at the ‘μ•-construct (). It has two premises conjoined by ∗ (in fact, all premises in this language are separated, with no use of sharing conjunction). The first premise expects a simple value, corresponding to the input of the operator. The second premise is more complex, and represents the body/continuation of the `let`-expression. It binds two variables, corresponding to the two outputs, which can then be used in later `let`-expressions or the outputs. The bound variables all have type wire and usage annotation 1 (written u1 in Agda) to make them behave linearly.

```
data ‘GraphL : Set where
  ‘η○ ‘μ○ ‘η• ‘μ• ‘end : LTree → ‘GraphL

GraphL : System
GraphL = ‘GraphL ▷ λ where
  (‘η○ s) →   ⟨ [ u1 • wire ]ᶜ ‘⊢ bundle s ⟩ ‘⇾ bundle s
  (‘μ○ s) →   ⟨ []ᶜ ‘⊢ wire ⟩ ‘∗ ⟨ []ᶜ ‘⊢ wire ⟩ ‘∗ ⟨ [ u1 • wire ]ᶜ ‘⊢ bundle s ⟩
              ‘⇾ bundle s
  (‘η• s) →   ⟨ []ᶜ ‘⊢ wire ⟩ ‘∗ ⟨ []ᶜ ‘⊢ bundle s ⟩ ‘⇾ bundle s
  (‘μ• s) →   ⟨ []ᶜ ‘⊢ wire ⟩ ‘∗ ⟨ [ u1 • wire ]ᶜ ++ᶜ [ u1 • wire ]ᶜ ‘⊢ bundle s ⟩
              ‘⇾ bundle s
  (‘end s) →  end-premises s ‘⇾ bundle s
```

The required premises of the ‘end-rule are calculated by end-premises. This follows the tree structure of the shape, requiring a wire-term in an unextended context for each leaf.

```
end-premises : LTree → Premises
end-premises [-] = ⟨ []ᶜ ‘⊢ wire ⟩
```

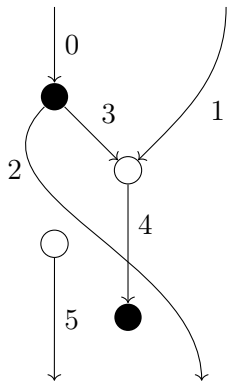
```
end-premises ε = 'I*
end-premises (s <+> t) = end-premises s '* end-premises t
```

A graph is a term of the graph language. Specifically, a graph with inputs of shape s and outputs of shape t has the following type — a term in a context of shape s, all of whose entries are wire variables with annotation 1, whose result type is bundle t.

```
Graph : LTree → LTree → Set
Graph s t = [ GraphL , ∞ ] ctx {s} (λ _ → u1) (λ _ → wire) ⊢ bundle t
```

Finally, I give an example term myGraph. As I said at the end of section 6.2, writing a term out in full is tedious, so I instead choose to use the machinery I discuss in detail in section 8.1. The elab-unique tool elaborates a well typed term into a well typed and usage-correct term as long as it can infer assignments of usage annotations that satisfy the constraints. I add the prefix u to signify the unannotated (just well typed) terms. All of the checks required by the elaboration procedure are done at Agda's type checking time, so we know that the program listed below can be elaborated by virtue of the whole Agda definition being type-checked.

The uvar# tool is like the var# tool I used earlier — allowing us to refer to variables by counting from the left-hand end of the context. I add these numbers to the picture of the graph for convenience. Variables never go out of scope, despite being used, so these numbers do not change with any bindings.




```
myGraph : Graph ([-] <+> [-]) ([-] <+> [-])
myGraph = elab-unique GraphL
  (uμ• (uvar# 0)
  (uμ○ (uvar# 3) (uvar# 1)
  (uη○
  (uη• (uvar# 4)
  (uend (uvar# 5 *ᶜ⟨ _ ⟩ (uvar# 2)))))))
  (λ _ → u1)
```

6.3.2 The system $\mu\tilde{\mu}$

I encode a usage-annotated version of System L/the $\mu\tilde{\mu}$-calculus [Curien and Herbelin, 2000] — a syntax for classical logic — in such a way that contexts capture the undistinguished parts of the sequent. As such, the generic substitution lemma we get in section 7.6 is the form of substitution required in standard $\mu\tilde{\mu}$-calculus metatheory. Though the $\mu\tilde{\mu}$-calculus is originally described as a sequent calculus [Curien and Herbelin, 2000], I use the techniques of Herbelin [2005, p. 12] and Lovas and Crary [2006] to present it using hypothetical judgements, thus giving a notion of *variable* to the system.

Unlike the single judgement form of $\lambda\mathcal{R}$ and standard simply typed λ-calculi, the $\mu\tilde{\mu}$-calculus has three judgement forms: terms (traditional notation: $\Gamma \vdash A \mid \Delta$), coterms ($\Gamma \mid A \vdash \Delta$), and commands ($\Gamma \vdash \Delta$). Read logically, terms and coterms are seen to, respectively, prove and refute propositions (types), while commands exhibit contradictions. This means that the abstract *Ty* in the generic framework is instantiated to Conc (for *conclusion*) as below, with Ty not being exposed directly to the generic framework. For now, I just consider multiplicative disjunction ⅋ (*par*) and negation/duality, beside an uninterpreted base type. These are enough to exhibit classical behaviour.

```
data Ty : Set where
  base : Ty
  _⅋_ : (rA sB : Ann × Ty) → Ty
  _^⊥ : (A : Ty) → Ty

data Conc : Set where
  com : Conc
  trm cot : (A : Ty) → Conc
```

With *Ty* instantiated as Conc, all terms are assigned Conc type, as are all the variables. No variables are given com type, similar to how in the bidirectional typing syntax of Allais et al. [2021, p. 25], no variables are given Check type. How to observe this invariant is covered in the latter paper, so we will not repeat it here (having not yet seen how to write traversals on terms).

The syntax comprises a *cut* between a term and a coterm of the same type, the eponymous μ and $\tilde{\mu}$ constructs for proof by contradiction, and then term and coterm (introduction and elimination) forms for negation and *par*. I give a traditional presen-

$$\frac{\mathcal{P} \le \mathcal{P}_l + \mathcal{P}_r \qquad \mathcal{Q} \le \mathcal{Q}_l + \mathcal{Q}_r \qquad \mathcal{P}_l\gamma \vdash A \mid \mathcal{Q}_l\delta \qquad \mathcal{P}_r\gamma \mid A \vdash \mathcal{Q}_r\delta}{\mathcal{P}\gamma \vdash \mathcal{Q}\delta}\ \textsc{Cut}_A$$

$$\frac{\Gamma \vdash 1A, \Delta}{\Gamma \vdash A \mid \Delta}\ \mu \qquad \frac{\Gamma, 1A \vdash \Delta}{\Gamma \mid A \vdash \Delta}\ \tilde{\mu} \qquad \frac{\Gamma \mid A \vdash \Delta}{\Gamma \vdash A^{\perp} \mid \Delta}\ \lambda \qquad \frac{\Gamma \vdash A \mid \Delta}{\Gamma \mid A^{\perp} \vdash \Delta}\ \tilde{\lambda}$$

$$\frac{\mathcal{P} \le r\mathcal{P}_l + s\mathcal{P}_r \qquad \mathcal{Q} \le r\mathcal{Q}_l + s\mathcal{Q}_r \qquad \mathcal{P}_l\gamma \vdash A \mid \mathcal{Q}_l\delta \qquad \mathcal{P}_r\gamma \vdash B \mid \mathcal{Q}_r\delta}{\mathcal{P}\gamma \vdash rA \parr sB \mid \mathcal{Q}\delta}\ \langle -, - \rangle$$

$$\frac{\Gamma, rA, sB \vdash \Delta}{\Gamma \mid rA \parr sB \vdash \Delta}\ \mu\langle -, - \rangle$$

Figure 6.1: A fragment of a usage-annotated $\mu\tilde{\mu}$-calculus presented in traditional sequent notation

tation of this fragment of the $\mu\tilde{\mu}$-calculus in figure 6.1, with my encoding of the rules below.

```
data `MMT : Set where
  `cut `μ `μ∼ : (A : Ty) → `MMT
  `λ `λ∼ : (A : Ty) → `MMT
  `⟨-,-⟩ `μ⟨-,-⟩ : (rA sB : Ann × Ty) → `MMT

MMT : System
MMT = `MMT ▷ λ where
  (`cut A)    → ⟨ []ᶜ `⊢ trm A ⟩ `∗ ⟨ []ᶜ `⊢ cot A ⟩   `⇾ com
  (`μ A)      → ⟨ [ 1# , cot A ]ᶜ `⊢ com ⟩             `⇾ trm A
  (`μ∼ A)     → ⟨ [ 1# , trm A ]ᶜ `⊢ com ⟩             `⇾ cot A
  (`λ A)      → ⟨ []ᶜ `⊢ cot A ⟩                       `⇾ trm (A ^⊥)
  (`λ∼ A)     → ⟨ []ᶜ `⊢ trm A ⟩                       `⇾ cot (A ^⊥)
  (`⟨-,-⟩ rA@(r , A) sB@(s , B)) →
    r `· ⟨ []ᶜ `⊢ cot A ⟩ `∗ s `· ⟨ []ᶜ `⊢ cot B ⟩    `⇾ cot (rA ⅋ sB)
  (`μ⟨-,-⟩ rA@(r , A) sB@(s , B)) →
    ⟨ [ r , cot A ]ᶜ ++ᶜ [ s , cot B ]ᶜ `⊢ com ⟩      `⇾ trm (rA ⅋ sB)
```

6.3.3 Duplicability and L/nL

In section 4.4.2, I introduced a bunched connective $\square^{0+}$, used to ensure that such a premise is derived in a context that allows it to be duplicated and discarded. Syntax for this connective can be added to Premises and worked through the various parts of the framework, though I do not show these parts in the text of this thesis for the sake of simplicity and brevity. Additionally, variations such as $\square^{0}$ (requiring only that the context is ≤ 0) and $\square^{0+*}$ (requiring, in addition to $\square^{0+}$, closure under multiplication by any scalar) can be added to the framework independently of each other and of $\square^{0+}$.

As well as the use case of inductive types, as shown in section 4.4.2, I use the $\square^{0+*}$-modality to encode the restriction to intuitionistic variables in the intuitionistic judgement forms of the L/nL calculus [Benton, 1994]. I will explain this in detail in due course.

Linear/non-Linear Logic has two sorts of types: linear types A, B, C and intuitionistic types X, Y, Z. I encode these two sorts in the Agda types Ty lin and Ty int, respectively. Using an indexed type lets me take the total space of Ty, named ΣTy, as the evident dependent pair type. ΣTy becomes the type of types seen by the framework. The types themselves are, on the linear side, a tensor unit tI, a tensor product _t⊗_, and a linear function space _t⊸_; on the intuitionistic side, a unit t1, a product _t×_, and an intuitionistic function space _t→_; and adjoint type formers tF and tG. I also include a linear base type ι.

```
data Frag : Set where
  lin int : Frag

data Ty : Frag → Set where
  ι : Ty lin
  tI : Ty lin
  _t⊗_ : (A B : Ty lin) → Ty lin
  _t⊸_ : (A B : Ty lin) → Ty lin
  tF : Ty int → Ty lin
```

$$A, B, C ::= I \mid A \otimes B \mid A \multimap B \mid FX$$
$$X, Y, Z ::= 1 \mid X \times Y \mid X \to Y \mid GA$$
$$\mathcal{J} ::= \Theta; \Gamma \vdash_{\mathcal{L}} A \mid \Theta \vdash_{\mathcal{C}} X$$

$$\frac{}{\Theta; A \vdash_{\mathcal{L}} A}\ \mathcal{L}\text{-VAR} \qquad \frac{}{\Theta, X \vdash_{\mathcal{C}} X}\ \mathcal{C}\text{-VAR} \qquad \frac{}{\Theta; \cdot \vdash_{\mathcal{L}} I}\ I\text{I} \qquad \frac{\Theta; \Gamma \vdash_{\mathcal{L}} I \quad \Theta; \Delta \vdash_{\mathcal{L}} C}{\Theta; \Gamma, \Delta \vdash_{\mathcal{L}} C}\ I\text{E}$$

$$\frac{\Theta; \Gamma \vdash_{\mathcal{L}} A \quad \Theta; \Delta \vdash_{\mathcal{L}} B}{\Theta; \Gamma, \Delta \vdash_{\mathcal{L}} A \otimes B}\ \otimes\text{I} \qquad \frac{\Theta; \Gamma \vdash_{\mathcal{L}} A \otimes B \quad \Theta; \Delta, A, B \vdash_{\mathcal{L}} C}{\Theta; \Gamma, \Delta \vdash_{\mathcal{L}} C}\ \otimes\text{E}$$

$$\frac{\Theta; \Gamma, A \vdash_{\mathcal{L}} B}{\Theta; \Gamma \vdash_{\mathcal{L}} A \multimap B}\ \multimap\text{I} \qquad \frac{\Theta; \Gamma \vdash_{\mathcal{L}} A \multimap B \quad \Theta; \Delta \vdash_{\mathcal{L}} A}{\Theta; \Gamma, \Delta \vdash_{\mathcal{L}} B}\ \multimap\text{E} \qquad \frac{\Theta \vdash_{\mathcal{C}} X}{\Theta; \cdot \vdash_{\mathcal{L}} FX}\ F\text{I}$$

$$\frac{\Theta; \Gamma \vdash_{\mathcal{L}} FX \quad \Theta, X; \Delta \vdash_{\mathcal{L}} C}{\Theta; \Gamma, \Delta \vdash_{\mathcal{L}} C}\ F\text{E} \qquad \frac{}{\Theta \vdash_{\mathcal{C}} 1}\ 1\text{I} \qquad (\text{no } 1\text{E}) \qquad \frac{\Theta \vdash_{\mathcal{C}} X \quad \Theta \vdash_{\mathcal{C}} Y}{\Theta \vdash_{\mathcal{C}} X \times Y}\ \times\text{I}$$

$$\frac{\Theta \vdash_{\mathcal{C}} X_0 \times X_1}{\Theta \vdash_{\mathcal{C}} X_i}\ \times\text{E}_i \qquad \frac{\Theta, X \vdash_{\mathcal{C}} Y}{\Theta \vdash_{\mathcal{C}} X \to Y}\ \to\text{I} \qquad \frac{\Theta \vdash_{\mathcal{C}} X \to Y \quad \Theta \vdash_{\mathcal{C}} X}{\Theta \vdash_{\mathcal{C}} Y}\ \to\text{E}$$

$$\frac{\Theta; \cdot \vdash_{\mathcal{L}} A}{\Theta \vdash_{\mathcal{C}} GA}\ G\text{I} \qquad \frac{\Theta \vdash_{\mathcal{C}} GA}{\Theta; \cdot \vdash_{\mathcal{L}} A}\ G\text{E}$$

Figure 6.2: Linear/non-Linear Logic in traditional sequent notation

```
t1 : Ty int
_t×_ : (X Y : Ty int) → Ty int
_t→_ : (X Y : Ty int) → Ty int
tG : Ty lin → Ty int
```

As in the other examples, I define a data type of rule labels `'LnL` before defining the type system `LnL`. The system `LnL` has types `ΣTy` and usage annotations $\{0, 1, \omega\}$, as I have used before for linear systems. For reference, I give a standard presentation of the rules of L/nL in figure 6.2.

We want to be able to embed the L/nL judgement forms $\Theta; \Gamma \vdash_{\mathcal{L}} A$ and $\Theta \vdash_{\mathcal{C}} X$, where Θ is made of intuitionistic variables and Γ is made of linear variables. I take the subscript on $\vdash_{\mathcal{L}}$ and $\vdash_{\mathcal{C}}$ to be a redundant annotation of whether the conclusion type is `lin` or `int`, so it does not appear in the Agda code, but I retain it in all non-mechanised presentations.

I will generally ensure that all intuitionistic variables will all have usage annotation ω, while linear variables will each be annotated either 1 or 0. These assignments of usage annotations are enforced wherever the rules bind new variables. By usage subsumption, intuitionistic variables may also end up with annotation 1 or 0, but we see this as a (meaningless) refinement of the ω annotation, not causing a soundness problem. The key invariant is that linear variables never get annotation ω.

The key difference in form between linear and intuitionistic sequents is that intuitionistic sequents cannot contain linear variables. However, syntax descriptions give us no ability to directly state such a restriction on the kinds of variables. Therefore, any encoded syntax will allow us to propose ill formed sequents like $1A \vdash_{\mathcal{C}} X$, where a linear variable is to be used for an intuitionistic conclusion. To make sure that this is not a problem I use the $\Box^{0+*}$ bunched modality to guard many of the rules, making sure that variables with usage annotation 1 cannot occur in *provable* intuitionistic sequents. Specifically, every rule targeting an intuitionistic judgement has its premises wrapped in a $\Box^{0+*}$. Additionally, rules $F\text{I}$ and $G\text{E}$, which target linear judgements but have intuitionistic premises, similarly get a $\Box^{0+*}$. Note that having the annotation 0 means that well formed intuitionistic sequents can contain linear variables annotated 0, but these variables are not usable.

For ease of *producing* terms, we may prefer a "garbage in; garbage out" style with a minimal number of uses of $\Box^{0+*}$, which could be achieved in two different ways:

- Ensure that no sequent of the form $\Gamma, 1A \vdash_{\mathcal{C}} X$ is derivable. This leaves boxes only on rules 1-I and G-I — the two logical rules where an intuitionistic conclusion is derived without any intuitionistic premises.

- Ensure that any rule targeting a well formed conclusion has only well formed premises. This leaves boxes only on rules F-I and G-E — the two rules where a linear conclusion is derived from intuitionistic premises.

It would be interesting to show the equivalence of these two approaches with the one I have taken in this thesis, but that is left to future work.

The rules are presented in Agda code below, and also using bunched inference rule

$$\frac{I^*}{\vdash_{\mathcal{L}} I}\ I\text{I} \qquad \frac{\vdash_{\mathcal{L}} I \ * \ \vdash_{\mathcal{L}} C}{\vdash_{\mathcal{L}} C}\ I\text{E} \qquad \frac{\vdash_{\mathcal{L}} A \ * \ \vdash_{\mathcal{L}} B}{\vdash_{\mathcal{L}} A \otimes B}\ \otimes\text{I}$$

$$\frac{\vdash_{\mathcal{L}} A \otimes B \ * \ 1A, 1B \vdash_{\mathcal{L}} C}{\vdash_{\mathcal{L}} C}\ \otimes\text{E} \qquad \frac{1A \vdash_{\mathcal{L}} B}{\vdash_{\mathcal{L}} A \multimap B}\ \multimap\text{I} \qquad \frac{\vdash_{\mathcal{L}} A \multimap B \ * \ \vdash_{\mathcal{L}} A}{\vdash_{\mathcal{L}} B}\ \multimap\text{E}$$

$$\frac{\square(\vdash_{\mathcal{C}} X)}{\vdash_{\mathcal{L}} FX}\ F\text{I} \qquad \frac{\vdash_{\mathcal{L}} FX \ * \ \omega X \vdash_{\mathcal{L}} C}{\vdash_{\mathcal{L}} C}\ F\text{E} \qquad \frac{\square \mathrm{i}}{\vdash_{\mathcal{C}} 1}\ 1\text{I} \qquad (\text{no } 1\text{E})$$

$$\frac{\square(\vdash_{\mathcal{C}} X \ \dot{\times} \ \vdash_{\mathcal{C}} Y)}{\vdash_{\mathcal{C}} X \times Y}\ \times\text{I} \qquad \frac{\square(\vdash_{\mathcal{C}} X_0 \times X_1)}{\vdash_{\mathcal{C}} X_i}\ \times\text{E}_i \qquad \frac{\square(\omega X \vdash_{\mathcal{C}} Y)}{\vdash_{\mathcal{C}} X \to Y}\ \to\text{I}$$

$$\frac{\square(\vdash_{\mathcal{C}} X \to Y \ \dot{\times} \ \vdash_{\mathcal{C}} X)}{\vdash_{\mathcal{C}} Y}\ \to\text{E} \qquad \frac{\square(\vdash_{\mathcal{L}} A)}{\vdash_{\mathcal{C}} GA}\ G\text{I} \qquad \frac{\square(\vdash_{\mathcal{C}} GA)}{\vdash_{\mathcal{L}} A}\ G\text{E}$$

Figure 6.3: Linear/non-Linear Logic in bunched notation using $\{0, 1, \omega\}$ usage annotations and with $\square^{0+*}$ abbreviated to $\square$

notation in figure 6.3.

```
data 'LnL : Set where
  'Ii : 'LnL
  'Ie : (C : Ty lin) → 'LnL
  '⊗i : (A B : Ty lin) → 'LnL
  '⊗e : (A B C : Ty lin) → 'LnL
  '—∘i '—∘e : (A B : Ty lin) → 'LnL
  'Fi : (X : Ty int) → 'LnL
  'Fe : (X : Ty int) (C : Ty lin) → 'LnL

  '1i : 'LnL
  '×i : (X Y : Ty int) → 'LnL
  '×e : (i : Hand) (X Y : Ty int) → 'LnL
  '→i '→e : (X Y : Ty int) → 'LnL
  'Gi 'Ge : (A : Ty lin) → 'LnL

LnL : System
LnL = 'LnL ▷ λ where
```

$$
\begin{array}{lll}
\text{`li} & \to \text{`}I^* & \text{`}\multimap \mathsf{lin} \mathbin{,} \mathsf{tl} \\
(\text{`le } C) & \to \langle\, []^c \text{ `}\vdash \mathsf{lin} \mathbin{,} \mathsf{tl} \,\rangle \text{ `}* \langle\, []^c \text{ `}\vdash \mathsf{lin} \mathbin{,} C \,\rangle & \text{`}\multimap \mathsf{lin} \mathbin{,} C \\
(\text{`}\otimes\mathsf{i}\ A\ B) & \to \langle\, []^c \text{ `}\vdash \mathsf{lin} \mathbin{,} A \,\rangle \text{ `}* \langle\, []^c \text{ `}\vdash \mathsf{lin} \mathbin{,} B \,\rangle & \text{`}\multimap \mathsf{lin} \mathbin{,} A\ \mathsf{t}\otimes\ B \\
(\text{`}\otimes\mathsf{e}\ A\ B\ C) & \to & \\
\quad \langle\, []^c \text{ `}\vdash \mathsf{lin} \mathbin{,} A\ \mathsf{t}\otimes\ B \,\rangle \text{ `}* \langle\, [\, \mathsf{u1} \mathbin{,} \mathsf{lin} \mathbin{,} A \,]^c +\!\!+^c [\, \mathsf{u1} \mathbin{,} \mathsf{lin} \mathbin{,} B \,]^c \text{ `}\vdash \mathsf{lin} \mathbin{,} C \,\rangle & & \\
\quad & & \text{`}\multimap \mathsf{lin} \mathbin{,} C \\
(\text{`}\multimap\mathsf{i}\ A\ B) & \to \langle\, [\, \mathsf{u1} \mathbin{,} \mathsf{lin} \mathbin{,} A \,]^c \text{ `}\vdash \mathsf{lin} \mathbin{,} B \,\rangle & \text{`}\multimap \mathsf{lin} \mathbin{,} A\ \mathsf{t}\multimap\ B \\
(\text{`}\multimap\mathsf{e}\ A\ B) & \to & \\
\quad \langle\, []^c \text{ `}\vdash \mathsf{lin} \mathbin{,} A\ \mathsf{t}\multimap\ B \,\rangle \text{ `}* \langle\, []^c \text{ `}\vdash \mathsf{lin} \mathbin{,} A \,\rangle & & \text{`}\multimap \mathsf{lin} \mathbin{,} B \\
(\text{`}\mathsf{Fi}\ X) & \to \text{`}\square^{\mathsf{o}+*} \langle\, []^c \text{ `}\vdash \mathsf{int} \mathbin{,} X \,\rangle & \text{`}\multimap \mathsf{lin} \mathbin{,} \mathsf{tF}\ X \\
(\text{`}\mathsf{Fe}\ X\ C) & \to & \\
\quad \langle\, []^c \text{ `}\vdash \mathsf{lin} \mathbin{,} \mathsf{tF}\ X \,\rangle \text{ `}* \langle\, [\, \mathsf{u}\omega \mathbin{,} \mathsf{int} \mathbin{,} X \,]^c \text{ `}\vdash \mathsf{lin} \mathbin{,} C \,\rangle & & \text{`}\multimap \mathsf{lin} \mathbin{,} C \\
\text{`}\mathsf{1i} & \to \text{`}\square^{\mathsf{o}+*} \text{ `}\mathsf{i} & \text{`}\multimap \mathsf{int} \mathbin{,} \mathsf{t1} \\
(\text{`}\times\mathsf{i}\ X\ Y) & \to & \\
\quad \text{`}\square^{\mathsf{o}+*} (\langle\, []^c \text{ `}\vdash \mathsf{int} \mathbin{,} X \,\rangle \text{ `}\dot{\times} \langle\, []^c \text{ `}\vdash \mathsf{int} \mathbin{,} Y \,\rangle) & & \text{`}\multimap \mathsf{int} \mathbin{,} X\ \mathsf{t}\times\ Y \\
(\text{`}\times\mathsf{e}\ i\ X\ Y) & \to \text{`}\square^{\mathsf{o}+*} \langle\, []^c \text{ `}\vdash \mathsf{int} \mathbin{,} X\ \mathsf{t}\times\ Y \,\rangle & \text{`}\multimap \mathsf{int} \mathbin{,} [\, X > i < Y \,] \\
(\text{`}\to\mathsf{i}\ X\ Y) & \to \text{`}\square^{\mathsf{o}+*} \langle\, [\, \mathsf{u}\omega \mathbin{,} \mathsf{int} \mathbin{,} X \,]^c \text{ `}\vdash \mathsf{int} \mathbin{,} Y \,\rangle & \text{`}\multimap \mathsf{int} \mathbin{,} X\ \mathsf{t}\to\ Y \\
(\text{`}\to\mathsf{e}\ X\ Y) & \to & \\
\quad \text{`}\square^{\mathsf{o}+*} (\langle\, []^c \text{ `}\vdash \mathsf{int} \mathbin{,} X\ \mathsf{t}\to\ Y \,\rangle \text{ `}\dot{\times} \langle\, []^c \text{ `}\vdash \mathsf{int} \mathbin{,} X \,\rangle) & & \text{`}\multimap \mathsf{int} \mathbin{,} Y \\
(\text{`}\mathsf{Gi}\ A) & \to \text{`}\square^{\mathsf{o}+*} \langle\, []^c \text{ `}\vdash \mathsf{lin} \mathbin{,} A \,\rangle & \text{`}\multimap \mathsf{int} \mathbin{,} \mathsf{tG}\ A \\
(\text{`}\mathsf{Ge}\ A) & \to \text{`}\square^{\mathsf{o}+*} \langle\, []^c \text{ `}\vdash \mathsf{int} \mathbin{,} \mathsf{tG}\ A \,\rangle & \text{`}\multimap \mathsf{lin} \mathbin{,} A
\end{array}
$$

In section 8.4, I will give translations between the original L/nL system, this encoding of L/nL, and $\lambda\mathcal{R}$, providing some assurance that the correct logic has been encoded.

6.4 Conclusion

I have shown via examples that the syntax descriptions defined in this chapter can capture a significant range of simply typed usage-aware calculi. Together with the examples of posemirings given in chapter 4, the whole framework is capable of encoding

many specific substructural calculi. However, the language of descriptions has several restrictions, with more and less clear practical consequences. Many of these restrictions are shared with the syntax descriptions of Allais et al. [2021], but some pertain to usage annotations.

One clear limitation is that the forms of judgements do not allow enough interdependency to capture dependently typed calculi. In particular, for any given calculus, there is one fixed set of types, with no indexing over contexts. This allows contexts to have a flat list-like structure rather than a telescope, and means that the type of a term does not depend on that context, but clearly restricts the expressiveness of the calculi which can be encoded. As well as dependently typed calculi, this also prevents the encoding of the well formed types of System F_ω.

Less clear than describability of System F_ω is describability of System F. Though I have not presented the details, I believe it should be possible to encode calculi with polymorphism, such as System F and the usage-annotated system found in Abel and Bernardy [2020], via a two-stage process. First, we make a syntax description for the language of types, and then use the well typed terms arising from that description as the set of types when making the description for the System F terms. Such an encoding would yield all of the benefits of the framework presented in this thesis separately to the language of types and the language of terms, but how these two languages interact is unclear. For example, in chapter 7, I provide a syntax-generic substitution operation. This operation would yield substitution for type variables in types and substitution for term variables in terms, but substitution for type variables in terms would have to be provided separately, and would likely be more complicated. It would be interesting to see future work in this direction, possibly inspired by how Autosubst [Schäfer et al., 2015] handles the same issue.

As for restrictions peculiar to the usage-aware setting, we of course have the general limitations of posemiring annotations listed in section 4.6, which I will not repeat here. Additionally, there are things we could wish to do with posemiring-based usage annotations which we do not have access to with the descriptions given in this chapter. For example, an issue one may have with the encoding of Linear/non-Linear

Logic given in section 6.3.3 is that linear variables and intuitionistic variables, despite having morally distinct usage disciplines, necessarily share the $\{0, 1, \omega\}$ posemiring of usage annotations. If one wanted to generalise this construction to different pairs of usage disciplines, one may well find that they do not combine adequately into a single posemiring. A possible solution would be to have a range of kinds, and for each kind to have its own set of types and its own posemiring — for example, a *linear* kind and an *intuitionistic* kind, using $\{0, 1, \omega\}$ and the 1-element posemiring, respectively. However, the descriptions given in this thesis only allow for one posemiring.

Finally, while there is a simple and well motivated core of premise connectives — $\dot{1}$, $\dot{\times}$, I^*, $*$, and $r \cdot (-)$ — the various $\Box$-modalities feel like ad hoc additions. While I have shown that $\Box^{0+}$ and $\Box^{0+*}$ have several practical uses, it is unclear whether there should be more modalities. Furthermore, I have not made an effort to isolate the properties that make these modalities well behaved. We will see in section 7.2 that any such modality will need to be functorial in the appropriate sense, but other examples of generic programming, such as the usage elaborator which will be detailed in section 8.1, may need further properties.

Chapter 7

Generic usage-aware semantics

Having fixed a universe of syntaxes, in which we can build terms, the next thing to do is to write recursive functions on terms. With terms being given by an inductive data type definition, they already come with a recursion and an induction principle. However, these principles do not handle variable-binding, which we have seen with the fact that we had to write the bind helper function for renaming and substitution in section 5.3.

In this chapter, the central construct is a function semantics which, for a $\mathcal{V}$-environment $\rho : \Gamma \xRightarrow{\mathcal{V}} \Delta$, maps a term $M : \Delta \vdash A$ to some semantic value in the type $\mathcal{C}\,\Gamma\,A$. This is a direct adaptation of the semantics function of section 2.4, which has the same kind of action on intuitionistic terms, given similar operations on $\mathcal{V}$ and $\mathcal{C}$ as what we had earlier. The semantics function recurses on the term M, updating ρ whenever new variables are bound. In our usage-aware case, ρ is also updated whenever we come across linear combinations induced by premise connectives I^*, $*$, and $r \cdot$.

This chapter is structured as follows. I start by giving a quick introduction to linear relations — a generalisation of linear maps — with reference to their use in mechanised algebraic reasoning, in section 7.1. Using linear relations, I give a functorial *map* operation to a single layer of syntax in section 7.2. I then adapt the Kripke function space to the usage-aware setting in section 7.3. Then I apply the Kripke function space, along with much of the machinery I have introduced in previous chapters, to give the semantics function in section 7.4. Finally, I show that the Kripke function space simplifies

under certain conditions in section 7.5, and I use that case to give the syntax-generic simultaneous renaming and substitution operations in section 7.6.

7.1 Linear relations in Agda

In section 5.1, I defined *usage-annotated environments* (definition 5.1.7). One component of a usage-annotated environment is a linear map Ψ which, when applied to the target usage vector, gives a vector compatible with the source usage vector.

When it comes to mechanisation, I prefer to replace an assertion "$\mathcal{P} \leq \mathcal{Q}\Psi$", involving a linear map Ψ, by an assertion "$\mathcal{P}\Psi\mathcal{Q}$", where Ψ is now a linear *relation* said to relate $\mathcal{P}$ and $\mathcal{Q}$. I define linear relations as follows, where the reader may wish to check that a linear map gives rise to a linear relation via the expression $\mathcal{P} \leq \mathcal{Q}\Psi$.

Definition 7.1.1. Given a posemiring $\mathscr{R}$ and modules $\mathscr{M}$ and $\mathscr{N}$ over $\mathscr{R}$, a *linear relation* between $\mathscr{M}$ and $\mathscr{N}$ is a relation Ψ between the underlying sets of $\mathscr{M}$ and $\mathscr{N}$ such that the following properties hold of all $\mathcal{P}, \mathcal{P}', \mathcal{P}_l, \mathcal{P}_r \in \mathscr{M}$ and all $\mathcal{Q}, \mathcal{Q}', \mathcal{Q}_l, \mathcal{Q}_r \in \mathscr{N}$.

$$\mathcal{P}' \leq \mathcal{P} \ \wedge \ \mathcal{P}\Psi\mathcal{Q} \ \wedge \ \mathcal{Q} \leq \mathcal{Q}' \implies \mathcal{P}'\Psi\mathcal{Q}'$$

$$(\exists \mathcal{Q}.\ \mathcal{P}\Psi\mathcal{Q} \ \wedge \ \mathcal{Q} \leq 0) \implies \mathcal{P} \leq 0$$

$$(\exists \mathcal{Q}.\ \mathcal{P}\Psi\mathcal{Q} \ \wedge \ \mathcal{Q} \leq \mathcal{Q}_l + \mathcal{Q}_r) \implies (\exists \mathcal{P}_l, \mathcal{P}_r.\ \mathcal{P} \leq \mathcal{P}_l + \mathcal{P}_r \ \wedge \ \mathcal{P}_l\Psi\mathcal{Q}_l \ \wedge \ \mathcal{P}_r\Psi\mathcal{Q}_r)$$

$$\big(\exists \mathcal{Q}.\ \mathcal{P}\Psi\mathcal{Q} \ \wedge \ \mathcal{Q} \leq r\mathcal{Q}'\big) \implies \big(\exists \mathcal{P}'.\ \mathcal{P} \leq r\mathcal{P}' \ \wedge \ \mathcal{P}'\Psi\mathcal{Q}'\big)$$

I write $\mathscr{M} \nrightarrow \mathscr{N}$ as the type of linear relations between $\mathscr{M}$ and $\mathscr{N}$.

Relations have several advantages over functions when doing mechanised algebra in type theory. For one, what are compound expressions in functional style — for example $x \leq f(y) + g(z)$ — become collections of simple relationships in relational style — for example $\exists v, w.\ vfy \ \wedge \ wgz \ \wedge \ \mathrm{Add}\, x\, v\, w$. The advantage of this is that we have immediate access to all of the expressions and subexpressions, and the proofs of the relationships between them. This means that there is no need for congruence or monotonicity lemmas, and correspondingly no need to explicitly describe the syntactic

context in which some algebraic manipulation is being applied and we rely less on the unifier. Another advantage is that one can design relations so that pattern-matching suggestively decomposes complex relationships. For example, given $F : \mathscr{M} \nrightarrow \mathscr{M}'$ and $G : \mathscr{N} \nrightarrow \mathscr{N}'$, we can define a relation $F \oplus G : \mathscr{M} \oplus \mathscr{N} \nrightarrow \mathscr{M}' \oplus \mathscr{N}'$ pointwise, so that a proof of $(x, x')(F \oplus G)(y, y')$ is a proof of xFy together with a proof of $x'Gy'$. Pattern-matching on such a proof immediately gives us these constituent parts, whereas proofs of the corresponding statement involving functions would require using a lemma to get the parts. There is a dual advantage when producing one of these proofs, where we can introduce the canonical constructor (for pairs, in this example) rather than having to find the appropriate lemma.

Relations also have several disadvantages, though I have found that for my use case, these are outweighed by the advantages. For example, automated algebraic solvers are better developed for function-based algebraic expressions, and sometimes the fact that functions satisfy unitality and associativity up to decidable judgemental equality means that some proofs can be avoided. The handling of compound expressions can also be a disadvantage in that it necessitates lots of new variable names and obscures goal and context displays. Finally, in predicative systems such as Agda, relations typically live in a larger universe than the corresponding functions. In practice, this means quantifying over an extra level variable for each relation involved in general lemmas.

There are more relations than there are functions, so statements involving relations are more general than the corresponding statements involving functions. However, one part of the development requires functions rather than relations, so I impose functionality on relations after the fact. The appropriate notion of functional relation I use is slightly different to the standard one, in that I take account of the order on the codomain, and thus ask for the *largest* solution rather than the *unique* solution.

Definition 7.1.2. A linear relation Ψ between $\mathscr{M}$ and $\mathscr{N}$ is *(right-to-left) functional* if, for every $\mathcal{Q} \in \mathscr{N}$, there exists universally a $\mathcal{P} \in \mathscr{M}$ such that $\mathcal{P}\Psi\mathcal{Q}$. Universality means that, for all $\mathcal{P}'$ such that $\mathcal{P}'\Psi\mathcal{Q}$, we have $\mathcal{P}' \leq \mathcal{P}$ (i.e. $\mathcal{P}$ is the largest solution).

In Agda code, Ψ becomes Ψ and the fact that Ψ relates $\mathcal{P}$ and $\mathcal{Q}$ (in this section written $\mathcal{P}\Psi\mathcal{Q}$) is rendered as Ψ .rel P Q. That Ψ respects the orders on its arguments

is given by Ψ .rel-$\leq_m$, and the various linearity properties are given by Ψ .rel-0_m, Ψ .rel-$+_m$, and Ψ .rel-$*_m$.

7.2 A layer of syntax is functorial

A basic property of the universe of syntaxes is that every syntax supports a functorial action on subterms, realised by a function map-s. Its type says that to map a function f over a layer of syntax, there must be a linear map Ψ relating the domain and codomain usage contexts, and f should be usable wherever the domain and codomain usage contexts are similarly related by Ψ.

```
map-s : (s : System) →
  (∀ {Θ P′ Q′} → Ψ .rel P′ Q′ → X Θ (ctx Q′ δ) ⊸ Y Θ (ctx P′ γ)) →
  (∀ {P Q} → Ψ .rel P Q → ⟦ s ⟧s X (ctx Q δ) ⊸ ⟦ s ⟧s Y (ctx P γ))
```

This generality is needed because usage contexts change between a term and its immediate subterms—they are decomposed according to the bunched connectives used in the rules. X and Y are ExtOpenFams, with Θ being the context extension for a subterm (i.e., the variables newly bound in that subterm). Unlike usage annotations, types in the contexts γ and δ, and the conclusion types implicit here, are preserved throughout. This is the essence of the usage annotation based approach—we use traditional techniques for variable binding, with the usage annotations layered on top.

The heart of map-s is map-p, which recursively works through the structure ps of premises of the rule applied, acting on each subterm it finds. Here, particularly in the clauses for $`*$ and $`\cdot$, we see why it is not enough for the function on subterms to apply at usage contexts P and Q — rather, it also needs to apply at any similarly related P' and Q'. In the case of $`*$, we have that $\mathcal{P} \leq \mathcal{P}_M + \mathcal{P}_N$, with M and N being collections of subterms in usage contexts $\mathcal{P}_M$ and $\mathcal{P}_N$, respectively. Linearity of Ψ yields $\mathcal{Q}_M$ and $\mathcal{Q}_N$ such that $\mathcal{Q} \leq \mathcal{Q}_M + \mathcal{Q}_N$ and we use map-p recursively at $(\mathcal{P}_M, \mathcal{Q}_M)$ and $(\mathcal{P}_N, \mathcal{Q}_N)$ on M and N. The cases for $`\cdot$ and $`I^*$ are similar, each using a different aspect of linearity. In contrast, the cases for $`\dot{\mathsf{I}}$ and $`\dot{\times}$, which are the only constructors used in fully structural systems, do not involve any changes in the usage contexts.

The linearity of relation Ψ is given by fields rel-0_m, rel-$+_m$, and rel-$*_m$ (with the subscript-m being a mnemonic for *module*, as opposed to scalar).

```
map-p : (ps : Premises) →
  (∀ {Θ P′ Q′} → Ψ .rel P′ Q′ → X Θ (ctx Q′ δ) ⇾ Y Θ (ctx P′ γ)) →
  (∀ {P Q} → Ψ .rel P Q → ⟦ ps ⟧p X (ctx Q δ) → ⟦ ps ⟧p Y (ctx P γ))
map-p ⟨ Γ ⊢ A ⟩ f r M                 = f r M
map-p ⌜i f r _                        = _
map-p (ps ⌜× qs) f r (M , N)          = map-p ps f r M , map-p qs f r N
map-p ⌜I* f r I*c⟨ sp0 ⟩               = I*c⟨ Ψ .rel-0m (r , sp0) ⟩
map-p (ps ⌜* qs) f r (M *c⟨ sp+ ⟩ N)   =
  let rM ↘, sp+′ ,↙ rN = Ψ .rel-+m (r , sp+) in
  map-p ps f rM M *c⟨ sp+′ ⟩ map-p qs f rN N
map-p (p ⌜· ps) f r (⟨ sp* ⟩.c M)      =
  let r′ , sp*′ = Ψ .rel-*m (r , sp*) in
  ⟨ sp*′ ⟩.c map-p ps f r′ M
```

I have also extended map-p to handle the various □-modalities described in section 6.3.3. The Agda code for this extension is not particularly readable, so I do not include it in this document. However, this extension is notable as the only part of the framework requiring that the linear relation Ψ be functional (i.e., total and deterministic).

7.3 The Kripke function space

At this point I introduce a minor generalisation to OpenFam and ExtOpenFam (as defined in section 6.2): I —OpenFam and I —ExtOpenFam. We obtain the definition of I —OpenFam by replacing the textual occurrence of Ty by the parameter I, though there is still reference to the ambient Ty via Ctx. The main value I am interested in I taking, other than Ty, is Ctx — for example, the type family of $\mathcal{V}$-environments, for a given $\mathcal{V}$, is a Ctx —OpenFam _. I use this generality in the type of extend in the next section.

```
_—OpenFam_ : ∀ {i} → Set i → ∀ ℓ → Set (i ⊔ suc ℓ)
I —OpenFam ℓ = Ctx → I → Set ℓ

_—ExtOpenFam_ : ∀ {i} → Set i → ∀ ℓ → Set (i ⊔ suc ℓ)
I —ExtOpenFam ℓ = Ctx → I —OpenFam ℓ
```

The definition Kripke $\mathcal{V}$ $\mathcal{C}$ Δ is a kind of function space that describes a $\mathcal{C}$-value parametrised by Δ-many additional $\mathcal{V}$-values (all correctly typed and usage-annotated). It is used to describe how to go under binders in a Higher-Order Abstract Syntax style: To go under a binder we must provide semantic interpretations for all the additional variables.

```
Kripke : (𝒱 : OpenFam v) (𝒞 : I —OpenFam c) → I —ExtOpenFam _
Kripke = Wrap λ 𝒱 𝒞 Δ Γ A → □ʳ ([ 𝒱 ]_⇒ᵉ Δ ⊸ [ 𝒞 ]_⊨ A) Γ
```

Wrap is a device that turns any type family into an equivalent type family that is definitionally injective in its indices, which helps with Agda's type inference. It turns the type family into a parametrised record with a single field get whose type is the type in the body of the λ-abstraction. For understanding the meaning of Kripke, Wrap can be ignored.

If Δ is of the form $s_1 B_1, \ldots, s_n B_n$, then Kripke $\mathcal{V}$ $\mathcal{C}$ Δ Γ A is equivalent to $\Box^r$ ($s_1 \cdot$ [$\mathcal{V}$]$\_\vDash B_1 \mathrel{-\!\!*} \cdots \mathrel{-\!\!*} s_n \cdot$ [$\mathcal{V}$]$\_\vDash B_n \mathrel{-\!\!*}$ [$\mathcal{C}$]$\_\vDash A$) Γ by Currying. That is to say, the Kripke function is expecting a value for each newly bound variable, at the multiplicity of its annotation, together with the resources supporting each of those values. We use the "magic wand" function space here to enforce the invariant that the freshly bound variables have usage annotations that are added to the existing variables, not shared with them. The use of the $\Box^r$ modality ensures that we can still use it in the presence of additional variables introduced by weakening.

Kripke is functorial in the $\mathcal{C}$ argument, as witnessed by the mapK$\mathcal{C}$ function, which is essentially post-composition:

```
mapK𝒞 :  ∀ {A B} → [ 𝒞 ]_⊨ A ⊸ [ 𝒞′ ]_⊨ B →
         ∀ {Δ Γ} → Kripke 𝒱 𝒞 Δ Γ A → Kripke 𝒱 𝒞′ Δ Γ B
mapK𝒞 f b .get ren .app* sp ρ = f (b .get ren .app* sp ρ)
```

7.4 Semantic traversal

We can now state the data required to implement a traversal assigning semantics to terms. For open families $\mathcal{V}$ and $\mathcal{C}$, interpreting variables and terms respectively, we assume that $\mathcal{V}$ is renameable (i.e., that $\models^{\mathcal{V}} A \twoheadrightarrow \Box\left(\models^{\mathcal{V}} A\right)$ for all A), that $\mathcal{V}$ is embeddable in $\mathcal{C}$, and that we have an algebra for a layer of syntax, where bound variables are handled using the Kripke function space:

```
record  Semantics (d : System) (𝒱 : OpenFam v) (𝒞 : OpenFam c)
        : Set (suc 0ℓ ⊔ v ⊔ c) where
  field
    ren^𝒱   : ∀ {A} → Renameable ([ 𝒱 ]_⊨ A)
    ⟦var⟧   : 𝒱                    ↠ 𝒞
    ⟦con⟧   : ⟦ d ⟧s (Kripke 𝒱 𝒞)  ↠ 𝒞
```

We mutually define the action semantics and its lemma body. The purpose of semantics is to turn a term into a $\mathcal{C}$-value using a $\mathcal{V}$-environment and the fields of Semantics. Meanwhile, body does a similar job, but also deals with newly bound variables. In particular, body takes a term in a context extended by Θ, and produces a Kripke function from $\mathcal{V}$-values for Θ to $\mathcal{C}$-values.

```
semantics : ∀ {Γ Δ} → [ 𝒱 ] Γ ⇒ᵉ Δ → ∀ {sz} →
  [ d , sz ] Δ ⊢_ ↠ [ 𝒞 ] Γ ⊨_
body : ∀ {Γ Δ} → [ 𝒱 ] Γ ⇒ᵉ Δ → ∀ {sz Θ} →
  Scope [ d , sz ]_⊢_ Θ Δ ↠ Kripke 𝒱 𝒞 Θ Γ
```

To implement the new recursor semantics, we use the standard recursor, which in one case gives us a variable v, and in the other gives us a structure of subterms M, each of which is in an extended context. To deal with a variable v, we look it up in the environment ρ, then use the ⟦var⟧ field to map the resulting $\mathcal{V}$-value to a $\mathcal{C}$-value. To deal with a structure of subterms M, we use the functoriality of the syntactic structure to consider each subterm separately. On a subterm, we apply body, which amounts to a recursive call to semantics with an extended environment. Recall that relocate (lemma 5.2.1) adjusts the environment ρ to work in the usage contexts of the subterms.

```
semantics ρ ('var v) = ⟦var⟧ $ ρ .lookup (ρ .fit-here) v
semantics ρ ('con M) = ⟦con⟧ $
  map-s (ρ .Ψ) d (λ r → body (relocate ρ r)) (ρ .fit-here) M
```

For body, we are given a subterm M, to which we want to apply semantics. To do so, we need an extended version of the initial environment ρ. We express this as the generation of a Kripke function that produces the extended environment given interpretations of the fresh variables. We take ρ, which is an environment covering Δ, and σ, which is an environment covering Θ, and glue them together using the inductive rules for generating environments, after having renamed ρ via corollary 5.2.11 to make it fit the new context Γ^+ (intended to be Γ $++^c$ Θ):

```
extend : ∀ {Γ Δ Θ} →
  [ V ] Γ ⇒ᵉ Δ → Kripke V ([ V ]_⇒ᵉ_) Θ Γ (Δ ++ᶜ Θ)
extend ρ .get ren .app* sp σ = ++ᵉ (ren^Env ren^V ρ ren *⟨ sp ⟩ σ)
```

To define body, we use mapK$\mathcal{C}$ to post-compose the environment extension by the λ-function taking an extended environment and acting with it on M.

```
body ρ M = mapK𝒞 (λ σ → semantics σ M) (extend ρ)
```

semantics is the fundamental lemma of the framework. With it proven, I move onto corollaries and specific applications.

7.5 Reifying the Kripke function space

A result I will use throughout the rest of this thesis is *reification*. When we have an index-preserving mapping from usage-checked variables to $\mathcal{V}$-environments, we can construct environments of the form $\Gamma \overset{\mathcal{V}}{\Longrightarrow} \Gamma$ (identity environments) for all Γ. This lets us write the reify function, which simplifies our obligations in giving a Semantics by coercing Kripke functions into just $\mathcal{C}$-values in an extended context.

Lemma 7.5.1 (reify)**.** If $\mathcal{V}$ is an open family such that there is a function $v : \exists \rightharpoonup \mathcal{V}$, we get a function of type Kripke $\mathcal{V}\,\mathcal{C} \rightharpoonup$ Scope $\mathcal{C}$ for any $\mathcal{C}$.

Proof. Let $b : \mathrm{Kripke}\, \mathcal{V}\, \mathcal{C}\, \Delta\, \Gamma\, A$. That is, b is a Kripke function yielding $\mathcal{C}$-computations We want to apply b so as to get a $\mathcal{C}\,(\Gamma, \Delta)\,A$. Let $\mathcal{P}\gamma = \Gamma$ and $\mathcal{Q}\delta = \Delta$. The $\Box^r$ in the type of b allows us to reverse-rename Γ to $\Gamma, 0\delta$. Then we give the $-\!\!*$-function an argument in context $0\gamma, \Delta$, noting that $(\Gamma, 0\delta) + (0\gamma, \Delta) = (\Gamma, \Delta)$, as we wanted from the result. The argument needs type $0\gamma, \Delta \overset{\mathcal{V}}{\Longrightarrow} \Delta$. We produce this via corollary 5.2.10 from an environment $\rho : 0\gamma, \Delta \overset{\mathcal{V}}{\Longrightarrow} 0\gamma, \Delta$ created using v and a renaming which is the complement to that used on $\Box^r$. □

All of the $\mathcal{V}$s used in examples in this paper support identity environments. However, Allais et al. [2021, p. 27] give some important examples that do not support identity environments, and thus cannot use reify (lemma 7.5.1). The feature that causes the lack of support for identity environments is that a semantics can make use of the fact that only variables of particular kinds are bound by the syntax. In the examples of Allais et al., a bidirectionally typed language only binds variables that synthesise their type, as opposed to those whose type is checked. The semantics of type-checking and elaboration rely on variables synthesising their type, so $\mathcal{V}$ is chosen to cover only those variables. Instead of using reify, we must observe that each syntactic form only binds such synthesising variables. Similar phenomena would appear in, say, a call-by-value language where variables are values (not computations), or a polarised language where variables always have a polarity matching their type.

7.6 Renaming and substitution

The final completely syntax-generic result I present is simultaneous substitution. I derive this as I did in the simply typed case in section 2.4: I first show that a syntactic kit can be turned into a semantics, and then by instantiating the notion of kit for, in turn, renaming and substitution, the general semantic traversal gives the result we want.

The notion of Kit is essentially the same as in the simply typed case, once we allow for changes to the basic definitions of variables, terms, and environments (in particular, renamings).

```
record Kit (d : System) (𝒱 : OpenFam v) : Set (suc 0ℓ ⊔ v) where
  field
    ren^𝒱  : ∀ {A} → Renameable ([ 𝒱 ]_⊨ A)
    vr     : _∃_   → 𝒱
    tm     : 𝒱     → [ d , ∞ ]_⊢_
```

The first two fields of Semantics derive directly from fields of Kit. Meanwhile, to handle term constructors, we first reify to get a collection of traversed subterms, and then use ‘con to assemble these subterms into a similarly shaped syntactic form as we started with. The vr field is used implicitly in reify, as it is used to show that $\mathcal{V}$-identity environments exist.

```
kit→sem : Kit d 𝒱 → Semantics d 𝒱 [ d , ∞ ]_⊢_
kit→sem          K .ren^𝒱   = K .ren^𝒱
kit→sem          K .⟦var⟧   = K .tm
kit→sem {d = d}  K .⟦con⟧   = ‘con ∘ map-s′ d reify
  where open Kit K using (identityEnv)
```

The action of a syntactic traversal on logical rules is basically fixed: we preserve the logical rule and extend the environment with any newly bound variables according to vr. Meanwhile, the action on variables is relatively unconstrained: we look up the variable in the environment to get a $\mathcal{V}$-value, then transform that $\mathcal{V}$-value into a term using tm.

The idea of simultaneous renaming is that variables replace variables, whereas with simultaneous substitution, terms replace variables. This translates to environments for renaming containing $\exists$-values (variables), and environments for substitution containing $\vdash$-values (terms).

```
Ren-Kit : Kit d _∃_
Ren-Kit = record { ren^𝒱 = ren^∃ ; vr = id ; tm = ‘var }
```

Notice that ren^⊢, witnessing the fact that terms are renameable, is a corollary of Ren-Kit.

```
Sub-Kit : Kit d [ d , ∞ ]_⊢_
Sub-Kit = record { ren^𝒱 = ren^⊢ ; vr = `var ; tm = id }
```

7.7 Conclusion

In this chapter, I have completed my generalisation of the core of the work of Allais et al. [2021]. The key decision to be made in this generalisation process was that the Kripke function spaces would be based on the $-\!\!*$ bunched connective. That this change works justifies the idea of a deep connection between linear syntaxes and bunched-logic-style handling of contexts.

The main pay-off we have seen so far from this and the previous chapter is that all syntaxes we can describe respect substitution. However, in the following chapter, I give more specific worked examples using the generic semantic traversal and operations derived from it.

Chapter 8

Applications

In this chapter, I provide four example uses of semantic traversals: a usage elaborator, a normalisation by evaluation algorithm, a denotational semantics, and translations between calculi. The reader is also encouraged to see the far greater range of examples in the work of Allais et al. [2021], which should adapt to our usage-annotated setting. The usage elaborator (section 8.1) gives an example of a program that is generic in both syntax and usage annotations, as well as being an essential tool for practical use of the framework. The normalisation by evaluation algorithm (section 8.2) is for a specific syntax, but exercises parts of the generic semantic traversal — particularly, its handling of environments — to good effect. The denotational semantics in terms of world-indexed relations (section 8.3) presents an interesting semantics for semiring-annotated type systems, but with minimal use of operations provided by the framework. Finally, the translations between $\lambda\mathcal{R}$ and L/nL (section 8.4) both justify the encoding of L/nL given in section 6.3.3 and give examples of syntactic work benefiting from the generic renaming and substitution operations.

8.1 A usage elaborator

Using the constructs we have seen so far, producing example terms soon becomes extremely tedious. We can achieve some abbreviation by using pattern synonyms to wrap around ‘con expressions, but we still have to produce essentially bespoke proofs when-

ever we use a usage-sensitive part of the syntax. The size of each of these proofs is roughly proportional to the number of free variables, so the amount of proof we have to write grows roughly quadratically with the size of terms. An additional factor, which we can't see on paper but is nonetheless significant, is that type checking time for these proofs soon becomes prohibitive to interactive development.

Our aim in this section is to automate usage constraint proofs, making terms both easier to write and more performant to check. We invoke the automation by writing terms in a syntax where usage constraints have been trivialised, and then use a semantic traversal over the simplified syntax to try to produce a fully elaborated term in the original syntax. We write the automation in a way that is generic in the syntax description, thus avoiding repetition and facilitating the prototyping of new type systems.

The type of syntax descriptions depends on the type of usage annotations because of variable binding. For example, in the $!r$-E rule of figure 4.4, the right premise binds a new variable with annotation r, where r is drawn from the ambient posemiring. The scaling connective also makes direct reference to the posemiring. To produce a simplified syntax description, where usage constraints are trivialised, we set the ambient posemiring to the 1-element **0** posemiring. In contrast to syntax descriptions, even though types can contain usage annotations, the type of types does not depend on the type of usage annotations. This means that, in our simplified syntax, terms have types from the original system even though variables have trivial usage annotations. We define the **0** posemiring as follows, being careful to use the 0-field record type $\top$ so that everything algebraic gets solved by Agda's η-laws. Indeed, in this very definition, all of the semiring operations and laws are canonically inferred.

```
0-poSemiring : PoSemiring 0ℓ 0ℓ 0ℓ
0-poSemiring = record
  { Carrier = ⊤; _≈_ = λ _ _ → ⊤; _≤_ = λ _ _ → ⊤ }
```

The elaboration process is monadic. In particular, we use the List/non-determinism monad to give *all* of the possible annotation choices on the free variables of a term. We believe the commitment to multiple solutions is inherent when the syntax contains `i. For example, in the intermediate stages of elaborating $(\vdash \lambda x.\ (*,*)) : A \multimap \top \otimes \top$

with a usage counting posemiring (assuming reasonable rules for $\top$ and $\otimes$), it is unclear whether to use the variable x in the left $*$ or the right $*$. This uncertainty should be reflected in the final result.

The non-deterministic choices we make during elaboration are enumerated by the fields of NonDetInverses. These choices are driven by the typing rules and a candidate usage vector for the conclusion. For example, $+^{-1}\ r$ is needed when we encounter a $`*$ in the syntax and the candidate usage annotation we are considering is r. Then, $+^{-1}\ r$ is a list of pairs of annotations p and q that r can split into, together with a proof of the splitting. For $0\#^{-1}$ and $1\#^{-1}$, inverses to constants, we are given the candidate r and typically return an empty list if the constraint cannot be satisfied, or a singleton list containing a proof. $*^{-1}$ is used when we encounter scaling, in which case we know both the scaling factor r (from the syntax description) and the candidate q. These inverse operations combine monadically (in fact, applicatively) to give inverses to the vector operations of zero, addition, scaling, and basis.

```
record NonDetInverses : Set where
  field
    0#⁻¹ : (r : Ann) → List (r ≤ 0#)
    +⁻¹ : (r : Ann) → List (∃ \ ((p , q) : _ × _) → r ≤ p + q)
    1#⁻¹ : (r : Ann) → List (r ≤ 1#)
    *⁻¹ : (r q : Ann) → List (∃ \ p → q ≤ r * p)
```

We choose the $\mathcal{V}$ of our semantics to be (unannotated) variables. For the $\mathcal{C}$, we consider *functions* from candidate usage vectors R to the list of elaborated derivations with usage annotations given by R. The protocol this encodes is that the user will provide an unannotated term together with a candidate usage context R, and usage elaboration returns a list of possible ways the term could be annotated such that the conclusion has usage context R. The module name U refers to the fact that we are taking the ambient posemiring to be **0** in OpenFam. The effect on OpenFam is that the usage annotations of any contexts we consider are uninformative (hence the _ on the left).

```
𝒞 : System → U.OpenFam _
𝒞 sys (U.ctx _ γ) A = ∀ R → List ([ sys , ∞ ] ctx R γ ⊢ A)
```

To traverse the unannotated terms, we produce a Semantics over the unannotated system uSystem *sys*. To write it, we make use of idiom brackets ⦇ ... ⦈, which have the effect of replacing top-level spines of applications by (List-)applicative applications. Field by field, we already know that variables are renameable. To interpret a variable, we consider all the possible proofs that such a variable could be well annotated, and package them up as a variable term via the applicative machinery. Finally, for compound terms, we first reify the unannotated subterms, and then combine the subterms via a lemma.

```
elab-sem : ∀ sys → U.Semantics (uSystem sys) U._∃_ (𝒞 sys)
elab-sem sys .ren^𝒱 = U.ren^∃
elab-sem sys .⟦var⟧ (U.lvar i q _) R =
  ⦇ 'var ⦇ (lvar i q) (⟨ i ⟩⁻¹ R) ⦈ ⦈
elab-sem sys .⟦con⟧ b R =
  let rb = U.map-s′ (uSystem sys) U.reify b in
  ⦇ 'con (lemma sys rb) ⦈
```

The lemma essentially goes through the shape of the premises, combining the collections of subterms in the natural way. For example, at each _⨯̇_, we take the Cartesian product of the possibilities of each half, and at each _*_, we non-deterministically split the usage annotations coming in, and then take the Cartesian product. When it comes to newly bound variables, the *syntax description* tells us their annotations, so there is no further non-determinism introduced here.

```
lemma : ∀ (sys : System) {A Γ} →
  U.⟦ uSystem sys ⟧s (U.Scope (𝒞 sys)) (uCtx Γ) A →
  List (⟦ sys ⟧s (Scope [ sys , ∞ ]_⊢_) Γ A)
```

To actually use elab-sem on terms, we take the associated semantics and pass it the identity environment (an identity *renaming* in this case, because $\mathcal{V}$ is a family of variables). We use elab-unique, which further checks statically that exactly one derivation

is returned. The candidate usage vector R will be [] for closed terms, and otherwise we have to supply the intended usage annotations.

We can now use the elaborator to automatically infer the usage annotations for the example at the end of section 6.2. This allows us to write:

```
cojoin-!ω : ∀ {A} → [ λR , ∞ ] []ᶜ ⊢ (! ω# A ⊸ ! ω# (! ω# A))
cojoin-!ω = elab-unique _ (⊸I (!E (var# 0) (!I (!I (var# 1))))) []
```

We have instantiated the usage elaborator so that: $0\#^{-1}$ is a singleton on 0 and ω, and empty on 1; $1\#^{-1}$ is a singleton on 1 and ω, and empty on 0; $+^{-1}$ gives $0 \mapsto [(0,0)]$, $1 \mapsto [(0,1),(1,0)]$, and $\omega \mapsto [(\omega,\omega)]$; and $*^{-1}$ gives $(\omega,0) \mapsto [0]$, $(\omega,1) \mapsto []$, and $(\omega,\omega) \mapsto [\omega]$ (omitting $(0,\_)$ and $(1,\_)$ cases for brevity). Note that we do not consider splitting ω up as, say, $1+\omega$, because this splitting would introduce more non-determinism but not allow any more terms to be typed. As such, the only non-determinism comes when we have variables annotated 1 and need to do an additive split, like when we apply the !E rule below. At this point, the variable can become either 0-annotated in the left subterm and 1-annotated on the right, or vice-versa. We will find that, because the left subterm wants to use that variable, the former choice will be rejected. The function var# is a convenience for converting statically known natural numbers, representing De Bruijn *levels*, into variable terms.

8.2 Normalisation by evaluation

To give an example of a common operation on programming languages, I present a normalisation by evaluation (NbE) algorithm [Berger and Schwichtenberg, 1991] for a fragment of $\lambda\mathcal{R}$. The algorithm contains not only a structural induction on a term using an environment, but also functions which create terms. It therefore gives an example of various parts of the framework.

Given the linear theme of $\lambda\mathcal{R}$, one may consider using the size-based normalisation proof given by Girard [1987, p. 71] and Krishnaswami [2013]. However, this fails for two reasons for $\lambda\mathcal{R}$, both of which also apply to the fragment I consider here defined by $A, B ::= \iota \mid !_r A \multimap B$. Firstly, we can use the ! connective in the linear instantiation of

$\lambda\mathcal{R}$ to introduce contractable variables. The size maintenance/reduction principles do not hold in the presence of contraction, because substitution for a contracted variable can make the term grow arbitrarily larger. Secondly, there are instantiations other than the linear $\{0, 1, \omega\}$ semiring, many of which allow for non-linear behaviour even without the !-connective. For example, with the trivial semiring, we recover an intuitionistic calculus with no usage restrictions. We therefore have to use a normalisation proof which also works for standard Simply Typed λ-Calculus.

Though it is easy to define a syntax of normal and neutral forms, I do not do so here. Converting between terms and normal/neutral forms introduces significant overheads, and the syntax of normal/neutral forms is somewhat complicated by the invariant that all variables are of neutral type. It would be worthwhile future work to work out this refinement, thereby showing that the normaliser presented in this section actually puts terms into normal form. Another limitation is that I do no reasoning about equivalence of terms, so there is no proof that the resultant term is equivalent to the original.

For the sake of simplicity and conciseness, I consider just the fragment containing a unary function type, but whose argument carries a usage annotation. The partially ordered semiring of usage annotations is arbitrary throughout. I also include an arbitrary set of base types $BaseTy$ via constructor ι.

```
data Ty : Set where
  ι : BaseTy → Ty
  _⊸_ : (rA : Ann × Ty) (B : Ty) → Ty
```

With only function types, the only term formers are application and λ-abstraction. These are similar to the corresponding term formers in $\lambda\mathcal{R}$, but application scales its argument in accordance with the usage annotation of the function type, and λ-abstraction binds a variable with the given usage annotation. For convenience, I define an alias Term for the open family of terms in this type system of unspecified (∞) size.

```
data L : Set where
  'app 'lam : (rA : Ann × Ty) (B : Ty) → L
```

```
system : System
system = L ▷ λ where
  ('app rA@(r , A) B) → ⟨ []ᶜ '⊢ rA ⊸ B ⟩ '∗ r '· ⟨ []ᶜ '⊢ A ⟩ '↠ B
  ('lam rA B) → ⟨ [ rA ]ᶜ '⊢ B ⟩ '↠ rA ⊸ B

Term = [ system , ∞ ]_⊢_
```

The NbE model of types is given by `_⊨_`. It is defined by induction on the type. The base type is interpreted by terms of base type, while the function type is interpreted by the Kripke function space (essentially a special case of Kripke with $\Delta = rA$, though reusing the Kripke definition causes difficulty for the termination checker). The interpretation of the base type additionally requires a Lift of its universe level, because $\Box^r$ produces a large type (a type in Set_1). The largeness of $\Box^r$ comes from the largeness of environments, and hence renamings, because of the relationally specified linear map.

```
_⊨_ : OpenFam (suc 0ℓ)
Γ ⊨ (ι α) = Lift (suc 0ℓ) (Term Γ (ι α))
Γ ⊨ (r , A) ⊸ B = □ʳ ((r · _⊨ A) ⇻ _⊨ B) Γ
```

`_⊨_` is a renameable family, as can be seen by cases on the type. It is renameable at the base type because terms are renameable, and it is renameable at function type because any open type formed using $\Box^r$ is renameable by composition of renamings. I refer to this proof of renameability as `ren^⊨`.

Now that I am working with a specific syntax, I introduce the function `reify[]`. This is a variant of the `reify` function given in section 7.5 usable only for rules that bind no new variables — i.e., the parameter Δ to Kripke is the empty context.

Lemma 8.2.1 (`reify[]`)**.** We have a function of type $\mathrm{Kripke}\,\mathcal{V}\,\mathcal{C}\,(\cdot) \twoheadrightarrow \mathcal{C}$ for any open families $\mathcal{V}$ and $\mathcal{C}$.

Proof. Let $b : \mathrm{Kripke}\,\mathcal{V}\,\mathcal{C}\,(\cdot)\,\Gamma\,A$. We give b the identity renaming (corresponding to no extension of the context), yielding $b\,1 : \left((-) \overset{\mathcal{V}}{\Longrightarrow} \cdot \twoheadrightarrow (-) \models^{\mathcal{C}} A\right)\Gamma$. By lemma 5.2.2, we have that $(-) \overset{\mathcal{V}}{\Longrightarrow} \cdot$ is equivalent to I^*, and by monoidal closure, $\left(I^* \twoheadrightarrow (-) \models^{\mathcal{C}} A\right)$ is equivalent to $(-) \models^{\mathcal{C}} A$, so the result follows. □

The first part of the NbE algorithm I give is the evaluator. The evaluator interprets terms into the model via an environment of values from the model, making it a clear instance of the generic semantic traversal. Specifically, I fix both $\mathcal{V}$ and $\mathcal{C}$ to $\_\vDash\_$ (allowing the conversion from semantic values to semantic terms to be given by the identity function id), and provide the other required data as below. The case for λ-abstraction is almost trivial, given that its job is to convert the Kripke function M coming from the generic traversal into a value in the model of function type (i.e. a Kripke function). The main thing we need to do is to wrap the argument to the function into a singleton environment using $[\text{-}]^e$. In the case for function application, we use reify[] to get rid of the extraneous Kripke functions coming from the generic traversal. Having done so for both subterms, we apply the value corresponding to M (which is a Kripke function) to the value corresponding to N in an unextended context (1^r), with usage information divided up as in the original application.

```
evalSem : Semantics system _⊨_ _⊨_
evalSem .ren^𝒱 {A} = ren^⊨ {A}
evalSem .⟦var⟧ = id
evalSem .⟦con⟧ (`app _ _ , refl , M *⟨ sp+ ⟩ (⟨ sp* ⟩· N)) =
  reify[] M 1ʳ .app* sp+ (⟨ sp* ⟩· reify[] N)
evalSem .⟦con⟧ (`lam _ _ , refl , M) ρ .app* sp+ [N] =
  M .get ρ .app* sp+ ([-]ᵉ [N])
```

The function eval is given by the appropriate instantiation of the generic traversal semantics. Note that we have not yet shown that $\_\vDash\_$ supports identity environments (which requires the reify and reflect functions), so we cannot yet evaluate arbitrary open terms.

```
eval : ∀ {Γ Δ} → [ _⊨_ ] Γ ⇒ᵉ Δ → {A : Ty} → Term Δ A → Γ ⊨ A
eval ρ M = semantics evalSem ρ M
```

The final pieces of NbE are the *reify* and *reflect* functions. I give these the names nbeReify and nbeReflect respectively to disambiguate from the reify and reify[] functions of the framework. nbeReify and nbeReflect are defined mutually by recursion on the type,

as is standard in NbE. Therefore, neither uses the generic semantic traversal, though both make use of some of the general components I have developed in this thesis.

At base type, values in the model are terms (modulo Lift), so the conversion between these and terms is trivial. At function type, nbeReify is given a Kripke function v and aims to turn it into a normal form of function type. It does so in the canonical way — introducing a λ-abstraction — and then aims to produce a term in an extended context. Because of the context extension, we apply v using the renaming $\swarrow^r : \Gamma +\!\!+^c [\ 0\# \bullet A\ ]^c$ to get it "back" to the original context. The splitting $sp+$ adds together $\Gamma, 0A$ and $0\Gamma, rA$ to get Γ, rA, while sp^* scales down $0\Gamma, rA$ to $0\Gamma, 1A$. Semantic value v then wants a semantic value of type A, which it gets from reflecting the newly bound variable into the model. Finally, v produces a semantic value, which a recursive call to nbeReify at the smaller type B converts into a term (in fact, a normal form). Meanwhile, nbeReflect at function type is given a syntactic function (in fact, a neutral form) and aims to turn it into a Kripke function. To produce a Kripke function, we consider a renaming ρ into Γ, splittings of the resultant context, and a semantic value N of type A. To produce the value of type B, we recursively call nbeReflect at smaller type B, but with a larger term formed by reifying N (again of smaller type) and applying it to the term M. To make the application work, we must rename both M and N to ignore the (empty) context extensions introduced by the application rule, and we must also rename M by ρ to put M into the extended context the reflection is being done at.

```
nbeReify : ∀ {Γ} A → Γ ⊨ A → Term Γ A
nbeReflect : ∀ {Γ} A → Term Γ A → Γ ⊨ A

nbeReify (ι α) (lift v) = v
nbeReify ((r , A) ⊸ B) v =
  ‘con (‘lam _ _ , refl ,
    nbeReify B (v ↙ʳ .app* (mkᶜ sp+) (⟨ sp* ⟩·ᶜ
      nbeReflect A (‘var (lvar (↘ here) refl (≤*-refl ++ₙ [ ≤-refl ]ₙ))))))
  where
  sp+ : ∀ {s t} {P : Vector Ann s} {Q : Vector Ann t} →
    (P ++ Q) ≤[ (P ++ 0*) +* (0* ++ Q) ]
```

```
  sp+ = +*-identity↘ _ ++ₙ +*-identity↙ _

  sp* : ∀ {s r} → (0* {s} ++ [ r ]) ≤[ r *ₗ (0* ++ [ 1# ]) ]
  sp* = (mk λ _ → ≤-reflexive (sym (annihilʳ _))) ++ₙ [ *.identity .proj₂ _ ]ₙ

nbeReflect (ι α) M = lift M
nbeReflect ((r , A) ⊸ B) M ρ .app* (mkᶜ sp+) (⟨ sp* ⟩·ᶜ N) =
  nbeReflect B (`con (`app _ _ , refl ,
    ren (↙ʳ′ []ₙ ⟩⟩ʳ ρ) M *ᶜ⟨ sp+ ⟩ ⟨ sp* ⟩·ᶜ nbeReify A (ren^⊨ N (↙ʳ′ []ₙ))))
```

Finally, we compose eval, nbeReify, and nbeReflect to get the normaliser nbe. The definition of nbe uses eval to interpret the term into the model, and then uses nbeReify to read the semantic value back as a term (a normal form). However, as noted earlier, we need to give an environment of type $\Gamma \overset{\vDash}{\Longrightarrow} \Gamma$ to eval in order to interpret an open term in its own context. I provide this environment via lemma 5.2.7, which says that if we have a mapping from variables in Γ to values in Γ, we can get the desired identity environment. The required map is given in identityEnv^⊨ by turning the variable into a term using `var and then applying nbeReflect. The definition identityEnv^⊨ is marked as an **instance** so that it is picked up by id^Env.

```
instance
  identityEnv^⊨ : IdentityEnv _⊨_
  identityEnv^⊨ .pure x = nbeReflect _ (`var x)

nbe : ∀ {Γ A} → Term Γ A → Term Γ A
nbe M = nbeReify _ (eval id^Env M)
```

The main limitation of this example is that the results are not specified as being in normal form by their type. Ideally, we would have a syntax of normal and neutral forms for nbe and supporting functions to target. We can produce such a syntax — similarly to how Allais et al. [2021] produce a syntax for bidirectional type checking — but this syntax is difficult to work with because of the crucial restriction on the allowable kind of variables — namely that variables can only appear as neutral forms,

and not normal forms. I believe that this difficulty could be worked around via carefully placed assertions that various contexts are well formed — and in particular that the extended context introduced by a $\Box^r$ is well formed.

Also, I have not given an account of many of the connectives of $\lambda\mathcal{R}$. Of particular interest would be the types with pattern-matching eliminators — I, $\otimes$, 0, $\oplus$, and $!$. I believe these could be handled by ordinary techniques for NbE for sums — probably not doing η-expansion for these types.

8.3 A denotational semantics

Standard denotational semantics falls out as a somewhat trivial case of Semantics. A lot of work is done in the generic traversal semantics to maintain a $\mathcal{V}$-environment, where $\mathcal{V}$ is a reasonably variable-like semantic family. For a simple denotational semantics, we set $\mathcal{V}$ to be variables

As an example denotational semantics for a semiring-annotated calculus, I take a semantics in world-indexed relations. This is a refinement of the semantics given by Abel and Bernardy [2020], which gives a way to extract parametricity theorems from substructurally typed programs. Example theorems are that all linear terms act as permutations on some fixed set of resources, and all monotonically typed terms are really monotonic in the way the typing suggests they are.

For the sake of brevity, I use the same term language as in section 8.2, with a set of base types and function types with a usage-annotated domain type.

As a running example, I take the usage annotations to be the 4-element variance posemiring (example 4.2.4). I establish the property that all terms are monotonic in their free variables. This monotonicity can be covariant or contravariant (or neither or both) depending on the annotation of each free variable (respectively, $\uparrow\uparrow$, $\downarrow\downarrow$, $??$, and $\sim\sim$). This provides an additional example to those of Abel and Bernardy [2020].

In the semantics of this section, types are interpreted as *world-indexed relations* [Abel and Bernardy, 2020, Atkey and Wood, 2018]. A world-indexed relation (WRel) over a poset of worlds W is a set set over which we have a W-indexed binary relation ren satisfying a presheaf-like property *subres* with respect to the order on W.

I declare the record type WRel to have **no-eta-equality** so that the usual η-law of records does not apply to it. The lack of an η-law allows the connectives on WRel (e.g. $\_\otimes^R\_$ and $\_\multimap^R\_$, defined below) to be definitionally injective, helping the unification in Agda's elaborator.

```
record WRel {W : Set} (_≤_ : Rel W 0ℓ) : Set₁ where
  no-eta-equality
  field
    set : Set 0ℓ
    rel : (a b : set) → W → Set
    subres : ∀ {a b w w′} → w′ ≤ w → rel a b w → rel a b w′
```

Example 8.3.1. When W is the 1-element set, a world-indexed relation is just a set equipped with a binary relation.

Morphisms (WRelMor) between world-indexed relations R and S consist of a mapping set⇒ between the underlying sets such that, at each world w, the mapping preserves relatedness from R to S. In the type of the field rel⇒, the world index is handled implicitly via $\_\twoheadrightarrow\_$.

```
record WRelMor {W ≤ʷ} (R S : WRel {W} ≤ʷ) : Set where
  constructor wRelMor
  field
    set⇒ : R .set → S .set
    rel⇒ : ∀ {x y} → R .rel x y ↠ S .rel (set⇒ x) (set⇒ y)
```

When the poset of worlds forms a (relational) commutative monoid, such world-indexed relations support a symmetric monoidal closed structure, with objects denoted I^R, $\_\otimes^R\_$, and $\_\multimap^R\_$,. These reuse the bunched connectives I^*, $*$, and $-\!\!*$, now over worlds rather than contexts. Their definitions are listed below.

```
Iᴿ : WRel _≤ʷ_
Iᴿ .set = ⊤
```

```
I^R .rel _ _ = I*
I^R .subres sub I*⟨ sp ⟩ = I*⟨ ε-mono sub sp ⟩

_⊗^R_ : (R S : WRel _≤^w_) → WRel _≤^w_
(R ⊗^R S) .set = R .set × S .set
(R ⊗^R S) .rel (a , b) (a′ , b′) = R .rel a a′ ∗ S .rel b b′
(R ⊗^R S) .subres sub (r ∗⟨ sp ⟩ s) =
  r ∗⟨ •-mono sub ≤^w-refl ≤^w-refl sp ⟩ s

_⊸^R_ : (R S : WRel _≤^w_) → WRel _≤^w_
(R ⊸^R S) .set = R .set → S .set
(R ⊸^R S) .rel f g =
  I⋂ (_ × _) \ (x , y) → R .rel x y ⇻ S .rel (f x) (g y)
(R ⊸^R S) .subres sub rf .app∗ sp xx =
  rf .app∗ (•-mono ≤^w-refl sub ≤^w-refl sp) xx
```

The final piece of semantics we need is a *bang* operator. I allow the semantic *bang* to be an arbitrary annotation-indexed functor on world-indexed relations. This functor must respect all of the structure on the indices, making it a graded comonad over multiplication, as well as being lax monoidal at any particular index r.

```
record Bang : Set₁ where
  field
    !^R : Ann → WRel _≤^w_ → WRel _≤^w_
    !^R-map : ∀ {r R S} → WRelMor R S → WRelMor (!^R r R) (!^R r S)
    !^R-0 : ∀ {r R} → r ≤ 0# → WRelMor (!^R r R) I^R
    !^R-+ : ∀ {r p q R} → r ≤ p + q →
      WRelMor (!^R r R) (!^R p R ⊗^R !^R q R)
    !^R-1 : ∀ {r R} → r ≤ 1# → WRelMor (!^R r R) R
    !^R-* : ∀ {r p q R} → r ≤ p * q → WRelMor (!^R r R) (!^R p (!^R q R))
    !^R-I : ∀ {r} → WRelMor I^R (!^R r I^R)
    !^R-⊗ : ∀ {r R S} → WRelMor (!^R r R ⊗^R !^R r S) (!^R r (R ⊗^R S))
```

The properties required of the semantic *bang* operator are slightly weaker than those given by Abel and Bernardy [2020]. Specifically, the morphisms given by $!^R$-1, $!^R$-I, and $!^R$-$\otimes$ only go one way, rather than being bi-implications as they are for Abel and Bernardy [2020]. Abel and Bernardy need the stronger laws because of their strong eliminator for tensor products, discussed in section 4.4.1. Also, I do not use the *non-idempotent intersection* operators (which handle worlds in the same way as I^R and $\_\otimes^R\_$, but keep the underlying set the same, rather than using products of the underlying sets), and thus do not have axioms for them. I avoid the need for non-idempotent intersections by giving the semantics all at once, rather than first giving a Set-semantics and then giving a WRel-semantics on top of it.

Example 8.3.2. With W being the 1-element set and annotations coming from the variance semiring, we can define the following *bang*. It is always the identity on the set component, while the relation component consists of flipping the relation for contravariance and taking conjunctions to achieve both covariance and contravariance. When we want neither covariance nor contravariance, we use the always true predicate on worlds $\dot{\mathsf{1}}$. With the worlds being trivial, so too is the property subres.

$$!^R : \mathsf{WayUp} \to \mathsf{WRel}\ \_\leq^w\_ \to \mathsf{WRel}\ \_\leq^w\_$$
$$!^R\ a\ R\ .\mathsf{set} = R\ .\mathsf{set}$$
$$!^R \uparrow\uparrow\ R\ .\mathsf{rel} = R\ .\mathsf{rel}$$
$$!^R \downarrow\downarrow\ R\ .\mathsf{rel}\ x\ y = R\ .\mathsf{rel}\ y\ x$$
$$!^R\ ??\ R\ .\mathsf{rel}\ x\ y = \dot{\mathsf{1}}$$
$$!^R\ \sim\sim\ R\ .\mathsf{rel}\ x\ y = R\ .\mathsf{rel}\ x\ y \mathbin{\dot{\times}} R\ .\mathsf{rel}\ y\ x$$
$$!^R\ a\ R\ .\mathsf{subres}\ \_ = \mathsf{id}$$

Let us assume that we have a mapping $\iota[\![\_]\!]$ from base types to world-indexed relations. We extend this interpretation to all types via $[\![\_]\!]$, shown below, with the function type being interpreted using $\multimap^R$ and $!^R$. Contexts are interpreted by $[\![\_]\!]^c$, using $\otimes^R$ and I^R, and $!^R$ for singleton contexts. Terms are interpreted as morphisms by the open family $[\![\_ \vdash \_]\!]$. Variables are interpreted by lookup^R (definition omitted).

```
⟦_⟧ : Ty → WRel _≤ʷ_
⟦ ι α ⟧ = ι⟦ α ⟧
⟦ (r , A) ⊸ B ⟧ = !ᴿ r ⟦ A ⟧ ⊸ᴿ ⟦ B ⟧

⟦_⊢_⟧ : OpenFam 0ℓ
⟦ Γ ⊢ A ⟧ = WRelMor ⟦ Γ ⟧ᶜ ⟦ A ⟧

lookupᴿ : ∀ {Γ A} → Γ ∋ A → ⟦ Γ ⊢ A ⟧
```

The laws $!^R$-0, $!^R$-+, and $!^R$-* lift from singleton contexts to all contexts via lemmas ctx-0, ctx-+, and ctx-*, with the latter saying that we have morphisms of type $[\![(r\mathcal{P})\gamma]\!] \to !^R r[\![\mathcal{P}\gamma]\!]$. We can see each of these three lemmas as ways to turn the algebraic structure of contexts into structured world-indexed relations — allowing us to use general facts about I^R, $\_\otimes^R\_$, and $!^R$, respectively.

Now I give a Semantics. The choice of $\mathcal{V}$ as $\_\exists\_$ is somewhat arbitrary, given that a standard denotational semantics would not use intermediate environments in the same sense as renaming, substitution, and evaluation. The only requirements are that the choice of $\mathcal{V}$ respects renaming and yields inhabitants of the target family $[\![\_\vdash\_]\!]$. Picking $\_\exists\_$ means that we can use the already proven lemma ren^∃ for the former requirement; and we use lookup^R — which we would have to provide somewhere anyway — to yield semantic objects.

Meanwhile, for the logical rules, we ignore environments by using reify and reify[] to just deal with morphisms in an extended context. The crucial structure of world-indexed relations is given by lemmas curry^R and $\mathsf{uncurry}^R$, which translate between WRelMor $(R \otimes^R S)$ T and WRelMor R $(S \multimap^R T)$. Also, map-$\otimes^R$ implements the tensor product of world-indexed relation morphisms. With these lemmas, the interpretations of λ-abstraction and application are straightforward. For λ-abstraction, we have $[\![\Gamma, rA]\!] = [\![\Gamma]\!] \otimes^R !^R r[\![A]\!]$, so we just need to use curry^R to get the desired semantic function in the original context. For application, we are producing the composition pictured in figure 8.1. In this picture, I ignore the possible subusaging between $\mathcal{P} + r\mathcal{Q}$ and the original context for the sake of notational suggestiveness.

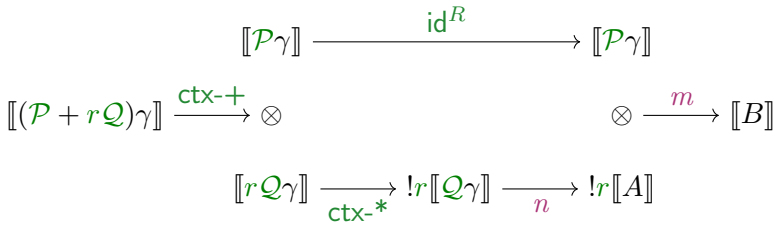


Figure 8.1: Interpretation of application in the world-indexed relation semantics

```
Wrel : Semantics system _∃_ ⟦_⊢_⟧
Wrel .ren^𝒱 = ren^∃
Wrel .⟦var⟧ = lookup^R
Wrel .⟦con⟧ (`lam (r , A) B , ≡.refl , m) = curry^R (reify m)
Wrel .⟦con⟧ (`app (r , A) B , ≡.refl , m *^c⟨ sp+ ⟩ (⟨ sp* ⟩·^c n)) =
  uncurry^R (reify[] m)
    ∘^R map-⊗^R id^R (!^R-map (reify[] n) ∘^R ctx-* sp*)
    ∘^R ctx-+ sp+
```

Then, the semantics of terms is given by the function semantics Wrel 1^r, where 1^r is the identity renaming.

```
wrel : ∀ {sz Γ A} → [ system , sz ] Γ ⊢ A → ⟦ Γ ⊢ A ⟧
wrel = semantics Wrel 1^r
```

Example 8.3.3. We can make a subtraction function from primitive addition and negation on integers. Subtraction is covariant in its first argument and contravariant in its second argument. We give the definition in pseudocode, though it is also amenable to the usage elaborator of section 8.1, suitably instantiated.

$$\sim\sim p : \uparrow\uparrow\mathbb{Z} \multimap \uparrow\uparrow\mathbb{Z} \multimap \mathbb{Z}, \sim\sim n : \downarrow\downarrow\mathbb{Z} \multimap \mathbb{Z} \vdash \mathit{minus} : \uparrow\uparrow\mathbb{Z} \multimap \downarrow\downarrow\mathbb{Z} \multimap \mathbb{Z}$$

$$\mathit{minus} \coloneqq \lambda x.\ \lambda y.\ p\,x\,(n\,y)$$

After feeding in Agda's addition and negation functions as the interpretations of the

free variables (noting that they are both monotonic in the required way), we get the following free theorem.

$$\mathsf{thm} : x\ \mathbb{Z}.\leq x' \to y'\ \mathbb{Z}.\leq y \to x\ \mathbb{Z}.+\ (\mathbb{Z}.\text{-}\ y)\ \mathbb{Z}.\leq x'\ \mathbb{Z}.+\ (\mathbb{Z}.\text{-}\ y')$$

8.4 Translating between $\lambda\mathcal{R}$ and L/nL

In section 6.3.3, I gave what I claimed to be an encoding of Linear/non-Linear logic [Benton, 1994] as a syntax description. In this section, I rigorously state and prove the correspondence between Benton's definition of L/nL and my encoding of it. Then, I give translations from this encoding to my encoding of $\lambda\mathcal{R}$, and vice versa, using two generic semantic traversals. These results together should give us confidence that the encoding of L/nL is correct up to logical equivalence.

8.4.1 Encoding L/nL

I will present translations between the systems given by figures 6.2 and 6.3.

I use S to range over both linear and intuitionistic variables. In this section, I use the notations $\Gamma \vdash A$ and $\Gamma \vdash X$ without subscripts on the turnstile to refer to the encoded version of the calculus. This notation keeps the encoded calculus distinct from the reference L/nL calculus I am translating from and to.

The main difference between the original L/nL calculus and the encoded version is that the encoded version contains some extra "junk", not corresponding to anything in the original L/nL calculus. This junk includes all of the wrinkles we saw when translating to and from DILL in section 4.5.1 — where in the semiring-annotated system, variables annotated ω (corresponding to intuitionistic variables) can slip into having annotation 1 or 0 whenever there are any algebraic manipulations of the context. In linear/non-linear logic, this slipping causes extra problems because intuitionistic variables are supposed to be of a distinct sort to other (i.e., linear) variables. Additionally, we have no means in the framework to correlate types with usage annotations, so we must deal with free variables carefully to ensure the required correlation between linear and intuitionistic types and annotations.

With the above remarks in mind, I take it as clear how to translate the original L/nL calculus into the encoded version, and just state the type of the translation in proposition 8.4.1 without including a proof. In contrast, I spend most of this subsection on the reverse translation, which I provide a proof sketch of in proposition 8.4.4.

Proposition 8.4.1. We can construct the following translations.

$$(\Theta \vdash_{\mathcal{C}} X) \to (\omega\Theta \vdash X) \tag{8.1}$$

$$(\Theta; \Gamma \vdash_{\mathcal{L}} A) \to (\omega\Theta, 1\Gamma \vdash A) \tag{8.2}$$

The key property needed to sensibly do the reverse translation is *linear well-formedness*, as given by definition 8.4.2. Linear well-formedness says that variables of linear type have linear usage annotations. It does not say anything about intuitionistic types and usage annotations for two reasons. First, talking about ω is not sufficiently stable. As we work up a derivation, the "slip" described earlier says that usage annotations will tend to get larger, i.e. more precise. Therefore, it makes more sense to make conditions of being greater than or equal to some collection of usage annotations. Second, it turns out to be unnecessary to add any conditions on variables with intuitionistic type, because we can just treat them as if they were annotated ω. We can forget the specificness of annotations 0 and 1 when not needed, because whatever a specifically annotated variable can do can be done by an ω-annotated variable.

Definition 8.4.2. A semiring-annotated context for L/nL is *linearly well formed* when all linear variables are annotated either 0 or 1.

Lemma 8.4.3. If Γ is linearly well formed and $M : (\Gamma \vdash S)$, then the context of every subterm of M is linearly well formed.

Proof. This lemma follows by inspection of the syntax description (figure 6.3). In the subterms, the linear variables in Γ are only changed by binding new variables (all instances of which maintain linear well formedness) and by existing variables being shared out or coerced (which never produces ω annotations from 0 or 1). □

Linear well-formedness is the only condition needed for the translation given by

proposition 8.4.4. We can translate a derivation with any such context, with no further specification of its shape. In particular, the shape is not calculated from an original L/nL context.

Proposition 8.4.4. Let $\Gamma_{\mathcal{C}}$ be the list of variables of intuitionistic type in Γ. Let $\Gamma_{\mathcal{L}}$ be the list of variables of linear type in Γ with usage annotation 1. Then, whenever Γ is linearly well formed, we can construct the following translations.

$$(\Gamma \vdash X) \to (\Gamma_{\mathcal{C}} \vdash_{\mathcal{C}} X) \tag{8.3}$$

$$(\Gamma \vdash A) \to (\Gamma_{\mathcal{C}}; \Gamma_{\mathcal{L}} \vdash_{\mathcal{L}} A) \tag{8.4}$$

Proof. We proceed by mutual induction on the derivations.

First, I consider variables. Suppose we have $\Gamma \ni S$. If S is a linear type A, then Γ contains one variable of type A annotated 1, while all of the other linear variables are annotated 0 (by linear well formedness, we have no linear variables annotated ω). Therefore, $\Gamma_{\mathcal{L}} = A$, and $\mathcal{L}$-VAR applies. Otherwise, if S is an intuitionistic type X, then $\mathcal{C}$-VAR applies to yield $\Gamma_{\mathcal{C}} \vdash_{\mathcal{C}} X$.

For the logical rules, linear well formedness means that all variables of linear type act linearly. Additionally, every L/nL rule requiring there to be no linear variables in scope is guarded by $\square^{0+*}$ or I^* in the syntax description, excluding linear variables. Given these behaviours, the two calculi correspond closely, and it is a matter of inspection (and use of lemma 8.4.3 when using the induction hypothesis) to complete the proof. □

8.4.2 Translating between L/nL and $\lambda\mathcal{R}$

Benton [1994, §3.3] gives syntactic translations back and forth between Linear/non-Linear logic and the presentation of intuitionistic linear logic given by Benton et al. [1993]. In this section, I give analogous translations between my encodings of L/nL and $\lambda\mathcal{R}$ as instances of the generic traversal semantics. More precisely, I instantiate $\lambda\mathcal{R}$ to the $\{0 > \omega < 1\}$ posemiring and restrict it to the fragment containing connectives I, $\otimes$, $\multimap$, and $!\omega$, matching the fragment of L/nL presented by Benton and in this section. Notably, this fragment of $\lambda\mathcal{R}$ excludes $!0$ and $!1$. I write $!\omega$ as just $!$, as in traditional

$$
\begin{aligned}
(-)^* &: \mathrm{Ty}_{\mathrm{L/nL}} \to \mathrm{Ty}_{\lambda\mathcal{R}} \\
I^* &= I \\
(A \otimes B)^* &= A^* \otimes B^* \\
(A \multimap B)^* &= A^* \multimap B^* \\
(FX)^* &= !X^* \\
1^* &= I \\
(X \times Y)^* &= !X^* \otimes !Y^* \\
(X \to Y)^* &= !X^* \multimap Y^* \\
(GA)^* &= A^*
\end{aligned}
\qquad
\begin{aligned}
(-)^\circ &: \mathrm{Ty}_{\lambda\mathcal{R}} \to \mathrm{Ty}_{\mathrm{lin}} \\
I^\circ &= I \\
(A \otimes B)^\circ &= A^\circ \otimes B^\circ \\
(A \multimap B)^\circ &= A^\circ \multimap B^\circ \\
(!A)^\circ &= GFA^\circ
\end{aligned}
$$

$$
\begin{aligned}
(-)^\circ &: 01\omega \times \mathrm{Ty}_{\lambda\mathcal{R}} \to \Sigma_f\ \mathrm{Ty}_f \\
(0A)^\circ &= (\mathrm{lin}, A^\circ) \\
(1A)^\circ &= (\mathrm{lin}, A^\circ) \\
(\omega A)^\circ &= (\mathrm{int}, GA^\circ)
\end{aligned}
$$

$$
\begin{aligned}
(-)^* &: \mathrm{Ctx}_{\mathrm{L/nL}} \to \mathrm{Ctx}_{\lambda\mathcal{R}} \\
((\mathcal{R}\gamma)^*)_i &= \mathcal{R}_i \gamma_i^*
\end{aligned}
\qquad
\begin{aligned}
(-)^\circ &: \mathrm{Ctx}_{\lambda\mathcal{R}} \to \mathrm{Ctx}_{\mathrm{L/nL}} \\
((\mathcal{R}\gamma)^\circ)_i &= \mathcal{R}_i(\mathcal{R}_i\gamma_i)^\circ
\end{aligned}
$$

Figure 8.2: Translation of types between L/nL and $\lambda\mathcal{R}$

linear logic.

Under the above restrictions and conventions, Benton's translations between the types of ILL and L/nL can be used verbatim, and are reproduced in figure 8.2. Notably, the image of each ILL type under the $(-)^o$-translation falls in the *linear* types of L/nL. In the other direction (the $(-)^*$-translation), we make extensive use of the !-type former to translate the intuitionistic types of L/nL.

I extend $(-)^*$ to contexts pointwise on the type context. Note that this differs from Benton's translation of contexts in that the intuitionistic variables of L/nL are translated using usage annotation ω, rather than type former !. A translation from L/nL to DILL would probably similarly use DILL's intuitionistic variables rather than !.

For $(-)^\circ$, I use an extra step to avoid producing contexts that are not linearly well formed per definition 8.4.2. Specifically, wherever there is an annotation ω in a $\lambda\mathcal{R}$ context, the corresponding type is wrapped in a G to make it intuitionistic. For example, $(0A, 1B, \omega C)^\circ = (0A^\circ, 1B^\circ, \omega GC^\circ)$.

In Agda code, I define the $(-)^\circ$ operator on $01\omega \times \mathrm{Ty}_{\lambda\mathcal{R}}$ (the one that introduces a G

for each ω) via a *view* LIView [McBride and McKinna, 2004]. I define usage annotations 0 and 1 (u0 and u1 in Agda) to be *linear*, with only ω (uω) being *intuitionistic*. LIView is a view because of the existence of liview : $\forall\ x \to$ LIView x, and well behaved in the sense that any two views of the same usage annotation are equal (witnessed by liview-prop, not shown here). The translation from $\lambda\mathcal{R}$ to L/nL takes cases between linear and non-linear annotations at many points, so having these cases expressed as a view avoids duplication of arguments between the cases for 0 and 1.

```
data Linear : Ann → Set where
  u0-lin : Linear u0
  u1-lin : Linear u1

data LIView : Ann → Set where
  view-lin : ∀ {x} (l : Linear x) → LIView x
  view-int : LIView uω
```

Theorem 8.4.5. Let S be an L/nL type, either linear or intuitionistic. Then, we can translate any L/nL term to a $\lambda\mathcal{R}$ term as follows.

$$(\Gamma \vdash_{\mathrm{L/nL}} S) \to (\Gamma^* \vdash_{\lambda\mathcal{R}} S^*) \tag{8.5}$$

Proof. The proof is mostly straightforward, largely following Benton's translation. Similarly to the denotational semantics of section 8.3, we may use a Semantics with $\mathcal{V}$ set to $\exists_{\mathrm{L/nL}}$, and then set $\mathcal{C}\,\Gamma\,S \coloneqq \Gamma^* \vdash_{\lambda\mathcal{R}} S^*$. Whenever we have induction hypotheses of Kripke type, we use the reify function for $\lambda\mathcal{R}$ to get $\lambda\mathcal{R}$ terms. Therefore, we are essentially just doing a proof by induction on the structure of the input term.

Translating the $F\mathrm{I}$ rule into $!\mathrm{I}$ relies on the equivalence between duplicability (as expressed by the $\square$ premise connective) and having been scaled by ω (as expressed by the $\omega\cdot$ premise connective). This holds of the 01ω semiring, but not of general semirings (and is not even stateable for general semirings because of the mention of ω). The same reasoning is needed when translating the introduction rules for intuitionistic connectives, because they always have $\square$ed premises and are translated using !.

As an example of translating an intuitionistic elimination rule, let us consider the $\times\mathrm{E}_0$ rule. I reproduce it here with explicit contexts.

$$\frac{\Gamma' \leq \Gamma \qquad \Gamma \text{ duplicable} \qquad \Gamma \vdash_{\mathrm{L/nL}} X \times Y}{\Gamma' \vdash_{\mathrm{L/nL}} X} \times\mathrm{E}_0$$

Recall that $(X \times Y)^* = !X^* \otimes !Y^*$. This means that we must pattern-match the hypothesis to get variables $\omega X^*, \omega Y^*$ so as to be able to use the X^* for the conclusion and discard the Y^*. The formal derivation is as follows. We are able to copy Γ^* between all of these subterms because its usage annotations are the same as those of Γ, which is duplicable by the assumption in $\times\mathrm{E}_0$. The distinction between Γ' and Γ is minor. When we apply the $\otimes\mathrm{E}$ rule, we use the fact that $\Gamma' \leq \Gamma + \Gamma$, by the fact that $\Gamma' \leq \Gamma$ and $\Gamma \leq \Gamma + \Gamma$. From then on, we need only think about Γ, which behaves well because it is duplicable.

$$\frac{\Gamma^* \vdash_{\lambda\mathcal{R}} !X^* \otimes !Y^* \;(\text{IH}) \qquad \dfrac{\dfrac{}{\Gamma^*, 1!X^*, 0!Y^* \vdash_{\lambda\mathcal{R}} !X^*}\,\textsc{Var} \qquad \nabla}{\Gamma^*, 1!X^*, 1!Y^* \vdash_{\lambda\mathcal{R}} X^*}\,!\mathrm{E}}{\Gamma^* \vdash_{\lambda\mathcal{R}} X^*}\,\otimes\mathrm{E}$$

where $\nabla :=$

$$\frac{\dfrac{}{\Gamma^*, 0!X^*, 1!Y^*, \omega X^* \vdash_{\lambda\mathcal{R}} !Y^*}\,\textsc{Var} \qquad \dfrac{}{\Gamma^*, 0!X^*, 0!Y^*, \omega X^*, \omega Y^* \vdash_{\lambda\mathcal{R}} X^*}\,\textsc{Var}}{\Gamma'^*, 0!X^*, 1!Y^*, \omega X^* \vdash_{\lambda\mathcal{R}} X^*}\,!\mathrm{E}$$

In the Agda proof, renaming is required to perform lots of minor functions that we would ignore on paper. For example, the equation $(\Gamma, \Delta)^* = \Gamma^*, \Delta^*$ — required when induction hypotheses have newly bound variables — is not true definitionally. Furthermore, because of the functional representation I use for contexts, it is not even provable without function extensionality. Therefore, renaming is the simplest way to get the required coercion. Such renamings perhaps could be avoided in most cases if contexts were represented as non-functional lists. □

Theorem 8.4.6. We can translate from $\lambda\mathcal{R}$ to the linear fragment of L/nL.

$$(\Gamma \vdash_{\lambda\mathcal{R}} A) \to (\Gamma^\circ \vdash_{\text{L/nL}} A^\circ) \tag{8.6}$$

Proof. We use the same simple induction scheme as in theorem 8.4.5, but with the places of $\lambda\mathcal{R}$ and L/nL switched. However, some of the rule translations are more complex, mainly caused by the complexity of the $(-)^\circ$ translation on contexts.

For variables, we must consider separately the cases where the variable being used is annotated 1 and ω. The case for a variable $1A$ is straightforward, and translates directly to an L/nL variable $1A^\circ$. A variable ωA, however, is translated to $\omega G A^\circ$, so we must eliminate the G in order to get the conclusion A°. The $G\text{E}$ rule requires its context to be duplicable, which is true by the fact that all of the unused variables are annotated either 0 or ω (because they are all less than or equal to 0), and the variable being used is annotated ω.

Most logical rules are handled very similarly to each other, so I will just translate the $\otimes\text{I}$ rule as an example. I reproduce it here with explicit contexts (split into their usage and typing contexts).

$$\frac{\mathcal{R} \leq \mathcal{P} + \mathcal{Q} \qquad \mathcal{P}\gamma \vdash_{\lambda\mathcal{R}} A \qquad \mathcal{Q}\gamma \vdash_{\lambda\mathcal{R}} B}{\mathcal{R}\gamma \vdash_{\lambda\mathcal{R}} A \otimes B}\ \otimes\text{I}$$

We cannot do a naïve translation to the corresponding application of the $\otimes\text{I}$ rule of L/nL because $(\mathcal{R}\gamma)^\circ$, $(\mathcal{P}\gamma)^\circ$, and $(\mathcal{Q}\gamma)^\circ$ may all have different typing contexts. For example, consider the following instance. There are two problems. First, our induction hypotheses give us contexts containing C°, where our conclusion wants a context containing GC°. Second, $1GC^\circ$ is not linearly well formed, because an intuitionistic type is given a linear annotation. It is therefore difficult to work with such a sequent.

$$\frac{1C \vdash_{\lambda\mathcal{R}} A \qquad 1C \vdash_{\lambda\mathcal{R}} B}{\omega C \vdash_{\lambda\mathcal{R}} A \otimes B}\ \otimes\text{I} \quad \rightsquigarrow \quad \frac{\begin{array}{cc} 1C^\circ \vdash_{\text{L/nL}} A^\circ & 1C^\circ \vdash_{\text{L/nL}} B^\circ \\ \vdots\ ? & \vdots\ ? \\ 1GC^\circ \vdash_{\text{L/nL}} A^\circ & 1GC^\circ \vdash_{\text{L/nL}} B^\circ \end{array}}{\omega GC^\circ \vdash_{\text{L/nL}} A^\circ \otimes B^\circ}\ \otimes\text{I}$$

To fix these issues, firstly I overwrite the context-splitting given by the input term to conform to the bottom-up style of definition 4.5.2. This means precisely that wherever ω appears in the context of the conclusion, it will also appear in the context of all the hypotheses. This gives the following partial derivation.

$$\dfrac{\begin{array}{cc} 1C^\circ \vdash_{\mathrm{L/nL}} A^\circ & 1C^\circ \vdash_{\mathrm{L/nL}} B^\circ \\ \vdots\, ? & \vdots\, ? \\ \omega GC^\circ \vdash_{\mathrm{L/nL}} A^\circ & \omega GC^\circ \vdash_{\mathrm{L/nL}} B^\circ \end{array}}{\omega GC^\circ \vdash_{\mathrm{L/nL}} A^\circ \otimes B^\circ}\ \otimes\mathrm{I}$$

Then, I fix the discrepancy in types using substitution. In the running example, the substitution needed for both subterms is of the type $\omega GC^\circ \overset{\vdash_{\mathrm{L/nL}}}{\Longrightarrow} 1C^\circ$, which amounts to a term of type $\omega GC^\circ \vdash_{\mathrm{L/nL}} C^\circ$, as follows. Note that $G\mathrm{E}$ is only applicable thanks to changing the usage annotation from 1 to ω in the previous step.

$$\dfrac{\dfrac{}{\omega GC^\circ \vdash_{\mathrm{L/nL}} GC^\circ}\ \textsc{Var}}{\omega GC^\circ \vdash_{\mathrm{L/nL}} C^\circ}\ G\mathrm{E}$$

More generally, we may have to produce substitutions of type

$$0A^\circ, 1B^\circ, \omega GC^\circ, \omega GD^\circ \overset{\vdash_{\mathrm{L/nL}}}{\Longrightarrow} 0A^\circ, 1B^\circ, 1C^\circ, \omega GD^\circ.$$

These can be produced pointwise, from substitutions of types $0A^\circ \overset{\vdash_{\mathrm{L/nL}}}{\Longrightarrow} 0A^\circ$ and $1B^\circ \overset{\vdash_{\mathrm{L/nL}}}{\Longrightarrow} 1B^\circ$ and $\omega GC^\circ \overset{\vdash_{\mathrm{L/nL}}}{\Longrightarrow} 1C^\circ$ and $\omega GD^\circ \overset{\vdash_{\mathrm{L/nL}}}{\Longrightarrow} \omega GD^\circ$. We have just seen how to produce the third of these, and the rest are identity substitutions.

The two sui generis rules to translate are the rules for the !-modality, with the ! of intuitionistic linear logic becoming the composite FG in linear/non-linear logic. The way of handling $!\mathrm{I}$ is similar to the way of handling rules like $\otimes\mathrm{I}$, but involving multiplication/scaling by ω rather than addition. We have a similar difficult instance, shown below, where an ω gets specialised to a 1 via the algebraic operation.

$$\dfrac{\omega \leq \omega \cdot 1 \qquad 1C \vdash_{\lambda\mathcal{R}} A}{\omega C \vdash_{\lambda\mathcal{R}} !A}\ !\mathrm{I}$$

Again, the solution is to fix up the operation to keep the more general ω, allowing us

to apply F_{I} and G_{I}, and the same substitution as before possibly including an application of G_{E} as necessary.

Finally, the translation of $!_{\mathrm{E}}$ is simple, but worth checking. I reproduce the rule below with explicit contexts.

$$\frac{\mathcal{R} \leq \mathcal{P} + \mathcal{Q} \qquad \mathcal{P}\gamma \vdash_{\lambda\mathcal{R}} !A \qquad \mathcal{Q}\gamma, \omega A \vdash_{\lambda\mathcal{R}} B}{\mathcal{R}\gamma \vdash_{\lambda\mathcal{R}} B}\ !_{\mathrm{E}}$$

The main thing to note is that the translation of the context of the right-hand premise, $\mathcal{Q}\gamma, \omega A$, is $(\mathcal{Q}\gamma)^\circ, \omega G A^\circ$ — i.e., the translation of contexts gives us a G thanks to the ω usage annotation. Therefore, we do not have to eliminate the G, because the right-hand subterm is expected to do it for us. Indeed, we have seen in previous cases many uses of G_{E} already.

I translate the $!_{\mathrm{E}}$ rule as follows. As in the $\otimes_{\mathrm{I}}$ case, I pick $\mathcal{P}'$ and $\mathcal{Q}'$ to fit bottom-up form, and use the same substitutions as in that case to mend the terms arriving from the induction hypotheses. Then, just F_{E} suffices.

$$\frac{\overline{\mathcal{R} \leq \mathcal{P}' + \mathcal{Q}'} \qquad \begin{array}{c}\vdots\\ (\mathcal{P}'\gamma)^\circ \vdash_{\mathrm{L/nL}} FGA^\circ\end{array} \qquad \begin{array}{c}\vdots\\ (\mathcal{Q}'\gamma)^\circ, \omega G A^\circ \vdash_{\mathrm{L/nL}} B^\circ\end{array}}{(\mathcal{R}\gamma)^\circ \vdash_{\mathrm{L/nL}} B^\circ}\ F_{\mathrm{E}}$$

□

8.5 Conclusion

In this chapter, I have provided a range of example syntaxes and operations on them, defined with the help of the generic semantic traversal. Where Allais et al. [2021] give examples which focus on using the semantic environment effectively, my examples focus more on problems we can only work on using usage restrictions. These examples tie together the two strands of this thesis — representation of syntax and semantics in Agda and usage restrictions via semiring annotations.

Though all of the applications have been successfully completed, there are some points in them where details of the framework have forced us into an unnatural or suboptimal approach, relative to what we could achieve with a reimplementation of each

specific calculus from first principles. Specifically, in the NbE example of section 8.2, I was significantly hampered by the inability to cleanly exclude variables from the syntax of normal forms, and in the encoding of a linear/non-linear calculus in section 8.4, we were forced to consider variables with linear type but intuitionistic usage annotation and vice versa. I will discuss these problems further, with suggestions of solutions, in chapter 9.

Chapter 9

Conclusions

In this thesis, I have developed a foundation for semiring-annotated calculi presented in natural deduction style. I have given a consolidated account of the semiring-annotated calculus $\lambda\mathcal{R}$, including its relations to existing linear and modal calculi. As part of this, I have adapted what I call *bunched connectives* from Rouvoet et al. [2020] as a way to state the typing rules of the calculus as well as to work with the metatheory. The distinction between sharing and separating conjunction given by the bunched connectives corresponds well to the notions of *additive* and *multiplicative* connectives in the object language, respectively. Following this, I have given a novel linear algebra-based definition of *environment* for semiring-annotated calculi, together with a motivation which may serve as a basis for the corresponding definition for other substructural systems. The adequacy of this definition of environment is shown by my implementation of simultaneous renaming and substitution, as well as other operations on environments, like composition of renamings and substitutions. The definition of environments, and hence simultaneous renaming/substitution, is novel, but acts like the appropriate generalisation of environments for simply typed theories.

With the details of $\lambda\mathcal{R}$ worked out, I then moved on to adapting the work of Allais et al. [2021] so as to make it able to capture semiring-based usage restrictions, as found in $\lambda\mathcal{R}$. The syntax descriptions of the resulting system are based on the bunched connectives, and are shown to be expressive enough to encode calculi of a variety of forms. In adapting the semantics, I am forced to be precise about *sharing* and *sepa-*

rating bunched connectives, but largely add these as a refinement of the work of Allais et al. [2021]. I provide the renaming and substitution operations for all expressible calculi. I also provide more specialised examples of semantic traversals: a usage elaborator, an NbE algorithm, a denotational semantics, and translations between different calculi. The usage elaborator gives an unexpected example of generic programming, which one could not write without syntax descriptions. Meanwhile, the NbE algorithm gives further justification that I have the correct notion of environment for semiring-annotated type systems. Together, the examples show the applicability and versatility of the framework I have developed.

9.1 Future work

To conclude, I discuss various possible future directions of the work started in this thesis.

Equality Perhaps the most fundamental missing piece from the metatheoretic account of semiring-annotated calculi I have given in this thesis is equations between terms. Reasoning about equality between terms and environments is a problem I have tried to solve, but I have not arrived at a satisfactory solution in the time available to me.

I believe that the basic difficulty of giving an account for equality in a linear setting is the proliferation of Σ-types. For example, describing equality between two applications of &-I is immediate: $\Gamma \vdash (M, N) = (M', N') : A \,\&\, B$ if and only if $\Gamma \vdash M = M' : A$ and $\Gamma \vdash N = N' : B$. However, to do the same with $\otimes$-I requires us to be careful about the contexts of the subterms. The two applications of $\otimes$-I may a priori split the context in different ways, and should only be equated when those splittings are equal (in the appropriate sense). If the splittings are equal, then the contexts of the subterms will line up, and only then can the subterms themselves be compared for equality. These multiple stages come about because $(T * U)\,\Gamma$ is a Σ-type, and equality of elements of Σ-types always follows this pattern. Such reasoning becomes even more complex for environments, which are equivalent to iterated $*$-families. Additionally, it is unclear what effect subsumption of contexts (like subusaging in this thesis or

explicit structural rules in other calculi) should have on equality, particularly when the subsumption commutes with parts of the subterms.

Polymorphism An important feature of most contemporary statically typed programming languages is polymorphism. In particular, parametric polymorphism over types can be used to significantly improve code reuse, and is well understood theoretically via System F and its variants. I have not considered polymorphism in this thesis, and neither did Allais et al. [2021] in their paper, so whether it can be supported in the framework presented earlier is an open question. However, both Schäfer et al. [2015] and Kaiser et al. [2018] have applied similar work to polymorphic calculi via special support, suggesting that it would be possible to modify the framework of this thesis in a similar way.

A separate but related question concerns polymorphism over usage annotations. The status of polymorphism over usage annotations is less well established both in practice and in theory. Orchard et al. [2019] present an implementation allowing for polymorphism of usage annotations, and even polymorphism over semirings, but provide no more than example programs to justify the feature. This thesis provides no advance on understanding annotation polymorphism, unless it can be encoded into a semiring to fit the framework.

Structure of contexts As I have presented it, the work of Allais et al. [2021] has two axes in which it is generic: the syntax, which can be controlled through syntax descriptions to produce a wide range of calculi and features; and the semantics, where we can produce a wide range of maps out of terms with the help of environments. To this, the work of this thesis has added a third axis of genericity: the usage discipline of variables, as described by a partially ordered semiring.

Starting at least with the bunched connectives in section 4.3, if not earlier when talking about usage contexts forming modules over the semiring of annotations, I have made productive use of abstractions over the basic usage annotations throughout this thesis. These abstractions suggest a next step of completely abstracting away much of the representation of contexts and their individual entries. One may imagine that it

is possible to develop a framework in which the required operations and properties of contexts are axiomatised, similar to how usage annotations are axiomatised to form a semiring in this thesis, and to how categories-with-families models are defined [Castellan et al., 2019, Dybjer, 1995]. Instances of such a framework would include the work of Rouvoet et al. [2020], which uses a very similar bunched connective abstraction over a very different representation of contexts, based on relational interleaving of lists.

The use of semirings is motivated in this thesis and elsewhere largely because they are general enough to cover a wide range of examples. However, I cannot claim to have a derivation from first principles of why we should choose partially ordered semirings over any of a range of similar algebraic structures. Additionally, some of the specific constructions done in this thesis fit somewhat unnaturally with the semiring-based approach. For example, when translating semiring-annotated systems to traditional systems, I tended to need to make a *bottom-up* assumption (definitions 4.5.2 and 4.5.7) so as to avoid some "junk" facts given by the semiring. Meanwhile, the usage elaborator of section 8.1 eschews the "forward" computation of semiring operations in favour of non-deterministic backwards computation, e.g., from a sum to the collection of possible summands. Possibly consciously working more abstractly, as described in the previous paragraph, would make a more natural structure appear.

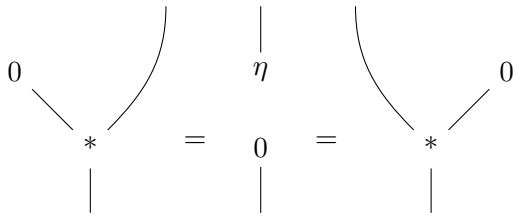
If we are to retain an annotation-based approach to usage restrictions, then a possible feature request that falls out of the encoding of linear/non-linear logic is to have some sort of kinding system by which different kinds of types are annotated using different sets of annotations. In the L/nL example, we would want linear types to be annotated with 0 and 1, and intuitionistic types to be annotated with ω (as the sole element of a trivial instance of an algebraic structure), with no crossover between the two kinds. Algebraic means to handle such mixed-kind usage vectors may be inspired by the work of Hart [1995], McBride and Nordvall Forsberg [2021] on dimensional analysis in linear algebra.

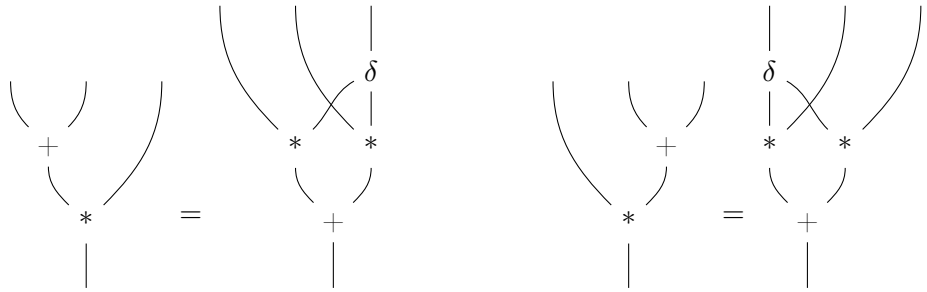
Partiality As we have seen, the way additive and multiplicative rules are realised algebraically is related to models of separation logic. Models of separation logic typically

use *partial* commutative monoids to model a heap, so it is tempting to generalise the commutative monoid of addition in our semirings to a *partial* commutative monoid. However, we find that the most natural notion of *partial semiring* is degenerate, in the sense that all partial semirings are actually (total) semirings.

Recall that a commutative monoid (or commutative monoid object) can be defined in any symmetric monoidal category. A partial commutative monoid is exactly a commutative monoid object in the category of sets and partial functions with the usual monoidal product given by pairing of objects and morphisms (like the Cartesian product in Set). However, semirings need a Cartesian category in order to state the interaction equations between addition and multiplication. While the category of sets and partial functions is not Cartesian, the standard way to manufacture a Cartesian category out of a symmetric monoidal category $\mathcal{C}$ is to take the category of cocommutative comonoids $\mathrm{CComon}(\mathcal{C})$. Intuitively, the cocommutative comonoid structure equips the underlying object M with a *delete* map $\eta : M \to I$ and a *duplicate* map $\delta : M \to M \otimes M$ which are coherent with respect to each other. All morphisms in $\mathrm{CComon}(\mathcal{C})$ must respect η and δ; in particular, both addition and multiplication must separately form bimonoids in $\mathcal{C}$ together with the cocommutative comonoid.

The distributivity laws of semirings are stated below. I include these to show that the cocommutative comonoids of a monoidal category give enough structure to state these laws. The other laws — that all morphisms respect η and δ, that addition forms a commutative monoid, and that multiplication forms a monoid — are standard in symmetric monoidal category theory.





It is well known that all commutative comonoids in $(\mathrm{Set}, \times)$, and indeed any Cartesian monoidal category, are trivial, in the sense that every object of Set gives rise to exactly one commutative comonoid. We find in the following two lemmas that this property also holds of $(\mathrm{Set}_{\mathrm{part}}, \otimes)$.

Lemma 9.1.1. For each object X in $(\mathrm{Set}_{\mathrm{part}}, \otimes)$, there is a cocommutative comonoid over X.

Proof. Let $\eta(x) \coloneqq ()$ and $\delta(x) \coloneqq (x, x)$, with both being defined for all x. Checking that these satisfy the cocommutative comonoid laws is routine. Alternatively, we can see that both η and δ, being total, are morphisms in Set, where it is well known that they form a cocommutative comonoid. The equations in Set carry over to $\mathrm{Set}_{\mathrm{part}}$. □

Lemma 9.1.2. For each object X in $(\mathrm{Set}_{\mathrm{part}}, \otimes)$, any comonoid over X is the one described in lemma 9.1.1.

Proof. The left unit law says that, for all x and x', we have $\exists y.\, \delta(x) = (y, x') \wedge \eta(y) = ()$ if and only if $x = x'$. Letting x' be x and reading from right to left, we get that there is some y such that $\delta(x) = (y, x)$ and $\eta(y) = ()$. Symmetrically, from the right unit law, we get some z such that $\delta(x) = (x, z)$ and $\eta(z) = ()$. But because δ, being a partial function, is deterministic, we have $(y, x) = (x, z)$, giving us that $y = z = x$, and $\delta(x) = (x, x)$. Moreover, because the chosen y is equal to x, we have for all x that $\eta(x) = ()$. □

That a morphism f respects the η given in lemma 9.1.1 is equivalent to saying that f is total. Therefore, all possible semiring operators in $\mathrm{CComon}(\mathrm{Set}_{\mathrm{part}}, \otimes)$ are total, meaning that there is a corresponding semiring in $(\mathrm{Set}, \times)$.

The above reasoning shows that semirings in the category of sets and partial functions are not worth studying. If we want partiality, there appear to be two options. The first option is to give up on multiplication. We could imagine replacing the binary multiplication operator by a set of unary modalities satisfying fewer laws. In particular, I make little use of addition on the left of a multiplication, and multiplying by 0 on the left (as done by $!0$) is unwanted in some cases (such as when encoding DILL and PD, as in section 4.5). With unary modalities, we could expect all of the required laws to be expressible in a symmetric monoidal category. The second option is to use a different notion of partiality. The notion of partiality given by the category of sets and partial functions is "strict", in that composing with an everywhere-undefined function yields an everywhere-undefined function. With a non-strict notion of partial function, we may be able to have interesting partial semirings.

A separate variable sort Allais et al. [2021] mention as a limitation of their work the fact that variables can have unrestricted type. This limitation carries over to the work of this thesis. In many languages — for example, the $\mu\tilde{\mu}$-calculus of section 6.3.2 and the normal/neutral forms of a λ-calculus — we would like to be able to restrict the syntax so as to disallow variables of certain kinds. Such a restriction would make it easier to construct traversals over these syntaxes, which I have not attempted in this thesis.

In private communication, Allais has suggested a simple solution to this problem by which the type of variable types and the type of term types are distinct, and related by an arbitrary relation R. Each use of the $\mathsf{`var}$ constructor then requires a proof that R relates the type of the variable picked to the type of the conclusion desired. For example, in the syntax of normal/neutral forms, we let the variable types be just the types, while the term types are types tagged with whether the term is normal or neutral. The relation R relates equal types where the term type is marked neutral. Allais also suggests that such a scheme could be used to bind patterns in the context, with the relatedness proof serving as a path through the pattern to a variable of the desired type. It remains non-trivial to work out the appropriate notion of renaming, and more

generally the appropriate notion of environment, in this setting, which I leave to future work.

Logical frameworks I discussed in section 3.3.5 two previous approaches to representing linear languages in logical frameworks. The work of this thesis should help in adapting the first approach — making a new logical framework with direct support for linearity — to a broader range of substructural calculi. In particular, the bunched premise connectives give candidates for the required type formers in such a logical framework. On the other hand, the second approach — to use an existing non-linear logical framework but include a `linear` predicate — should be easy to adapt to semiring-annotated systems by replacing the `linear` predicate by a relation between semiring elements and `term -> term` weak functions.

Fitch-style systems I mentioned in section 3.3.4 an alternative way to present modal logics, namely Fitch-style natural deduction systems. These appear to be largely irreconcilable with the work of this thesis due to the fact that the $\Box$-elimination rule is very sensitive to the context. A general treatment of Fitch-style syntaxes, comparable to the treatment of semiring-indexed syntaxes given in this thesis, would probably have to assume a $\Box$ connective whose behaviour was determined as part of the structural rules of the calculus.

Bibliography

M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991. doi: 10.1017/S0956796800000186.

M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A Core Calculus of Dependency. In *POPL '99*, pages 147–160, 1999.

Andreas Abel. Miniagda: Integrating sized and dependent types. In Ekaterina Komendantskaya, Ana Bove, and Milad Niqui, editors, *Partiality and Recursion in Interactive Theorem Provers, PAR@ITP 2010, Edinburgh, UK, July 15, 2010*, volume 5 of *EPiC Series*, pages 18–33. EasyChair, 2010. doi: 10.29007/322q. URL `https://doi.org/10.29007/322q`.

Andreas Abel. Check needed when $\infty < \infty$ is ok for sizes, 2015. URL `https://github.com/agda/agda/issues/1201`.

Andreas Abel and Jean-Philippe Bernardy. A unified view of modalities in type systems. *Proc. ACM Program. Lang.*, 4(ICFP), August 2020. doi: 10.1145/3408972. URL `https://doi.org/10.1145/3408972`.

Andreas Abel and Brigitte Pientka. Higher-order dynamic pattern unification for dependent types and records. In C.-H. Luke Ong, editor, *Typed Lambda Calculi and Applications - 10th International Conference, TLCA 2011, Novi Sad, Serbia, June 1-3, 2011. Proceedings*, volume 6690 of *Lecture Notes in Computer Science*, pages 10–26. Springer, 2011. doi: 10.1007/978-3-642-21691-6_5. URL `https://doi.org/10.1007/978-3-642-21691-6_5`.

The Agda Development Team. Agda 2.6.3, 2023. URL https://agda.readthedocs.io/en/v2.6.3/.

Guillaume Allais. Typing with Leftovers - A mechanization of Intuitionistic Multiplicative-Additive Linear Logic. In *TYPES 2017*, pages 1:1–1:22, 2018. ISBN 978-3-95977-071-2. doi: 10.4230/LIPIcs.TYPES.2017.1.

Guillaume Allais. Generic level polymorphic n-ary functions. In David Darais and Jeremy Gibbons, editors, *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development, TyDe@ICFP 2019, Berlin, Germany, August 18, 2019*, pages 14–26. ACM, 2019. doi: 10.1145/3331554.3342604. URL https://doi.org/10.1145/3331554.3342604.

Guillaume Allais, James Chapman, Conor McBride, and James McKinna. Type-and-scope safe programs and their proofs. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, page 195–207, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450347051. doi: 10.1145/3018610.3018613. URL https://doi.org/10.1145/3018610.3018613.

Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. A type- and scope-safe universe of syntaxes with binding: their semantics and proofs. *J. Funct. Program.*, 31:e22, 2021. doi: 10.1017/S0956796820000076. URL https://doi.org/10.1017/S0956796820000076.

Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In *In Computer Science Logic*, pages 453–468. Springer-Verlag, 1999.

Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. Monads need not be endofunctors. *Log. Methods Comput. Sci.*, 11(1), 2015. doi: 10.2168/LMCS-11(1:3)2015. URL https://doi.org/10.2168/LMCS-11(1:3)2015.

Michael Arntzenius. Tones and types. *URL:http://www.rntz.net/files/tones.pdf*, page 14, 2019.

Robert Atkey. The syntax and semantics of quantitative type theory. In *LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom*, 2018. doi: 10.1145/3209108.3209189.

Robert Atkey and James Wood. Context constrained computation. In *3rd Workshop on Type-Driven Development (TyDe '18), Extended Abstract*, 2018.

Brian Aydemir, Aaron Bohannon, and Stephanie Weirich. Nominal reasoning techniques in Coq. In *International Workshop on Logical Frameworks and Meta-Languages:Theory and Practice (LFMTP)*, Seattle, WA, USA, August 2006.

Andrew Barber. Dual intuitionistic linear logic. Technical report, University of Edinburgh, 1996.

H. P. Barendregt. *Lambda Calculi with Types*, page 117–309. Oxford University Press, Inc., USA, 1993. ISBN 0198537611.

Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic J. of Computing*, 10(4):265–289, dec 2003. ISSN 1236-6064.

Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. Strongly typed term representations in coq. *J. of Autom Reasoning*, 49(2), 8 2012. ISSN 0168-7433. doi: 10.1007/s10817-011-9219-0.

P. N. Benton, Gavin M. Bierman, Valeria de Paiva, and Martin Hyland. A term calculus for intuitionistic linear logic. In *Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, pages 75–90. Springer, 1993. doi: 10.1007/BFb0037099.

P.N. Benton. A mixed linear and non-linear logic: Proofs, terms and models. pages 121–135. Springer-Verlag, 1994.

Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed lambda-calculus. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*,

pages 203–211. IEEE Computer Society, 1991. doi: 10.1109/LICS.1991.151645. URL https://doi.org/10.1109/LICS.1991.151645.

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear haskell: Practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017. doi: 10.1145/3158093. URL https://doi.org/10.1145/3158093.

Marc Bezem, Thierry Coquand, Peter Dybjer, and Martín Escardó. Type theory with explicit universe polymorphism. In Delia Kesner and Pierre-Marie Pédrot, editors, *28th International Conference on Types for Proofs and Programs, TYPES 2022, June 20-25, 2022, LS2N, University of Nantes, France*, volume 269 of *LIPIcs*, pages 13:1–13:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi: 10.4230/LIPIcs.TYPES.2022.13. URL https://doi.org/10.4230/LIPIcs.TYPES.2022.13.

Aleš Bizjak and Lars Birkedal. On models of higher-order separation logic. *Electronic Notes in Theoretical Computer Science*, 336:57–78, 2018. ISSN 1571-0661. doi: https://doi.org/10.1016/j.entcs.2018.03.016. URL https://www.sciencedirect.com/science/article/pii/S1571066118300197. The Thirty-third Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIII).

V.A.J. Borghuis. *Coming to terms with modal logic : on the interpretation of modalities in typed lambda-calculus*. PhD thesis, Mathematics and Computer Science, 1994.

Edwin C. Brady. Idris 2: Quantitative type theory in practice. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPIcs*, pages 9:1–9:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi: 10.4230/LIPIcs.ECOOP.2021.9. URL https://doi.org/10.4230/LIPIcs.ECOOP.2021.9.

A. Brunel, M. Gaboardi, D. Mazza, and S. Zdancewic. A Core Quantitative Coeffect Calculus. In *ESOP 2014*, pages 351–370, 2014.

Simon Castellan, Pierre Clairambault, and Peter Dybjer. Categories with families: Unityped, simply typed, and dependently typed. *CoRR*, abs/1904.00827, 2019. URL http://arxiv.org/abs/1904.00827.

Iliano Cervesato and Frank Pfenning. A linear logical framework. *Inf. Comput.*, 179(1): 19–75, 2002. doi: 10.1006/inco.2001.2951. URL https://doi.org/10.1006/inco.2001.2951.

Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. *Theor. Comput. Sci.*, 232(1-2):133–163, 2000. doi: 10.1016/S0304-3975(99)00173-5. URL https://doi.org/10.1016/S0304-3975(99)00173-5.

Vikraman Choudhury and Neel Krishnaswami. Recovering purity with comonads and capabilities. *Proc. ACM Program. Lang.*, 4(ICFP), aug 2020. doi: 10.1145/3408993. URL https://doi.org/10.1145/3408993.

Alonzo Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(2):56–68, 1940. doi: 10.2307/2266170. URL https://doi.org/10.2307/2266170.

The Coq Team. The Coq reference manual, 2023. URL https://coq.inria.fr/refman/index.html#.

Karl Crary. Higher-order representation of substructural logics. *SIGPLAN Not.*, 45(9): 131–142, September 2010. ISSN 0362-1340. doi: 10.1145/1932681.1863565.

Pierre-Louis Curien and Hugo Herbelin. The duality of computation. *SIGPLAN Not.*, 35(9):233–243, September 2000. ISSN 0362-1340. doi: 10.1145/357766.351262. URL https://doi.org/10.1145/357766.351262.

Brian Day. On closed categories of functors. In *Reports of the Midwest Category Seminar IV*, volume 137 of *Lecture Notes in Mathematics*, pages 1–38. Springer-Verlag, 1970.

Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE*

28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021. doi: 10.1007/978-3-030-79876-5_37. URL https://doi.org/10.1007/978-3-030-79876-5_37.

Stephen Dolan and Leo White. Stack allocation for ocaml, 2022. URL http://stedolan.net/talks/ocaml22.

Peter Dybjer. Internal type theory. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers*, volume 1158 of *Lecture Notes in Computer Science*, pages 120–134. Springer, 1995. doi: 10.1007/3-540-61780-9_66. URL https://doi.org/10.1007/3-540-61780-9_66.

Thomas Ehrhard. An introduction to differential linear logic: proof-nets, models and antiderivatives. *Mathematical Structures in Computer Science*, 28(7):995–1060, 2018. doi: 10.1017/S0960129516000372.

Thomas Ehrhard and Laurent Regnier. The differential lambda-calculus. *Theoretical Computer Science*, 309(1):1–41, 2003. ISSN 0304-3975. doi: https://doi.org/10.1016/S0304-3975(03)00392-X. URL https://www.sciencedirect.com/science/article/pii/S030439750300392X.

M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *Proceedings. 14th Symposium on Logic in Computer Science (Cat. No. PR00158)*, pages 193–202, 1999. doi: 10.1109/LICS.1999.782615.

Marcelo Fiore. Second-order and dependently-sorted abstract syntax. In *2008 23rd Annual IEEE Symposium on Logic in Computer Science*, pages 57–68, 2008. doi: 10.1109/LICS.2008.38.

Marcelo Fiore and Makoto Hamana. Multiversal polymorphic algebraic theories: Syntax, semantics, translations, and equational logic. In *2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 520–529, 2013. doi: 10.1109/LICS.2013.59.

Marcelo Fiore and Chung-Kil Hur. Second-order equational logic (extended abstract). In Anuj Dawar and Helmut Veith, editors, *Computer Science Logic*, pages 320–335, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-15205-4.

Marcelo Fiore and Ola Mahmoud. Second-order algebraic theories. In Petr Hliněný and Antonín Kučera, editors, *Mathematical Foundations of Computer Science 2010*, pages 368–380, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-15155-2.

Marcelo Fiore and Dmitrij Szamozvancev. Formal metatheory of second-order abstract syntax. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022. doi: 10.1145/3498715. URL `https://doi.org/10.1145/3498715`.

Murdoch J. Gabbay. *A Theory of Inductive Definitions with alpha-Equivalence*. phdthesis, 2001. URL `http://www.gabbay.org.uk/papers.html#thesis`.

Murdoch J. Gabbay. The nom package, 2020. URL `https://hackage.haskell.org/package/nom-0.1.0.2/docs/Language-Nominal.html`.

Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2002.

Murdoch James Gabbay and Michael Gabbay. Representation and duality of the untyped λ-calculus in nominal lattice and topological semantics, with a proof of topological completeness. *Ann. Pure Appl. Log.*, 168(3):501–621, 2017. doi: 10.1016/j.apal.2016.10.001. URL `https://doi.org/10.1016/j.apal.2016.10.001`.

Dan R. Ghica and Alex I. Smith. Bounded linear types in a resource semiring. In *ESOP 2014*, pages 331–350, 2014.

Jean-Yves Girard. Linear logic. *Theor. Comp. Sci.*, 50:1–101, 1987.

Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. Bounded linear logic: A modular approach to polynomial-time computability. *Theor. Comput. Sci.*, 97(1):1–66, 1992. doi: 10.1016/0304-3975(92)90386-T. URL `https://doi.org/10.1016/0304-3975(92)90386-T`.

Ananda Guneratne, Chad Reynolds, and Aaron Stump. Project report: Dependently typed programming with lambda encodings in cedille. In David Van Horn and John Hughes, editors, *Trends in Functional Programming - 17th International Conference, TFP 2016, College Park, MD, USA, June 8-10, 2016, Revised Selected Papers*, volume 10447 of *Lecture Notes in Computer Science*, pages 115–134. Springer, 2016. doi: 10.1007/978-3-030-14805-8_7. URL https://doi.org/10.1007/978-3-030-14805-8_7.

Robert Harper, Furio Honsell, and Gordon D. Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, 1993. doi: 10.1145/138027.138060. URL https://doi.org/10.1145/138027.138060.

George W Hart. *Multidimensional analysis: algebras and systems for science and engineering*. Springer Science & Business Media, 1995.

Fritz Henglein. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289, 1993. doi: 10.1145/169701.169692. URL https://doi.org/10.1145/169701.169692.

Hugo Herbelin. C'est maintenant qu'on calcule, au cœur de la dualité. Habilitation, 2005.

André Hirschowitz, Tom Hirschowitz, Ambroise Lafont, and Marco Maggesi. Variable binding and substitution for (nameless) dummies. *CoRR*, abs/2209.02614, 2022. doi: 10.48550/arXiv.2209.02614. URL https://doi.org/10.48550/arXiv.2209.02614.

Martin Hofmann. Linear types and non-size-increasing polynomial time computation. *Inf. Comput.*, 183(1):57–85, 2003. doi: 10.1016/S0890-5401(03)00009-9. URL https://doi.org/10.1016/S0890-5401(03)00009-9.

William A Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

Jack Hughes, Daniel Marshall, James Wood, and Dominic Orchard. Linear Exponentials as Graded Modal Types. In *5th International Workshop on Trends in Linear Logic*

and Applications (TLLA 2021), Rome (virtual), Italy, June 2021. URL https://hal-lirmm.ccsd.cnrs.fr/lirmm-03271465.

Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1), apr 2016. ISSN 0360-0300. doi: 10.1145/2873052. URL https://doi.org/10.1145/2873052.

Andrej Ivašković, Alan Mycroft, and Dominic Orchard. Data-Flow Analyses as Effects and Graded Monads. In Zena M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, volume 167 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:23, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. ISBN 978-3-95977-155-9. doi: 10.4230/LIPIcs.FSCD.2020.15. URL https://drops.dagstuhl.de/opus/volltexte/2020/12337.

Jonas Kaiser, Steven Schäfer, and Kathrin Stark. Binder aware recursion over well-scoped de bruijn syntax. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, page 293–306, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355865. doi: 10.1145/3167098. URL https://doi.org/10.1145/3167098.

Neel Krishnaswami. Strong normalization without logical relations, 2013. URL https://semantic-domain.blogspot.com/2013/05/strong-normalization-without-logical.html.

Joachim Lambek. The mathematics of sentence structure. *The American Mathematical Monthly*, 65(3):154–170, 1958. doi: 10.1080/00029890.1958.11989160. URL https://doi.org/10.1080/00029890.1958.11989160.

Olivier Laurent. Preliminary report on the yalla library. Coq Workshop, 2018. URL https://perso.ens-lyon.fr/olivier.laurent/yalla/.

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The ocaml system, 2022. URL `https://v2.ocaml.org/releases/5.0/htmlman/index.html`.

Daniel R. Licata, Michael Shulman, and Mitchell Riley. A fibrational framework for substructural and modal logics. In *FSCD 2017*, pages 25:1–25:22, 2017. doi: 10.4230/LIPIcs.FSCD.2017.25.

William Lovas and Karl Crary. Structural normalization for classical natural deduction, 2006.

J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 47–57, New York, NY, USA, 1988. Association for Computing Machinery. ISBN 0897912527. doi: 10.1145/73560.73564. URL `https://doi.org/10.1145/73560.73564`.

Dhruv C. Makwana and Neelakantan R. Krishnaswami. NumLin: Linear Types for Linear Algebra. In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:25, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-111-5. doi: 10.4230/LIPIcs.ECOOP.2019.14. URL `http://drops.dagstuhl.de/opus/volltexte/2019/10806`.

Simon Marlow. Haskell 2010 language report. Technical report, 2010. URL `https://www.haskell.org/onlinereport/haskell2010/`.

Nicholas D. Matsakis and Felix S. Klock. The rust language. *Ada Lett.*, 34(3):103–104, oct 2014. ISSN 1094-3641. doi: 10.1145/2692956.2663188. URL `https://doi.org/10.1145/2692956.2663188`.

Conor McBride. Type-preserving renaming and substitution, 2005. URL `http://www.strictlypositive.org/ren-sub.pdf`.

Conor McBride. A polynomial testing principle. *URL:https://personal.cis.strath.ac.uk/conor.mcbride/PolyTest.pdf*, page 37, 2012.

Conor McBride. I got plenty o' nuttin'. In *A List of Successes That Can Change the World*, pages 207–233. Springer, 2016.

Conor McBride. Everybody's got to be somewhere. *Electronic Proceedings in Theoretical Computer Science*, 275:53–69, jul 2018. doi: 10.4204/eptcs.275.6. URL `https://doi.org/10.4204%2Feptcs.275.6`.

Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1): 69–111, 2004. doi: 10.1017/S0956796803004829. URL `https://doi.org/10.1017/S0956796803004829`.

Conor McBride and Fredrik Nordvall Forsberg. *Type systems for programs respecting dimensions*. Series on Advances in Mathematics for Applied Sciences. World Scientific Publishing Co. Pte Ltd, Singapore, January 2021. Advanced Mathematical and Computational Tools in Metrology and Testing XII, AMCTMT XII ; Conference date: 15-09-2020 Through 17-09-2020.

Craig McLaughlin, James McKinna, and Ian Stark. Triangulating context lemmas. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, page 102–114, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355865. doi: 10.1145/3167081. URL `https://doi.org/10.1145/3167081`.

Dale Miller. Unification under a mixed prefix. *J. Symb. Comput.*, 14(4):321–358, 1992. doi: 10.1016/0747-7171(92)90011-R. URL `https://doi.org/10.1016/0747-7171(92)90011-R`.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. The definition of standard ml. Technical report, 1997. URL `https://smlfamily.github.io/sml97-defn.pdf`.

Alan Mycroft. Polymorphic type schemes and recursive definitions. In Manfred Paul and Bernard J. Robinet, editors, *International Symposium on Programming, 6th Colloquium, Toulouse, France, April 17-19, 1984, Proceedings*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228. Springer, 1984. doi: 10.1007/3-540-12925-1_41. URL https://doi.org/10.1007/3-540-12925-1_41.

Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *BULLETIN OF SYMBOLIC LOGIC*, 5(2):215–244, 1999.

Dominic A. Orchard, Vilem Liepelt, and Harley Eades. Quantitative program reasoning with graded modal types. *Proceedings of the ACM on Programming Languages*, 3, June 2019.

Tomas Petricek. *Context-aware programming languages*. PhD thesis, University of Cambridge, 3 2017.

Tomas Petricek, Dominic A. Orchard, and Alan Mycroft. Coeffects: a calculus of context-dependent computation. In *ICFP 2014*, pages 123–135, 2014.

Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. In *Mathematical Structures in Computer Science*, page 2001, 1999.

Frank Pfenning and Carsten Schürmann. System description: Twelf - A meta-logical framework for deductive systems. In Harald Ganzinger, editor, *Automated Deduction - CADE-16, 16th International Conference on Automated Deduction, Trento, Italy, July 7-10, 1999, Proceedings*, volume 1632 of *Lecture Notes in Computer Science*, pages 202–206. Springer, 1999. doi: 10.1007/3-540-48660-7_14. URL https://doi.org/10.1007/3-540-48660-7_14.

Andrew M. Pitts. Locally nameless sets. *Proc. ACM Program. Lang.*, 7(POPL):488–514, 2023. doi: 10.1145/3571210. URL https://doi.org/10.1145/3571210.

Jeff Polakow. Embedding a full linear lambda calculus in haskell. *SIGPLAN Not.*, 50 (12):177–188, August 2015. ISSN 0362-1340. doi: 10.1145/2887747.2804309.

D. Prawitz. *Natural Deduction: A Proof-Theoretical Study*. Dover Books on Mathematics. Dover Publications, 1965. ISBN 9780486446554. URL https://books.google.co.uk/books?id=sJj3DQAAQBAJ.

J. Reed and B. C. Pierce. Distance makes the types grow stronger. In P. Hudak and S. Weirich, editors, *ICFP 2010*, pages 157–168, 2010.

Greg Restall. *An Introduction to Substructural Logics*. New York: Routledge, 1999.

Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for linear, session-typed languages. In *CPP 2020*, pages 284–298, 2020. ISBN 9781450370974. doi: 10.1145/3372885.3373818.

The Rust team. The Rust programming language, 2023. URL https://rust-lang.org/.

Steven Schäfer, Tobias Tebbi, and Gert Smolka. Autosubst: Reasoning with de bruijn terms and parallel substitutions. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, volume 9236 of *Lecture Notes in Computer Science*, pages 359–374. Springer, 2015. doi: 10.1007/978-3-319-22102-1_24. URL https://doi.org/10.1007/978-3-319-22102-1_24.

Miki Tanaka and John Power. A Unified Category-theoretic Semantics for Binding Signatures in Substructural Logics. *Journal of Logic and Computation*, 16(1):5–25, 02 2006. ISSN 0955-792X. doi: 10.1093/logcom/exi070. URL https://doi.org/10.1093/logcom/exi070.

The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

Christian Urban. Nominal techniques in Isabelle/HOL. *J. Autom. Reason.*, 40(4): 327–356, 2008. doi: 10.1007/s10817-008-9097-2. URL https://doi.org/10.1007/s10817-008-9097-2.

Nachiappan Valliappan, Fabian Ruch, and Carlos Tom'e Corti nas. Normalization for fitch-style modal calculi. *Proc. ACM Program. Lang.*, 6(ICFP):772–798, 2022. doi: 10.1145/3547649. URL `https://doi.org/10.1145/3547649`.

Philip Wadler. Propositions as sessions. *ACM SIGPLAN Notices*, 47(9):273–286, 2012.

Michael Winikoff and James Harland. Deterministic resource management for the linear logic programming language lygon. 1994.

James Wood and Robert Atkey. A linear algebra approach to linear metatheory. In Ugo Dal Lago and Valeria de Paiva, editors, *Proceedings Second Joint International Workshop on Linearity & Trends in Linear Logic and Applications, Online, 29-30 June 2020*, 2021. doi: 10.4204/EPTCS.353.10.

James Wood and Robert Atkey. A framework for substructural type systems. In Ilya Sergey, editor, *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, volume 13240 of *Lecture Notes in Computer Science*, pages 376–402. Springer, 2022. doi: 10.1007/978-3-030-99336-8_14. URL `https://doi.org/10.1007/978-3-030-99336-8_14`.

Hongwei Xi. Applied type system. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs*, pages 394–408, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24849-1.

Uma Zalakain and Ornela Dardha. π with leftovers: A mechanisation in agda. In Kirstin Peters and Tim A. C. Willemse, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 41st IFIP WG 6.1 International Conference, FORTE 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings*, volume 12719 of *Lecture Notes in Computer Science*, pages 157–174. Springer, 2021. doi: 10.1007/978-3-030-78089-0_9. URL `https://doi.org/10.1007/978-3-030-78089-0_9`.

Dengping Zhu and Hongwei Xi. Safe programming with pointers through stateful views. In Manuel V. Hermenegildo and Daniel Cabeza, editors, *Practical Aspects of Declarative Languages, 7th International Symposium, PADL 2005, Long Beach, CA, USA, January 10-11, 2005, Proceedings*, volume 3350 of *Lecture Notes in Computer Science*, pages 83–97. Springer, 2005. doi: 10.1007/978-3-540-30557-6_8. URL https://doi.org/10.1007/978-3-540-30557-6_8.