LINEAR/NON-LINEAR TYPES FOR EMBEDDED DOMAIN-SPECIFIC LANGUAGES

Jennifer Paykin

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2018

Supervisor of Dissertation

____

Steve Zdancewic
Professor of Computer and Information Science

Graduate Group Chairperson

____

Lyle Ungar
Professor of Computer and Information Science

Dissertation Committee
Stephanie Weirich, Professor of Computer and Information Science, University of Pennsylvania
Benjamin Pierce, Professor of Computer and Information Science, University of Pennsylvania
Andre Scedrov, Professor of Mathematics, University of Pennsylvania
Peter Selinger, Professor of Mathematics, Dalhousie University

LINEAR/NON-LINEAR TYPES
FOR EMBEDDED DOMAIN-SPECIFIC LANGUAGES

COPYRIGHT

2018

Jennifer Paykin

This work is licensed under a Creative Commons Attribution 4.0 International License.
To view a copy of this license, visit

http://creativecommons.org/licenses/by/4.0/

Acknowledgment

I have so many people to thank for helping me along the journey of this PhD. I could not have done it without the love and support of my partner and best friend, Jake, who was there for me every single step of the way. I also need to thank my parents Laurie and Lanny for giving me so many opportunities in my life, and my siblings Susan and Adam for helping me grow.

I owe so much thanks to advisor Steve Zdancewic, who has been an amazing advisor and who has made me into the researcher I am today. Steve, thank you for teaching me new things, encouraging me to succeed, listening to my ideas, and waiting patiently until I realized you were right all along.

I have been lucky to have many excellent mentors over the years. Thank you to Norman Danner, for introducing me to programming languages and pushing me to take advantage of the opportunities that come my way. Thank you to Stephanie Weirich, for inspiring me and supporting me over the years. Thank you to Benjamin Pierce, for making me into a better writer, speaker, and researcher. Thank you to the rest of my thesis committee—Peter Selinger and Andre Scedrov—for your feedback and encouragement. Finally, thank you to Neel Krishnaswami, Dan Licata, Val Tannen, and all my other collaborators, professors, and mentors for teaching me so much over the years.

My time at Penn would have been much less enjoyable without the wonderful friends I have made here. To everyone at PLClub, Monday night quizzo, cisters, and GETUP, thank you for your friendship, your commiseration, and your support. So many people have made my life better at Penn that I cannot possibly list them all, but I need to single out my closest confidants and conspirators—Antal, Leo, Robert, and Kenny. Thanks for everything!

Finally, my work has been supported financially by the following sources, whose contributions have been much appreciated: the NSF Graduate Research Fellowship Grant Number DGE-1321851; NSF Grant Number CCF-1421193; and ONR MURI No. FA9550-16-1-0082.

ABSTRACT

LINEAR/NON-LINEAR TYPES
FOR EMBEDDED DOMAIN-SPECIFIC LANGUAGES

Jennifer Paykin
Steve Zdancewic

Domain-specific languages are often embedded inside of general-purpose host languages so that the embedded language can take advantage of host-language data structures, libraries, and tools. However, when the domain-specific language uses *linear* types, existing techniques for embedded languages fall short. Linear type systems, which have applications in a wide variety of programming domains including mutable state, I/O, concurrency, and quantum computing, can manipulate embedded non-linear data via the linear type $!\sigma$. However, prior work has not been able to produce linear embedded languages that have full and easy access to host-language data, libraries, and tools.

This dissertation proposes a new perspective on linear, embedded, domain-specific languages derived from the linear/non-linear (LNL) interpretation of linear logic. The LNL model consists of two distinct fragments—one with linear types and another with non-linear types—and provides a simple categorical interface between the two. This dissertation identifies the linear fragment with the linear embedded language and the non-linear fragment with the general-purpose host language.

The effectiveness of this framework is illustrated via a number of examples, implemented in a variety of host languages. In Haskell, linear domain-specific languages using mutable state and concurrency can take advantage of the monad that arises from the LNL model. In Coq, the $\mathcal{Q}$WIRE quantum circuit language uses linearity to enforce the no-cloning axiom of quantum mechanics. In homotopy type theory, quantum transformations can be encoded as higher inductive types to simplify the presentation of a quantum equational theory. These examples serve as case studies that prove linear/non-linear type theory is a natural and expressive interface in which to embed linear domain-specific languages.

TABLE OF CONTENTS

ACKNOWLEDGMENT . . . . **iii**

ABSTRACT . . . . **iv**

1 Introduction **1**

1.1 Conventions 6

2 Linear type systems **7**

2.1 A simple linear type system 7

2.2 Linear connectives 9

2.3 The exponential modality ! 12

2.4 Dual Intuitionistic Linear Logic 15

2.5 Indexed modalities 17

2.6 Kind-based linear logic 19

2.7 Linear/non-linear logic 21

3 Embedded linear/non-linear types **25**

3.1 A linear embedded language 25

3.2 The linear/non-linear interface 26

3.3 Example: linear file handles 29

3.4 Monadic programming 29

3.5 Extensions 32

3.6 Example: session types 34

3.7 Discussion 37

4 Haskell Implementation **39**

4.1 Dependent types in Haskell 40

4.2 Linear types and type checking 42

4.3 Running linear programs 49

4.4 Monadic programming 54

4.5 Example: Arrays 57

4.6 Example: Session types 61

4.7 Discussion and Related Work 64

5 Embedded categorical semantics **68**

5.1 Background 68

5.2 Categories for multiplicative additive linear logic 71

5.3 Linear/non-linear categories 73

5.4 Embedded meta-theory 74

5.5 Conclusion 77

Case study: Quantum Computing **78**

6 A quantum/non-quantum type system **79**
6.1 Quantum computing background 81
6.2 The quantum/non-quantum (QNQ) calculus 85
6.3 Examples 87
6.4 Denotational semantics 89

7 Quantum equational theories in HoTT **93**
7.1 Background and main ideas 94
7.2 Equational theory of QNQ 95
7.3 Deriving equational rules in homotopy type theory 98
7.4 Equivalence of unitaries 101
7.5 Denotational Semantics 107
7.6 Discussion 109
7.7 Conclusion 110

8 $\mathcal{Q}$wire: Quantum circuits in Coq **111**
8.1 The $\mathcal{Q}$WIRE circuit language 112
8.2 Linear type checking in Coq 114
8.3 Surface language 117
8.4 Discussion 121

9 Future work **123**
9.1 Adapting LNL to other substructural type systems. 123
9.2 Formalizing the theory of embedded languages. 125
9.3 Variations to the structure of LNL 125
9.4 Drawing on the host language 125
9.5 Shortcomings and outstanding problems 125
9.6 Conclusion 126

BIBLIOGRAPHY **126**

LIST OF ILLUSTRATIONS

1.1 The linear/non-linear embedded programming model. 4

2.1 Linear implication 9
2.2 Multiplicative product $\otimes$ 10
2.3 Proof that `LUnit` is the unit of $\otimes$. 11
2.4 Multiplicative unit `LUnit` 11
2.5 Additive product 12
2.6 Additive unit: `LTop` 12
2.7 Additive sum 13
2.8 Additive unit: `LZero` 13
2.9 The linear/non-linear categorical model. The model consists of two categories related by functors `Lift` and `Lower` that form a categorical adjunction `Lift` $\dashv$ `Lower`; for details see Section 5.3. 22
2.10 LNL `Lift` connective 23
2.11 LNL `Lower` connective 23

3.1 Specification of an embedded linear lambda calculus as terms of type `LExp` Δ σ. 27
3.2 Interface of linear file handles, given as inference rules, writing $\Delta \vdash e : \sigma$ for $e :$ `LExp` Δ σ. 28
3.3 Recursive types in the linear embedded language 33
3.4 Polymorphism in the linear embedded language 34
3.5 Dependent types in the linear embedded language 35
3.6 Linear interface to session types. 36

4.1 Type families over linear typing contexts, enforcing the invariant that typing contexts are sorted. The custom type errors `AddError` and `RemoveError` provide better error reporting—see Section 4.7. 43
4.2 Haskell interface to linear additive connectives 47
4.3 Examples of linear programs embedded in Haskell 48
4.4 Values in the deep embedding associated with various linear connectives. . . 52
4.5 Interface to linear arrays. 58
4.6 Linear quicksort algorithm 61

5.1 β and η equivalence for the embedded linear lambda calculus. 75

6.1 Multiplicative exponential fragment of QNQ 86
6.2 Quantum teleportation. 90
6.3 Quantum Fourier transform in QNQ 90

7.1 Structural axioms 97
7.2 Groupoid axioms 97
7.3 Unitary equivalence axioms for $X : \mathcal{U}(\texttt{Qubit}, \texttt{Qubit})$, $\texttt{SWAP} : \mathcal{U}(\sigma \otimes \tau, \tau \otimes \sigma)$, and $\texttt{DISTR} : \mathcal{U}(\sigma \otimes (\tau_1 \oplus \tau_2), (\sigma \otimes \tau_1) \oplus (\sigma \otimes \tau_2))$. 98
7.4 Operations on open linear types 102

7.5 Inductive presentation of open type equivalence. 103
7.6 Proofs of Equations SWAP-INTRO and SWAP-ELIM. 106

8.1 Unitary and non-unitary gates in $\mathcal{Q}$WIRE. Different gate sets could have been chosen, for example by picking a different universal set of unitary gates, or by allowing arbitrary circuits to be frozen as gates, which is a feature allowed by many practical circuit languages. Rennela and Staton (2018) propose some extensions to $\mathcal{Q}$WIRE that expand the gate set to add sums and recursive data types. 112
8.2 Translation of QNQ expressions and boxes to $\mathcal{Q}$WIRE circuits and boxes. . . 119
8.3 Implementing QNQ syntax in $\mathcal{Q}$WIRE. 119

9.1 Lattice of substructural type systems 124

CHAPTER 1

Introduction

Resources like mutable state, I/O, and communication channels play a big role in many programming domains, but are subject to some very subtle bugs. Programming languages can alleviate some of this pain through abstractions, which simplify reasoning about otherwise unsafe effects. Often, particular programming domains need domain-specific abstractions, and the languages that provide them are called *domain-specific languages* (DSLs).

Implementing a standalone DSL can be a lot of work for the language designer, who must come up with useful domain-specific abstractions and also provide syntax, libraries, a programming environment, and tool support. Furthermore, working in a standalone DSL can be inconvenient for the user, who has to work with the new abstractions and also learn the syntax and features of a brand new language.

Embedded domain-specific languages (EDSLs) alleviate some of this work by defining the DSL inside of an existing general-purpose language. EDSLs let users take advantage of existing language constructs, libraries, and tools, ideally with very little overhead.

Unfortunately, not all programming abstractions are popular as EDSLs. Abstractions that use *linear* or *substructural* type systems (Girard, 1987) have been neglected because most general-purpose languages do not natively support linear resource management.

For example, a DSL for memory management might provide *linear mutable references*:

```
alloc    : α ⊸ LRef α
dealloc  : LRef α ⊸ Unit
lookup   : LRef α ⊸ α ⊗ LRef α
assign   : LRef α ⊸ α ⊸ LRef α
```

Here, $\multimap$ ("lollipop") denotes a linear function that uses its argument exactly once, and $\otimes$ ("tensor") denotes a linear pair.

Linearity enforces two major invariants: linear data can be neither duplicated nor discarded. For references, the fact that linear data cannot be duplicated means that once a reference has been deallocated, it cannot be accessed again.

```
illegal_assign ≡ let r  ≔ alloc "hello"
                 let () ≔ dealloc r in
                 assign r "world"     -- type error
```

The lack of duplication also prevents data races in the case of parallelism.

```
illegal_race ≡ let r ≔ alloc "hello" in
               fork (assign r "world") (assign r "goodbye") -- type error
```

The fact that linear data cannot be discarded means that, in any terminating, top-level program (in this case, a program of type `Unit`), every reference will eventually be

deallocated. This eliminates the need for garbage collection, which can improve performance, while ensuring there are no space leaks.

```
illegal_leak ≡ let _ ≔ alloc "hello world" in () -- type error
```

As it happens, linearity and related concepts are useful for a wide variety of domain-specific applications, not just mutable state.

- Perhaps most frequently, linear types provide abstractions for system resources including I/O (Wadler, 1990), file handles (Brady and Hammond, 2012), ownership/permissions of shared data (Pottier and Protzenko, 2013), and garbage collection (Fluet *et al.*, 2006). These abstractions are also supported by variations of linear type systems including affine, relevant, and ownership type systems, which are known collectively as *substructural* type systems.

- Linearity can be used as a logic in which to reason about such stateful systems. Separation logic uses linearity to reason about non-overlapping parts of a heap, so that properties about heaps can be extended to larger contexts in a modular way (Reynolds, 2002). The Linear Logical Framework (LLF) (Cervesato and Pfenning, 1996) uses linearity to facilitate logic meta-programming about languages with mutable state.

- Session types use linearity to ensure that at any given time, a communication channel has exactly two endpoints, governed by dual communication protocols (Kobayashi *et al.*, 1999; Gay and Vasconcelos, 2010). Formally, there is a Curry-Howard correspondence between session-typed π-calculus terms and linear logic (Caires and Pfenning, 2010; Wadler, 2014).

- Linearity can also be used to manage time as a resource. Girard (1998) shows how linearity can be used to characterize polynomial-time functions. Krishnaswami *et al.* (2012) use linear types combined with temporal logic to safely reason about functional reactive programs and graphical user interfaces (Krishnaswami and Benton, 2011).

- Bounded linear logic can be used to track properties of data, known as *coeffects*, like data flow, liveness, and privacy (Petricek *et al.*, 2014; Brunel *et al.*, 2014). Reed and Pierce (2010) use linear types in a language for differential privacy to bound the amount of private information leaked by statistical database queries.

- Quantum computing has the property that quantum data cannot be duplicated or discarded: the *no-cloning theorem*. Several languages for quantum computing use linear types to enforce this invariant (Selinger and Valiron, 2009; Ross, 2015).

Although most general-purpose languages do not natively support linear types, languages with dependent types can be used to *encode* linear typing judgments. Consider a type $\mathtt{LExp}\ \Delta\ \sigma$ of linear expressions, where $\sigma : \mathtt{LType}$ is a linear type and $\Delta : \mathtt{List}(\mathtt{LVar} \times \mathtt{LType})$ is a typing context mapping linear variables to linear types. The intention is that terms $e : \mathtt{LExp}\ \Delta\ \sigma$ represent linear expressions of type σ using linear variables from Δ.

Linearity is enforced by imposing constraints on typing contexts. For example, given a linear function $e_1 : \mathtt{LExp}\ \Delta_1\ (\sigma \multimap \tau)$ and an argument $e_2 : \mathtt{LExp}\ \Delta_2\ \sigma$, function application $e_1\hat{}e_2$ is defined exactly when the linear variables used by e_1 and e_2 do not overlap—in other words, when Δ_1 and Δ_2 are disjoint, written $\Delta_1 \perp \Delta_2$. In that case, $e_1\hat{}e_2 : \mathtt{LExp}\ (\Delta_1, \Delta_2)\ \tau$.

Linear EDSLs in this style are not common in practice. Host languages lack tools and techniques to manage linear variable binding and automatically check disjointedness conditions $\Delta_1 \perp \Delta_2$. Although isolated examples exist in the literature (see for example Mazurak *et al.* (2010); Polakow (2015); Kiselyov (2012)), full support of linear types is rare.

More importantly, the linear EDSLs that do exist are not designed in a way that take advantage of host-language data and libraries. In traditional presentations of linear types, all data is assumed to be linear unless its type has the form $!\sigma$ (pronounced "bang σ"). Types of this form, however, can be freely duplicated and discarded.

$$\texttt{duplicate} : !\sigma \multimap !\sigma \otimes !\sigma \qquad \texttt{discard} : !\sigma \multimap \texttt{LUnit}$$

To construct a term of type $!\sigma$, it suffices to produce a term of type σ, *as long as it is linearly closed*. That is, if $e : \texttt{LExp}\ \varnothing\ \sigma$ does not use any linear variables, then it can be duplicated or discarded by simply executing e twice or zero times, respectively.

$$\texttt{duplicate}\ (e) \equiv (e, e) \qquad \texttt{discard}\ (e) \equiv ()$$

If e had a non-empty linear context, the pair (e, e) would be ill-typed; the two components of the pair would use overlapping linear variables, violating the no-duplication property.

In the context of an EDSL, this implies that non-linear expressions should be part of the embedded language. Thus, the EDSL implementer has to design an easy-to-use type system that handles both linear and non-linear data—already a difficult task. Furthermore, the EDSL user cannot fully take advantage of the host language's abstractions and libraries, but instead has to duplicate all relevant libraries inside the EDSL.

As an example, consider the linear mutable references presented at the beginning of this section. Although references themselves are linear, they hold non-linear data; `lookup` duplicates its data and `assign` discards its data. More precisely, the types of `lookup` and `assign` should be non-linear:

```
lookup : LRef σ ⊸ !σ ⊗ LRef σ            assign : LRef σ ⊸ !σ ⊸ LRef σ
```

Consider a program `center_at` that updates the state of an xy-coordinate stored in a mutable reference. The program `center-at flag coord` sets the coordinate (x, y) in `coord` to (x, x) if `flag` is true, and otherwise sets (x, y) to (y, y).

```
centerAt : LRef Bool ⊸ LRef (LInt ⊗ LInt) ⊸ LRef Bool ⊗ LRef (LInt ⊗ LInt)
         ≡ λ flag. λ coord. let (b,flag) ≔ lookup flag in
                            let ((x,y),coord) ≔ lookup coord in
                            (flag, if b then assign (x,x) coord
                                        else assign (y,y) coord)
```

In order to implement this program, a linear EDSL would have to

- provide a type of linear booleans `LBool` and a library of boolean operations;
- provide a type of linear numbers `LInt` and a library of arithmetic operations; and
- implement type inference that automatically coerces linear integers and linear booleans to `!LInt` and `!LBool` respectively, so the expressions (x, x) and (y, y) are well-typed.

While numbers and booleans are not too large a challenge, this kind of code duplication goes against the very spirit of embedded DSLs. If the host language has its own libraries

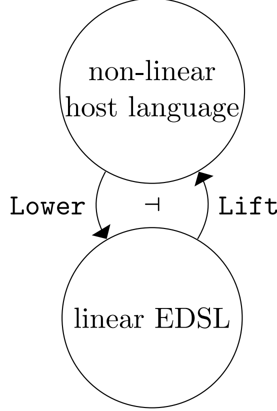


Figure 1.1: The linear/non-linear embedded programming model.

for numbers and booleans, our linear EDSL should be able to take advantage of them! For example, we can imagine that instead of being indexed by linear types, mutable references could instead be indexed by host language types. The program `centerAt` could then be implemented using built-in if statements and integers, which would be better for both implementers and users of the language.

However, it is not immediately clear whether such an abstraction makes sense. Consider the interface to mutable references that hold host-language data, denoted by the type α:

```
alloc   : α → LExp ∅ (LRef α)
dealloc : LExp Δ (LRef α) → LExp Δ Unit
lookup  : LExp Δ (LRef α) → LExp Δ (?? ⊗ LRef α)
assign  : LExp Δ (LRef α) → α → LExp Δ (LRef α)
```

The types of `alloc`, `dealloc` and `assign` are all straightforward, where `alloc` and `assign` allow the user to provide an ordinary host-language value to be stored in the reference cell. However, it is not clear what the type of `lookup` should be. On the one hand, `lookup` must return a linear expression or else we would not be able to enforce linearity at the top level. On the other hand, we want the output of `lookup` to be a host-language value, as it is used in `centerAt`.

The goals of linear DSLs and embedded DSLs seem to be at odds here. But with a small change in perspective, these two approaches can be reconciled.

Whereas traditional linear type systems treat data as linear unless marked with the type $!\sigma$, Benton's linear/non-linear (LNL) logic puts linear and non-linear data on equal ground and provides a simple interface, illustrated in Figure 1.1, to relate the two (Benton, 1995). This interface includes, for every non-linear type α, a linear type `Lower` α; and for every linear type σ, a non-linear type `Lift` σ.

Although the linear/non-linear model has been widely accepted as a semantic foundation of linear type systems (Melliès, 2003), it has had limited impact as a programming paradigm. For example, it is common knowledge that the LNL model gives rise to a monad, but how does this monad integrate with modern monadic programming techniques? Krishnaswami *et al.* (2015) use an LNL type system to integrate linear and dependent types, since

dependent types can only depend on non-linear values, but how does this system compare to modern dependently-typed languages?

In this work I propose that Benton's linear/non-linear interface exactly describes the relationship between embedded and host-language data in a linear EDSL. For example, the `lookup` operation should naturally return a `Lower`ed value.

```
lookup : LExp Δ (LRef α) → LExp Δ (Lower α ⊗ LRef α)
```

The type system that arises from this proposal is expressive and has practical applications for a number of different linear domains and host languages. But the embedded LNL type system is more than just a programming model—it is also a powerful framework for meta-programming and meta-reasoning about linear EDSLs. For example, if the host language supports monadic programming, we can define monadic wrappers around domain-specific linear operations. If the host language has dependent types, then the linear language inherits a limited form of dependent types for free. If the host language has support for theorem proving, then it can be used to formalize the meta-theory of the linear language, taking advantage of existing results about non-linear data used in the linear EDSL.

Thesis Statement. *Linear/non-linear logic is a simple and powerful programming model for linear embedded domain-specific languages. Embedded LNL type systems come with a rich and elegant meta-theory, have practical applications in a variety of linear domains and host languages, and facilitates powerful embedded meta-theory.*

To support its thesis, this dissertation makes the following contributions:

- Chapter 2 contains a tutorial and survey of linear types with a focus on different possible formulations of non-linear types in a linear type system.

- Chapter 3 develops the meta-theory of embedded LNL, illustrates the resulting language with examples, and exposes connections with monadic programming techniques.

- Chapter 4 presents the implementation of linear/non-linear EDSLs in Haskell, to demonstrate the LNL programming model in practice. The Haskell implementation is a general framework that can be instantiated with many different domain-specific languages, and the chapter presents two particular examples—linear arrays and concurrent session types.

  The meta-theory and examples of Chapters 3 and 4 were originally presented in the context of the Haskell implementation at the 2017 Haskell Symposium, in *The linearity monad* (Paykin and Zdancewic, 2017).

- Chapter 5 presents the category theory of linear/non-linear type systems and establishes a categorical semantics of embedded LNL using the host language as a meta-theory. The embedded meta-theory acts as a sanity check to ensure that the embedded LNL framework is sound and accurately represents Benton's original LNL model.

- Starting in Chapter 6, the dissertation focuses on a larger case study that uses embedded LNL for quantum computation. Chapter 6 describes a quantum/non-quantum (QNQ) term calculus, gives examples of quantum programming with dependent types, and develops the meta-theory of QNQ, focusing on its denotational semantics.

- Chapter 7 develops an equational theory for QNQ using homotopy type theory as a host language. The embedding encodes components of the embedded language (specifically, unitary transformations) in a higher inductive type—a feature unique to homotopy type theory. This encoding simplifies the resulting equational theory, and shows how features of the host language can directly benefit the design of the embedded language.
- Finally, Chapter 8 describes a variation of QNQ—an embedded quantum circuit language called $\mathcal{Q}$WIRE. Implemented in Coq, $\mathcal{Q}$WIRE relies heavily on the rich language features of its host language, including dependent types, to facilitate type checking.

  $\mathcal{Q}$WIRE was developed in conjunction with Robert Rand and Steve Zdancewic, and was originally presented in the proceedings of POPL 2017 as *$\mathcal{Q}$WIRE: A core language for quantum circuits* (Paykin *et al.*, 2017). The surface language described in Section 8.3 is new to this dissertation, however. The implementation in Coq was also developed later, and described in part in *$\mathcal{Q}$WIRE practice: Formal verification of quantum circuits in Coq* (Rand *et al.*, 2017); the formal verification aspects of the $\mathcal{Q}$WIRE project are not a contribution of this dissertation.

1.1 Conventions

This dissertation uses dependent types to define linear typing judgments and reason about the meta-theory of such type systems. To do this effectively we assume basic familiarity with dependent type theory such as Π types and Σ types for universal and existential quantification respectively. We also assume familiarity with inductively defined relations and predicates such as one would find in Coq or Agda. For more background, we refer the reader to Aspinall and Hofmann (2005).

In general, our use of dependent types is informal and language-agnostic, with the understanding that the reasoning principles are valid in a range of dependently typed languages. In Chapter 4 we use Haskell as a host language, and in Chapter 8 we use Coq; in those chapters we expect some basic familiarity with the languages but introduce advanced features as needed. In Chapter 7 we use homotopy type theory as a host language and provide the relevant background in that chapter.

The syntax we use in the language-agnostic chapters is loosely inspired by Haskell. We define functions by pattern matching over their arguments, and give type declarations above their definitions, as in:

```
isEven : Nat → Bool
isEven 0     ≡ true
isEven 1     ≡ false
isEven (n+2) ≡ isEven n
```

We write $x \equiv y$ to define x as y, and write $x = y$ for the proposition that x is equal to y.

Anonymous functions in the host language are written $\lambda a.b$ and application $a a'$. In contrast, functions in the embedded linear language are written $\hat{\lambda} x.e$ and application $e\,\hat{}\,e'$.

We write `Type` for the kind of host-language types, and we define inductive predicates and relations with the keyword `data` as follows:

```
data IsEven : Nat → Type where
  even0 : IsEven 0
  even2 : Π n, IsEven n → IsEven (n+2)
```

CHAPTER 2

Linear type systems

What features should we expect of a linear EDSL? In this chapter we review the basics of linear type systems, including the standard linear connectives for functions, pairs, and sums. In addition, we survey different ways to integrate non-linear data into a linear type system, starting with the traditional ! modality and moving through to linear/non-linear logic. Our goal is to identify how well different presentations work for linear EDSLs.

Linear logic is often referred to as a logic of resources (Girard, 1987). Linear type systems are used to reason *about* system resources like memory, locks, time, *etc.*, but they also treat linear variables as if they *are* consumable resources. This means that when a variable is used in a linear program, the resource it is associated with gets used up, and is no longer accessible to the rest of the program.

The interpretation of linear variables as resources is characterized by two structural rules:

1. Linear resources cannot be duplicated.

2. Linear resources cannot be discarded.

Type systems limited by such structural rules are called *substructural type systems.* If resources can be duplicated but not discarded, the system is called *relevant*, and if resources can be discarded but not duplicated, the system is called *affine.* Other structural rules can also be added; if resources can be neither duplicated, discarded, nor reordered in a context, then the system is called *ordered.* This work focuses on linear type systems, but many of the results are applicable to different substructural systems as well.

2.1 A simple linear type system

Consider a linear typing judgment of the form $\Delta \vdash_\ell e : \sigma$, where σ is a linear type and Δ is a typing context $x_1 : \sigma_n, \ldots, x_n : \sigma_n$ mapping linear variables x_i to types σ_i. The subscript ℓ is syntax to distinguish it from other typing judgments that will appear later. Intuitively, we can think of e as a *computation* that uses exactly the resources x_i to produce a result of type σ.

The fact that resources cannot be discarded means that every resource in a linear typing context Δ must be used at some point in the expression. Consider the following typing rule for variables, which says that the only resource being consumed is the variable x itself.

$$\frac{\Delta = x : \sigma}{\Delta \vdash_\ell x : \sigma}\ \textsc{var}$$

The fact that resources cannot be duplicated means that resources used in one part of a program cannot be used in another. For example, consider a `let` binding:

$$\frac{\Delta_1 \vdash_\ell e : \sigma \qquad \Delta_2, x : \sigma \vdash_\ell e' : \tau \qquad \Delta_1 \perp \Delta_2}{\Delta_1, \Delta_2 \vdash_\ell \texttt{let } x \coloneqq e \texttt{ in } e' : \tau}\ \text{LET}$$

The judgment $\Delta_1 \perp \Delta_2$ says that the domains of Δ_1 and Δ_2 are disjoint, meaning that e and e' draw on disjoint sets of resources. In general, (Δ_1, Δ_2) is only defined when $\Delta_1 \perp \Delta_2$.

$$\frac{}{\varnothing \perp \Delta_2} \qquad \frac{\Delta_1 \perp \Delta_2 \qquad x \notin \texttt{dom}(\Delta_2)}{(\Delta_1, x : \sigma) \perp \Delta_2}$$

The operational semantics of this linear type system, much like non-linear type systems, can be described using α-, β-, and η-equivalences. The α-equivalence rule for `let` bindings says that bound variables can be renamed inside an expression.

$$\texttt{let } x \coloneqq e \texttt{ in } e' \sim_\alpha \texttt{let } y \coloneqq e \texttt{ in } e'\{y/x\}$$

We write $e'\{e/x\}$ for the usual capture-avoiding substitution of e for x in e'. In the rest of this dissertation we omit α-equivalences, as they are completely standard.

Evaluation order is independent of linearity, so we can consider both call-by-value and call-by-name operational semantics. Evaluation contexts E^{V} and E^{N}, respectively for call-by-value and call-by-name, dictate where β-reductions occur in a term.

$$\frac{e \rightsquigarrow_{\text{V}} e'}{E^{\text{V}}[e] \rightsquigarrow_{\text{V}} E^{\text{V}}[e']} \qquad \frac{e \rightsquigarrow_{\text{N}} e'}{E^{\text{N}}[e] \rightsquigarrow_{\text{N}} E^{\text{N}}[e']}$$

An evaluation context is a term with a hole in it; we write $E[e]$ for the term obtained by filling the hole with e. Evaluation contexts reduce the bodies of `let` bindings for call-by-value contexts, but not for call-by-name.

$$E^{\text{V}} ::= \square \mid \texttt{let } x \coloneqq E^{\text{V}} \texttt{ in } e' \qquad \texttt{let } x \coloneqq v^{\text{V}} \texttt{ in } e' \rightsquigarrow_{\text{V}} e'\{v/x\}$$
$$E^{\text{N}} ::= \square \qquad \texttt{let } x \coloneqq e \texttt{ in } e' \rightsquigarrow_{\text{N}} e'\{e/x\}$$

Eta-equivalences for `let` bindings typically say that any expression e is equivalent to a `let` binding `let` $x \coloneqq e$ `in` x. However, such a rule can be generalized so that, if e is an expression that occurs linearly in another term e', then $e'\{e/x\}$ is equivalent to `let` $x \coloneqq e$ `in` e'. This rule both introduces a let binding and also commutes the `let` binding to the front of a term. Such rules are also called *commuting conversions* in literature surrounding proof theory (Girard *et al.*, 1989).

$$\frac{\Delta \vdash e : \sigma \qquad \Delta', x : \sigma \vdash e' : \tau \qquad \Delta \perp \Delta'}{e'\{e/x\} \sim_\eta \texttt{let } x \coloneqq e \texttt{ in } e'}$$

We write $e_1 \sim e_2$ for the smallest congruence containing α, β, and η equivalences.

$$\sigma, \tau ::= \cdots \mid \sigma \multimap \tau \qquad\qquad \frac{\Delta \vdash_\ell e : \sigma \multimap \tau}{e \sim_\eta \hat{\lambda} x.ex}$$

$$e ::= \cdots \mid \hat{\lambda} x.e \mid e \,\hat{}\, e'$$

$$\frac{\Delta, x : \sigma \vdash_\ell e : \tau \qquad \Delta \bot x : \sigma}{\Delta \vdash_\ell \hat{\lambda} x.e : \sigma \multimap \tau} \multimap\text{-I} \qquad \frac{\Delta \vdash_\ell e : \sigma \multimap \tau \qquad \Delta' \vdash_\ell e' : \sigma \qquad \Delta \bot \Delta'}{\Delta, \Delta' \vdash_\ell e \,\hat{}\, e' : \tau} \multimap\text{-E}$$

$$E^{\mathrm{V}} ::= \cdots \mid E^{\mathrm{V}} \,\hat{}\, e' \mid v \,\hat{}\, E^{\mathrm{V}} \qquad v^{\mathrm{V}} ::= \cdots \mid \hat{\lambda} x.e \qquad (\hat{\lambda} x.e') \,\hat{}\, v \rightsquigarrow_{\mathrm{V}} e'\{v/x\}$$

$$E^{\mathrm{N}} ::= \cdots \mid E^{\mathrm{N}} e' \qquad v^{\mathrm{N}} ::= \cdots \mid \hat{\lambda} x.e \qquad (\hat{\lambda} x.e') \,\hat{}\, e \rightsquigarrow_{\mathrm{N}} e'\{e/x\}$$

Figure 2.1: Linear implication

2.2 Linear connectives

Linear type systems are not expressive if they only contain constants and `let` bindings, and they usually come with a variety of standard connectives.

Linear functions. The type of linear functions is written $\sigma \multimap \tau$. Figure 2.1 summarizes the syntax, typing rules, and operational semantics of linear functions. The typing rules for abstraction $\hat{\lambda}.e$ and application $e \,\hat{}\, e'$ ensure that the resources used to produce a function and its argument are disjoint. The β and η rules are identical to those of the simply-typed lambda calculus. A lambda closure $\hat{\lambda} x.e$ is a value in both the call-by-name and call-by-value fragments, but call-by-value evaluation contexts evaluate the argument to a function before taking a β-step. The η-equivalence rule says that every linear function e is equivalent to $\hat{\lambda} x.e \,\hat{}\, x$.

Multiplicative product. The multiplicative product, also called tensor product and written $\otimes$, is linear in the sense that the two components of the pair cannot use any shared resources. The fragment of the type system with the multiplicative product is shown in Figure 2.2.

Unlike the non-linear/Cartesian product, the multiplicative product cannot be eliminated using projections $\pi_i : \sigma_1 \otimes \sigma_2 \multimap \sigma_i$, because such a projection uses only half of the resources of the original pair. Instead, the multiplicative product can be eliminated by a `let` binding, written $\texttt{let}\ (x_1, x_2) \coloneqq e\ \texttt{in}\ e'$, where the two components of the pair are bound to variables x_1 and x_2.

The η-equivalence rule says that, if e is an expression of type $\sigma_1 \otimes \sigma_2$ that occurs linearly in a larger term e', then $e'\{e/x\}$ is equivalent to $\texttt{let}\ (x_1, x_2) \coloneqq e\ \texttt{in}\ e'\{(x_1, x_2)/x\}$. That is, a `let` binding can always be commuted to the front of a term.

Linear functions can be curried and uncurried with respect to the multiplicative product:

$$\texttt{curry} : (\sigma_1 \otimes \sigma_2 \multimap \tau) \multimap \sigma_1 \multimap \sigma_2 \multimap \tau \qquad \texttt{uncurry} : (\sigma_1 \multimap \sigma_2 \multimap \tau) \multimap \sigma_1 \otimes \sigma_2 \multimap \tau$$

$$\texttt{curry} \equiv \hat{\lambda} f.\hat{\lambda} x_1.\hat{\lambda} x_2.f(x_1, x_2) \qquad \texttt{uncurry} \equiv \hat{\lambda} f.\hat{\lambda} x.\texttt{let}\ (x_1, x_2) \coloneqq x\ \texttt{in}\ f\ x_1\ x_2$$

Multiplicative unit. The multiplicative unit is written `LUnit` or sometimes 1; the fragment corresponding to this type is shown in Figure 2.4. Notice that the call-by-value and call-by-name rules are identical. `LUnit` is a unit of $\otimes$ in the sense that $\texttt{LUnit} \otimes \sigma$ (and also $\sigma \otimes \texttt{LUnit}$) is isomorphic to σ—there are morphisms between the two types that compose

$$\sigma, \tau ::= \cdots \mid \sigma \otimes \tau$$

$$e ::= \cdots \mid (e_1, e_2) \mid \texttt{let } (x_1, x_2) \coloneqq e \texttt{ in } e'$$

$$\frac{\Delta \vdash_\ell e : \sigma_1 \otimes \sigma_2 \qquad \Delta', x : \sigma_1 \otimes \sigma_2 \vdash_\ell e' : \tau \qquad \Delta \bot \Delta'}{e'\{e/x\} \sim_\eta \texttt{let } (x_1, x_2) \coloneqq e \texttt{ in } e'\{(x_1, x_2)/x\}}$$

$$\frac{\Delta_1 \vdash_\ell e_1 : \sigma_1 \qquad \Delta_2 \vdash_\ell e_2 : \sigma_2 \qquad \Delta_1 \bot \Delta_2}{\Delta_1, \Delta_2 \vdash_\ell (e_1, e_2) : \sigma_1 \otimes \sigma_2} \otimes\text{-I}$$

$$\frac{\Delta \vdash_\ell e : \sigma_1 \otimes \sigma_2 \qquad \Delta', x_1 : \sigma_1, x_2 : \sigma_2 \vdash_\ell e' : \tau \qquad \Delta \bot \Delta'}{\Delta, \Delta' \vdash_\ell \texttt{let } (x_1, x_2) \coloneqq e \texttt{ in } e' : \tau} \otimes\text{-E}$$

$$E^{\mathrm{V}} ::= \cdots \mid (E^{\mathrm{V}}, e_2) \mid (v_1, E^{\mathrm{V}}) \mid \texttt{let } (x_1, x_2) \coloneqq E^{\mathrm{V}} \texttt{ in } e' \qquad v^{\mathrm{V}} ::= \cdots \mid (v_1^{\mathrm{V}}, v_2^{\mathrm{V}})$$

$$E^{\mathrm{N}} ::= \cdots \mid \texttt{let } (x_1, x_2) \coloneqq E^{\mathrm{N}} \texttt{ in } e' \qquad v^{\mathrm{N}} ::= \cdots \mid (e_1, e_2)$$

$$\texttt{let } (x_1, x_2) \coloneqq (v_1, v_2) \texttt{ in } e' \rightsquigarrow_{\mathrm{V}} e'\{v_1/x_1, v_2/x_2\}$$

$$\texttt{let } (x_1, x_2) \coloneqq (e_1, e_2) \texttt{ in } e' \rightsquigarrow_{\mathrm{N}} e'\{e_1/x_1, e_2/x_2\}$$

Figure 2.2: Multiplicative product $\otimes$

to the identity:

$$\texttt{lunit}_\otimes : \sigma \multimap \sigma \otimes \texttt{LUnit} \qquad \texttt{lunit}'_\otimes : \sigma \otimes \texttt{LUnit} \multimap \sigma$$

$$\texttt{lunit}_\otimes \equiv \hat{\lambda} x.(x, ()) \qquad \texttt{lunit}'_\otimes \equiv \hat{\lambda} z.\texttt{let } (x, y) \coloneqq z \texttt{ in let } () \coloneqq y \texttt{ in } x$$

The proofs that $\texttt{lunit}_\otimes \circ \texttt{lunit}'_\otimes$ and $\texttt{lunit}'_\otimes \circ \texttt{lunit}_\otimes$ are identity functions are shown in Figure 2.3.

The definition of $\texttt{lunit}'_\otimes$ here is overly verbose; informally we write $\hat{\lambda}(x, ()).x$.

Additive product. Linear type systems often contain two different sorts of products. The *additive product*, written $\sigma \,\&\, \tau$ and pronounced "σ with τ," corresponds more closely with the non-linear/Cartesian product. Given a computation e of type $\sigma_1 \,\&\, \sigma_2$, the user can choose to use the first component or the second component, but not both, via projection. This means that to construct an additive pair, the two components of the pair, $[e_1, e_2]$, must use exactly the same resources. Intuitively, a computation of type $\sigma_1 \,\&\, \sigma_2$ provides a *choice* of either σ_1 or σ_2. This implies that $[e_1, e_2]$ is always a value; its components should not be evaluated until a choice is made. The typing and evaluation rules for the additive product are shown in Figure 2.5.

The unit of the additive product, `LTop`, can be interpreted as a computation throwing an error. Its rules are shown in Figure 2.6. The error computation is valid under any collection of resources: $\Delta \vdash_\ell \texttt{error} : \texttt{LTop}$ for any typing judgment Δ. However, there is no elimination form for `LTop`. Even so, $\texttt{LTop} \,\&\, \sigma$ is isomorphic to σ.

$$\texttt{lunit}_\& : \sigma \multimap \sigma \,\&\, \texttt{LTop} \qquad \texttt{lunit}'_\& : \sigma \,\&\, \texttt{LTop} \multimap \sigma$$

$$\texttt{lunit}_\& \equiv \hat{\lambda} x.[x, \texttt{error}] \qquad \texttt{lunit}'_\& \equiv \hat{\lambda} x.\pi_1 x$$

$$
\begin{aligned}
\texttt{lunit}_\otimes \circ \texttt{lunit}'_\otimes &\sim_\eta \hat\lambda z.\texttt{lunit}_\otimes(\texttt{lunit}'_\otimes\ z) \\
&\sim_\beta \hat\lambda z.(\texttt{lunit}'_\otimes\ z, ()) \\
&\sim_\beta \hat\lambda z.(\texttt{let}\ (x,y) \coloneqq z\ \texttt{in}\ \texttt{let}\ () \coloneqq y\ \texttt{in}\ x, ()) \\
&\sim_\eta \hat\lambda z.\texttt{let}\ (x,y) \coloneqq z\ \texttt{in}\ (\texttt{let}\ () \coloneqq y\ \texttt{in}\ x, ()) \\
&\sim_\eta \hat\lambda z.\texttt{let}\ (x,y) \coloneqq z\ \texttt{in}\ \texttt{let}\ () \coloneqq y\ \texttt{in}\ (x, ()) \\
&\sim_\eta \hat\lambda z.\texttt{let}\ (x,y) \coloneqq z\ \texttt{in}\ (x,y) \\
&\sim_\eta \hat\lambda z.z
\end{aligned}
$$

$$
\begin{aligned}
\texttt{lunit}'_\otimes \circ \texttt{lunit}_\otimes &\sim_\eta \hat\lambda x.\texttt{lunit}'_\otimes(\texttt{lunit}_\otimes x) \\
&\sim_\beta \hat\lambda x.\texttt{lunit}'_\otimes(x, ()) \\
&\sim_\beta \hat\lambda x.\texttt{let}\ (x,y) \coloneqq (x, ())\ \texttt{in}\ \texttt{let}\ () \coloneqq y\ \texttt{in}\ x \\
&\sim_\beta \hat\lambda x.\texttt{let}\ () \coloneqq ()\ \texttt{in}\ x \\
&\sim_\beta \hat\lambda x.x
\end{aligned}
$$

Figure 2.3: Proof that `LUnit` is the unit of $\otimes$.

$$
\begin{aligned}
\sigma, \tau &::= \cdots \mid \texttt{LUnit} \\
e &::= \cdots \mid () \mid \texttt{let}\ () \coloneqq e\ \texttt{in}\ e'
\end{aligned}
$$

$$
\frac{\Delta \vdash_\ell e : \texttt{LUnit} \qquad \Delta' \vdash_\ell e' : \tau \qquad \Delta \perp \Delta'}{e'\{e/x\} \sim_\eta \texttt{let}\ () \coloneqq e\ \texttt{in}\ e'\{()/x\}}
$$

$$
\frac{}{\varnothing \vdash_\ell () : \texttt{LUnit}}\ \texttt{LUnit-I}
$$

$$
\frac{\Delta \vdash_\ell e : \texttt{LUnit} \qquad \Delta' \vdash_\ell e' : \tau \qquad \Delta \perp \Delta'}{\Delta, \Delta' \vdash_\ell \texttt{let}\ () \coloneqq e\ \texttt{in}\ e' : \tau}\ \texttt{LUnit-E}
$$

$$
\begin{array}{lll}
E^{\mathrm{V}} ::= \cdots \mid \texttt{let}\ () \coloneqq E^{\mathrm{V}}\ \texttt{in}\ e' & v^{\mathrm{V}} ::= \cdots \mid () & \texttt{let}\ () \coloneqq ()\ \texttt{in}\ e' \rightsquigarrow_{\mathrm{V}} e' \\
E^{\mathrm{N}} ::= \cdots \mid \texttt{let}\ () \coloneqq E^{\mathrm{N}}\ \texttt{in}\ e' & v^{\mathrm{N}} ::= \cdots \mid () & \texttt{let}\ () \coloneqq ()\ \texttt{in}\ e' \rightsquigarrow_{\mathrm{N}} e'
\end{array}
$$

Figure 2.4: Multiplicative unit `LUnit`

$$\begin{aligned} \sigma, \tau &::= \cdots \mid \sigma \mathbin{\&} \tau \\ e &::= \cdots \mid [e_1, e_2] \mid \pi_1 e \mid \pi_2 e \\ v^{\mathrm{V}}, v^{\mathrm{N}} &::= \cdots \mid [e_1, e_2] \end{aligned}$$

$$\frac{\Delta \vdash_\ell e : \sigma_1 \mathbin{\&} \sigma_2}{e \sim_\eta [\pi_1 e, \pi_2 e]}$$

$$\frac{\Delta \vdash_\ell e_1 : \sigma_1 \qquad \Delta \vdash_\ell e_2 : \sigma_2 \qquad \Delta_1 \bot \Delta_2}{\Delta \vdash_\ell (e_1, e_2) : \sigma_1 \mathbin{\&} \sigma_2}\ \&\text{-I} \qquad \frac{\Delta \vdash_\ell e : \sigma_1 \mathbin{\&} \sigma_2}{\Delta \vdash_\ell \pi_1 e : \sigma_1}\ \&\text{-E}_1 \qquad \frac{\Delta \vdash_\ell e : \sigma_1 \mathbin{\&} \sigma_2}{\Delta \vdash_\ell \pi_2 e : \sigma_2}\ \&\text{-E}_2$$

$$\begin{array}{lll} E^{\mathrm{V}} ::= \cdots \mid \pi_i E^{\mathrm{V}} & v^{\mathrm{V}} ::= \cdots \mid [e_1, e_2] & \pi_i [e_1, e_2] \rightsquigarrow_{\mathrm{V}} e_i \\ E^{\mathrm{N}} ::= \cdots \mid \pi_i E^{\mathrm{N}} & v^{\mathrm{N}} ::= \cdots \mid [e_1, e_2] & \pi_i [e_1, e_2] \rightsquigarrow_{\mathrm{N}} e_i \end{array}$$

Figure 2.5: Additive product

$$\begin{aligned} \sigma, \tau &::= \cdots \mid \mathtt{LTop} \\ e &::= \cdots \mid \mathtt{error} \end{aligned} \qquad \frac{\Delta \vdash_\ell e : \mathtt{LTop}}{e \sim_\eta \mathtt{error}}$$

$$\frac{}{\Delta \vdash_\ell \mathtt{error} : \mathtt{LTop}}\ \text{LTop-I} \qquad v^{\mathrm{V}}, v^{\mathrm{N}} ::= \cdots \mid \mathtt{error}$$

Figure 2.6: Additive unit: `LTop`

There are no β rules for `LTop`, but there is an η equivalence: every computation of type `LTop` is equivalent to the error computation `error`.

Additive sum. A computation of type $\sigma \oplus \tau$ is either a computation of type σ or a computation of type τ; unlike the additive product, the introduction rules dictate which of σ or τ to provide. To eliminate a sum type, a user must be prepared to accept either result using case analysis. The rules for $\oplus$ are shown in Figure 2.7.

The unit of $\oplus$ is the linear void type, written `LZero` and shown in Figure 2.8. Like the non-linear void type, there are no constructors of type `LZero`, and having a term of type `LZero` is a contradiction, so it can be used to derive any type. In particular, given a computation $\Delta \vdash_\ell e : \mathtt{LZero}$, the computation $\mathtt{case}\ e\ \mathtt{of}\ ()$ can be given any type τ. Furthermore, $\mathtt{case}\ e\ \mathtt{of}\ ()$ vacuously uses resources not used by e itself.

$$\begin{array}{ll} \mathtt{lunit}_\oplus : \sigma \multimap \sigma \oplus \mathtt{LZero} & \mathtt{lunit}'_\oplus : \sigma \oplus \mathtt{LZero} \multimap \sigma \\ \mathtt{lunit}_\oplus \equiv \hat{\lambda} x. \iota_1 x & \mathtt{lunit}'_\oplus \equiv \hat{\lambda} z. \mathtt{case}\ z\ \mathtt{of}\ (\iota_1 x \rightarrow x \mid \iota_2 y \rightarrow \mathtt{case}\ y\ \mathtt{of}\ ()) \end{array}$$

2.3 The exponential modality !

As we argued in the introduction, it is not enough to just have linear resources; many domains naturally mix linear resources with non-linear, unrestricted data. Traditionally, linear logic accounts for unrestricted data with the ! modality (pronounced “bang”). Unlike

$$\sigma, \tau ::= \cdots \mid \sigma \oplus \tau$$
$$e ::= \cdots \mid \iota_1 e \mid \iota_2 e \mid \texttt{case}\ e\ \texttt{of}\ (\iota_1 x_1 \to e_1 \mid \iota_2 x_2 \to e_2)$$

$$\frac{\Delta \vdash_\ell e : \sigma_1 \oplus \sigma_2 \qquad \Delta', x : \sigma_1 \oplus \sigma_2 \vdash_\ell e' : \tau}{e'\{e/x\} \sim_\eta \texttt{case}\ e\ \texttt{of}\ (\iota_1 x_1 \to e'\{\iota_1 x_1/x\} \mid \iota_2 x_2 \to e'\{\iota_2 x_2/x\})}$$

$$\frac{\Delta_1 \vdash_\ell e_1 : \sigma_1}{\Delta_1, \Delta_2 \vdash_\ell \iota_1 : \sigma_1 \oplus \sigma_2}\ \oplus\text{-I}_1 \qquad \frac{\Delta_1 \vdash_\ell e_2 : \sigma_2}{\Delta_1, \Delta_2 \vdash_\ell \iota_2 : \sigma_1 \oplus \sigma_2}\ \oplus\text{-I}_2$$

$$\frac{\Delta \vdash_\ell e : \sigma_1 \oplus \sigma_2 \qquad \Delta', x_1 : \sigma_1 \vdash_\ell e_1 : \tau \qquad \Delta', x_2 : \sigma_2 \vdash_\ell e_2 : \tau \qquad \Delta \bot \Delta'}{\Delta, \Delta' \vdash_\ell \texttt{case}\ e\ \texttt{of}\ (\iota_1 x_1 \to e_1 \mid \iota_2 x_2 \to e_2) : \tau}\ \oplus\text{-E}$$

$$E^{\mathrm{V}} ::= \cdots \mid \iota_i E^{\mathrm{V}} \mid \texttt{case}\ E^{\mathrm{V}}\ \texttt{of}\ (\iota_1 x_1 \to e_1 \mid \iota_2 x_2 \to e_2) \qquad v^{\mathrm{V}} ::= \cdots \mid \iota_i v^{\mathrm{V}}$$
$$E^{\mathrm{N}} ::= \cdots \mid \texttt{case}\ E^{\mathrm{N}}\ \texttt{of}\ (\iota_1 x_1 \to e_1 \mid \iota_2 x_2 \to e_2) \qquad v^{\mathrm{N}} ::= \cdots \mid \iota_i e$$

$$\texttt{case}\ \iota_i v_i\ \texttt{of}\ (\iota_1 x_1 \to e_1 \mid \iota_2 x_2 \to e_2) \rightsquigarrow_{\mathrm{V}} e_i\{v_i/x_i\}$$
$$\texttt{case}\ \iota_i e_i\ \texttt{of}\ (\iota_1 x_1 \to e_1 \mid \iota_2 x_2 \to e_2) \rightsquigarrow_{\mathrm{N}} e_i\{e_i/x_i\}$$

Figure 2.7: Additive sum

$$\sigma, \tau ::= \cdots \mid \texttt{LZero}$$
$$e ::= \cdots \mid \texttt{case}\ e\ \texttt{of}\ ()$$

$$\frac{\Delta \vdash_\ell e : \texttt{LZero} \qquad \Delta', x : \texttt{LZero} \vdash e' : \tau \qquad \Delta \bot \Delta'}{e'\{e/x\} \sim_\eta \texttt{case}\ e\ \texttt{of}\ ()}$$

$$\frac{\Delta \vdash_\ell e : \texttt{LZero} \qquad \Delta \bot \Delta'}{\Delta, \Delta' \vdash_\ell \texttt{case}\ e\ \texttt{of}\ () : \tau}\ \textsc{LZero-E}$$

$$E^{\mathrm{V}} ::= \cdots \mid \texttt{case}\ E^{\mathrm{V}}\ \texttt{of}\ ()$$
$$E^{\mathrm{N}} ::= \cdots \mid \texttt{case}\ E^{\mathrm{N}}\ \texttt{of}\ ()$$

Figure 2.8: Additive unit: `LZero`

with linear resources, it *is* possible to duplicate and discard unrestricted data.

$$\texttt{duplicate} : !\sigma \multimap !\sigma \otimes !\sigma \qquad \texttt{discard} : !\sigma \multimap \texttt{LUnit}$$

One interpretation of the ! modality says that an expression of type $!\sigma$ can be thought of as a *suspended computation* that can be executed an arbitrary number of times.

The treatment of ! is one of the most important and delicate components in the design of a linear type system. It is important because the use of ! affects the usability of the linear system, and it is delicate because it is easy to get wrong. In fact, we start by presenting a simple but *unsound* version of !, originally popularized by Abramsky (1993).

Consider a linear computation $\varnothing \vdash_\ell e : \sigma$ that does not use any linear resources. This computation can be executed an arbitrary number of times, because each execution does not use up any resources. We denote such a suspended computation as $\varnothing \vdash_\ell !e : !\sigma$; this operation is called *promotion*.

More generally, if $x_1 : !\sigma_1, \ldots, x_n : !\sigma_n \vdash e : \tau$ uses resources that can themselves be duplicated, then e can be promoted. Every time e is executed, it will use up one copy of each of its duplicable resources. Forcing the suspended computation $!e$ executes the underlying computation, and is called *dereliction*.

$$\frac{!\Delta \vdash_\ell e : \sigma}{!\Delta \vdash_\ell !e : !\sigma}\ \textsc{promotion} \qquad \frac{\Delta \vdash_\ell e : !\sigma}{\Delta \vdash_\ell \texttt{derelict}\ e : \sigma}\ \textsc{dereliction}$$

Here, we write $!\Delta$ to refer to a context of the form $x_1 : !\sigma_1, \ldots, x_n : !\sigma_n$

Unrestricted resources are implicitly subject to the structural rules disallowed for plain linear types—unrestricted resources can be duplicated (also called *contraction*) and dropped (also called *weakening*).

$$\frac{\Delta', x : !\sigma, y : !\sigma \vdash_\ell e' : \tau}{\Delta, z : !\sigma, \Delta' \vdash_\ell e'\{z/x, z/y\} : \tau}\ \textsc{contraction} \qquad \frac{\Delta, \Delta' \vdash_\ell e : \tau}{\Delta, x : !\sigma, \Delta' \vdash_\ell e : \tau}\ \textsc{weakening}$$

Since we expect suspended computations of the form $!e$ to be evaluated many times, every suspended computation is a value, and we should never evaluate under a !. The β rule says that applying dereliction to a promoted expression $!e$ actually executes e.

$$\begin{array}{lll} E^{\mathrm{V}} ::= \cdots \mid \texttt{derelict}\ E^{\mathrm{V}} & v^{\mathrm{V}} ::= \cdots \mid !e & \texttt{derelict}(!e) \rightsquigarrow_{\mathrm{V}} e \\ E^{\mathrm{N}} ::= \cdots \mid \texttt{derelict}\ E^{\mathrm{N}} & v^{\mathrm{N}} ::= \cdots \mid !e & \texttt{derelict}(!e) \rightsquigarrow_{\mathrm{N}} e \end{array}$$

The η rule for $!\sigma$ says that every computation of $!\sigma$ is equivalent to a suspended computation.

$$\frac{\Delta \vdash_\ell e : !\sigma}{e \sim_\eta !(\texttt{derelict}\ e)}$$

A refinement of !. Abramsky's syntax above gives a good first approximation of !, and closely corresponds to Girard's original presentation as a logical system. However, Abramsky's syntax has two serious problems.

First, Abramsky's syntax is inconsistent with the substitution property (Wadler, 1992).

Consider $\Delta \vdash e : !\sigma$ where Δ may not necessarily have the form $!\Delta$—for example, $x : \texttt{LUnit} \vdash \texttt{let}\ () \coloneqq x\ \texttt{in}\ !() : !\texttt{LUnit}$. In addition, notice that a variable y of type $!\texttt{LUnit}$ *can* be promoted to $!y : !!\texttt{LUnit}$. However, the result of substituting $\texttt{let}\ () \coloneqq x\ \texttt{in}\ !()$ for y in $!y$ is *not* well-typed:

$$x : \texttt{LUnit} \nvdash_{\ell} !(\texttt{let}\ () \coloneqq x\ \texttt{in}\ !()) : !!\texttt{LUnit}$$

Benton *et al.* (1993) presented a variation of Abramsky's syntax that solves the substitution problem and soon became standard. The main difference from Abramsky's syntax is that Benton *et al.*'s syntax requires promotion to explicitly capture all the unrestricted resources being used.

$$\frac{\Delta_i \vdash e_i : !\sigma_i \qquad x_1 : !\sigma_1, \ldots, x_n : !\sigma_2 \vdash e : \tau}{\Delta_1, \ldots, \Delta_n \vdash \texttt{promote}\ \{e_i\ \texttt{as}\ x_i\}\ \texttt{in}\ e : !\tau}\ (\text{Benton } et\ al., 1993)$$

Benton *et al.*'s presentation restores the substitution principle by baking it into the promotion rule.

The second problem with both Abramsky's and also Benton *et al.*'s presentations is practical—it is inefficient and inconvenient to keep explicitly discarding and duplicating variables via the weakening and contraction rules. There must be a better way to program with non-linear data in a linear type theory!

Over the years, many styles have been proposed to deal with the problem of linear syntax, and the remainder of this chapter will highlight four of the most popular. For each, we also consider how well it models an *embedded* linear type system—whether an embedded presentation could use host-language data, libraries, and other tools for non-linear resources, instead of relying entirely on the embedding for manipulating non-linear data.

2.4 Dual Intuitionistic Linear Logic

Barber's Dual Intuitionistic Linear Logic (DILL) (1996) is based on the philosophy that linear and non-linear resources should be treated differently from each other. DILL's typing judgment has the form $\Theta; \Delta \vdash_{\text{D}} e : \sigma$, where σ is a linear type, and Θ and Δ are both typing contexts. Resources in Θ (on the left-hand-side of the semi-colon) are unrestricted in e, while resources in Δ are linear in e.

Since variables can be either linear or unrestricted, there are two ways to use variables in DILL. For a linear variable, it is not necessary to limit or keep track of the unrestricted resources, and for an unrestricted variable, it suffices to check there are no other linear resources.

$$\frac{x : \sigma \in \Theta}{\Theta; \varnothing \vdash_{\text{D}} x : \sigma}\ \text{DILL-NL-VAR} \qquad \frac{}{\Theta; x : \sigma \vdash_{\text{D}} x : \sigma}\ \text{DILL-L-VAR}$$

A suspended computation is one that uses no linear resources.

$$\frac{\Theta; \varnothing \vdash_{\text{D}} e : \sigma}{\Theta; \varnothing \vdash_{\text{D}} !e : !\sigma}\ \text{DILL-!-I}$$

The elimination rule for ! allows the result of a suspended computation to be bound to an

unrestricted variable.

$$\frac{\Theta; \Delta_1 \vdash_{\mathrm{D}} e : !\sigma \qquad \Theta, x : \sigma; \Delta_2 \vdash_{\mathrm{D}} e' : \tau \qquad \Delta_1 \perp \Delta_2}{\Theta; \Delta_1, \Delta_2 \vdash_{\mathrm{D}} \texttt{let } !x \coloneqq e \texttt{ in } e' : \tau} \text{ DILL-!-E}$$

Notice that the same unrestricted resources can be used in both e and e', even though their linear resources must be disjoint. This, combined with the intrinsic weakening of the unrestricted context Θ in the variable rules, means that the weakening and contraction structural rules need not be included explicitly; they can be derived from the remaining laws.

In fact, all of Abramsky's rules are derivable in this system. For example, Abramsky's dereliction operator can be derived as follows:

$$\frac{\Theta; \Delta \vdash_{\mathrm{D}} e : !\sigma}{\Theta; \Delta \vdash_{\mathrm{D}} \texttt{derelict } e : \sigma} \quad \equiv \quad \frac{\Theta; \Delta \vdash_{\mathrm{D}} e : !\sigma \qquad \overline{\Theta, x : \sigma; \varnothing \vdash_{\mathrm{D}} x : \sigma}}{\Theta; \Delta \vdash_{\mathrm{D}} \texttt{let } !x \coloneqq e \texttt{ in } x : \sigma}$$

The other rules for implication, pairs, and sums are all relatively straightforward. For example, the rules for linear functions introduce linear variables:

$$\frac{\Theta; \Delta, x : \sigma \vdash_{\mathrm{D}} e : \tau}{\Theta; \Delta \vdash_{\mathrm{D}} \hat{\lambda} x.e : \sigma \multimap \tau} \text{ DILL-}\multimap\text{-I} \qquad \frac{\Theta; \Delta_1 \vdash_{\mathrm{D}} e : \sigma \multimap \tau \qquad \Theta; \Delta_2 \vdash_{\mathrm{D}} e' : \sigma}{\Theta; \Delta_1, \Delta_2 \vdash_{\mathrm{D}} e e' : \tau} \text{ DILL-}\multimap\text{-E}$$

Alternatively, it's possible to derive syntax for non-linear functions: let us write $\sigma \to \tau$ for the type $!\sigma \multimap \tau$. We can derive an introduction rule that introduces the argument into the non-linear context:

$$\frac{\Theta, x : \sigma; \Delta \vdash_{\mathrm{D}} e : \tau}{\Theta; \Delta \vdash_{\mathrm{D}} \hat{\lambda} !x.e : \sigma \to \tau} \quad \equiv \quad \frac{\dfrac{\Theta; z : !\sigma \vdash_{\mathrm{D}} z : !\sigma \qquad \Theta, x : \sigma; \Delta \vdash_{\mathrm{D}} e : \tau}{\Theta; \Delta, z : !\sigma \vdash_{\mathrm{D}} \texttt{let } !x \coloneqq z \texttt{ in } e : \tau}}{\Theta; \Delta \vdash_{\mathrm{D}} \hat{\lambda} z.\texttt{let } !x \coloneqq z \texttt{ in } e : !\sigma \multimap \tau}$$

Related work. The inspiration for tracking linear and non-linear resources in different parts of a context started with Girard's *Logic of Unity* (LU) (1993), which unifies several logical frameworks including linear logic, intuitionistic logic, and classical logic. In LU, each logical fragment has a designated fragment of the context. Wadler (1994) restricted LU to its intuitionistic and linear fragments and considered it as a sequent-calculus style syntax for linear logic, but it was Barber's natural deduction style that caught on.

Embedded DILL. DILL might be the most popular style of linear type system in practice, but how does it fare as a model for embedded linear types? Our goal is for unrestricted variables in Θ to hold host-language data. This indicates that host types and linear types should overlap, since linear variables and non-linear variables have the same types in DILL. So we take `LType` to be `Type`, and we allow non-linear data to be embedded in a linear

expression as follows, where $\texttt{LExp}_{\text{D}}\ \Theta\ \Delta\ \alpha$ is the type of linear expressions:

$$\frac{a : \alpha}{\texttt{put}\ a : \texttt{LExp}_{\text{D}}\ \Theta\ \varnothing\ \alpha}$$

If linear types are just host-language types, then how do we distinguish linear connectives from ordinary non-linear connectives? Consider the linear function type $\alpha \multimap \beta$, which must now correspond to a type in the host language. If that type is inhabited—say, if $\alpha \multimap \beta$ is the normal function type $\alpha \to \beta$—then the $\texttt{put}$ constructor would violate linearity, as we would have $\texttt{put}(\lambda x.(x,x)) : \texttt{LExp}_{\text{D}}\ \Theta\ \varnothing\ (\alpha \multimap \alpha \times \alpha)$. On the other hand, we would like $\texttt{put}(\lambda x.(x,x)) : \texttt{LExp}_{\text{D}}\ \Theta\ \varnothing\ (!\alpha \multimap \alpha \times \alpha)$.

It is clear that a theory of embedded DILL would require significant changes to its meta-theory, so we look for another approach.

2.5 Indexed modalities

DILL syntactically separates non-linear variables Θ from linear variables Δ in its typing judgment, but one could equally consider a typing judgment that annotated each variable as either linear or unrestricted. This presentation, which we call *indexed resource modalities*, uses a single typing context Φ annotated with resource descriptors r:

$$\Phi ::= \varnothing \mid \Phi, x :_r \sigma \qquad\qquad r ::= 0 \mid 1 \mid \omega$$

The resource 1 stands for linear use, *i.e.*, the variable is used exactly once in a term, and ω stands for unrestricted use. The resource 0 stands for an unused resource, so if x does not appear in Φ, then Φ is equivalent to $\Phi, x :_0 \sigma$.

These resource descriptors form an algebraic structure known as a *rig*—a riNg without Negation:

$$0 + r = r + 0 = r \qquad\qquad 0 \cdot r = r \cdot 0 = 0 \qquad\qquad 1 \cdot r = r \cdot 1 = r$$

In addition, the unrestricted resource absorbs other resources.

$$1 + 1 = \omega \qquad\qquad \omega + r = r + \omega = \omega \qquad\qquad \omega \cdot \omega = \omega$$

The first equation says that when a linear resource (denoted with the resource descriptor 1) is used more than once in a system, then it it is unrestricted in the combined system. With a different collection of resources, *e.g.,* resources drawn from $\mathbb{Z}$, we could produce a more refined analysis; we discuss these more below. The second and third equations say that an unrestricted resource will always remain unrestricted.

We can extend the rig on resources to a semi-module on indexed typing contexts.

$$(\Delta_1, x :_{r_1} \sigma) + (\Delta_2, x :_{r_2} \sigma) \equiv (\Delta_1 + \Delta_2), x :_{r_1+r_2} \sigma$$
$$r \cdot (\Gamma, x :_{r'} \sigma) \equiv (r \cdot \Gamma), x :_{r \cdot r'} \sigma$$

The typing judgment has the form $\Phi \vdash_{\text{I}} e : \sigma$. Like in DILL, we want unrestricted data annotated with ω to have implicit weakening and contraction, which we can obtain by modifying how contexts are split. Instead of restricting typing rules to *disjoint* typing contexts, we simply use context addition to determine the output typing context from the

input contexts.

$$\frac{}{\omega \cdot \Phi, x :_r \sigma \vdash_{\text{I}} x : \sigma} \text{ I-VAR} \qquad \frac{\Phi \vdash_{\text{I}} e : \sigma \qquad \Phi', x :_r \sigma \vdash_{\text{I}} e' : \tau}{(r \cdot \Phi) + \Phi' \vdash_{\text{I}} \texttt{let } x \coloneqq e \texttt{ in } e' : \tau} \text{ I-LET}$$

In the variable rule, all the variables in $\omega \cdot \Phi$ are unrestricted, so they can be implicitly weakened. In the `let` rule, the resources Φ used to construct e are scaled by the number of times x is being used in the result.

The promotion rule says that any linear expression can be promoted, but the resources in the result are all scaled by ω, since the result could be used any number of times.

$$\frac{\Phi \vdash_{\text{I}} e : \sigma}{\omega \cdot \Phi \vdash_{\text{I}} !e : !\sigma} \textit{ i-!-I} \qquad \frac{\Phi \vdash_{\text{I}} e : !\sigma}{\Phi \vdash_{\text{I}} \texttt{derelict } e : \sigma} \textit{ i-!-E}$$

Function types can be annotated with the resource corresponding to how many times the argument is used.

$$\frac{\Phi, x :_r \sigma \vdash_{\text{I}} e : \tau}{\Phi \vdash_{\text{I}} \hat{\lambda} x.e : \sigma \to_r \tau} \textit{ i-}\to\text{-I} \qquad \frac{\Phi \vdash_{\text{I}} e : \sigma \to_r \tau \qquad \Phi' \vdash_{\text{I}} e' : \sigma}{\Phi + r \cdot \Phi' \vdash_{\text{I}} ee' : \tau} \textit{ i-}\to\text{-E}$$

Related work. Resource annotations have often been extended to different substructural type systems. The style seems to have originated with bounded linear logic (Girard *et al.*, 1992) annotating the exponential $!_n$ with a number n recording the precise number of times it is used. The type system presented above can easily accommodate exponentials indexed by arbitrary resources:

$$\frac{\Phi \vdash_{\text{I}} e : \sigma}{r \cdot \Phi \vdash_{\text{I}} !e : !_r\sigma} \textit{ i-}!_r\text{-I} \qquad \frac{\Phi \vdash_{\text{I}} e : !_r\sigma}{\Phi \vdash_{\text{I}} \texttt{derelict } e : \sigma} \textit{ i-}!_r\text{-E}$$

By including resources corresponding to affine or substructural use, resource annotations can express substructural typing systems, or coeffects like data flow, liveness analyses, or differential privacy (Petricek *et al.*, 2014; Brunel *et al.*, 2014; Reed and Pierce, 2010).

McBride (2016) uses resource annotations in a calculus for linear dependent types, where variables can be used in types with a resource annotation of $x :_0 \sigma$. McBride indexes not only variables, but also the typing judgment itself, with a resource: $\Phi \vdash_{\text{I}} e :_r \sigma$, which takes the place of the exponential $!_r$.

Bernardy *et al.* (2017) use resource annotations in a calculus that retrofits Haskell with linear types. Their typing judgment, though, has a unique interpretation: the typing judgment $\Phi \vdash e : \sigma$ in their system means that if the result of e is consumed exactly once, then the linear hypotheses in Φ will be consumed exactly once. However, any top-level expression *can* be consumed multiple times, to make the calculus backwards-compatible and facilitate code reuse between linear and non-linear types. This means that if a program wants to guarantee linear use of a piece of data, it must bind that data on the left-hand-side of a function type, as in $\sigma \to_1 \tau$. In practice this seems to result in a style of programming akin to continuation-passing style.

Embedded indexed modalities. Like DILL, the presentation in terms of indexed modalities requires that both linear and non-linear resources share the same kind of type. But now we can define the type $\alpha \to_r \beta$ as a wrapper for $\alpha \to \beta$ when r is ω, and otherwise as an empty type.

```
data α →r β where
  fun : (α → β) → (α →ω β)
```

Thus, non-linear functions $f : \alpha \to \beta$ can be coerced into a linear expression $\texttt{put}(\texttt{fun}\ f)$ of linear type $\alpha \to_\omega \beta$, but not into the type $\alpha \to_1 \beta$, which can only be constructed via the embedded $\hat{\lambda}$ constructor.

$$\frac{e : \texttt{LExp}_{\texttt{l}}\ (\Phi, x :_r \alpha)\ \beta}{\hat{\lambda} x.e : \texttt{LExp}_{\texttt{l}}\ \Phi\ (\alpha \to_r \beta)} \qquad \frac{e : \texttt{LExp}_{\texttt{l}}\ \Phi\ (\alpha \to_r \beta) \qquad e' : \texttt{LExp}_{\texttt{l}}\ \Phi'\ \alpha}{e\,\hat{}\,e' : \texttt{LExp}_{\texttt{l}}\ (\Phi + r \cdot \Phi')\ \beta}$$

Non-linear functions from host-language libraries can now be applied to linear arguments. For example, consider the `lookup` operation from the linear interface to mutable references discussed in Chapter 1.

$$\texttt{lookup} : \texttt{LExp}_{\texttt{l}}\ (\omega \cdot \Phi)\ (\texttt{LRef}\ \alpha \to_1 \alpha \otimes \texttt{LRef}\ \alpha)$$

We can lift arbitrary functions of type $\alpha \to \beta$ to the result of `lookup`:

$$\begin{aligned} op &: (\alpha \to \beta) \to \texttt{LExp}_{\texttt{l}}\ (\omega \cdot \Phi)\ (\texttt{LRef}\ \alpha \to_1 \beta \otimes \texttt{LRef}\ \alpha) \\ op &\equiv \lambda f. \hat{\lambda} r. \texttt{let}\ (x, r') \coloneqq \texttt{lookup}\ r\ \texttt{in}\ (\texttt{put}\ f\,\hat{}\,x, r') \end{aligned}$$

But what is the operational semantics of `put`? Is $(\texttt{put}\ f)$ a value? If so, then is `put f ^ v` a stuck term? What about `put f ^ put a`?

These questions may not be insurmountable, but they are not straightforward from the theory of indexed resource modalities.

2.6 Kind-based linear logic

The previous two presentations assume that, while linear and non-linear variables should be treated differently, all types are inherently the same. Kind-based presentations of linear logic suggest that linear data is inherently different from non-linear data, and the type system should distinguish them.

System F° (pronounced "F-pop"), introduced by Mazurak *et al.* (2010), has a kind $*$ ("star") for non-linear types and a kind $\circ$ ("pop") for linear types. The kinding judgment $\vdash_{\text{K}} \sigma : \kappa$ assigns each type σ a kind $\kappa \in \{*, \circ\}$. Like System F (Girard, 1971, 1986), System F° allows type variables and quantification over type variables X.

$$\frac{\Phi \vdash_{\text{K}} \sigma : *}{\Phi \vdash_{\text{K}} \sigma : \circ}\ \textsc{sub} \qquad \frac{\Phi \vdash_{\text{K}} \sigma_1 : \kappa_1 \qquad \Phi \vdash_{\text{K}} \sigma_2 : \kappa_2}{\Phi \vdash_{\text{K}} \sigma_1 \xrightarrow{\kappa} \sigma_2 : \kappa}\ \to$$

$$\frac{X : \kappa \in \Phi}{\Phi \vdash_{\text{K}} X : \kappa}\ \textsc{tvar} \qquad \frac{\Phi, X : \kappa_1 \vdash_{\text{K}} \sigma : \kappa_2}{\Phi \vdash_{\text{K}} \forall X : k_1.\ \sigma : k_2}\ \forall$$

In the first rule, non-linear types can be coerced into linear types. In the second rule, a function depends on three parameters: the kind of its argument; the kind of its result; and the kind of the function itself, which annotates the top of the arrow. For example, the linear identity type may be given the type $\sigma \xrightarrow{*} \sigma$ for $\vdash_{\mathrm{K}} \sigma : \circ$, because although the function uses its argument linearly, the function itself is linearly closed, and so can be used arbitrarily many times. So, although System F° doesn't include a ! operator on types, it can be approximated as $!\sigma \equiv \texttt{Unit} \xrightarrow{*} \sigma : *$, where $\texttt{Unit} : *$ is a non-linear unit type, and $\sigma : \circ$ is a linear type.

Typing contexts can either be separated according to the kind of the type being stored, as in DILL, or they can be combined as in the resource-annotated calculi. We choose the latter presentation, and we again write $\Phi_1 + \Phi_2$ for the linear merge of contexts Φ_1 and Φ_2. The subkinding relation can be written as a reflexive, transitive relation $\kappa_1 \geq \kappa_2$, with $* \geq \circ$. This relation can be extended to contexts $\Phi \geq \kappa$ to say that every type in Φ has kind κ' such that $\kappa' \geq \kappa$.

$$\frac{\Phi \geq *}{\Phi, x : \sigma \vdash_{\mathrm{K}} e : \sigma} \text{ F}^\circ\text{-VAR}$$

$$\frac{\Phi, x : \sigma \vdash_{\mathrm{K}} e : \tau \qquad \Phi \geq \kappa}{\Phi \vdash_{\mathrm{K}} \hat{\lambda} x.e : \sigma \xrightarrow{\kappa} \tau} \text{ F}^\circ\text{-}\to\text{-I} \qquad \frac{\Phi_1 \vdash_{\mathrm{K}} e : \sigma \xrightarrow{\kappa} \tau \qquad \Phi_2 \vdash_{\mathrm{K}} e' : \sigma}{\Phi_1 + \Phi_2 \vdash_{\mathrm{K}} e \,\hat{}\, e' : \tau} \text{ F}^\circ\text{-}\to\text{-E}$$

Related work. Later calculi including Alms (Tov and Pucella, 2011) and Quill (Morris, 2016) present variations of System F° with the addition of kind polymorphism and more nuanced subkinding. For example, in Alms (which is actually an affine type system), the identity function can be given the type $\forall \kappa, \forall(\sigma : \kappa), \sigma \xrightarrow{*} \sigma$. This makes it easier to reuse code and means that every well-typed program has a single most general type. However, the presentation of that most general type can be quite complex.

For example, in plain System F°, the expression $\hat{\lambda} x.\ \hat{\lambda} y.\ x$, which discards its second argument, can be given one of four types:

$$\forall(X : \circ)(Y : *), X \xrightarrow{*} Y \xrightarrow{\circ} X \qquad \forall(X : *)(Y : *), X \xrightarrow{*} Y \xrightarrow{*} X$$
$$\forall(X : \circ)(Y : *), X \xrightarrow{\circ} Y \xrightarrow{\circ} X \qquad \forall(X : *)(Y : *), X \xrightarrow{\circ} Y \xrightarrow{*} X$$

In Alms, however, this argument can be given a single most general type:

$$\forall(X : \kappa)(Y : *), X \xrightarrow{*} Y \xrightarrow{k} X \tag{2.1}$$

The four types above are all subtypes of Equation (2.1).

Embedded System F°. An embedded version of System F° would have its types of kind $*$ overlap with host-language types, but types of kind $\circ$ be distinct. Let `Kind` be a type with two constructors, $\circ$ and $*$, and define `Pop : Type` to be a data kind of linear types. Then we can define $[\![-]\!] : \texttt{Kind} \to \texttt{Type}$ so that $[\![\circ]\!] = \texttt{Pop}$:

```
data Pop where
  Var : Nat → Pop
  Sub : Type → Pop
```

```
  PopArrow : Π {κ₁ κ₂ : Kind}, ⟦κ₁⟧ → ⟦κ₂⟧ → Pop
data Kind where
  * : Kind
  ∘ : Kind
⟦*⟧ ≡ Pop
⟦∘⟧ ≡ Type
```

The linear type `Sub` α `: Pop` corresponds to the subkinding rule `sub`. The linear type `PopArrow` $\sigma_1\ \sigma_2$ corresponds only to $\sigma_1 \xrightarrow{\circ} \sigma_2$, as $\sigma_1 \xrightarrow{*} \sigma_2$ must be a host-language type. In particular, if σ_1 and σ_2 both have kind $*$, then $\sigma_1 \xrightarrow{*} \sigma_2$ should correspond to the regular host-language function type $\sigma_1 \to \sigma_2$. However, if either σ_1 or σ_2 has kind $\circ$, then the type should be uninhabited.

```
data StarArrow : Π (κ₁ κ₂ : Kind), ⟦κ₁⟧ → ⟦κ₂⟧ → Type where
  Arrow : Π (α β : Type), (α → β) → StarArrow α β
```

If made explicit, the subkinding relation from $*$ to $\circ$ could give some indication about the behavior of `put`, which injects host-language data into the linear embedded language. For example, we might expect host data $a : \alpha$ can be embedded into `put` a of $*$-type α, which can then be coerced to a linear type `Sub` α.

$$\frac{a : \alpha \qquad \Phi \geq *}{\texttt{put}\ a : \texttt{LExp}_{\text{K}}\ \Phi\ \alpha} \qquad \frac{e : \texttt{LExp}_{\text{K}}\ \Phi\ \alpha}{e : \texttt{LExp}_{\text{K}}\ \Phi\ (\texttt{Sub}\ \alpha)}$$

The subtyping relation should further coerce the type `Sub (StarArrow` $\sigma_1\ \sigma_2$`)` into `PopArrow` $\sigma_1\ \sigma_2$ so that `put(Arrow` f`)^put` a reduces to `put(`fa`)`. But this doesn't fully explain the semantics of `put`—what if the argument to `put Arrow` f does not have the form `put` a?

System F$^\circ$'s subkinding relation is implicit, but in an embedded language the two kinds would be explicit; thus F$^\circ$'s meta-theory does not characterize a linear embedding.

2.7 Linear/non-linear logic

Introduced by Benton (1995) and illustrated in Figure 2.9, linear/non-linear (LNL) logic makes a distinction between the syntax of linear and non-linear types, whereas System F$^\circ$ distinguishes them via a kinding judgment. Linear types, which we continue to denote with the meta-variable σ, are distinguished from non-linear types, which we denote α. The LNL system similarly consists of two kinds of variables (linear x and non-linear a), and two kinds of typing contexts (linear $\Delta ::= \varnothing \mid \Delta, x : \sigma$ and non-linear $\Gamma ::= \varnothing \mid \Gamma, a : \alpha$). LNL also has two kinds of terms, depending on whether the result is a linear or non-linear type.

A linear expression e can be thought of as a computation that consumes linear resources and also has access to non-linear variables. Its typing judgment is written $\Gamma; \Delta \vdash_{\text{B}} e : \sigma$, where Δ is a linear typing context and Γ is a non-linear typing context.[1] A non-linear term t cannot access any linear resources, so it can be thought of as a value instead of a computation. The typing judgment for non-linear data has the form $\Gamma \vdash_{\text{B}} t : \alpha$, where it only has access to non-linear variables. From these two typing judgments, there are two variable rules and three `let` bindings: non-linear variables bound in a non-linear term; non-linear

[1] The subscript $\vdash_{\text{B}}$ stands for Benton (1995).

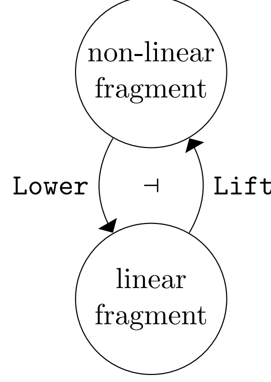


Figure 2.9: The linear/non-linear categorical model. The model consists of two categories related by functors `Lift` and `Lower` that form a categorical adjunction `Lift` ⊣ `Lower`; for details see Section 5.3.

variables bound in a linear term; and linear variables bound in a linear term.

$$\frac{}{\Gamma; x:\sigma \vdash_{\mathrm{B}} x:\sigma}\ \text{LNL-}\ell\text{-VAR} \qquad \frac{a:\alpha \in \Gamma}{\Gamma \vdash_{\mathrm{B}} a:\alpha}\ \text{LNL-N}\ell\text{-VAR}$$

$$\frac{\Gamma;\Delta \vdash_{\mathrm{B}} e:\sigma \qquad \Gamma;\Delta', x:\sigma \vdash_{\mathrm{B}} e':\tau \qquad \Delta\bot\Delta'}{\Gamma;\Delta,\Delta' \vdash_{\mathrm{B}} \mathtt{let}\ x \coloneqq e\ \mathtt{in}\ e':\tau}\ \text{LNL-LET-}\ell\text{-IN-}\ell$$

$$\frac{\Gamma \vdash_{\mathrm{B}} t:\alpha \qquad \Gamma, a:\alpha;\Delta \vdash_{\mathrm{B}} e:\tau}{\Gamma;\Delta \vdash_{\mathrm{B}} \mathtt{let}\ a \coloneqq t\ \mathtt{in}\ e:\tau}\ \text{LNL-LET-N}\ell\text{-}\ell \qquad \frac{\Gamma \vdash_{\mathrm{B}} t:\alpha \qquad \Gamma, a:\alpha \vdash_{\mathrm{B}} t':\beta}{\Gamma \vdash_{\mathrm{B}} \mathtt{let}\ a \coloneqq t\ \mathtt{in}\ t':\beta}\ \text{LNL-LET-N}\ell\text{-N}\ell$$

Because there are both linear and non-linear types and terms, there must be two sorts of every connective—two kinds of products, two kinds of functions, *etc.*. The non-linear product, for example, is only relevant in the non-linear typing judgment, and conversely for the linear products.

$$\frac{\Gamma \vdash_{\mathrm{B}} t_1:\alpha_1 \qquad \Gamma \vdash_{\mathrm{B}} t_2:\alpha_2}{\Gamma \vdash_{\mathrm{B}} (t_1,t_2):\alpha_1 \times \alpha_2}\ \text{LNL-}\times\text{-I} \qquad \frac{\Gamma \vdash_{\mathrm{B}} t:\alpha_1 \times \alpha_2}{\Gamma \vdash_{\mathrm{B}} \pi_i t:\alpha_i}\ \text{LNL-}\times\text{-E}$$

$$\frac{\Gamma;\Delta_1 \vdash_{\mathrm{B}} e_1:\sigma_1 \qquad \Gamma;\Delta_2 \vdash_{\mathrm{B}} e_2:\sigma_2 \qquad \Delta_1\bot\Delta_2}{\Gamma;\Delta_1,\Delta_2 \vdash_{\mathrm{B}} (e_1,e_2):\sigma_1 \otimes \sigma_2}\ \text{LNL-}\otimes\text{-I}$$

$$\frac{\Gamma;\Delta \vdash_{\mathrm{B}} e:\sigma_1 \otimes \sigma_2 \qquad \Gamma;\Delta', x_1:\sigma_1, x_2:\sigma_2 \vdash_{\mathrm{B}} e':\tau \qquad \Delta\bot\Delta'}{\Gamma;\Delta,\Delta' \vdash_{\mathrm{B}} \mathtt{let}\ (x_1,x_2) \coloneqq e\ \mathtt{in}\ e':\tau}\ \text{LNL-}\otimes\text{-E}$$

The exponential operator ! is broken up into two parts in LNL, as illustrated in Figure 2.9.

The first operator takes a linear type σ and lifts it to a non-linear type, `Lift` σ, summarized in Figure 2.10. The introduction and elimination rules for `Lift` correspond to the promotion and dereliction rules for !, which we write in this case as `suspend` and `force`.

$$\alpha ::= \cdots \mid \texttt{Lift}\ \sigma \qquad t ::= \cdots \mid \texttt{suspend}\ e \qquad e ::= \cdots \mid \texttt{force}\ t$$

$$\frac{\Gamma \vdash_{\mathrm{B}} t : \texttt{Lift}\ \sigma}{t \sim_\eta \texttt{suspend}(\texttt{force}\ t)}$$

$$\frac{\Gamma; \varnothing \vdash_{\mathrm{B}} e : \sigma}{\Gamma \vdash_{\mathrm{B}} \texttt{suspend}\ e : \texttt{Lift}\ \sigma}\ \texttt{Lift-I} \qquad \frac{\Gamma \vdash_{\mathrm{B}} t : \texttt{Lift}\ \sigma}{\Gamma; \varnothing \vdash_{\mathrm{B}} \texttt{force}\ t : \sigma}\ \texttt{Lift-E}$$

$$E^{\mathrm{V}} ::= \cdots \mid \texttt{force}\ E^{\mathrm{V}} \qquad v^{\mathrm{V}} ::= \cdots \mid \texttt{suspend}\ e \qquad \texttt{force}(\texttt{suspend}\ e) \rightsquigarrow_{\mathrm{V}} e$$
$$E^{\mathrm{N}} ::= \cdots \mid \texttt{force}\ E^{\mathrm{N}} \qquad v^{\mathrm{N}} ::= \cdots \mid \texttt{suspend}\ e \qquad \texttt{force}(\texttt{suspend}\ e) \rightsquigarrow_{\mathrm{N}} e$$

Figure 2.10: LNL `Lift` connective

$$\sigma ::= \cdots \mid \texttt{Lower}\ \alpha \qquad e ::= \cdots \mid \texttt{put}\ t \mid \texttt{let}\ !a \coloneqq e\ \texttt{in}\ e'$$

$$\frac{\Gamma; \Delta \vdash_{\mathrm{B}} e : \texttt{Lower}\ \alpha \qquad \Gamma; \Delta', x : \texttt{Lower}\ \alpha \vdash_{\mathrm{B}} e' : \tau \qquad \Delta \bot \Delta'}{e'\{e/x\} \sim_\eta \texttt{let}\ !a \coloneqq e\ \texttt{in}\ e'\{\texttt{put}\ a/x\}}$$

$$\frac{\Gamma \vdash_{\mathrm{B}} t : \alpha}{\Gamma; \varnothing \vdash_{\mathrm{B}} \texttt{put}\ t : \texttt{Lower}\ \alpha}\ \texttt{Lower-I}$$

$$\frac{\Gamma; \Delta \vdash_{\mathrm{B}} e : \texttt{Lower}\ \alpha \qquad \Gamma, a : \alpha; \Delta' \vdash_{\mathrm{B}} e' : \tau \qquad \Delta \bot \Delta'}{\Gamma; \Delta, \Delta' \vdash_{\mathrm{B}} \texttt{let}\ !a \coloneqq e\ \texttt{in}\ e' : \tau}\ \texttt{Lower-E}$$

$$E^{\mathrm{V}} ::= \cdots \mid \texttt{put}\ E^{\mathrm{V}} \mid \texttt{let}\ !a \coloneqq E^{\mathrm{V}}\ \texttt{in}\ e' \qquad v^{\mathrm{V}} ::= \cdots \mid \texttt{put}\ v \qquad \texttt{let}\ !a \coloneqq \texttt{put}\ v\ \texttt{in}\ e' \rightsquigarrow_{\mathrm{V}} e'\{v/a\}$$
$$E^{\mathrm{N}} ::= \cdots \mid \texttt{let}\ !a \coloneqq E^{\mathrm{N}}\ \texttt{in}\ e' \qquad v^{\mathrm{N}} ::= \cdots \mid \texttt{put}\ t \qquad \texttt{let}\ !a \coloneqq \texttt{put}\ t\ \texttt{in}\ e' \rightsquigarrow_{\mathrm{N}} e'\{t/a\}$$

Figure 2.11: LNL `Lower` connective

As for !, we should never evaluate under a suspended computation.

The second operator takes a non-linear type α to a linear type `Lower` α, as shown in Figure 2.11. Any host-language term t of type α can be coerced into a (linear) computation `put` t of type `Lower` α that returns the value of t. A computation of type `Lower` α can be bound to a non-linear variable a, so that the result can be used non-linearly in the continuation.

The original ! modality can be derived from the composition of `Lower` and `Lift`:

$$\frac{\Gamma; \varnothing \vdash_{\mathrm{B}} e : \sigma}{\Gamma; \varnothing \vdash_{\mathrm{B}} !e : !\sigma} \quad \equiv \quad \frac{\dfrac{\Gamma; \varnothing \vdash_{\mathrm{B}} e : \sigma}{\Gamma \vdash_{\mathrm{B}} \texttt{suspend}\ e : \texttt{Lift}\ \sigma}}{\Gamma; \varnothing \vdash_{\mathrm{B}} \texttt{put}(\texttt{suspend}\ e) : \texttt{Lower}(\texttt{Lift}\ \sigma)}$$

$$\frac{\Gamma; \Delta \vdash_{\mathrm{B}} e : !\sigma}{\Gamma; \Delta \vdash_{\mathrm{B}} \texttt{derelict}\ e : \sigma} \equiv \frac{\Gamma; \Delta \vdash_{\mathrm{B}} e : \texttt{Lower}(\texttt{Lift}\ \sigma) \qquad \dfrac{\dfrac{}{\Gamma, a : \texttt{Lift}\ \sigma \vdash_{\mathrm{B}} a : \texttt{Lift}\ \sigma}}{\Gamma, a : \texttt{Lift}\ \sigma; \varnothing \vdash_{\mathrm{B}} \texttt{force}\ a : \sigma}}{\Gamma; \Delta \vdash_{\mathrm{B}} \texttt{let}\ !a \coloneqq e\ \texttt{in force}\ a : \sigma}$$

Related work. The idea of having separate typing judgments for different kinds of types is closely related to Levy's call-by-push-value (CBPV), which makes the distinction between values and computations of two different syntactic forms (Levy, 2003). In CBPV, only values can be bound to variables, and since computations correspond to linear data in LNL, this means linear type checking is not needed in CBPV.

The relationship between values and computations is further emphasized by polarized logic (Girard, 1991; Laurent, 2002), which associates different type connectives with values (positive) or computations (negative). In particular, the multiplicative product $\otimes$ and additive sum $\oplus$ are positive types corresponding to values, but function types $\multimap$ and the additive product & are negative types, corresponding to computations. Polarized logic suggests that positive types correspond to a call-by-value evaluation order, and negative types correspond to a call-by-name evaluation order. Every value can be coerced into a trivial computation, corresponding to the shift operator $\uparrow$ from polarized logic or the `Lower` operator from LNL. Computations can be suspended (turned into values) via the shift operator $\downarrow$ of polarized logic or the `Lift` operator from LNL (Zeilberger, 2008).

Linear dependent types, where types can depend only on non-linear values, were introduced by Cervesato and Pfenning (1996), and Krishnaswami *et al.* (2015) present them in a calculus based on linear/non-linear logic. Vákár (2014) also develops a categorical semantics of linear dependent types based on the LNL categorical model (see Chapter 5).

Embedded linear/non-linear types. Unlike in the previous presentations, an embedded linear/non-linear type system suggests that linear types are explicitly different from non-linear types. So while non-linear types can correspond to host-language types, linear types can be defined separately. Surprisingly, this observation actually simplifies the theory of LNL. Instead of a non-linear typing judgment, we can use arbitrary terms in the host language. Instead of a non-linear typing context, we let the host language manage non-linear variables.

Embedded LNL thus consists of a single typing judgment $\Delta \vdash_{\mathrm{B}} e : \sigma$ of embedded linear terms that have access to the non-linear host language through the interface of `Lift` and `Lower`. The semantic behavior of `Lift` and `Lower` is completely characterized by the theory of LNL, so the soundness of the LNL type system extends directly to the embedded language. While it initially seems redundant to have both linear and non-linear pairs and other data types, we already expect this duplication for embedded languages, and the host language now provides half of the LNL system for free.

The surprising simplicity of embedded LNL makes for a sound, robust, and expressive system. In the next chapter we explore the details of the theory of embedded LNL, and give a number of examples of practical domain-specific applications.

CHAPTER 3

Embedded linear/non-linear types

In this section we develop the theory of linear/non-linear embedded languages. As we argued in Section 2.7, Benton's linear/non-linear (LNL) lambda calculus can be interpreted as a simple language of linear expressions `LExp` Δ σ embedded inside a non-linear host language with dependent types. The embedded language can interact with the host language via the two operators `Lift` and `Lower` from Figure 1.1.

3.1 A linear embedded language

We start with a purely linear type system with the linear connectives explored in Section 2.2: functions, multiplicative and additive products, additive sums, and unit types. We define these linear types as an algebraic data type as follows:

```
data LType where
    (⊸)   : LType → LType → LType
  | LUnit : LType | (⊗) : LType → LType → LType
  | LTop  : LType | (&) : LType → LType → LType
  | LZero : LType | (⊕) : LType → LType → LType
```

The parentheses around constructors indicate that they can be used infix, as in $\sigma \multimap \tau$. We use the meta-variable σ : `LType` to refer to linear types, and α : `Type` to refer to host-language types.

When defining any programming language, one notoriously difficult decision is how to represent variables, variable binding, and typing contexts. These choices are largely tangential to the design of the LNL embedding, so we put off discussion of these issues to the implementations in Chapters 4 and 8. For now, we write x : `Var` for the type of linear variables, and Δ : `Ctx` for linear typing contexts, which map variables to linear types. We require some basic operations over linear contexts:

- Every context Δ : `Ctx` has an associated finite set `dom`(Δ) of variables;
- $\varnothing$ is the empty context;
- $x : \sigma$ is the singleton context containing just the variable x of type σ;
- given two typing contexts Δ_1 and Δ_2, we write $\Delta_1 \perp \Delta_2$ if their domains are disjoint; and
- if $\Delta_1 \perp \Delta_2$, then their disjoint merge (Δ_1, Δ_2) is defined.

The type of linear expressions `LExp` Δ σ is indexed by a typing context Δ and a linear type σ. This Church-style, intrinsically typed presentation of the judgment means that

there is no untyped syntax of linear expressions, only well-typed and well-scoped linear expressions. A Curry-style judgment with untyped syntax and an external typing judgment is also consistent with the LNL framework; we use such a judgment in Chapter 8.

Linear expressions are given by the rules in Figure 3.1, which exactly correspond to the rules from Section 2.1. Syntactically we make a distinction between host-language variables a and embedded variables x; between host-language functions $\lambda a.b$ and embedded functions $\hat{\lambda} x.e$; and between host-language application $f a$ and embedded application $e\,\hat{}\,e'$.

We use the notation $\Delta \vdash e : \sigma$ to mean that e is a term of type `LExp` $\Delta\ \sigma$, and we sometimes write $\Delta \vdash - : \sigma$ for the type `LExp` $\Delta\ \sigma$ itself. If we wanted to, we could restate the rules from Figure 3.1 as the typing judgments from Section 2.2. For example, the following two rules for lambda abstraction are identical.

$$\frac{e : \texttt{LExp}\ (\Delta, x : \sigma)\ \tau}{\hat{\lambda} x.e : \texttt{LExp}\ \Delta\ (\sigma \multimap \tau)} \qquad \frac{\Delta, x : \sigma \vdash e : \tau}{\Delta \vdash \hat{\lambda} x.e : \sigma \multimap \tau}$$

We define α, β, and η rules as relations $\sim_\alpha$, $\rightsquigarrow_\beta$, and $\sim_\eta$ on linear expressions, exactly as discussed in Section 2.2; we do not restate those rules here. We write $\sim$ for the smallest congruence on linear expressions that contains $\sim_\alpha$, $\rightsquigarrow_\beta$, and $\sim_\eta$, and we write $e\{e_1/x_1, \ldots, e_n/x_n\}$ for the simultaneous capture-avoiding substitution of e_i for x_i in e.

3.2 The linear/non-linear interface

The linear/non-linear interface is given by the two type operators `Lift` and `Lower`, as illustrated back in Figure 1.1.

For any host-language type α, we have a linear type `Lower` α, for which we extend the definition of linear types.

```
data LType where
  ...
  | Lower : Type → LType
```

According to the typing rules for `Lower`, for any host-language term $a : \alpha$, there is a linear expression $\varnothing \vdash \texttt{put}\ a : \texttt{Lower}\ \alpha$.

$$\frac{a : \alpha}{\texttt{put}\ a : \texttt{LExp}\ \varnothing\ (\texttt{Lower}\ \alpha)}\ \texttt{Lower-I}$$

To eliminate expressions of type `Lower` α, it suffices to bind them against a continuation that uses a host-language variable $a : \alpha$. What we wrote in Section 2.7 as $\texttt{let}\ !a \coloneqq e\ \texttt{in}\ e'$ is now replaced by $e >!\ f$, pronounced e "let-bang" f, where f is a host-language function of type $\alpha \to \texttt{LExp}\ \Delta'\ \tau$. We will still use the notation $\texttt{let}\ !a \coloneqq e\ \texttt{in}\ e'$ for $e >!\ \lambda a.e'$.

$$\frac{e : \texttt{LExp}\ \Delta\ (\texttt{Lower}\ \alpha) \qquad f : \alpha \to \texttt{LExp}\ \Delta'\ \tau \qquad \Delta \perp \Delta'}{e >!\ f : \texttt{LExp}\ (\Delta, \Delta')\ \tau}\ \texttt{Lower-E}$$

The β and η rules for `Lower` α are identical to those described in Section 2.7. If `put` a is bound against f, the result $\texttt{put}\ a >!\ f$ reduces to $f a$. Further, if a computation e' has a sub-expression e of (linear) type `Lower` α, then it is equivalent to first bind e against $>!$

$$\frac{}{x : \texttt{LExp}\ (x:\sigma)\ \sigma}\ \textsc{var} \qquad \frac{e : \texttt{LExp}\ \Delta\ \sigma \qquad e' : \texttt{LExp}\ (\Delta', x:\sigma)\ \tau \qquad \Delta \bot \Delta'}{\texttt{let}\ x \coloneqq e\ \texttt{in}\ e' : \texttt{LExp}\ (\Delta, \Delta')\ \tau}\ \textsc{let}$$

$$\frac{e : \texttt{LExp}\ (\Delta, x:\sigma)\ \tau}{\hat{\lambda} x.e : \texttt{LExp}\ \Delta\ (\sigma \multimap \tau)}\ \multimap\text{-I} \qquad \frac{e : \texttt{LExp}\ \Delta\ (\sigma \multimap \tau) \qquad e' : \texttt{LExp}\ \Delta'\ \sigma \qquad \Delta \bot \Delta'}{e\,\hat{}\,e' : \texttt{LExp}\ (\Delta, \Delta')\ \tau}\ \multimap\text{-E}$$

$$\frac{}{() : \texttt{LExp}\ \varnothing\ \texttt{LUnit}}\ \texttt{LUnit-I} \qquad \frac{e : \texttt{LExp}\ \Delta\ \texttt{LUnit} \qquad e' : \texttt{LExp}\ \Delta'\ \tau \qquad \Delta \bot \Delta'}{\texttt{let}\ () \coloneqq e\ \texttt{in}\ e' : \texttt{LExp}\ (\Delta, \Delta')\ \tau}\ \texttt{LUnit-E}$$

$$\frac{e_1 : \texttt{LExp}\ \Delta_1\ \sigma_1 \qquad e_2 : \texttt{LExp}\ \Delta_2\ \sigma_2 \qquad \Delta_1 \bot \Delta_2}{(e_1, e_2) : \texttt{LExp}\ (\Delta_1, \Delta_2)\ (\sigma_1 \otimes \sigma_2)}\ \otimes\text{-I}$$

$$\frac{e : \texttt{LExp}\ \Delta\ (\sigma_1 \otimes \sigma_2) \qquad e' : \texttt{LExp}\ (\Delta', x_1 : \sigma_1, x_2 : \sigma_2)\ \tau \qquad \Delta \bot \Delta'}{\texttt{let}\ (x_1, x_2) \coloneqq e\ \texttt{in}\ e' : \texttt{LExp}\ (\Delta, \Delta')\ \tau}\ \otimes\text{-E}$$

$$\frac{}{\texttt{error} : \texttt{LExp}\ \Delta\ \texttt{LTop}}\ \textsc{LTop-I} \qquad \frac{e : \texttt{LExp}\ \Delta\ \texttt{LZero} \qquad \Delta \bot \Delta'}{\texttt{case}\ e\ \texttt{of}\ () : \texttt{LExp}\ \Delta, \Delta'\ \tau}\ \textsc{LZero-E}$$

$$\frac{e : \texttt{LExp}\ \Delta\ \sigma_1}{\iota_1 e : \texttt{LExp}\ \Delta\ (\sigma_1 \oplus \sigma_2)}\ \oplus\text{-I}_1 \qquad \frac{e : \texttt{LExp}\ \Delta\ \sigma_2}{\iota_2 e : \texttt{LExp}\ \Delta\ (\sigma_1 \oplus \sigma_2)}\ \oplus\text{-I}_2$$

$$\frac{\begin{array}{c} e : \texttt{LExp}\ \Delta\ (\sigma_1 \oplus \sigma_2) \\ e_1 : \texttt{LExp}\ (\Delta', x_1 : \sigma_1)\ \tau \qquad e_2 : \texttt{LExp}\ (\Delta', x_2 : \sigma_2)\ \tau \qquad \Delta \bot \Delta' \end{array}}{\texttt{case}\ e\ \texttt{of}\ (\iota_1 x_1 \to e_1 \mid \iota_2 x_2 \to e_2) : \texttt{LExp}\ (\Delta, \Delta')\ \tau}\ \oplus\text{-E}$$

$$\frac{e_1 : \texttt{LExp}\ (\Delta)\ \sigma_1 \qquad e_2 : \texttt{LExp}\ (\Delta)\ \sigma_2}{[e_1, e_2] : \texttt{LExp}\ (\Delta)\ (\sigma_1 \& \sigma_2)}\ \&\text{-I}$$

$$\frac{e : \texttt{LExp}\ \Delta\ (\sigma_1 \& \sigma_2)}{\pi_1 e : \texttt{LExp}\ \Delta\ \sigma_1}\ \&\text{-E}_1 \qquad \frac{e : \texttt{LExp}\ \Delta\ (\sigma_1 \& \sigma_2)}{\pi_2 e : \texttt{LExp}\ \Delta\ \sigma_2}\ \&\text{-E}_2$$

Figure 3.1: Specification of an embedded linear lambda calculus as terms of type `LExp` Δ σ.

```
data LType where ⋯ | Handle : LType
```

$$\frac{s : \texttt{String}}{\varnothing \vdash \texttt{open}\ s : \texttt{Handle}}\ \textsc{open} \qquad \frac{\Delta \vdash e : \texttt{Handle}}{\Delta \vdash \texttt{close}\ e : \texttt{LUnit}}\ \textsc{close}$$

$$\frac{\Delta \vdash e : \texttt{Handle}}{\Delta \vdash \texttt{read}\ e : \texttt{Handle} \otimes \texttt{Lower Char}}\ \textsc{read} \qquad \frac{c : \texttt{Char} \qquad \Delta \vdash e : \texttt{Handle}}{\Delta \vdash \texttt{write}\ c\ e : \texttt{Handle}}\ \textsc{write}$$

Figure 3.2: Interface of linear file handles, given as inference rules, writing $\Delta \vdash e : \sigma$ for $e : \texttt{LExp}\ \Delta\ \sigma$.

and then continue as e'.

$$\texttt{put}\ a >!\ f \rightsquigarrow_\beta f a \qquad \frac{\Delta \vdash e : \texttt{Lower}\ \alpha \qquad \Delta', x : \texttt{Lower}\ \alpha \vdash e' : \tau \qquad \Delta \bot \Delta'}{e'\{e/x\} \sim_\eta e >!\ \lambda a. e'\{\texttt{put}\ a/x\}}$$

The operational behavior only differs from that of Section 2.7 in that the evaluation order of `put` a is determined not by our choice of call-by-name or call-by-value of the linear language, but instead by the existing evaluation order of the host language. We say that for any term $a : \alpha$ in the host language, `put` a is a value in the term language. If the host language uses a call-by-value semantics, then a will be evaluated when it is used as an argument to `put`; if the host language is lazy, it may not be evaluated until the result is actually required.

For any linear type σ, the LNL interface also specifies that there should be a host-language type `Lift` σ of suspended linear computations. In particular, we can define `Lift` σ as a host-language data type with a constructor for suspending a linear computation.

```
data Lift (σ : LType) where
  suspend : LExp ∅ σ → Lift σ
```

We argued in Section 2.7 that the operational behavior of `suspend` should always be lazy, meaning that linear expressions should never be evaluated under `suspend`. Depending on how the embedding is defined, this should be taken into account in the definition of `Lift`. In many embeddings, evaluation of linear computations will be a relation of the form `eval` : `LExp` Δ σ → `LExp` Δ σ → `Type`. In that case, using `suspend` e will never accidentally evaluate its argument. If, however, evaluation of linear expressions is built into the type `LExp`, then it may be necessary to add a thunk to the type of `suspend`, *e.g.,* () → `LExp` ∅ σ.

To eliminate terms of type `Lift` σ, we can define an ordinary host-language function `force` a that pattern matches against the data type `Lift` σ to expose the underlying computation.

```
force : Lift σ → LExp ∅ σ
force (suspend e) ≡ e
```

Since `force` is defined by ordinary pattern matching, the β and η semantics is given by the semantics of the host language itself.

3.3 Example: linear file handles

To illustrate the linear/non-linear interface in practice, Figure 3.2 shows a simple EDSL for linear file handles. It consists of four operations: `open`, `close`, `read`, and `write`. Writing out the types more explicitly,[2] we have

```
open  : String → LExp ∅ Handle
close : LExp Δ Handle → LExp Δ LUnit
read  : LExp Δ Handle → LExp Δ (Handle ⊗ Lower Char)
write : Char → LExp Δ Handle → LExp Δ Handle
```

Linearity rules out two specific kinds of errors here. First, it ensures that file handles cannot be used more than once in a term, so once a handle has been closed, it cannot be read from or written to again. Second, linearity ensures that all open handles are eventually closed (at least for terminating computations) since variables of type `Handle` cannot be dropped. Linearity allows us to think of a file handle as a consumable resource that gets used up when it is closed.

Note that linearity does not prevent all runtime errors: `open` could fail if there is a problem with the file name, or `read` could fail with an end-of-file error, etc.

An alternative interface Instead of presenting handles as a series of inference rules as in Figure 3.2, we could have presented them more compactly using the `Lift` type:

```
openF  : String → Lift Handle
closeF : Lift (Handle ⊸ LUnit)
readF  : Lift (Handle ⊸ Handle ⊗ Lower Char)
writeF : Char → Lift (Handle ⊸ Handle)
```

The two presentations are inter-derivable, but whereas the second interface is more compact and readable, it is also harder to use. Consider the following function that opens a file, reads a single character, then writes that character back to the file twice:

```
readWriteTwice ≡ let h      ≔ open "foo.txt" in
                 let (!c,h) ≔ read h in
                 close (write c (write c h))

readWriteTwiceF ≡ let h      ≔ force (openF "foo.txt") in
                  let (!c,h) ≔ force readF ^ h in
                  force closeF ^ (force (writeF c) ^ (force (writeF c) ^ h))
```

The second presentation can only access the file handle interface using `force` and embedded application (^), which makes it clunky compared to the first presentation.

3.4 Monadic programming

Haskell and other functional programming languages frequently use monads to encode effects in a purely functional setting (Moggi, 1989). For example, instead of the linear interface to file handles, a nonlinear, monadic interface might have the following form, where `withFile` opens a file handle, performs some operations on it, and then closes the file at the end.

[2] Technically, the second presentation allows for partial application of `open`, `close`, `read`, and `write`, while the first presentation in Figure 3.2 requires total application. Both styles are acceptable, and we consider the question of partial application to be orthogonal to the point being made here.

```
withFile : String → (Handle → M ()) → M ()
writeM   : Char → Handle → M ()
readM    : Handle → M Handle
```

The monad M is an operator on types; a term of type $M\alpha$ can be thought of as a computation that returns a value of type α. Monadic programming is popular because it is expressive, functional, and easy to use, as evidenced by the popularity of monadic programming in Haskell. Given a monadic computation $m : M\alpha$ and a function $f : \alpha \to M\beta$, we can *bind* m against f, written $m \gg\!= f$, to obtain a computation of type $M\beta$. In addition, for $a : \alpha$, there is a trivial computation $\mathtt{return}\ a : M\alpha$. These two operators should satisfy the following laws:

$$\mathtt{return}\ a \gg\!= f = f a \qquad (\text{M-}\beta)$$

$$(m \gg\!= f) \gg\!= g = m \gg\!= (n\ x \to f x \gg\!= g) \qquad (\text{M-ASSOC})$$

$$\frac{m : M\alpha}{m = m \gg\!= \mathtt{return}}\ \text{M-}\eta$$

In practice, the bind operation $\gg\!=$ is often written with the much more readable do-notation, writing $\mathtt{do}\ x \leftarrow a; b$ for $a \gg\!= \lambda x.b$:

```
readWriteTwice ≡ withFile "foo.txt" (λ h. do c ← readM h
                                               _ ← writeM c h
                                               writeM c h )
```

Unlike linear expressions, monadic computations do not restrict how resources are used, so the non-linear monadic interface cannot provide the same correctness guarantees. For example, even though `withFile` closes its file handle, a malicious actor could escape the file handle from its scope.

```
unsafeM ≡ do h ← withFile "foo.txt" return
             write 'c' h -- error: write after close
```

Monads are a powerful abstraction because they apply both to built-in effects like I/O (as in the case of file handles), as well as derived algebraic effects like state. The state monad $\mathtt{State}\ \alpha\ \beta \equiv \alpha \to \alpha \times \beta$ can be given instances for `return` and $\gg\!=$, as well as $\mathtt{get} : \mathtt{State}\ \alpha\ \alpha$ and $\mathtt{put} : \alpha \to \mathtt{State}\ \alpha\ ()$.

```
return b ≡ λ a. (a,b)                          get   ≡ λ a. (a,a)
s ≫= f   ≡ λ a. let (a',b') ≔ s a in f b' a'   put a ≡ λ _. (a,())
```

3.4.1 Linear monads

The instances of `return` and $\gg\!=$ for the state monad are linear in their arguments, which means the linear type $\mathtt{LState}\ \sigma\ \tau \equiv \sigma \multimap \sigma \otimes \tau$ is a monad in its own right. In fact, many of the algebraic non-linear monads, such as the option monad $\sigma \oplus \mathtt{LUnit}$ or the continuation-passing monad $(\sigma \multimap \tau) \multimap \tau$, are linear in the same way.

Consider the state monad. Unlike the non-linear interface to state, the linear interface does not admit `get` and `put`, which are non-linear in their arguments. However, the linear state monad is still a useful abstraction. For example, we can derive monadic formulations of `withFile`, `readM`, and `writeM` in the linear state monad.

```
withFile : String → LExp Δ (LState Handle LUnit) → LExp Δ LUnit
withFile filename e ≡ let h     ≔ open filename in
```

```
                        let (h,x) ≔ e ^ h         in
                        let ()    ≔ close h       in x
readM  : LExp ∅ (LState Handle (Lower Char))
readM  ≡ read
writeM : Char → LExp ∅ (LState Handle LUnit)
writeM c ≡ λ̂ h. write c h
```

Using do-notation can make using this interface easier to work with.

```
readWriteTwiceM : LExp ∅ LUnit
readWriteTwiceM ≡ withFile "foo.txt" (do !c ← readM
                                         () ← writeM c
                                         writeM c)
```

3.4.2 The linearity monad

Monads and linear types are both ways to manage effects in functional languages, and in fact there is a strong connection between the two approaches. Chen and Hudak (1997) show that mutable, linear abstract data types can automatically be given monadic interfaces for use in functional languages. Haskell's built-in `ST` monad encapsulates global state precisely because it always treats its state linearly (Launchbury and Peyton Jones, 1995). Benton and Wadler (1996) show how a monadic lambda calculus can be soundly translated into a linear/non-linear type system.

Benton and Wadler's translation relies on the fact that the composition `Lift` ∘ `Lower` forms a monad in the non-linear fragment of LNL. This is true because of the categorical relationship underlying the LNL model, which we explore in Chapter 5. But in the meantime we ask: can we program with this monad?

We write `Lin` α for `Lift(Lower` α`)`, and we call it *the linearity monad*. The operations `return` and ≫= are defined as follows:

```
return : α → Lin α
return a ≡ suspend (put a)
(≫=) : Lin α → (α → Lin β) → Lin β
m ≫= f ≡ suspend (let !a ≔ force m in force (f a))
```

These definitions satisfy the monad laws up to equivalence $\sim$ of the linear embedded language, which we defined to be the smallest congruence relation that contains $\sim_\alpha$, $\rightsquigarrow_\beta$, and $\sim_\eta$. For example, for the β rule for monads we have

```
return a ≫= f ≡ suspend (let !a ≔ force (suspend (put a)) in force (f a))
              = suspend (let !a ≔ put a in force (f a))
             ⇝β suspend (force (f a))
```

which is η-equivalent to $f\,a$ itself. Similar reasoning proves that η and associativity equivalences hold.

The type `Lin` α denotes a computation that takes place in the linear language, but does not return any linear data. Thus, `Lin` is a non-linear interface to the linear language, which can be used to limit exposure of linear types to end users. For example, the `withFile` operation can be formulated to return a computation in the linearity monad.

```
withFileM : String → Lift (LState Handle LUnit) → Lin Unit
withFileM filename op ≡ suspend (let h      ≔ open filename in
                                 let (h,()) ≔ force op ^ h  in
```

```
                                   let ()     ≔ close h       in
                                   put ())
```

3.4.3 The linearity monad transformer

The prospect of monadic programming can be pushed even further by combining monads in the linear embedded language, such as `LState` σ τ, with the linearity monad `Lift(Lower` α`)`. We will prove in Chapter 5 that the LNL model actually gives rise to a *monad transformer*—for any monad M on linear types, there is a monad `Lift`$\circ M \circ$`Lower` on non-linear types. We write `LinT` M α for `Lift(`M`(Lower` α`))`, and its interface builds on the monadic interface to M as follows:

```
return : α → LinT M α
return a ≡ suspend (return (put a))
(≫=) : LinT M α → (α → LinT M β) → LinT M β
m ≫= f ≡ suspend (force m ≫= λ̂ (x : Lower α). let !a ≔ x in force (f a))
```

The monad transformer turns out to be useful for many of the domain-specific interfaces we have considered, particularly when the linear monad returns a lowered host value. For example, the input to `withFileM` had the form `Lift (LState Handle LUnit)`, but could have instead returned the type `Lower ()`:

```
withFileT : String → LinT (LState Handle) () → Lin Unit
withFileT filename op ≡ suspend (let h      ≔ open filename in
                                   let (h,x)  ≔ force op ^ h   in
                                   let ()     ≔ close h       in
                                   x)
```

Furthermore, by reformulating `read` and `write` with respect to `LinT`, we can expose an entirely non-linear interface to linear file handles while keeping the safety guarantees of linearity.

```
readT : LinT (LState Handle) Char
readT ≡ suspend (λ̂ h. read h)

writeT : Char → LinT (LState Handle) Unit
writeT c ≡ suspend (λ̂ h. (write c h, put ()))

readWriteTwice : Lin ()
readWriteTwice ≡ withFileT "foo.txt" (do c ← readT
                                         _ ← writeT c
                                         writeT c)
```

We sometimes write `LStateT` σ α for the common idiom `LinT (LState` σ`)` α.

3.5 Extensions

The embedded LNL framework makes it easy to integrate various extensions to the embedded language; we consider a few of them here.

3.5.1 Linear data structures

Linear algebraic data types can be added to the linear embedded language in a few different ways.

```
data LType where ⋯ | μ : (LType → LType) → LType
```

$$\frac{\Delta \vdash e : F(\mu F)}{\Delta \vdash \texttt{fold}\ e : \mu F}\ \mu\text{-I} \qquad \frac{\Delta \vdash e : \mu F}{\Delta \vdash \texttt{unfold}\ e : F(\mu F)}\ \mu\text{-E}$$

Figure 3.3: Recursive types in the linear embedded language

The first naive approach adds a particular data structure as a one-off type, for example by extending `LType`s with `LList` σ, a linear list with values of type σ. Then we add the usual constructors and case analysis for lists:

$$\frac{}{\varnothing \vdash [\,] : \texttt{LList}\ \sigma} \qquad \frac{\Delta \vdash e : \sigma \qquad \Delta' \vdash e' : \texttt{LList}\ \sigma}{\Delta, \Delta' \vdash (e :: e') : \texttt{LList}\ \sigma}$$

$$\frac{\Delta \vdash e : \texttt{LList}\ \sigma \qquad \Delta' \vdash e_0 : \tau \qquad \Delta', x : \sigma, xs : \texttt{LList}\ \sigma \vdash e' : \tau}{\Delta, \Delta' \vdash \texttt{case}\ e\ \texttt{of}\ ([\,] \to e_0 \mid x :: xs \to e') : \tau}$$

Option two is to extend the language with recursive data types in general, as shown in Figure 3.3. Then, for example, lists can be encoded as `LList` $\sigma \equiv \mu(\lambda\tau.\texttt{LUnit} \oplus \sigma \otimes \tau)$.

Option three is, using dependent types, to encode length-indexed lists as n-tuples. That is, we can define a host level function $(\bigotimes) : \texttt{Nat} \to \texttt{LType} \to \texttt{LType}$ as follows:[3]

$$\begin{aligned} 0 \bigotimes \sigma &\equiv \texttt{LUnit} \\ n+1 \bigotimes \sigma &\equiv \sigma \otimes (n \bigotimes \sigma) \end{aligned}$$

The advantage of this approach is that it keeps the linear language itself very simple, and enforces a richer type discipline.

3.5.2 Recursion

Arbitrary linear recursion is not well typed, in that there is no linear Y combinator Y_l of type $(\sigma \multimap \sigma) \multimap \sigma$. If such a fixpoint did exist, and $\Delta \vdash f : \sigma \multimap \sigma$, then $f\,\hat{}\,(Y_l\,\hat{}\,f)$ would not be well-typed. However, the following is derivable, provided the host language also has general recursion:

```
lfix : Lift (σ ⊸ σ) → Lift σ
lfix f ≡ suspend (force f (force (lfix f)))
```

If the language does not have general recursion, like Coq or Agda for example, then `lfix` can be added as a constant to the linear language.

$$\frac{\varnothing \vdash e : \sigma \multimap \sigma}{\varnothing \vdash \texttt{lfix}\ e : \sigma}\ \text{LFIX} \qquad \texttt{lfix}\ f \rightsquigarrow_\beta f\,\hat{}\,(\texttt{lfix}\ f)$$

[3] Note the difference between $\sigma \otimes \tau$ where $\sigma, \tau : \texttt{LType}$ and $n \bigotimes \tau$, where $n : \texttt{Nat}$ and $\tau : \texttt{LType}$.

```
data LType where ⋯ | ∀ : (LType → LType) → LType
                   | ∃ : (LType → LType) → LType
```

$$\frac{\Pi(\sigma : \texttt{LType}), \Delta \vdash e : F\sigma}{\Delta \vdash e : \forall F}\ \forall\text{-I} \qquad \frac{\Delta \vdash e : \forall F \qquad \sigma : \texttt{LType}}{\Delta \vdash e[\sigma] : F\sigma}\ \forall\text{-E}$$

$$\frac{\Delta \vdash e : F\sigma}{\Delta \vdash (\sigma, e) : \exists F}\ \exists\text{-I} \qquad \frac{\Delta \vdash e : \exists F \qquad \prod_{\sigma} \Delta', x : F\sigma \vdash e' : \tau}{\Delta, \Delta' \vdash \texttt{let}\ (\sigma, x) \coloneqq e\ \texttt{in}\ e' : \tau}\ \exists\text{-E}$$

Figure 3.4: Polymorphism in the linear embedded language

Alternatively, given a function $f : \texttt{Lift}(\sigma \multimap \sigma)$ and a natural number n, we can define $f^n : \texttt{Lift}(\sigma \multimap \sigma)$ that applies f n times to its argument.

$$f^0 \equiv \texttt{id}_l$$
$$f^{n+1} \equiv \texttt{suspend}(\hat{\lambda} x.\ \texttt{force}\ f^n x)$$

3.5.3 Polymorphism

Conveniently, polymorphism in the host language gives rise to polymorphism in the linear language. Consider the linear identity function, where we get polymorphism for free:

$$\texttt{id}_l : \Pi(\sigma : \texttt{LType}), \texttt{Lift}(\sigma \multimap \sigma).$$

For more fine-grained control we can add general polymorphism, illustrated in Figure 3.4.

3.5.4 Dependent types

The embedded language inherits limited forms of dependent types for free from its host language, such as the length-indexed tuples $n \otimes \sigma$ shown above. In this context, a dependent linear type is one that depends on the value of a host language (non-linear) term; this restriction is generally accepted in the literature (Cervesato and Pfenning, 1996; Gaboardi *et al.*, 2013; Vákár, 2014; Krishnaswami *et al.*, 2015).

The syntax for Π and Σ types that depend on host-language terms is shown in Figure 3.5. If α is a host type and $F : \alpha \to \texttt{LType}$ is a function from α to linear types $\texttt{LType}$, then $\Pi^\ell_\alpha F$ and $\Sigma^\ell_\alpha F$ are linear types corresponding to universal and existential quantification over α, respectively. For Π^ℓ, a linear expression $\hat{\lambda} f$ of type $\Pi^\ell_\alpha F$ is constructed from a function f from values $a : \alpha$ to linear expressions of type Fa. To apply $\hat{\lambda} f$ to an argument of type $a : \alpha$, it suffices to just apply f to a.

Dually, a linear expression of type $\Sigma^\ell_\alpha F$ is a pair of a value of type a and a linear expression of type Fa. The elimination form binds a value of type $\Sigma^\ell_\alpha F$ against a function from $a : \alpha$ to linear expressions using $x : Fa$, and is written $\texttt{let}\ (!a, x) \coloneqq e\ \texttt{in}\ e'$.

3.6 Example: session types

Session types are a language mechanism for describing communication protocols between two actors (Honda, 1993; Kobayashi *et al.*, 1999). A session is a channel with exactly

```
data LType where ⋯ | Πℓ  : Π (α : Type), (α → LType) → LType
                   | Σℓ  : Π (α : Type), (α → LType) → LType
```

$$\frac{f : \Pi(a:\alpha), \Delta \vdash - : F a}{\Delta \vdash \hat{\lambda} f : \Pi^{\ell}_{\alpha} F}\ \Pi^{\ell}\text{-I} \qquad \frac{\Delta \vdash e : \Pi^{\ell}_{\alpha} F \qquad a : \alpha}{\Delta \vdash e\,\hat{}\,a : F a}\ \Pi^{\ell}\text{-E}$$

$$\frac{a : \alpha \qquad \Delta \vdash e : F a}{\Delta \vdash (a, e) : \Sigma^{\ell}_{\alpha} F}\ \Sigma^{\ell}\text{-I} \qquad \frac{\Delta \vdash e : \Sigma^{\ell}_{\alpha} F \qquad \prod_{a:\alpha} \Delta', x : F a \vdash e' : \tau \qquad \Delta \bot \Delta'}{\Delta, \Delta' \vdash \texttt{let}\ (!a, x) \coloneqq e\ \texttt{in}\ e' : \tau}\ \Sigma^{\ell}\text{-E}$$

Figure 3.5: Dependent types in the linear embedded language

two endpoints. Linearity ensures that the protocols for both endpoints of the channel are always in sync. Here we present a simple interface for session types, inspired by Lindley and Morris's GV calculus (2015).

A session type is given by the following grammar:

```
data Session where
  | (⟨!⟩) : LType → Session → Session
  | (⟨?⟩) : LType → Session → Session
  | End   : Session
```

```
data LType where
  ⋯
  | Channel : Session → LType
```

For a linear type σ and a session type S, a channel with session type $\sigma\langle ! \rangle S$ promises to send a value of type σ and then continue with the protocol S. On the other hand, a channel with session type $\sigma\langle ? \rangle S$ will receive a value of type σ and then continue as S. The session type `End` denotes the end of a communication process.

It is clear that every session type S has a corresponding dual type $S^{\bot}$ such that, if Alice has access to one end of the channel with session type S and Bob has access to the other end of the channel, then Bob's channel has the session type $S^{\bot}$:

$$(\sigma\langle ! \rangle S)^{\bot} \equiv \sigma\langle ? \rangle S^{\bot} \qquad (\sigma\langle ? \rangle S)^{\bot} \equiv \sigma\langle ! \rangle S^{\bot} \qquad \texttt{End}^{\bot} \equiv \texttt{End}$$

The linear type `Channel` S is a channel with session type S, and the interface to these channels is shown in Figure 3.6. A channel can be created by spawning a process that consumes the opposite end of the channel. A channel of session type `End` can be closed, resulting in a unit type. Users can send and receive values on channels, and any two channels of dual session types can be linked together, so that the communications of one are forwarded along to the other.

Lindley and Morris (2015) present an operational semantics of this language, which draws on process calculi like the π-calculus (Milner, 1999). In Chapter 4 we implement a different session-typed language and describe its operational behavior in more detail.

An echo server. Consider an echo server that receives a message, echoes that message back over the channel, then terminates. That protocol can be expressed as a session type and implemented as a process as follows:

```
EchoProtocol ≡ Lower String ⟨?⟩ Lower String ⟨!⟩ End
echoServer : Lift (Chan EchoProtocol ⊸ LUnit)
```

$$\frac{\Delta \vdash e : \texttt{Channel}\ S^{\perp} \multimap \texttt{LUnit}}{\Delta \vdash \texttt{spawn}\ e : \texttt{Channel}\ S}\ \textsc{new} \qquad \frac{\Delta \vdash e : \texttt{Channel End}}{\Delta \vdash \texttt{close}\ e : \texttt{LUnit}}\ \textsc{close}$$

$$\frac{\Delta \vdash e : \sigma \qquad \Delta' \vdash e' : \texttt{Channel}(\sigma\langle!\rangle S)}{\Delta, \Delta' \vdash \texttt{send}\ e\ e' : \texttt{Channel}\ S}\ \textsc{send} \qquad \frac{\Delta \vdash e : \texttt{Channel}(\sigma\langle?\rangle S)}{\Delta \vdash \texttt{receive}\ e : \sigma \otimes \texttt{Channel}\ S}\ \textsc{recv}$$

$$\frac{\Delta_1 \vdash e_1 : \texttt{Channel}\ S \qquad \Delta_2 \vdash e_2 : \texttt{Channel}(S^{\perp})}{\Delta_1, \Delta_2 \vdash \texttt{link}\ e_1\ e_2 : \texttt{LUnit}}\ \textsc{link}$$

Figure 3.6: Linear interface to session types.

```
echoServer ≡ suspend (λ̂ c. let (message, c) ≔ receive c in
                           let c ≔ send message c in
                           close c)
```

Meanwhile, a client of the echo server will check that they receive the same message that they sent. Note that `EchoProtocol`$^{\perp}$ is equal to `Lower String ⟨!⟩ Lower String ⟨?⟩ End`.

```
echoClient : Lift (Chan EchoProtocol⊥ ⊸ Lower Bool)
echoClient ≡ suspend (λ̂ c. let (!message,c) ≔ receive (send "ping" c) in
                           put (message == "ping"))
```

First class channels. Session types can be used to send channels themselves over a connection. For example, suppose Alice and Bob want to communicate via a session protocol S, but they need a third party to connect them. This intermediary has connections to Alice and Bob, but wants to verify their identities via passwords before connecting them. Here, the type `LMaybe` σ is just $\sigma \oplus$ `LUnit`; we write `LJust` for ι_1 and `LNothing` for $\iota_2()$.

```
intermediary : Suspend (Chan (Lower String ⟨?⟩ LMaybe (Chan S) ⟨!⟩ End)
                      ⊸ Chan (Lower String ⟨?⟩ LMaybe (Chan S⊥) ⟨!⟩ End)
                      ⊸ LUnit)
intermediary aliceChan bobChan ≡ suspend (
  let (!alicePasswd, aliceChan) ≔ receive aliceChan in
  let (!bobPasswd,   bobChan)   ≔ receive bobChan   in
  if verifyAlice alicePasswd && verifyBob bobPasswd
  then let c ≔ spawn (λ̂ c'. close (send (LJust c') aliceChan)) in
       close (send (LJust c) bobChan)
  else let () ≔ close (send LNothing aliceChan) in
       close (send LNothing bobChan) )
```

Notice that the intermediary can check the validity of Alice's and Bob's passwords via arbitrarily complex, non-linear procedures.

As a client, Alice can interact with the intemediary by sending her password and receiving back a channel, with which she can communicate with Bob. We write `aliceToBob` of type `Suspend (Chan S ⊸ LUnit)` for Alice's communication with Bob.

```
alice : Suspend (Chan (Lower String ⟨!⟩ LMaybe (Chan S) ⟨?⟩ End) ⊸ LUnit
alice ≡ suspend (λ̂ c. let c      ≔ send "my Password" c in
```

```
let (b, c) ≔ receive c in
let ()     ≔ close c in
case b of
| LJust bobChan → force aliceToBob ^ bobChan
| LNothing      → ())
```

A simple marketplace. Many real-life protocols describe diverging branches of communication, where users have a choice of which branch to take. For example, consider the following protocol for a simple online marketplace:

1. Receive an order from a customer;
2. If the item is in stock:
   (a) Send price to customer;
   (b) Receive credit card number from customer;
   (c) Send receipt to customer;
   (d) End transaction.
3. If the item is not in stock, end transaction.

The nested protocol, items 2a–2d, can be easily defined as follows:

```
InStockProtocol ≡ Lower Int ⟨!⟩ Lower Int ⟨?⟩ Lower String ⟨!⟩ End
inStockOp : String → Lift (Chan InStockProtocol ⊸ LUnit)
inStockOp item ≡ suspend (λ̂ c. let c       ≔ send (put (price item)) c in
                               let (!cc,c) ≔ receive c in
                               let c ≔ send (put "Thank you for your order!") c in
                               close c )
```

But how can we use session types to represent the choice of whether the item is in stock or not? We can encode this protocol as a session type $S_1\langle + \rangle S_2$ that makes a choice between two protocols. Dually, the session type $S_1\langle \& \rangle S_2$ *offers* a choice between two protocols.

$$S_1\langle + \rangle S_2 \equiv (\texttt{Channel } S_1^\perp \oplus \texttt{Channel } S_2^\perp)\langle ! \rangle \texttt{End}$$
$$S_1\langle \& \rangle S_2 \equiv (\texttt{Channel } S_1 \oplus \texttt{Channel } S_2)\langle ? \rangle \texttt{End}$$

The protocol of the online marketplace can now be described with the session type `MarketProtocol` and can be implemented as follows:

```
MarketProtocol ≡ Lower String ⟨?⟩ ( InStockProtocol ⟨+⟩ End )
marketplace : Lift (Chan MarketProtocol ⊸ LUnit)
marketplace ≡ suspend (λ̂ c. let (!item,c)  ≔ receive c in
                            if inStock item
                            then close (send (fork (inStockOp item)) c)
                            else close c)
```

3.7 Discussion

In the rest of this dissertation, we build on the embedded linear/non-linear lambda calculus described in this chapter. We consider several additional DSLs compatible with the LNL

embedding, including arrays and quantum computing. Some domains require changes to the basic LNL interface, but we will see that such changes do not necessarily affect the usability of the framework.

For example, the calculus in Chapter 6 does not include higher-order linear functions, and to compensate we replace the unary type constructor `Lift` σ with a binary type constructor `Box` σ τ, which represents first-order linear functions from σ to τ. Chapter 8 describes a language for quantum circuits that does not include the `put` constructor, as arbitrary host-language data structures should not be stored on a quantum circuit. Nevertheless, the LNL interface is still useful in this situation, since we can use >! to extract boolean values that have been obtained from quantum measurement on the circuit.

In Chapter 4 and Chapter 8 we show how to implement the LNL embedded framework in real host languages—Haskell and Coq respectively. The details of these implementations are specific to the tools available to each host language—in Haskell we use type classes to enforce linearity, and in Coq interactive tactics. But in both settings we find the basic LNL structure to be a robust interface to the linear embedding.

CHAPTER 4

Haskell Implementation

In this chapter we describe `LNLHask`, a framework for implementing linear EDSLs in Haskell.[4] `LNLHask` is based on the LNL embedding described in Chapter 3, and can be extended to a wide range of domain-specific applications.

Haskell has rich type classes and support for dependent types, which we exploit to enforce linearity. Our implementation draws on the techniques of previous embeddings of linear types in Haskell by Polakow (2015) and Eisenberg *et al.* (2012). Compared to the prior work, our implementation makes several contributions.

- The LNL interface lets us use arbitrary Haskell data structures in the linear type `Lower` α and use existing libraries to manipulate that data. Prior work is based on the traditional presentations of linear logic using !, and as a result has limited access to host-language data.

- `LNLHask` is an extensible framework that can be instantiated in a variety of domain-specific applications. In this chapter we describe two particular instantiations of the framework: mutable functional arrays and session types. The implementations of these examples require at most a few hundred lines of code on top of the base `LNLHask` library.

- Haskell has good support for monadic programming, making it a natural choice in which to implement the linearity monad described in Section 3.4. The monadic programming style arises directly from the LNL model and makes it easy to write high-level functional code such as an in-place quicksort algorithm (see Section 4.5).

- `LNLHask` performs linear type checking by drawing on several advanced language extensions for dependently typed programming provided by the Glasgow Haskell Compiler (GHC) version 8.[5] The newer features result in several improvements compared to the design of previous embeddings, which we discuss in Section 4.7. The implementation of `LNLHask` can also be seen as a case study for using state-of-the-art dependent types in Haskell.

Implementing the LNL model requires us to make a number of important design decisions that were left out of the discussion in Chapter 3 about how to represent linear types and typing contexts and how to enforce linearity. In `LNLHask`, variables are represented by type-level natural numbers and typing contexts are represented by type-level lists. Variable

[4] `https://github.com/jpaykin/LNLHaskell`
[5] `https://www.haskell.org/ghc/`

binding is by higher-order abstract syntax (Pfenning and Elliott, 1988), meaning that linear functions are represented as Haskell functions. To enforce linearity, we use type class constraints to define relations on typing contexts, such as the fact that two typing contexts are disjoint. Typing judgments are defined using a final tagless encoding (Kiselyov, 2012), where the interface to the embedded language is presented as an extensible type class whose implementation is opaque to the user.

The rest of this chapter presents the details of `LNLHask` and gives examples of how to use the framework. Section 4.1 introduces dependently-typed programming in Haskell. Section 4.2 describes the implementation of `LNLHask` including how to encode linear types, type checking, and linear expressions. Section 4.3 describes how to evaluate linear programs, and Section 4.4 develops monadic programming techniques; both chapters use linear file handles as a running example. Sections 4.5 and 4.6 develop two additional instantiations of the LNL framework—mutable arrays and session types. Finally, Section 4.7 discusses the design decisions of `LNLHask` and related work.

4.1 Dependent types in Haskell

In this chapter we assume the reader has basic familiarity with Haskell programming including inductive data types, GADT's, and type classes, but we take this opportunity to introduce some more advanced techniques for dependently-typed programming in Haskell.

In the past several years, Haskell has incorporated many features of dependently-typed languages. However, there are still several differences between dependent types in Haskell and languages with full dependent types like Coq or Agda. Haskell lacks Π and Σ types, and enforces a strict distinction between types and terms. The distinction between types and terms means that all types are erased at runtime, but it also means that types and terms live in different namespaces, which leads to complications when terms appear in types.

To get around the distinction between terms and types, every data type in Haskell is simultaneously a *data kind,* and the term constructors for those data types are promoted to type constructors (Yorgey *et al.*, 2012). For example, $[\kappa]$ is the kind of type-level lists holding types of kind κ, with constructors `'[] :: [`κ`]` and `(`α` ': `α`s) :: [`κ`]`, for α :: κ and α`s :: [`κ`]`. The tick marks on the constructors `'[]` and `':` distinguish the type constructors from the corresponding term constructors, though we can omit the tick mark when the promoted constructor is unambiguous.

Because Haskell does not have Π-types, terms that appear in a type as promoted data kinds cannot also appear in a term. However, it is often useful to have a dynamic representation of such type-level data. In the case of type-level natural numbers of type `Nat`, defined in `GHC.TypeLits`, the type class `KnownNat n` has a method to produce the integer corresponding to the type n.

```
natVal :: KnownNat n ⇒ Proxy n → Integer
```

A more general approach to term-level representations of type-level data is singletons (Eisenberg and Weirich, 2012), but we do not need them in this work.

The type `Proxy n` in the type of `natVal` identifies the type-level argument n.

```
data Proxy (α :: κ) = Proxy
```

For example, we can instantiate `natVal` with the type-level `Nat` 0 to obtain the integer 0.

```
myZero = natVal (Proxy :: Proxy 0)
```

Haskell's type class mechanism will generate an instance of the type class `KnownNat n` for any concrete natural number *e.g.,* 0.

Proxies can also be avoided by using the function `natVal'` below, which uses visible type application to specify the value of n (Eisenberg *et al.*, 2016).

```
natVal' :: forall n. KnownNat n ⇒ Integer
natVal' = natVal @n Proxy
```

Here, the `forall` quantifier brings `n` into scope of the function body, and `@n` denotes a type argument to `natVal`.

Because we use type-level `Nat`'s liberally in this work, we introduce here some basic operations on them. The `TypeLits` library provides a type-level comparison operator `CmpNat :: Nat → Ord` that produces a type of kind `Ordering = LT | EQ | GT`. We can use closed type families to pattern match against promoted orderings as follows:

```
type family CompareOrd (ord :: Ordering) (lt :: α) (eq :: α) (gt :: α) :: α where
  CompareOrd LT lt eq gt = lt
  CompareOrd EQ lt eq gt = eq
  CompareOrd GT lt eq gt = gt
```

The `CmpNat` type family lets us statically compare two type-level nats, but we often want dynamic comparison as well. To do this, we define a data type `COrdering m n` that encodes the type-level behavior of `CmpNat m n`.

```
data COrdering m n where
  CLT :: Dict (CmpNat m n ~ LT) → COrdering m n
  CEQ :: Dict (CmpNat m n ~ EQ) → COrdering m n
  CGT :: Dict (CmpNat m n ~ GT) → COrdering m n
```

The type `Dict c`, defined in the `constraints` package,[6] encodes a dictionary constraint—either a type class or an equality constraint of the form $\alpha \sim \beta$. The `Dict` type is used to carry around a first-class constraint argument, which can be used later by pattern matching against the term.

```
data Dict :: Constraint → * where
  Dict :: a ⇒ Dict a
withDict :: Dict a → (a ⇒ r) → r
withDict d r = case d of Dict → r
```

In the `LNLHask` development, the constructors of `COrdering m n` carry additional information, such as the fact that `CmpNat n m ~ GT` in the constructor of `CLT`; for the sake of this presentation we have simplified the types to one constraint per constructor.

By comparing the term-level representations of type-level `Nat`'s, we can generate type-level information about the behavior of `CmpNat`.

```
cmpNat :: (KnownNat m, KnownNat n) ⇒ Proxy m → Proxy n → COrdering m n
cmpNat m n = case compare (natVal m) (natVal n) of
               LT → CLT $ unsafeCoerce (Dict :: Dict ())
               EQ → CEQ $ unsafeCoerce (Dict :: Dict ())
               GT → CGT $ unsafeCoerce (Dict :: Dict ())
```

The use of the primitive `unsafeCoerce` :: $\alpha \to \beta$ here is justified because of the meta-reasoning that `natVal m < natVal n` implies `m < n`, and so on.

[6] `https://hackage.haskell.org/package/constraints`

4.2 Linear types and type checking

We implement linear types as a data type, promoted to a data kind. To start, consider linear functions and unit types. We use the infix notation $\sigma \multimap \tau$ as a synonym for the promoted type `Lolli` σ τ.

```
data LType = LUnit | Lolli LType LType
type σ ⊸ τ = Lolli σ τ
```

Variables are represented as type-level natural numbers, and typing contexts as type-level lists that map variables to `LType`s.

```
type Ctx = [(Nat,LType)]
```

We use γ as a meta-variable for typing contexts in this chapter. (Haskell requires that variables start with lower case letters, so the usual meta-variable Δ is not acceptable.)

Type checking linear expressions requires two main relations on typing contexts: (1) that γ_1 and γ_2 are disjoint and can be merged; and (2) that x does not occur in the domain of γ, and can be added to it.

Example 4.2.1. *Consider a linear typing judgment $y : \tau \vdash (\hat{\lambda} x.x)\hat{\ } y : \tau$. In order to check that this judgment holds, we need to split the context $y : \tau$ into two parts: $\Delta_1 \vdash \hat{\lambda} x.x : \sigma \multimap \tau$ and $\Delta_2 \vdash y : \sigma$. Since y is just a variable, we can infer that Δ_2 must have the form $y : \sigma$. From there it is reasonable to infer that Δ_1 is the remainder of the context that does not occur in Δ_1—so $\Delta_1 = \Delta - \Delta_2 = \varnothing$.*

To check that $\varnothing \vdash \hat{\lambda} x.x : \sigma \multimap \tau$, we must check that x does not occur in $\varnothing$, and check that $x : \sigma \vdash x : \tau$. Unifying these constraints, we conclude that $\sigma = \tau$.

In this example, type checking is a bidirectional process. To implement such a process in Haskell, we start by defining a number of type-level functions on typing contexts, and then combining them into type classes to implement the bidirectional reasoning.

4.2.1 Type families

Figure 4.1 defines several type families on linear variables x and typing contexts γ. In each of them, we enforce the invariant that typing contexts are sorted with respect to their domain.

- `Fresh` γ produces a variable that does not occur in the context γ.
- `Lookup` γ `x` returns `Just` σ if (x, σ) is in γ, and `Nothing` if x is not in the domain of γ.
- `AddF x` σ γ adds the binding `(x,`σ`)` to γ if x does not already occur in γ; if x does occur in γ, then `AddF x` σ γ is undefined.
- `MergeF` γ_1 γ_2 combines the typing contexts γ_1 and γ_2 if they are disjoint; if they have overlapping variables, their merge is undefined.
- `Remove x` γ removes the variable x from γ; it is undefined if x does not occur in γ
- `Div` γ γ_0 removes the variables in γ_0 from γ; if $\gamma_0 \neg \subseteq \gamma$, then the result is undefined.

```
type family Fresh (γ :: Ctx) :: Nat where
  Fresh '[]          = 0
  Fresh '[ '(x,_) ] = x+1
  Fresh (_ ': γ)     = Fresh γ
type family Lookup (γ :: Ctx) (x :: Nat) :: Maybe LType where
  Lookup '[] _         = Nothing
  Lookup ('(y,σ):γ) x = CompareOrd (CmpNat x y)
                          Nothing       -- if x < y
                          (Just σ)      -- if x = y
                          (Lookup γ x)  -- if x > y
type family AddF (x :: Nat) (σ :: LType) (γ :: Ctx) :: Ctx where
  AddF x σ '[] = '[ '(x,σ) ]
  AddF x σ ('(y,τ) ': γ) = CompareOrd (CmpNat x y)
                             ('(x,σ) ': '(y,τ) ': γ) -- if x < y
                             (AddError x ('(y,τ) γ)) -- if x = y
                             ('(y,τ) ': AddF x σ γ)  -- if x > y
type family MergeF (γ1 :: Ctx) (γ2 :: Ctx) :: Ctx where
  MergeF '[] γ2 = γ2
  MergeF ('(x,σ) ': γ1) γ2 = AddF x σ (MergeF γ1 γ2)
type family Remove (x :: Nat) (γ :: Ctx) :: Ctx where
  Remove x '[]           = RemoveError x '[]
  Remove x ('(y,σ) ': γ) = CompareOrd (CmpNat x y)
                             (RemoveError x ('(y,σ) : γ)) -- if x < y
                             γ                            -- if x = y
                             ('(y,σ) ': Remove x γ)       -- if x > y
type family Div (γ :: Ctx) (γ0 :: Ctx) :: Ctx where
  Div γ '[]            = γ
  Div γ ('(x,_) ': γ0) = Remove x (Div γ γ0)
```

Figure 4.1: Type families over linear typing contexts, enforcing the invariant that typing contexts are sorted. The custom type errors `AddError` and `RemoveError` provide better error reporting—see Section 4.7.

4.2.2 Type classes

As we saw in Example 4.2.1, it is not enough to compute the merge of two typing contexts; we must also must be able to partition a context using type inference where possible. Notice that, if `γ' ~ AddF x σ γ`, then `Remove x γ' ~ γ` and `Lookup γ' x ~ Just σ`. We can encode this reasoning into a multi-parameter type class with functional dependencies, which tell Haskell how to search for proofs of this judgment.

```
class ( γ' ~ AddF x σ γ, γ ~ Remove x γ', Lookup γ' x ~ Just σ
      , KnownNat x, WFCtx γ, WFCtx γ')
    ⇒ CAddCtx (x :: Nat) (σ :: LType) (γ :: Ctx) (γ' :: Ctx)
     | x σ γ → γ', x γ' → σ γ
```

The `KnownNat` and `WFCtx` constraints say that `Nat`s and `Ctx`s respectively are well-formed, and they are true for any concrete types. In particular, the `WFCtx γ` constraint ensures that γ behaves reasonably with respect to merging and dividing by the empty context.

```
type WFCtx γ = (Div γ '[] ~ γ, Div  γ γ ~ '[], MergeF '[] γ ~ γ, MergeF γ '[] ~ γ)
```

The instance declaration for the `CAddCtx` type class is trivial:

```
instance ( γ' ~ AddF x σ γ, γ ~ Remove x γ', Lookup γ' x ~ Just σ
         , KnownNat x, WFCtx γ, WFCtx γ')
    ⇒ CAddCtx (x :: Nat) (σ :: LType) (γ :: Ctx) (γ' :: Ctx)
```

This same technique can be used to generalize the `MergeF` type family into a type class such that knowing any two of γ_1, γ_2, or γ is enough to infer the third.

```
class    ( γ ~ MergeF γ1 γ2, γ ~ MergeF γ2 γ1, Div γ γ2 ~ γ1, Div γ γ1 ~ γ2
         , WFCtx γ1, WFCtx γ2, WFCtx γ )
        ⇒ CMerge γ1 γ2 γ | γ1 γ2 → γ, γ1 γ → γ2, γ2 γ → γ1
instance ( γ ~ MergeF γ1 γ2, γ ~ MergeF γ2 γ1, Div γ γ2 ~ γ1, Div γ γ1 ~ γ2
         , WFCtx γ1, WFCtx γ2, WFCtx γ )
        ⇒ CMerge γ1 γ2 γ
```

4.2.3 Typing Judgments

A linear expression $\gamma \vdash e : \tau$ is represented as a Haskell term `e :: exp γ τ` in final tagless style (Carette *et al.*, 2009), where the typing judgment `exp :: Ctx → LType → Type` is given as a type class, the methods of which correspond to typing rules.

```
class HasLolli (exp :: Ctx → LType → Type) where
  var  :: KnownNat x ⇒ proxy x → Var exp x σ
  λ̂    :: (x ~ Fresh γ, CAdd x σ γ γ')
       ⇒ (Var exp x σ → exp γ' τ) → exp γ (σ ⊸ τ)
  (^) :: CMerge γ1 γ2 γ ⇒ exp γ1 (σ ⊸ τ) → exp γ2 σ → exp γ τ
type Var exp x σ = exp '[ '(x,σ) ] σ
```

The `HasLolli` type class asserts that the typing judgment `exp` provides variables, abstraction ($\hat{\lambda}$), and application (^) operations.

- Variables are constructed from proxies, where `Var exp x σ` refers to a term of the singleton typing context `x :: exp '[ '(x,σ) ] σ`.

- The type of application corresponds closely to the inference rules given in Chapter 3, where `CMerge` encodes the disjoint union of contexts.

- Abstraction uses higher-order abstract syntax to bind variables in a linear function. Let's take a look at the type of $\hat{\lambda}$ without the type class constraints:

```
(Var exp x σ → exp γ' τ) → exp γ (σ ⊸ τ)
```

This type says that, in order to construct a linear function $\sigma \multimap \tau$, it suffices to provide an ordinary Haskell function from variables of type σ to expressions of type τ. In order to ensure that the bound variable is used linearly in a term, we have the following constraints:

```
(x ~ Fresh γ, CAdd x σ γ γ')
```

The first constraint says that x is fresh in γ, and the second constraint says that the body of the function, of type `exp γ' τ`, satisfies the relation $\gamma' = \gamma, x : \sigma$. Put in a more functional notation, the type of $\hat{\lambda}$ could be written as

```
(exp [x:σ] σ → exp (γ,x:σ) τ) → exp γ (σ ⊸ τ)
```

The HOAS encoding leads to fairly natural-looking code. The identity function is `λ̂ (\x → x)`, while composition is defined as:

```
compose :: HasLolli exp ⇒ exp '[] ((τ2 ⊸ τ3) ⊸ (τ1 ⊸ τ2) ⊸ (τ1 ⊸ τ3))
compose = λ̂ $ \g → λ̂ $ \f → λ̂ $ \x → g ^ (f ^ x)
```

We do not have to add any special infrastructure to handle polymorphism; Haskell takes care of it for us.

4.2.4 Linear connectives

The final tagless typing judgment can be easily extended by defining new type classes that offer different syntax. This makes it easy to extend the language to other operators of linear logic, such as units, products, and sums.

For the linear multiplicative unit, we have the following class:

```
class HasLUnit exp where
  unit    :: exp '[] LUnit
  letUnit :: CMerge γ1 γ2 γ ⇒ exp γ1 LUnit → exp γ2 τ → exp γ τ
```

For the tensor product, we first need to extend the syntax of linear types. We could add a constructor corresponding to $\otimes$ directly to the definition of `LType`, but doing so would commit to a particular choice of linear connectives. Instead, we extend `LType`'s with `MkLType`, a way to existentially introduce a new linear type:

```
data LType where MkLType :: ext LType → LType
```

Extensions, denoted with the meta-variable `ext :: Type → Type`, are universally quantified in `MkLType`. The multiplicative product can be encoded as an extension `TensorExt`:

```
data TensorExt ty = MkTensor ty ty
type σ ⊗ τ = MkLType (MkTensor σ τ)
```

We overload the notation $(\otimes)$ as syntax for terms. The HOAS version of the `let` binding, which we write `letPair`, introduces two variables, similar to the type of $\hat{\lambda}$ above.

```
class HasTensor exp where
  (⊗) :: CMerge γ1 γ2 γ ⇒ exp γ1 τ1 → exp γ2 τ2 → exp γ (τ1 ⊗ τ2)
  letPair :: ( CAdd x1 σ1 γ2 γ2', CAdd x2 σ2 γ2' γ2''
```

```
    , x₁ ~ Fresh γ₂, x₂ ~ Fresh γ₂'
    , CMerge γ₁ γ₂ γ)
  ⇒ exp γ₁ (σ₁ ⊗ σ₂)
  → ((Var exp x₁ σ₁, Var exp x₂ σ₂) → exp γ₂'' τ)
  → exp γ τ
```

The variables $\mathtt{x}_1$ and $\mathtt{x}_2$ are fresh variables generated by γ_2 and $\gamma_2' = \gamma_2, x_1 : \sigma_1$ respectively The continuation of the `letPair` is in the context $\gamma_2'' = \gamma_2, x_1 : \sigma_1, x_2 : \sigma_2$. As an inference rule, it is clear how `letPair` corresponds to the usual `let` binding for ⊗.

$$\frac{\gamma_1 \vdash e : \sigma_1 \otimes \sigma_2 \qquad \gamma_2, x_1 : \sigma_1, x_2 : \sigma_2 \vdash f(\mathtt{var}\ x_1, \mathtt{var}\ \mathtt{x}_2) : \tau}{\gamma_1, \gamma_2 \vdash \mathtt{letPair}\ e\ f : \tau}$$

We often use `letPair` infix, as in `λ̂ $ \x → x ‘letPair‘ \(y,z) → z ⊗ y`, of type $\sigma \otimes \tau \multimap \tau \otimes \sigma$. It would certainly be more natural to write `λ̂ $ \(y,z) → z ⊗ y` directly, but type checking for nested pattern matching has turned out to be a difficult problem. We can however define a top-level pattern match `λ̂pair`, and write our example as `λ̂pair $ \ (y,z) → z ⊗ y`. We discuss the issue of type checking and nested pattern matching more in Section 4.7.

The interfaces for additive sums, products, and units are shown in Figure 4.2.

4.2.5 The `Lift` and `Lower` Types

The LNL connective `Lower` can be added to the linear language using `MkLType` as we did for the other linear connectives. However, `Lower` takes an argument of kind `Type`—the kind of Haskell types.

```
data LowerExp ty = MkLower Type
type Lower α = MkLType (MkLower α)
```

The introduction and elimination forms for `Lower` are the same as those presented in Chapter 3. Given a Haskell term `a :: α`, there is a linear expression `put a :: exp '[] (Lower α)`, and an expression `e :: exp γ₁ (Lower α)` can be eliminated against `f :: α → exp γ₂ τ`, written `e >! f`.

```
class HasLower exp where
  put :: α → exp '[] (Lower α)
  (>!) :: CMerge γ₁ γ₂ γ
      ⇒ exp γ₁ (Lower α) → (α → exp γ₂ τ) → exp γ τ
```

We define `Lift` as an ordinary Haskell data type with a single constructor `Suspend`, and we define `force` by pattern matching against `Suspend`.

```
data Lift exp τ = Suspend (exp '[] τ)
force :: Lift exp τ → exp '[] τ
force (Suspend e) = e
```

For convenience, we define `HasMELL` for the class of constraints corresponding to Multiplicative Exponential Linear Logic (⊸, ⊗, `LUnit`, and `Lower`), and `HasMALL` for that class with the addition of Additive sums and products (& and ⊕).

```
type HasMILL exp = (HasLower exp, HasLolli exp, HasLUnit exp, HasTensor exp)
type HasMALL exp = (HasMILL exp, HasWith exp, HasPlus exp)
```

```
-- Additive Sum --
data PlusSig ty = PlusSig ty ty
type (⊕) (σ :: LType) (τ :: LType) = MkLType ('PlusSig σ τ)

class HasPlus exp where
  inl :: exp γ τ₁ → exp γ (τ₁ ⊕ τ₂)
  inr :: exp γ τ₂ → exp γ (τ₁ ⊕ τ₂)
  caseof :: ( CAddCtx x σ₁ γ₂ γ₂₁, CAddCtx x σ₂ γ₂ γ₂₂
            , x ~ Fresh γ, CMerge γ₁ γ₂ γ )
         ⇒ exp γ₁ (σ₁ ⊕ σ₂)
         → (Var exp x σ₁ → exp γ₂₁ τ)
         → (Var exp x σ₂ → exp γ₂₂ τ)
         → exp γ τ

-- Additive Product --
data WithSig ty = WithSig ty ty
type (σ :: LType) & (τ :: LType) = MkLType ('WithSig σ τ)

class HasWith exp where
  (&) :: exp γ τ₁ → exp γ τ₂ → exp γ (τ₁ & τ₂)
  proj₁ :: exp γ (τ₁ & τ₂) → exp γ τ₁
  proj₂ :: exp γ (τ₁ & τ₂) → exp γ τ₂

-- Zero --
data ZeroSig ty = ZeroSig
type Zero = MkLType 'ZeroSig

class HasZero exp where
  absurd :: CMerge γ₁ γ₂ γ ⇒ exp γ₁ Zero → exp γ τ

-- Top --
data TopSig ty = TopSig
type Top = MkLType 'TopSig

class HasTop exp where
  abort :: exp γ Top
```

Figure 4.2: Haskell interface to linear additive connectives

```
curry   :: HasMILL exp ⇒ Lift exp ((σ₁ ⊗ σ₂ ⊸ τ) ⊸ σ₁ ⊸ σ₂ ⊸ τ)
curry   = Suspend . λ̂ $ \f → λ̂ $ \x₁ → λ̂ $ \x₂ → f ^ (x₁ ⊗ x₂)
uncurry :: HasMILL exp ⇒ Lift exp ((σ₁ ⊸ σ₂ ⊸ τ) ⊸ σ₁ ⊗ σ₂ ⊸ τ)
uncurry = Suspend . λ̂ $ \f → λ̂ $ \x →
          x `letPair` \(x₁,x₂) → f ^ x₁ ^ x₂

type Bang τ = Lower (Lift τ)
dup  :: HasMELL exp ⇒ Lift exp (Bang τ ⊸ Bang τ ⊗ Bang τ)
dup  = Suspend . λ̂ $ \x → x >$ \a → put a ⊗ put a
drop :: HasMELL exp ⇒ Lift exp (Bang τ ⊸ LUnit)
drop = Suspend . λ̂ $ \x → x >$ \_ → unit
```

Figure 4.3: Examples of linear programs embedded in Haskell

Figure 4.3 shows some examples of linear code embedded in Haskell. This includes currying and uncurrying, as well as operations for the type `Bang` τ ≡ `Lower (Lift` τ`)`.

4.2.6 File Handles

Recall the example of linear file handles from Section 3.3, where linearity prevents use-after-close errors and memory leaks. We can give an interface for file handles in `LNLHask` as a type class that builds off of MELL.

```
class HasMELL exp ⇒ HasFH exp where
  open  :: String → exp '[] Handle
  read  :: exp γ Handle → exp γ (Handle ⊗ Lower Char)
  write :: exp γ Handle → Char → exp γ Handle
  close :: exp γ Handle → exp γ LUnit
```

The `open` and `write` operations take ordinary Haskell data (strings and characters) as input. Because strings are just lists of chars, we can write an entire string to a file by folding over the string.

```
writeString :: HasFH exp ⇒ String → exp γ Handle → exp γ Handle
writeString s h = foldl write h s
```

File handles interact nicely with the other linear connectives. The following function reads a character from a file and writes that same character back to the file twice:

```
readWriteTwice :: HasFH exp ⇒ Lift exp (Handle ⊸ Handle)
readWriteTwice = λ̂ $ \h → read h `letPair` \(h,x) →
                          x >! \c →
                          writeString [c,c] h
```

The type class mechanism really does enforce linearity, which prevents domain-speciifc errors like read-after-close. For example, the following code does not type check:

```
readAfterClose :: Lift exp (Handle ⊸ Handle ⊗ Lower Char)
readAfterClose = λ̂ $ \h → close h `letUnit` read h -- type error
```

4.3 Running linear programs

The goal of embedded DSLs is not just to express domain-specific programs, but to actually run those programs. In this section we define two different implementations of the type classes `HasLolli`, `HasFH`, *etc.* of the previous sections.

4.3.1 Values and effects

Instances of `LNLHask` have three components: a typing judgment `exp :: Ctx → LType → Type`; a value judgment `val :: LType → Type`, and a monadic effect `m :: Type → Type`. For example, the type of values associated with file handles will be Haskell's primitive file handles, and the monadic effect will be `IO`. Different domains require different representations of values and effects, so we structure these components as data and type families respectively.

```
type Exp = Ctx → LType → Type
data family LVal   (exp :: Exp) (τ :: LType) :: Type
type family Effect (exp :: Exp) :: Type → Type
```

To evaluate an expression, we use evaluation environments that map free variables to values. Each evaluation environment ρ `:: ECtx exp` γ is indexed by its respective typing context γ, and we maintain the invariant that, if $x : \sigma$ is in the domain of γ, then ρ maps x to a value of type σ. The type-safe interface to evaluation contexts is given as follows:

```
lookupECtx :: (KnownNat x, Lookup γ x ~ Just σ)
           ⇒ proxy x → ECtx exp γ → LVal exp σ
eEmpty     :: ECtx exp '[]
addECtx    :: KnownNat x
           ⇒ proxy x → LVal exp σ → ECtx exp γ → ECtx exp (AddF x σ γ)
splitECtx  :: γ ~ MergeF γ1 γ2
           ⇒ ECtx exp γ → (ECtx exp γ1 , ECtx exp γ2)
```

The function `eEmpty` is an empty evaluation context, `addECtx` adds a value to an evaluation context, and `splitECtx` partitions an evaluation context according to a valid merge.

Under the hood, evaluation contexts are implemented as simply-typed maps—specifically integer maps from the `containers` library.[7] The map holds values whose types are existentially hidden, so that evaluation contexts can hold values of different types.

```
data EVal sig where
  EVal :: LVal sig σ → EVal sig
newtype ECtx sig γ = ECtx (M.IntMap (EVal sig))
```

The function `lookupECtx` uses `unsafeCoerce` to coerce an untyped `EVal` into a well-typed `LVal`; as long as the evaluation context is always constructed from `addECtx`, this coercion will always succeed.

```
lookupECtx x (ECtx ρ) = let v = ρ ! natVal x
                        in unsafeCoerce v
```

The `eEmpty` and `splitECtx` operations are defined using the corresponding operations on `IntMaps`.

```
eEmpty = ECtx empty
addECtx x v (ECtx ρ) = ECtx $ insert (natVal x) (EVal v) γ
```

[7] https://hackage.haskell.org/package/containers

Finally, `split` is a no-op on its input ρ, since if $x : \sigma$ is in the domain of either γ_1 or γ_2, then it is the domain of `MergeF` γ_1 γ_2, and thus in the domain of ρ.

```
splitECtx (ECtx ρ) = (ECtx ρ, ECtx ρ)
```

The big-step operational semantics of the embedded language is defined as a type class `Eval exp` with a method `eval` that, given an evaluation context for γ and an expression $\gamma \vdash e : \tau$, evaluates e to a value inside the monad `Effect exp`.

```
class Eval exp where
  eval :: Monad (Effect exp) ⇒ exp γ τ → ECtx exp γ → Effect exp (LVal exp τ)
```

Deep versus shallow embeddings. EDSLs may either be *deeply* or *shallowly* embedded in their host language. A shallow embedding is one in which embedded programs are represented as host-language programs, and evaluating embedded programs is done by simply evaluating the program in the host language. On the other hand, a deep embedding encodes the embedded program in some data in the host language, for example as an abstract syntax tree.

There are many tradeoffs between deep and shallow embeddings. A shallow embedding may be more efficient, but in a deep embedding it is possible to reason about small-step operational semantics and perform meta-operations like optimizations, which is not possible in a shallow embedding. `LNLHask` supports both deep and shallow embeddings (Kiselyov, 2012).

4.3.2 A Deep Embedding

First we consider a deep embedding, where linear lambda terms are defined as a GADT in Haskell. The `Deep` data type bears a strong resemblance to the `HasLolli` type class, although without higher-order abstract syntax.

```
data Deep γ τ where
  Var :: KnownNat x ⇒ Proxy x → Deep '[ '(x,σ) ] τ
  Abs :: CAddCtx x σ γ γ'
      ⇒ Proxy x → Deep γ' τ → Deep γ (σ ⊸ τ)
  App :: CMerge γ1 γ2 γ
      ⇒ Deep γ1 (σ ⊸ τ) → Deep γ2 σ → Deep exp γ τ
```

To instantiate the `HasLolli` type class, it is enough to produce a proxy variable for the fresh variable being generated by the higher-order abstract syntax.

```
instance HasLolli Deep where
  λ̂         :: forall x (σ :: LType) γ γ' γ'' τ.
               (x ~ Fresh γ, CAddCtx x σ γ γ')
            ⇒ (Var Deep x σ → Deep γ' τ) → Deep γ (σ ⊸ τ)
  λ̂ f       = Abs x (f $ Var x) where x = (Proxy :: Proxy x)
  e1  ^  e2 = App e1 e2
```

Values are given as data instances according to their `LType`. A value of type $\sigma \multimap \tau$ is a closure containing an evaluation context paired with the body of the abstraction.

```
data instance LVal Deep (σ ⊸ τ) where
  VAbs :: CAddCtx x σ γ γ'
       ⇒ ECtx Deep γ → Proxy x → Deep γ' τ → LVal Deep (σ ⊸ τ)
```

Evaluating programs in the deep embedding is done by case analysis on `Deep` expressions.

```
instance Eval Deep where
  eval :: Monad (Effect Deep) ⇒ Deep γ τ → ECtx Deep γ → Effect Deep (LVal Deep τ)
  eval (Var x) (ECtx γ) = return $ lookupECtx x γ
  eval (Abs x e) ρ      = return $ VAbs γ x e
  eval (App (e₁ :: Deep γ₁ (σ ⊸ τ)) (e₂ :: Deep γ₂ σ)) ρ =
    do let (ρ₁,ρ₂) = splitECtx @γ₁ @γ₂ ρ
       VAbs ρ' x e₁' ← eval e₁ ρ₁
       v₂            ← eval e₂ ρ₂
       eval e₁' (addECtx x v₂ ρ')
```

Extending the deep embedding. To make the deep embedding extensible we extend the `Deep` data class by a parameterized typing judgment called a *domain*.

```
data Deep γ τ where ⋯
  Dom :: Domain dom ⇒ dom γ τ → Deep γ τ
```

A domain `dom` has kind `Ctx → LType → Type`, the same as the kind of typing judgments, and will be instantiated with AST's corresponding to different language extensions. The domain corresponding to the `Lower` type has the following form:

```
data LowerDom :: Ctx → LType → Type where
  Put     :: α → LowerDom '[] (Lower α)
  LetBang :: CMerge γ₁ γ₂ γ ⇒ Deep γ₁ (Lower α) → (α → Deep γ₂ τ) → LowerDom γ τ
data instance HasLower Deep where
  put    = Dom . Put
  e >! f = Dom $ LetBang e f
data instance LVal Deep (Lower α) = VPut α
```

Notice that the data structure `LowerDom` exactly mirrors the type class `HasLower`.

The `Domain` type class gives a way to evaluate the new domain.

```
class Domain dom where
  evalDomain :: Monad (Effect Deep)
             ⇒ dom γ σ → ECtx Deep γ → Effect Deep (LVal Deep σ)
instance Domain LowerExp where
  evalDomain (Put a) _ = return $ VPut a
  evalDomain (LetBang (e₁ :: Deep γ₁ (Lower α)) (e₂ :: α → Deep γ₂ τ)) ρ =
    do let (ρ₁,ρ₂) = split @γ₁ @γ₂ ρ
       VPut a ← eval e₁ ρ₁
       eval (e₂ a) ρ₂
```

The `eval` function can now be extended to arbitrary domains:

```
eval (Dom e) γ = evalDomain e γ
```

Implementing the other linear connectives are fairly straightforward. The values associated with various connectives are shown in Figure 4.4.

File handles. We can extend the deep embedding to file handles by specifying the effect of the language to be `IO` and taking values of type `Handle` to be built-in `IO` file handles.

```
type instance Effect Deep = IO
data instance LVal Deep Handle = VHandle (IO.Handle)
```

The file handle domain mirrors the structure of the `HasFH` type class.

```
data instance LVal Deep LUnit     = VUnit
data instance LVal Deep (σ1 ⊗ σ2) = VPair (LVal Deep σ1) (LVal Deep σ2)
data instance LVal Deep (σ1 ⊕ σ2) = VInl (LVal Deep σ1) | VInr (LVal Deep σ2)
data instance LVal Deep LTop      = VError
data instance LVal Deep (σ1 & σ2) where
  VWith :: ECtx Deep γ → Deep γ σ1 → Deep γ σ2  → LVal Deep (σ1 & σ2)
```

Figure 4.4: Values in the deep embedding associated with various linear connectives.

```
data FHExp :: Ctx → LType → Type where
  Open  :: String → FHExp '[] Handle
  Read  :: Deep γ Handle → FHExp γ (Handle ⊗ Lower Char)
  Write :: Deep γ Handle → Char → FHExp γ Handle
  Close :: Deep γ Handle → FHExp γ LUnit
```

All that remains is to give an instance of the `Domain` class, invoking the Haskell `IO` library.

```
instance Domain FHExp where
  evalDomain (Open s)    _ = VHandle <$> IO.openFile s IO.ReadWriteMode
  evalDomain (Read e)    ρ = do VHandle h ← eval e ρ
                               c ← IO.hGetChar h
                               return $ VPair (VHandle h) (VPut c)
  evalDomain (Write e c) ρ = do VHandle h ← eval e ρ
                               IO.hPutChar h c
                               return $ VHandle h
  evalDomain (Close e)   ρ = do VHandle h ← eval e ρ
                               IO.hClose h
                               return VUnit
```

4.3.3 A Shallow Embedding

Next we consider a shallow embedding, where `exp` γ τ is represented as a monadic function from evaluation contexts for γ to values of type τ. Evaluation just unpacks this function.

```
newtype Shallow γ τ =
  SExp { runSExp :: ECtx Shallow γ → Effect Shallow (LVal Shallow τ) }
class Eval Shallow where
  eval = runSExp
```

Values in the shallow embedding are almost the same as those in the deep embedding, except that a value of type $\sigma \multimap \tau$ in the shallow embedding is represented as a function from values of type σ to values of type τ, instead of as an explicit closure.

```
newtype instance LVal Shallow (σ ⊸ τ) =
  VAbs (LVal Shallow σ  →  Effect Shallow (LVal Shallow τ))
```

We can show that the shallow embedding simulates all the features of our linear language by instantiating the type classes for `HasLolli`, `HasLower`, *etc.* Unsurprisingly, all of these constructions mirror the evaluation functions from the deep embedding.

```
instance Monad (Effect Shallow) ⇒ HasLolli Shallow where
  λ̂ f = SExp $ \(γ :: ECtx Shallow γ) → return . VAbs $ \s →
          let x = (Proxy :: Proxy (Fresh γ))
```

```
          in runSExp (f $ var x) (add x s γ)
  f ^ e = SExp $ \γ → do let (γ1,γ2) = split γ
                         VAbs f' ← runSExp f γ1
                         v       ← runSExp e γ2
                         f' v
instance Monad (Effect Shallow) ⇒ HasLower Shallow where
  put a  = SExp $ \_ → return $ VPut a
  e >! f = SExp $ \γ → do let (γ1,γ2) = split γ
                          VPut a ← runSExp e γ1
                          runSExp (f a) γ2
```

File Handles. Like in the deep embedding, values of type `Handle` are built-in `IO` file handles, and the effect is also `IO`.

```
data instance LVal Shallow Handle = VHandle IO.Handle
type instance Effect Shallow = IO
```

The file handle operations are easily given by their `IO` counterparts.

```
instance HasFH Shallow where
  open s    = SExp $ \ρ → VHandle <$> IO.openFile s IO.ReadWriteMode
  read e    = SExp $ \ρ → do VHandle h ← runSExp e ρ
                             c ← IO.hGetChar h
                             return $ VPair (VHandle h) (VPut c)
  write e c = SExp $ \ρ → do VHandle h ← runSExp e ρ
                             IO.hPutChar h c
                             return $ VHandle h
  close e   = SExp $ \ρ → do VHandle h ← runSExp e ρ
                             IO.hClose h
                             return VUnit
```

4.3.4 Laws and Correctness

To make sure our implementations are correct, we should check them against a specification. These specifications characterize the β and η equivalences for each connective, and we express them with respect to the behavior of `eval`. We start by defining substitution.

```
subst  :: forall x σ τ γ1 γ2 γ1' γ exp.
         ( Eval exp, HasLolli exp, Monad (Effect exp)
         , CAddCtx x σ γ1 γ1', CMerge γ1 γ2 γ, x ~ Fresh γ2)
       ⇒ ( Var exp x σ → exp γ1' τ) → exp γ2 σ
       → ECtx exp γ → Effect exp (LVal exp τ)
subst f e ρ = do let (ρ1, ρ2) = splitECtx @γ1 @γ2 ρ
                 v ← eval e ρ2
                 eval (f $ var x) (addECtx x v ρ1)
  where x = (Proxy :: Proxy x)
```

The β and η rules for linear implication characterize the evaluation of linear expressions.

```
eval (λ̂ f ^ e) ρ = subst f e ρ               [β]
eval e ρ         = eval (λ̂ $ \x → e ^ x) ρ   [η]
```

The laws for `Lower` α parallel the monad laws described in Section 3.4. Here we write $e \approx e'$ when, for all evaluation contexts ρ, `eval e ρ = eval e' ρ`.

```
put a >! f    ≈ f a               [β]
e             ≈ (e >! put)        [η]
(e >! f) >! g ≈ e >! \a → f a >! g  [assoc]
```

We cannot formally verify these laws in Haskell, but we can reason informally about their correctness.

Proposition 4.3.1. *The shallow embedding satisfies the* `Lower` *laws.*

Proof. We start with the β rule. Unfolding definitions we see that

```
eval (put a >! f) ρ = do let (ρ1,ρ2) = split ρ
                         VPut a ← return $ VPut a
                         runSExp (f a) ρ2
```

Here ρ_1 is empty, so it must be the case that $\rho_2 = \rho$. But because `Effect Shallow` is a monad, the result is equal to `runSExp (f a)` γ, as expected.

The proof of the η law is similar. Unfolding definitions we see that

```
eval (e >! put) ρ = let (ρ1,ρ2) = split ρ
                    in runSExp e ρ1 >>= \(VPut a) → runSExp (put a) ρ2
```

In this case ρ_2 is empty and so ρ_1 is equal to ρ, so the expression above is equal to

```
SExp $ \ρ → runSExp e ρ >>= \(VPut a) → return $ VPut a
```

Again, the monad laws for `Effect Shallow` says that this is equivalent to `SExp $` $\backslash\gamma\rightarrow$ `runSExp e` γ which, by η-equivalence of shallow expressions, is just `e`.x

The associativity law is similarly obtained by unfolding definitions and applying the monad laws. □

Proposition 4.3.2. *The deep embedding satisfies the* `Lower` *laws.*

Proof. Again, by unfolding definitions and applying the monad laws. □

4.4 Monadic programming

We saw in Section 3.4.3 how the composition of `Lift` and `Lower` form a monad and a monad transformer. In Haskell, monads can be given instances of the `Monad` type class, which gives access to `do` notation and other abstractions from the standard library.

We start by giving a `newtype` declaration for the linearity monad, which, like `Lift`, must be indexed by the typing judgment `exp`.

```
newtype Lin exp α = Lin (Lift exp (Lower α))
```

It will be convenient to use the notation `suspend` and `force` to coerce data into and out of the linearity monad, as we do for `Lift`. We define a type class `Suspendable` to indicate that a Haskell data type can suspend closed linear computations.

```
class Suspendable exp τ lift | lift → exp τ where
  suspend :: exp '[] τ → lift
  force   :: lift → exp '[] τ
instance Suspendable exp τ (Lift exp τ) where
  suspend = Suspend
  force (Suspend e) = e
instance Suspendable exp (Lower α) (Lin exp α) where
  suspend = Lin . suspend
  force (Lin e) = force e
```

Computations in `Lin` have the property that they can be lifted to computations using the underlying effect of the embedding; we call this operation **run**. Note that it should always be possible to extract a Haskell value of type α from a linear value of type `Lower` α; we add this constraint to the `Eval` type class:

```
class Eval exp where
  eval      :: Monad (Effect exp) ⇒ exp γ τ → ECtx exp γ → Effect exp (LVal exp τ)
  fromVPut :: Monad (Effect exp) ⇒ LVal exp (Lower α) → Effect exp α
```

Running a computation in the linearity monad evaluates the underlying program, and extracts the underlying value.

```
run :: (Monad (Effect exp), Eval exp) ⇒ Lin exp a → Effect exp a
run e = eval (force e) eEmpty >>= fromVPut
```

To give `Lin` a monad instance, we must first give it instances for the `Functor` and `Applicative` type classes.

```
instance (HasLower exp) ⇒ Functor (Lin exp) where
  fmap f e = suspend $ force e >! (put . f)
instance (HasLower exp) ⇒ Applicative (Lin exp) where
  pure     = suspend . put
  f <*> e = suspend $ force e >! \a →
                      force f >! \g →
                      put $ g a
instance HasLower exp ⇒ Monad (Lin exp) where
  e >>= f = suspend $ force e >! \a →
                      force (f a)
```

Giving a type class instance is only half the battle, however; we must also check that it satisfies the monad laws up to evaluation.

```
pure a >>= f    ≈ f a                    [β]
e >>= pure      ≈ e                      [η]
(e >>= f) >>= g ≈ e >>= (\x → f x >>= g) [assoc]
```

In this case we write `a ≈ a'` to indicate that `a,a' : Lin exp` α are equal up to evaluation: that `run a = run a'`. Notice that if $e \approx e'$ then `suspend` $e \approx$ `suspend` e'.

Proposition 4.4.1. *If the* `Lower` *laws hold for a typing judgment* `exp`*, then* `Lin exp` *satisfies the monad laws up to evaluation.*

Proof. For the β rule, expanding the definitions of `pure` and `>>=` we have that

```
pure a >>= f = suspend $ force (pure a) >! force . f
             = suspend $ put a >! force . f
```

By the β rule for `Lower`, this is equivalent to `suspend (force $ f a)`, which is η-equivalent to `f a` itself.

The η and associativity laws are similarly proven by unfolding definitions and applying the `Lower` laws. □

4.4.1 Monads in the linear category

Next we discuss monads over linear types, such as the linear state monad $\sigma \multimap \sigma \otimes \tau$. The usual `Monad` type class only characterizes monads of kind `Type → Type`, which correspond to endofunctors on the category of Haskell types.

We can imagine defining a type class of linear monads `LMonad m`, where `m` has kind `LType → LType`, with linear versions of `return` and `≫=`.

To make an instance declaration for the linear state monad, we first try to define a type synonym `LState` σ τ for $\sigma \multimap \sigma \otimes \tau$. Unfortunately, this means that the monad `LState` σ is a partially defined type synonym, which is undefined in Haskell. The ordinary solution would be to define a `newtype` synonym for `LState` σ τ, but `newtype`s (and regular algebraic data types) produce `Type`s, not `LType`s.

Our solution is to use a trick called defunctionalization (Eisenberg and Stolarek, 2014). The `singletons` library[8] provides a type-level arrow `k1 ⤳ k2` that describes unsaturated type-level functions between kinds `k1` and `k2`. To define a defunctionalized arrow, we first define an empty data type for the unsaturated version of `LState`, and then define a type instance for the (infix) type family `(@@)`, which has kind `(k1 ⤳ k2) → k1 → k2`.

```
data LState' (σ :: LType) :: LType ⤳ LType
type instance LState' σ @@ τ = σ ⊸ σ ⊗ τ
```

We can then define `LState` σ τ `= LState'` σ `@@` τ. Instead of defining the `LMonad` type class for monads of kind `LType → LType`, we instead define it for defunctionalized arrows of kind `LType ⤳ LType`.

```
class LMonad exp (m :: LType ⤳ LType) where
  lreturn :: exp γ τ → LExp γ (m @@ τ)
  lbind   :: exp 'Empty (m @@ σ ⊸ (σ ⊸ m @@ τ) ⊸ m @@ τ)
```

When convenient, we use the notation `e ⋙= f` for `lbind ^ e ^ f`.

The laws for linear monads are the same as for those in Haskell, up to evaluation.

```
lreturn e ⋙= f ≈ f ^ e                  [β]
e ⋙= lreturn    ≈ e                      [η]
(e ⋙= f) ⋙= g  ≈ e ⋙= (\x → f x ⋙= g)  [assoc]
```

We can now define our monad instance.

```
instance HasMILL exp ⇒ LMonad exp (LState' σ) where
  lreturn e = λ̂ $ \s → s ⊗ e
  lbind     = λ̂ $ \st → λ̂ $ \f → λ̂ $ \s →
                st ^ s `letPair` \(s,x) → f ^ x ^ s
```

[8] https://hackage.haskell.org/package/singletons

4.4.2 The Monad Transformer

We saw in Section 3.4 that the adjunction `Lower ⊣ Lift` also induces an `LMonad` transformer. Given an `LMonad` of kind `LType ⇝ LType`, we can define a Haskell monad `LinT m`. As we did for `Lin`, we give it an instance of the `Suspendable` type class.

```
newtype LinT exp (m :: LType ⇝ LType) (α :: LType) = LinT (Lift exp (m @@ (Lower α)))
instance τ ~ (f @@ (Lower α)) ⇒ Suspendable exp τ (LinT exp f α) where
    suspend = LinT . suspend
    force (LinT x) = force x
```

We can define the `Monad` instance just as we did for `Lin`:

```
instance (LMonad m, HasLower exp) ⇒ Monad (LinT exp m) where
  return = suspend . lreturn . put
  x ≫= f = suspend $ force x ⋙= λ̂ $ \y → y >! (force . f)
```

Proposition 4.4.2. *If* `m` *satisfies the* `LMonad` *laws, then* `LinT exp m` *satisfies the* `Monad` *laws up to evaluation.*

Proof. By unfolding definitions and applying the `LMonad` laws. □

4.5 Example: Arrays

In this section we instantiate `LNLHask` with linear mutable arrays. In his paper "Linear types can change the world!", Wadler (1990) argues that mutable data structures like arrays can be given a pure functional interface if they are only accessed linearly. To understand why, consider a non-linear program with purely functional arrays that writes two values to index 0 of the array one after another, and then looks up the value of index 0.

```
let arr₁ = write 0 arr "hello" in
let arr₂ = write 0 arr "world" in arr₁[0]
```

If `write` were to update the array in place, the program would return `"world"` instead of `"hello"`. Linear types force us to serialize the operations on arrays so that reasonable equational laws still hold, even when performing destructive updates.

Here we expand Wadler's example to describe *slices* of an array. Consider an operation `slice arr i` that partitions an array `arr` around the index `i`. As long as the operations on each slice are restricted to their domains, the implementation of `slice` can alias the same array. Furthermore, as long as we keep track of when two slices alias the same array, we can merge slices back together with zero cost.

To implement linear arrays in `LNLHask`, we first add a new type for arrays of non-linear values.

```
data ArraySig ty = MkArray Type Type
type Array token α = MkLType (MkArray token α)
```

The first argument k to `Array k α` is a token that keeps track of the array being aliased—different arrays will be initialized with different tokens. The second argument α is the type of values stored in the array.

The interface to linear arrays is given in Figure 4.5. The interface can allocate new arrays and drop the pointers to existing ones. The `size` operation returns the length of a particular slice of an array; the `read` and `write` operations will fail at runtime if their

```
class HasMELL exp ⇒ HasArray exp where
  alloc   :: CAddCtx x (Array k α) γ γ'
          ⇒ Int → α → (Var exp (Array k α) → exp γ' σ) → exp γ σ
  dealloc :: exp γ (Array k α) → exp γ LUnit
  size    :: exp γ (Array token a) → exp γ (Array token a ⊗ Lower Int)
  read    :: Int → exp γ (Array k α) → exp γ (Array k α ⊗ Lower α)
  write   :: Int → exp γ (Array k α) → α → exp γ (Array k α)
  slice   :: Int → exp γ (Array k α) → exp γ (Array k α ⊗ Array k α)
  combine :: CMerge γ1 γ2 γ
          ⇒ exp γ1 (Array k α) → exp γ2 (Array k α) → exp γ (Array k α)
```

Figure 4.5: Interface to linear arrays.

arguments are not in the domain of their slice. In other words, we should think of a slice of an array as a standalone piece of data, indexed starting from zero.

The operation `slice` takes an index and an array, and outputs two aliases to the same array with domains partitioned around the index. Dually, `combine` takes two aliases to the same array and combines their bounds.

4.5.1 Implementation

We instantiate the `HasArray` signature by extending the shallow embedding. A value of type `Array k` α consists of a primitive `IO` array as well as a list of indices corresponding to the current slice of an array. Because these indices tend to be grouped into ranges, they are represented as a set of intervals. We write `Range` for the type `(Int,Int)` of inclusive ranges of integers, and the type `[Range]` for ordered, non-overlapping lists of ranges.

```
newtype instance LVal Shallow (Array k α) = VArray ([Range],IOArray Int α)
type instance Effect Shallow = IO
```

The implementations of `alloc`, `read`, and `write` use the corresponding operations on `IOArrays`. In `read` and `write`, the index i must be less than the length of the current slice. To convert i into an index in the gloval array, we offset i by the range of the current slice; `offset i rs` outputs the ith index in `rs`.

```
instance HasArray Shallow where
  alloc n a k = SExp $ \(ρ :: ECtx Shallow γ) →
    do arr ← IO.newArray (0,n-1) a
       let v = VArray ([(0,n-1)],arr)
           x = (Proxy :: Proxy (Fresh γ))
       runSExp (k $ var x) (addECtx x v ρ)

  read i e = SExp $ \ρ → do
        VArray (rs,arr) ← runSExp e ρ
        if i < size rs then do let x = offset i rs
                               a ← IO.readArray arr x
                               return $ VPair (VArray (rs,arr)) (VPut a)
        else error $ "Read " ++ show i ++ " out of bounds of " ++ show rs

  write i e a = SExp $ \ρ → do
        VArray (rs,arr) ← runSExp e ρ
```

```
        if i < size rs then do let x = offset i rs
                               IO.writeArray arr x a
                               return $ VArray (rs,arr)
        else error $ "Write " ++ show i ++ " out of bounds " ++ show rs
```

The implementation of dealloc simply returns a unit value—it does not explicitly deallocate the array, which would be inappropriate when dropping partial slices. Furthermore, the IO library does not expose a deallocation primitive for arrays. The size operation looks up the size of the underlying set of ranges.

```
dealloc e = SExp $ \ρ → runSExp e ρ ≫ return VUnit
size e    = SExp $ \ρ → do VArray (rs,arr) ← runSExp e ρ
                           let n = size rs
                           return $ VPair (VArray (rs,arr)) (VPut n)
```

The slice operation partitions the bounds of its input array according to its index, while combine evaluates its arguments and merges the resulting bounds. Neither actually affects the underlying array.

```
slice i e = SExp $ \ρ →
  do VArray (rs,arr) ← runSExp e ρ
     if i < size rs
     then let x          = offset i rs
              (rs₁,rs₂) = partition x rs
          in return $ VPair (VArray (rs₁,arr)) (VArray (rs₂,arr))
     else error $ "Slice " ++ show i ++ " out of bounds of " ++ show rs

combine e₁ e₂ = SExp $ \ρ → do let (ρ₁,ρ₂) = splitECtx ρ
                               VArray (rs₁,arr) ← runSExp e₁ ρ₁
                               VArray (rs₂,_)   ← runSExp e₂ ρ₂
                               return $ VArray (union rs₁ rs₂, arr)
```

Alternatively, combine can be implemented concurrently by evaluating the two subexpressions in separate threads. For very concurrent operations this could be more efficient, but in many cases it introduces too much overhead.

```
concurrentCombine e₁ e₂ = SExp $ \ρ → do let (ρ₁,ρ₂) = split ρ
                                         v₁ ← newEmptyMVar
                                         v₂ ← newEmptyMVar
                                         forkIO $ runSExp e₁ ρ₁ ≫= putMVar v₁
                                         forkIO $ runSExp e₂ ρ₂ ≫= putMVar v₂
                                         VArray (rs₁,arr) ← takeMVar v₁
                                         VArray (rs₂,_)   ← takeMVar v₂
                                         return $ VArray (union rs₁ rs₂, arr)
```

4.5.2 Arrays in the Lifted State Monad

The read, write, and size operations can be naturally lifted to the linear state monad transformer; recall that we write LStateT sig σ α for LinT sig (LState' σ) α.

```
readT  :: HasArray exp ⇒ Int → LStateT sig (Array k α) α
writeT :: HasArray exp ⇒ Int → α → LStateT sig (Array k α) ()
sizeT  :: HasArray exp ⇒ LStateT sig (Array k α) Int
```

We can combine allocation and deallocation of an array into a single LStateT operation.

```
allocT :: HasArray exp
       ⇒ Int → α → (forall k. LStateT exp (Array k α) β) → Lin exp β
allocT n a op = suspend $ alloc n a $ \arr →
                          force op ^ arr `letPair` \(arr,b) →
                          dealloc arr `letUnit` b
```

Finally, we can derive a lifted operation that combines slicing and rejoining slices. The function `sliceT` takes an index and two state transformations on arrays. The resulting state transformation takes in an array, slices it around the given index, and applies the two state transformations to the two sub-arrays.

```
sliceT :: HasArray exp
       ⇒ Int → LStateT sig (Array k α) () → LStateT sig (Array k α) ()
       → LStateT sig (Array k α) ()
sliceT i st₁ st₂ = Suspend . λ̂ ! \arr →
    slice i arr `letPair` \(arr₁,arr₂) →
    forceT st₁ ^ arr₁ `letPair` \(arr₁,res) → res >! \_ →
    forceT st₂ ^ arr₂ `letPair` \(arr₂,res) → res >! \_ →
    combine arr₁ arr₂ ⊗ put ()
```

The bound i in `sliceT i` is inclusive—index i will be included in one of the two slices. In practice, we sometimes want a variant where an operation is applied to indices less than i and greater than i, but not equal to i itself. The function `slice3 i op` slices the array into three parts, and applies `op` to the slice of indices less than i, and to the slice of indices greater than i, but not to index i itself. The bounds checking ensures that every slice is smaller than the original array.

```
-- slice3 i op applies op on indices < i and indices > i, does nothing at index i
-- precondition: 0 ≤ i < length array
slice3 :: HasArray sig
        ⇒ Int → LStateT sig (Array token α) () → LStateT sig (Array token α) ()
slice3 i op =
      do len ← sizeT
         if len ≤ 2       then return ()
         else if i== 0     then sliceT 1 (return ()) op
         else if i== len-1 then sliceT i op (return ())
         else                   sliceT i op $ sliceT 1 (return ()) op
```

4.5.3 Quicksort

We will use the `LStateT` interface to implement an in-place quicksort.

First, the operation `swap i j` swaps the indices i and j in the underlying array.

```
swap :: HasArray sig ⇒ Int → Int → LStateT sig (Array token α) ()
swap i j = do a ← readT i
              b ← readT j
              writeT i b
              writeT j a
```

Quicksort relies on a helper function `partition pivot (i,j)` that, given a pivot value, swaps elements between indices i and j so that values less than `pivot` occur on the left-hand-side of the range, and values greater than `pivot` occur on the right-hand-side of the range. It returns the index of the largest element in the range that is less than `pivot`. The

```
partition :: (HasArray sig, Ord α)
          ⇒ α → (Int,Int) → LStateT sig (Array token α) Int
                      -- if the range is empty, return the index
                      -- of the  largest element < pivot
partition pivot (i,j) | i ≥ j    = do a ← readT i
                                     if a < pivot then return i else return (i-1)
                      -- otherwise, if arr[i]<pivot, recurse on the range (i+1,j)
                      -- and if arr[i]>pivot, move array[i] to the right-hand-side
                      -- of the array and recurse on the range (i,j-1)
                      | otherwise = do a ← readT i
                                     if a < pivot then partition pivot (i+1,j)
                                     else do swap i j
                                             partition pivot (i,j-1)

quicksort :: (HasArray sig, Ord α)
          ⇒ LStateT sig (Array token α) ()
quicksort = do len ← sizeT
               if len ≤ 1 then return ()
               else do pivot ← readT 0
                       idx ← partition pivot (1,len-1)
                       swap 0 idx
                       slice3 idx quicksort
```

Figure 4.6: Linear quicksort algorithm

main `quicksort` algorithm selects the initial pivot element as the value at index 0. It calls `partition` to obtain the middle of the array, and moves the pivot element there. Then, it recursively sorts the two subarrays surrounding the pivot element using `slice3`.

The definitions of both `partition` and the main `quicksort` algorithm are shown in Figure 4.6.

4.6 Example: Session types

Section 3.6 introduced an interface to a linear EDSL for session-typed channels in which a session of linear type `Chan S` is governed by its session type S:

```
S ::= σ <!> S | σ <?> S | S <&> S | S <+> S | End
```

In that setting, a channel can either send a value (`<!>`), receive a value (`<?>`), offer a choice between two protocols (`<&>`), or make a choice between two protocols (`<+>`). There, a channel is a linear data type, and the domain-specific operations `send`, `receive`, *etc.* are used to maintain these channels.

Caires and Pfenning (2010) present an alternative interpretation of session types through the lens of the Curry-Howard correspondence. They claim that *any* linear type can be viewed as a session protocol, according to the following correspondence:

linear type	session type	protocol
$\sigma \multimap \tau$	$\sigma\langle ? \rangle\tau$	input σ and continue as τ
$\sigma \otimes \tau$	$\sigma\langle ! \rangle\tau$	output σ and continue as τ
$\sigma \& \tau$	$\sigma\langle \& \rangle\tau$	offer choice between σ and τ
$\sigma \oplus \tau$	$\sigma\langle + \rangle\tau$	make choice between σ and τ

A linear expression $\Delta \vdash e : \tau$ can be interpreted as a process communicating over channels $x : \sigma \in \Delta$, as well as over an output channel of type τ.

In this section we implement Caires and Pfenning's type system, not by adding new operations to send and receive over channels, but instead by implementing the usual linear logical connectives as processes that communicate over linear variables.

4.6.1 Example: an echo server

As in Section 3.6, we can implement an echo server, which takes a string as input and echoes it back to the user.

```
type EchoProtocol = Lower String ⊸ Lower String ⊗ LUnit
```

An echo server is a suspended computation of type `EchoProtocol`.

```
server :: HasMALL ⇒ Lift exp EchoProtocol
server = suspend . λ̂bang $ \s → put s ⊗ unit
```

A client is one who uses the `EchoProtocol`—she sends the string "Testing" to the server, and checks whether she receives the same string back.

```
client :: HasMALL ⇒ Lift exp (EchoProtocol ⊸ Lower Bool)
client = suspend . λ̂ $ \s → s  ^  put "Testing" `letpair` \(x,y) →
                            y `letUnit` x >! \s →
                            put $ s == "Testing"
```

4.6.2 Implementation

Although we use the same syntax as the pure linear lambda calculus, what we really want is an implementation that communicates data over channels. Since the type of a session changes over time, sessions should be implemented using untyped channels. We use `unsafeCoerce` to send and receive typed data over these untyped channels—the linear protocols will ensure that whenever a value of type σ is sent on the channel, it will be coerced back into the same type σ by the recipient.

A session is implemented as a pair `UChan` of untyped channels. We use a pair so that each component has a fixed direction—the left channel will always be used to receive data, and right channel will always be used to send data. Every time we construct a `UChan`, we also construct its swap, which corresponds to the other end of the channel.

```
type UChan = (Chan Any, Chan Any)
newU :: IO (UChan,UChan)
newU = do c1 ← IO.newChan
          c2 ← IO.newChan
          return ((c1,c2),(c2,c1))
```

These channels are untyped, and we use `unsafeCoerce` to send and receive data of arbitrary types. As long as `recvU` is only instantiated at types inferred from the session protocol, it should never cause a segfault.

```
sendU :: UChan → α → IO ()
sendU (cin,cout) a = writeChan cout $ unsafeCoerce a
recvU :: UChan → IO α
recvU (cin,cout) = unsafeCoerce <$> readChan cin
```

The operation linkU takes as input two channels and forwards data between them in both directions.

```
linkU :: UChan → UChan → IO ()
linkU c1 c2 = (forkIO $ forward c1 c2) >> forward c2 c1
  where
    forward c c' = recvU c >>= sendU c' >> forward c c'
```

A new shallow embedding. We use a variant of the shallow embedding to encode expressions. An expression is a function from evaluation contexts plus an extra UChan to IO (). The extra UChan is the output channel of the expressions; an expression of type $\sigma \otimes \tau$ will send a value of type σ on its output channel.

```
data Sessions γ τ = SExp {runSExp :: ECtx Sessions γ → UChan → IO ()}
data instance Effect Sessions = IO
```

Values in this setting, no matter the type, are all UChan's.

```
data instance LVal Sessions τ = Chan UChan
```

To evaluate an expression, we first construct a new UChan, which gives us two endpoints to a new channel. We pass one of the endpoints to the expression using runSExp, and the other endpoint is returned as the value of the expression.

```
instance Eval Sessions where
  eval e ρ = do (c,c') ← newU
                forkIO $ runSExp e ρ c
                return $ Chan c'
```

The HasLower type class. To construct an expression of type Lower τ via put a, we simply send the Haskell value a over the output channel.

```
    put a = SExp $ \_ c → sendU c a
```

To implement e >! f, we spawn a new channel and pass one end to e. We wait for a value from the other end, to which we apply f.

```
    e >! f = SExp $ \ρ c → do let (ρ1,ρ2) = split ρ
                              (x,x') ← newU
                              forkIO $ runSExp e ρ1 x
                              a ← recvU x'
                              runSExp (f a) ρ2 c
```

Sending and receiving values. An expression of type $\sigma \multimap \tau$ is a process that receives a channel of type σ and then continues as τ. So the expression $\hat{\lambda} x.e$ receives a value on its output channel and binds that value to x in the continuation.

```
λ̂ f = SExp $ \(ρ :: ECtx Sessions γ) c → do v ← recvU c
                                             let x = (Proxy :: Proxy (Fresh γ))
                                             runSExp (f $ var x) (add x v ρ) c
```

An application $e_1 \,\hat{}\, e_2$ needs to connect the output of e_2 to the input of e_1. The process starts by creating a new channel, x, on which e_2 can send its output. Next, we send the other end of the channel, called x', to e_1, which we do by initializing another new channel y. One endpoint of `y` is passed to e_1, and then the remainder is forwarded along to the original output of the process.

```
e₁ ^ e₂ = SExp $ \ρ c →
       do let (ρ₁,ρ₂) = split ρ
          (x,x') ← newU
          forkIO $ runSExp e₂ ρ₂ x -- e₂ sends output to x

          (y,y') ← newU
          sendU y' (Chan x')       -- send x' to e₁ via y
          forkIO $ runSExp e₁ ρ₁ y -- e₁ receives input from y
          linkU c y'               -- e₁ forwards result to (e₁ ^ e₂)
```

The implementation of the `HasTensor` type class is exactly dual to `HasLolli`—where $\multimap$ receives a value, $\otimes$ sends a value, and vice versa.

Making and offering choices. A process of type $\sigma_1 \& \sigma_2$ accepts a boolean flag as input, either `Left ()` or `Right ()`, to indicate which of σ_1 or σ_2 to continue as.

```
instance HasWith Sessions where
  e₁ & e₂ = SExp $ \ρ (c :: UChan) → recvU c ≫= \case
     Left  () → runSExp e₁ ρ c
     Right () → runSExp e₂ ρ c

  proj1 e = SExp $ \ρ c → do sendU c (Left ())
                              runSExp e ρ c
  proj₂ e = SExp $ \ρ c → do sendU c (Right ())
                              runSExp e ρ c
```

The `HasPlus` class is exactly the dual.

4.7 Discussion and Related Work

4.7.1 Design of the Embedded Language

The embedding described in this chapter is very similar to the work of Polakow (2015) and Eisenberg *et al.* (2012), who also embed linear lambda calculi in Haskell using dependently-typed features of GHC to enforce linearity. We adapt features from both embeddings: Polakow introduces higher-order abstract syntax (HOAS) for linear types, but to achieve this he uses an algorithmic typing judgment $\gamma_{in}/\gamma_{out} \vdash e : \tau$ that threads an input context into every judgment (Walker, 2005). Eisenberg *et al.* use the standard typing judgment $\gamma \vdash e : \tau$ but without HOAS, which makes programming in the embedded language awkward.

In this paper we combine the two representations to get a HOAS encoding of the direct-style typing judgment. Doing so has its limitations, however, specifically when constructing open linear expressions. Consider the `lpure` instance for the linear state monad:

```
lpureLState :: HasMILL exp ⇒ exp γ σ → exp γ (LState ρ σ)
```

We should be able to define `lpureLState e` as the expression `λ $ \r → r ⊗ e`, but the type system cannot derive the fact that `CAddCtx x σ γ (AddF x σ γ)`, where `x` is

`Fresh` γ. We can solve this problem in one of two ways. First, we could manually introduce a proof `wfFresh @`ρ `@`γ of type `Dict (WFVar (Fresh `γ`) `ρ` `γ`)` into the context.

```
lpureLState = withDict (wfFresh @ρ @γ) . λ $ \r → r ⊗ e
```

Alternatively, a more user-friendly approach is to define `lpureLState` as a closed function, and then apply it to the argument e:

```
lpureLState e = force lpure' ^ e
  where lpure' = suspend $ λ $ \ x → λ $ \ r → r ⊗ x
```

The second approach is conceptually much simpler, but it has the disadvantage of adding an extra β-redex to the evaluation of a program.

To improve type checking in the direct style, it may be possible to improve the expressiveness of type classes or implement a type checker plugin that uses an external solver to search for intermediate typing contexts. For example, the Coq implementation in Chapter 8 defines a theory of linear type constraints that could be adapted to Haskell.

Crucially, the contribution of this work in contrast to that of Eisenberg *et al.* and Polakow is not so much the design of the embedding in Haskell, but rather the use of the linear/non-linear model and the linearity monad it gives rise to. Eisenberg *et al.* and Polakow introduce $!\alpha$ as an embedded connective, which, compared to the LNL decomposition, requires significantly more maintenance in the linear system.

This chapter also makes some simplifications over the version of `LNLHask` presented at the 2017 Haskell Symposium (Paykin and Zdancewic, 2017). The biggest change is in the representation of type contexts and evaluation contexts. In the original paper, typing contexts are represented as lists of option `LType`s, and variables are represented as unary type-level natural numbers. There, a variable x maps to σ in γ if $\gamma[x]$ `= Just` σ. The representation of typing contexts as `[(Nat,LType)]` in this chapter is more standard and is simpler to implement. The runtime representation of natural numbers as Haskell integers versus unary natural numbers is an order of magnitude more space efficient. Finally, in the Haskell symposium version, evaluation contexts were represented as functions, which over time build up large thunks:

```
data ECtx sig γ where
  ECtx :: (∀ x σ. Lookup γ x ~ Just σ ⇒ Sing x → LVal sig σ)
       ⇒ ECtx sig γ
```

By comparison, the `IntMap`s used in this chapter are strict and highly optimized.

4.7.2 Performance

The goal of `LNLHask` is to represent and run linear programs, but the focus has not so far been on efficiency. Preliminary tests of the quicksort algorithm described in Section 4.5 indicate that `LNLHask` introduces significant constant-time overhead. We expect that further profiling and optimizing could significantly improve performance.

4.7.3 Error Messages

In the development, we use custom type errors in the type families `AddF`, `MergeF`, *etc.* to give informative messages when type errors fail. For example, the type checker will fail on $\hat{\lambda}$ `(\x → x ⊗ x)` with the error message

- `Error adding 0`

```
    to context '['(0, σ)]
• In the expression: x ⊗ x
```

This error arises roughly from the following sequence of steps:

- To show $\hat{\lambda} f$ has type `exp ∅ (σ ⊸ σ ⊗ σ)`, Haskell must show that f has the type `(Var exp x σ → exp γ' (σ ⊗ σ))`, where:
  - x is equal to `Fresh ∅` (so `x ~ 0`);
  - `Var exp x σ` is equal to `exp '[(0,σ)] σ`; and
  - `CAddCtx x σ ∅ γ'`. Type class resolution for `CAddCtx` lets the type checker infer that `γ'~[(0,σ)]`.
- To show `\x → x ⊗ x` has type `(exp '[(0,σ)] σ → exp [(0,σ)] (σ ⊗ σ))`, we need to show there exist contexts γ_1 and γ_2 such that `x : exp` γ_1 σ, `x : exp` γ_2 σ, and `CMerge` γ_1 γ_2 `[(0,σ)]`. Since `x : exp '[(0,σ)] σ`, we would thus need to show that `CMerge [(0,σ)] [(0,σ)] [(0,σ)]`.
- The type class mechanism infers that in order for `CMerge` γ_1 γ_2 γ to be true, it must be the case that γ `~ MergeF` γ_1 γ_2. However, the type family invocation `MergeF` γ_1 γ_2 is undefined when γ_1 and γ_2 share any variables x, and gives rise to the error `Error adding x to context` γ_1, which is the output of `AddError` as described in Figure 4.1.

In other situations where the type checking fails, such as the `lreturnLState` operation described above, we have not been able to improve the default error message:

```
• Couldn't match type 'MergeF (AddF (Fresh γ) ρ γ) '[]'
                 with 'AddF (Fresh γ) ρ γ'
    arising from a use of 'λ'
• In the expression: λ $ \ r → r ⊗ e
```

This message is generated because Haskell could not find a proof that two types are equal, not from an error generated from a type family. Unfortunately, bad error messages are a problem for many embedded languages.

4.7.4 Deep versus Shallow Embeddings

The prior implementations by Polakow (2015) and Eisenberg *et al.* (2012) describe shallow embeddings, which should be more efficient than deep embeddings. However, the shallow embedding is not technically *adequate* because it is possible to write down terms of type `LExp Shallow` γ τ that do not correspond to anything in the linear lambda calculus. For example, `SExp (\γ → VPut ())` has type `LExp Shallow γ (Lower ())` for any context γ. This can be acceptable in some cases, as there are two different consumers of our framework: DSL *implementers* and DSL *users*. Implementers have access to unsafe features of the embedding, and so they must be careful to only expose an abstract linear interface (*e.g.,* one not containing the `SExp` constructor) to the *clients* of the language. This will enforce the linearity invariants on the clients, but not necessarily for the implementers.

In the deep embedding, linear expressions are correct by construction, although of course the language implementer could make an error defining evaluation. The deep embedding also makes it possible to express program transformations and optimizations, which we cannot do in a shallow embedding.

4.7.5 Further Integration with Haskell

A recent proposal by Bernardy *et al.* (2017) suggests how to integrate linear types directly into GHC, based on a model of linear logic that uses weighted annotations on arrows instead of $!\alpha$ or the adjoint decomposition considered here. Their proposal could allow the implementation of efficient garbage collection and explicit memory management, and could conceivably be adapted to a wide variety of different domains using foreign function interface calls.[9] Compared to our approach, the proposal requires significant changes to GHC; our framework works out-of-the box. We hypothesize that the linearity monad arises in their work as the (linear) CPS monad: $(\alpha \multimap \bot) \multimap \bot$.

Bernardy *et al.*'s proposal is also adamant about eliminating code duplication, meaning that data structures and operations on data structures should be parametric over linear versus non-linear data. It is certainly a drawback of our work that the user may have to duplicate Haskell code in the linear fragment, as we saw when defining the linear versions of the monad type classes in Section 4.4. Future work could address this by using Template Haskell[10] to define data structures and functions with implementations in both the linear and non-linear fragments. This is the approach taken by the Singletons library for unifying types and terms, and a similar approach could unify linear and non-linear terms.

[9] `https://wiki.haskell.org/Foreign_Function_Interface`
[10] `https://wiki.haskell.org/Template_Haskell`

CHAPTER 5

Embedded categorical semantics

Linear type systems have always been closely coupled with categorical semantics through the Curry-Howard correspondence. The categorical model of linear/non-linear logic, which we describe in detail in this chapter, is made up of two categories—a linear monoidal category and a non-linear cartesian category—connected via a (symmetric monoidal) adjunction. In Section 5.4 we formalize the category theory of LNL inside of the embedded language framework, which is a sanity check that our linear type system is sound and that the embedded LNL structure actually corresponds to traditional presentations of LNL.

Studying the category theory of LNL also provides insight into the meaning of type operators like $!\sigma$, `Lin` α, `Lift` σ, and `Lower` α. The relationships between these operators arise from basic concepts from category theory—monads, comonads, and adjunctions. Understanding the categorical structure of our type system can even give rise to new abstractions and ways of programming through the Curry-Howard correspondence.

In this chapter we review the categorical foundations of linear/non-linear logic and establish that the embedded linear lambda calculus described in Chapter 3 forms a linear/non-linear model with the category of host language terms.

5.1 Background

We begin by reviewing some basic definitions of category theory up through natural transformations and adjunctions. For the sake of this chapter we assume familiarity with the topics in this section. There are several excellent resources for those not already familiar; see Pierce (1991) or Awodey (2010).

5.1.1 Categories, functors, and natural transformations

Definition 5.1.1. *A* category $\mathcal{C}$ *consists of the following components:*

- *A class* $\text{Obj}(\mathcal{C})$ *of* objects *of* $\mathcal{C}$*;*
- *For* $A, B \in \text{Obj}(\mathcal{C})$*, a class* $\mathcal{C}(A,B)$ *of* morphisms *with domain* A *and codomain* B*;*
- *For morphisms* $f \in \mathcal{C}(A,B)$ *and* $g \in \mathcal{C}(B,C)$*, there is a morphism* $g \circ f \in \mathcal{C}(A,C)$ *such that*

$$(h \circ g) \circ f = h \circ (g \circ f);$$

- *For each object* $A \in \text{Obj}(\mathcal{C})$*, there is an identity morphism* $1_A \in \mathcal{C}(A,A)$ *such that, for all*

$f \in \mathcal{C}(A, B)$,

$$1_B \circ f = f = f \circ 1_A.$$

Definition 5.1.2. *Let $\mathcal{C}$ and $\mathcal{D}$ be categories. A* functor *F from $\mathcal{C}$ to $\mathcal{D}$, written $F : \mathcal{C} \Rightarrow \mathcal{D}$, is a map that associates:*

- *For each object $A \in \mathrm{Obj}(\mathcal{C})$, an object $FA \in \mathrm{Obj}(\mathcal{D})$;*

- *For each morphism $f \in \mathcal{C}(A, B)$, a morphism $F(f) \in \mathcal{D}(FA, FB)$ such that*

$$F(1_A) = 1_{FA} \qquad \text{and} \qquad F(g \circ f) = F(g) \circ F(f).$$

Definition 5.1.3. *Let $F, G : \mathcal{C} \Rightarrow \mathcal{D}$ be functors. A natural transformation $\eta : F \Rightarrow G$ is a family of morphisms $\{\eta_A \in \mathcal{D}(FA, GA)\}_{A \in \mathrm{Obj}(\mathcal{C})}$ such that, for every $f \in \mathcal{C}(A, B)$, the following diagram commutes:*

$$\begin{array}{ccc} FA & \xrightarrow{F(f)} & FB \\ \downarrow{\scriptstyle \eta_A} & & \downarrow{\scriptstyle \eta_B} \\ GA & \xrightarrow[G(f)]{} & GB \end{array}$$

5.1.2 Products and coproducts

Definition 5.1.4. *Given $A, B \in \mathrm{Obj}(\mathcal{C})$, the* binary product *of A and B is an object $A \,\&\, B$ in $\mathcal{C}$, along with morphisms $\pi_1 \in \mathcal{C}(A \,\&\, B, A)$ and $\pi_2 \in \mathcal{C}(A \,\&\, B, B)$, satisfying the following universal property: for pairs of morphisms $f \in \mathcal{C}(C, A)$ and $g : \mathcal{C}(C, B)$, there is a unique morphism $[f, g] \in \mathcal{C}(C, A \times B)$ such that*

$$\pi_1 \circ [f, g] = f \qquad \text{and} \qquad \pi_2 \circ [f, g] = g.$$

The notion of binary product $A \,\&\, B$ extends to the product of a finite product $A_1, \ldots, A_n \in \mathrm{Obj}(\mathcal{C})$ of objects in $\mathcal{C}$: the product $A_1 \,\&\, \cdots \,\&\, A_n \in \mathrm{Obj}(\mathcal{C})$ exists if there are morphisms $\pi_i \in \mathcal{C}(A_1 \,\&\, \cdots \,\&\, A_n, A_i)$ for each i, and, given morphisms $f_i \in \mathcal{C}(C, A_i)$, there is a unique morphism $[f_1, \cdots, f_n \in \mathcal{C}(C, A_1 \,\&\, \cdots \,\&\, A_n)$ such that $\pi_j \circ [f_1, \ldots, f_n] = f_j$.

A category $\mathcal{C}$ is called Cartesian *if it has all finite products, in the sense that every number n and objects $A_1, \ldots, A_n \in \mathrm{Obj}(\mathcal{C})$, there is a product $A_1 \,\&\, \cdots \,\&\, A_n$.*

Definition 5.1.5. *Dually to finite products, a category $\mathcal{C}$ has* finite sums *if, for every number n and objects $A_1, \ldots, A_n \in \mathrm{Obj}(\mathcal{C})$, there is an object $A_1 \oplus \cdots \oplus A_n$ and morphisms $\iota_i \in \mathcal{C}(A_i, A_1 \oplus \cdots \oplus A_n)$ such that, given morphisms $f_i \in \mathcal{C}(A_i, C)$, there is a unique morphism $[f_1; \ldots; f_n] \in \mathcal{C}(A_1 \oplus \cdots \oplus A_n, C)$ such that $[f_1; \ldots; f_n] \circ \iota_j = f_j$.*

5.1.3 Monads, comonads, and adjunctions

Definition 5.1.6. *A* monad *on a category $\mathcal{C}$ is a functor $M : \mathcal{C} \Rightarrow \mathcal{C}$ along with natural transformations $\{\eta_A \in \mathcal{C}(A, MA)\}_{A \in \mathrm{Obj}(\mathcal{C})}$ and $\{\mu_A \in \mathcal{C}(MMA, MA)\}_{A \in \mathrm{Obj}(\mathcal{C})}$ such that the following diagrams hold:*

$$\begin{array}{ccc} MA & \xrightarrow{\eta_{MA}} & MMA \\ {\scriptstyle M(\eta_A)}\downarrow & \searrow^{1_{MA}} & \downarrow{\scriptstyle \mu_A} \\ MMA & \xrightarrow[\mu_A]{} & MA \end{array} \qquad \begin{array}{ccc} MMMA & \xrightarrow{\mu_{MA}} & MMA \\ {\scriptstyle M(\mu_A)}\downarrow & & \downarrow{\scriptstyle \mu_A} \\ MMA & \xrightarrow[\mu_A]{} & MA \end{array}$$

Dually, a comonad *on $\mathcal{C}$ is a functor $M : \mathcal{C} \Rightarrow \mathcal{C}$ along with natural transformations $\epsilon_A \in \mathcal{C}(MA, A)$ and $\delta_A \in \mathcal{C}(MA, M(MA))$ such that the following diagrams hold:*

$$\begin{array}{ccc} MA & \xrightarrow{\delta_A} & MMA \\ {\scriptstyle \delta_A}\downarrow & \searrow^{1_{MA}} & \downarrow{\scriptstyle \epsilon_{MA}} \\ MMA & \xrightarrow[M(\epsilon_A)]{} & MA \end{array} \qquad \begin{array}{ccc} MA & \xrightarrow{\delta_A} & MMA \\ {\scriptstyle \delta_A}\downarrow & & \downarrow{\scriptstyle \delta_{MA}} \\ MMA & \xrightarrow[M(\delta_A)]{} & MMMA \end{array}$$

Although categorical monads are usually presented in terms of η and μ, they could equally be given by the interface using the return and bind operators that are popular in programming.

Proposition 5.1.7. *A functor $M : \mathcal{C} \Rightarrow \mathcal{C}$ forms a monad if and only if there is a natural transformation $\{\mathtt{return}_A \in \mathcal{C}(A, MA)\}_{A \in \mathrm{Obj}(\mathcal{C})}$ and, for every $f \in \mathcal{C}(A, MB)$ there is a morphism $\mathtt{bind}\ f$ such that $\mathtt{bind}\ \mathtt{return}_A = 1_{MA}$ and the following diagrams commute:*

$$\begin{array}{ccc} A & \xrightarrow{\mathtt{return}_A} & MA \\ & \searrow^{f} & \downarrow{\scriptstyle \mathtt{bind}\ f} \\ & & MB \end{array} \qquad \begin{array}{ccc} MA & \xrightarrow{M(f)} & MMB \\ {\scriptstyle \mathtt{bind}\ f}\downarrow & & \downarrow{\scriptstyle \mathtt{bind}(M(g))} \\ MB & \xrightarrow[\mathtt{bind}\ g]{} & MC \end{array}$$

Proof. Given a monad (M, η, μ), we take $\mathtt{return}_A$ to be η_A and $\mathtt{bind}\ f$ to be $\mu_B \circ M(f)$.

In the other direction, given `return` and `bind` as defined above, we take η_A to be $\mathtt{return}_A$ and μ_A to be $\mathtt{bind}\ 1_{MA}$.

In both cases, the fact that the relevant diagrams commute can be easily checked by diagram chases. □

Definition 5.1.8. *An* adjunction *$L \dashv R$ is a pair of functors $L : \mathcal{C} \Rightarrow \mathcal{D}$ and $R : \mathcal{D} \Rightarrow \mathcal{C}$ such that for every $C \in \mathrm{Obj}(\mathcal{C})$ and $D \in \mathrm{Obj}(\mathcal{D})$, there is an isomorphism between $\mathcal{D}(LC, D)$ and $\mathcal{C}(C, RD)$. We also say that L is* left adjoint *to R.*

Alternatively, $L \dashv R$ if and only if there are natural transformations $\eta_C : \mathcal{C}(C, RLC)$ and $\epsilon_D : \mathcal{D}(LRD, D)$ such that

$$\begin{array}{ccc} RD & \xrightarrow{\eta_{RD}} & RLRD \\ & \searrow^{1_{RD}} & \downarrow R(\epsilon_D) \\ & & RD \end{array} \qquad \begin{array}{ccc} LC & \xrightarrow{L(\eta_C)} & LRLC \\ & \searrow^{1_{LC}} & \downarrow \epsilon_{LC} \\ & & LC \end{array}$$

Adjunctions have a very close relationship with monads and comonads. We saw in Section 3.4 that the `Lift` and `Lower` functors (which we will show form an adjunction) give rise to $!\sigma \equiv \texttt{Lower}(\texttt{Lift}\ \sigma)$ and $\texttt{Lin}\ \alpha \equiv \texttt{Lift}(\texttt{Lower}\ \alpha)$, which form a comonad and monad respectively. These relationships are a consequence of the following general theorem about adjoint functors:

Proposition 5.1.9. *If $L : \mathcal{C} \Rightarrow \mathcal{D}$ and $R : \mathcal{D} \Rightarrow \mathcal{C}$ form an adjunction $L \dashv R$, then $R \circ L$ is a monad on $\mathcal{C}$, and $L \circ R$ is a comonad on $\mathcal{D}$.*

Proof. Given the adjunction (L, R, η, ϵ), then η and ϵ are the units of the monad and comonad respectively. We take μ_A to be $R(\epsilon_{LA})$ and δ_A to be $L(\eta_{RA})$. □

In Section 3.4 we saw that not only did the LNL model give rise to a monad and comonad, it also gave rise to a monad transformer. This fact also generalizes to arbitrary adjunctions.

Proposition 5.1.10. *If $L : \mathcal{C} \Rightarrow \mathcal{D}$ and $R : \mathcal{D} \Rightarrow \mathcal{C}$ form an adjunction $L \dashv R$, and if $M : \mathcal{D} \Rightarrow \mathcal{D}$ is a monad, then $R \circ M \circ L : \mathbb{C} \Rightarrow \mathbb{C}$ is also a monad.*

Proof. Let $(L, R, \eta^{RL}, \epsilon^{LR})$ be an adjunction, with $\eta^{RL}_C \in \mathcal{C}(C, RLC)$ and $\epsilon^{LR}_D \in \mathcal{D}(LRD, D)$, and let (M, η^M, μ^M) be a monad, with $\eta^M_D \in \mathcal{D}(D, MD)$ and $\mu^M_D \in \mathcal{D}(MMD, MD)$. Then $(RML, \eta^{RML}, \mu^{RML})$ forms a monad on $\mathcal{C}$, where η^{RML}_C is

$$C \xrightarrow{\eta^{RL}_C} RLC \xrightarrow{R(\eta^M_{LC})} RMLC$$

and μ^{RML}_C is

$$RMLRMLC \xrightarrow{RM(\epsilon^{LR}_{MLC})} RMMLC \xrightarrow{R(\mu^M_{LC})} RMLC.$$

□

5.2 Categories for multiplicative additive linear logic

What kind of categorical structure do we need to interpret a linear type system? Following the Curry-Howard correspondence, we are looking for a category $\mathcal{L}$ such that:

- For every type σ and typing context Δ, there are objects $[\![\sigma]\!], [\![\Delta]\!] \in \mathrm{Obj}(\mathcal{L})$; and
- For every linear term $\Delta \vdash e : \sigma$, there is a morphism $[\![e]\!] \in \mathcal{L}([\![\Delta]\!], [\![\sigma]\!])$; such that
- Whenever $e_1 \sim e_2$, we have $[\![e_1]\!] = [\![e_2]\!]$.

Such a category is known as a model of the type system.

Each linear connective—implication $\multimap$, multiplicative product $\otimes$, additive sum $\oplus$ and additive product &—requires some corresponding categorical, which we develop piece by piece.

Definition 5.2.1. *A* symmetric monoidal category *is a category* $\mathcal{C}$ *equipped with a bifunctor* $\otimes$*, an object* I*, and the following natural isomorphisms:*

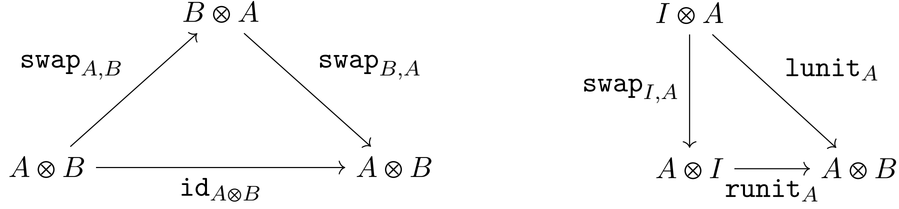
$$
\begin{array}{rlrl}
\texttt{assoc}_{A_1,A_2,A_3} : & A_1 \otimes (A_2 \otimes A_3) \to (A_1 \otimes A_2) \otimes A_3 \qquad & \texttt{lunit}_A : & I \otimes A \to A \\
\texttt{swap}_{A,B} : & A \otimes B \to B \otimes A & \texttt{runit}_A : & A \otimes I \to A
\end{array}
$$

These must satisfy the following coherence conditions:

$$
\begin{array}{ccccc}
 & & A_1 \otimes ((A_2 \otimes A_3) \otimes A_4) & & \\
 & \nearrow^{\texttt{id}_{A_1} \otimes \texttt{assoc}_{A_2,A_3,A_4}} & & \searrow^{\texttt{assoc}_{A_1,A_2 \otimes A_3,A_4}} & \\
A_1 \otimes (A_2 \otimes (A_3 \otimes A_4)) & & & & (A_1 \otimes (A_2 \otimes A_3)) \otimes A_4 \\
\downarrow{\scriptstyle \texttt{assoc}_{A_1,A_2,A_3 \otimes A_4}} & & & & \downarrow{\scriptstyle \texttt{assoc}_{A_1,A_2,A_3} \otimes \texttt{id}_{A_4}} \\
(A_1 \otimes A_2) \otimes (A_3 \otimes A_4) & & \xrightarrow[\texttt{assoc}_{A_1 \otimes A_2,A_3,A_4}]{} & & ((A_1 \otimes A_2) \otimes A_3) \otimes A_4
\end{array}
$$

$$
\begin{array}{ccccc}
 & & (A \otimes I) \otimes B & & \\
 & \nearrow^{\texttt{assoc}_{A,I,B}} & & \searrow^{\texttt{runit}_A \otimes \texttt{id}_B} & \\
A \otimes (I \otimes B) & & \xrightarrow[\texttt{id}_A \otimes \texttt{lunit}_B]{} & & A \otimes B
\end{array}
$$

$$
\begin{array}{ccc}
A_1 \otimes (A_2 \otimes A_3) & \xrightarrow{\texttt{id}_{A_1} \otimes \texttt{swap}_{A_2,A_3}} & A_1 \otimes (A_3 \otimes A_2) \\
\downarrow{\scriptstyle \texttt{assoc}_{A_1,A_2,A_3}} & & \downarrow{\scriptstyle \texttt{assoc}_{A_1,A_3,A_2}} \\
(A_1 \otimes A_2) \otimes A_3 & & (A_1 \otimes A_3) \otimes A_2 \\
\downarrow{\scriptstyle \texttt{swap}_{A_1 \otimes A_2,A_3}} & & \downarrow{\scriptstyle \texttt{swap}_{A_1,A_3} \otimes \texttt{id}_{A_2}} \\
A_3 \otimes (A_1 \otimes A_2) & \xrightarrow[\texttt{assoc}_{A_3,A_1,A_2}]{} & (A_3 \otimes A_1) \otimes A_2
\end{array}
$$



The characteristic semantics of linear implication $\sigma \multimap \tau$ is its relationship with the multiplicative product, in that morphisms from $\sigma \otimes \tau$ to ρ are isomorphic to morphisms from σ to $\tau \multimap \rho$.

Definition 5.2.2. *A* symmetric monoidal closed category *(SMCC) $\mathcal{C}$ is a symmetric monoidal category where, for every object $C \in \mathrm{Obj}(\mathcal{C})$, the functor $- \otimes C : \mathcal{C} \Rightarrow \mathcal{C}$ has a right adjoint, which we write as $C \multimap -$. In other words, the hom-set $\mathcal{C}(A \otimes C, B)$ is isomorphic to $\mathcal{C}(A, C \multimap B)$.*

Theorem 5.2.3 (Bierman, 1995)**.** *Let $\mathcal{L}$ be a symmetric monoidal closed category. Then $\mathcal{L}$ is a model of linear logic with $\otimes$, $\multimap$, and* `LUnit`*.*

The additive connectives & and $\oplus$ are accounted for by ordinary products and coproducts of Definitions 5.1.4 and 5.1.5. These operators are in addition to the ordinary monoidal structure $\otimes$.

Theorem 5.2.4 (Bierman, 1995)**.** *Let $\mathcal{L}$ be a symmetric monoidal closed category with finite products and coproducts. Then $\mathcal{L}$ is a model of multiplicative and additive linear logic.*

In contrast, if $\mathcal{C}$ is a symmetric monoidal closed category where the regular monoidal product $\otimes$ is Cartesian in the sense of Definition 5.1.4, then it is called a *Cartesian closed category* (CCC) and forms the basis of most categorical models of non-linear lambda calculi.

5.3 Linear/non-linear categories

Benton's linear/non-linear model consists of three main components:

- A symmetric monoidal closed category $\mathcal{L}$ to model linear expressions;

- A Cartesian closed category $\mathcal{C}$ to model non-linear expressions; and

- An adjunction $\mathtt{Lower} \dashv \mathtt{Lift}$ for functors $\mathtt{Lower} : \mathcal{C} \Rightarrow \mathcal{L}$ and $\mathtt{Lift} : \mathcal{L} \Rightarrow \mathcal{C}$.

Furthermore, the adjunction $\mathtt{Lower} \dashv \mathtt{Lift}$ needs to respect the monoidal structures of $\mathcal{L}$ and $\mathcal{C}$. In the next few definitions we characterize what it means for a functor, natural transformation, and adjunction to respect the monoidal structure.

Definition 5.3.1. *A* symmetric monoidal functor *$F : \mathcal{C} \Rightarrow \mathcal{C}'$ between symmetric monoidal categories $(\mathcal{C}, \otimes, I, \mathtt{ASSOC}, \mathtt{lunit}, \mathtt{runit}, \mathtt{SWAP})$ and $(\mathcal{C}', \otimes', I', \mathtt{ASSOC}', \mathtt{lunit}', \mathtt{runit}', \mathtt{SWAP}')$ is a functor F along with a map $m_I^F : I' \to FI$ and a natural transformation $m_{A,B}^F : F(A) \otimes' F(B) \to F(A \otimes B)$ that satisfies the following coherence conditions:*

$$
\begin{array}{ccc}
(F(A_1) \otimes' F(A_2)) \otimes' F(A_3) & \xrightarrow{\texttt{ASSOC}'} & F(A_1) \otimes' (F(A_2) \otimes' F(A_3)) \\
mF^F_{A_1,A_2} \otimes' I \Big\downarrow & & \Big\downarrow I' \otimes' m^F_{A_2,A_3} \\
F(A_1 \otimes A_2) \otimes' F(A_3) & & F(A_1) \otimes' F(A_2 \otimes A_3) \\
m^F_{A_1 \otimes A_2, A_3} \Big\downarrow & & \Big\downarrow m^F_{A_1, A_2 \otimes A_3} \\
F((A_1 \otimes A_2) \otimes A_3) & \xrightarrow{F(\texttt{ASSOC})} & F(A_1 \otimes (A_2 \otimes A_3))
\end{array}
$$

$$
\begin{array}{ccc}
F(A) \otimes' F(B) & \xrightarrow{\texttt{SWAP}'} & F(B) \otimes' F(A) \\
m^F_{A,B} \Big\downarrow & & \Big\downarrow m^F_{B,A} \\
F(A \otimes B) & \xrightarrow{F(\texttt{SWAP})} & F(B \otimes A)
\end{array}
$$

$$
\begin{array}{ccc}
I' \otimes' F(A) & \xrightarrow{\texttt{lunit}'} & F(A) \\
m^F_I \otimes' I' \Big\downarrow & & \Big\uparrow F(\texttt{lunit}_A) \\
F(I) \otimes' F(A) & \xrightarrow[m^F_{I,A}]{} & F(I \otimes A)
\end{array}
\qquad
\begin{array}{ccc}
F(A) \otimes' I' & \xrightarrow{\texttt{runit}'} & F(A) \\
I' \otimes' m^F_I \Big\downarrow & & \Big\uparrow F(\texttt{runit}) \\
F(A) \otimes' F(I) & \xrightarrow[m^F_{A,I}]{} & F(A \otimes I)
\end{array}
$$

Definition 5.3.2. *Let F and G be symmetric monoidal functors $F, G : \mathcal{C} \Rightarrow \mathcal{C}'$. A monoidal natural transformation $t : F \to G$ is a natural transformation satisfying*

$$t_{A \otimes B} \circ m^F_{A,B} = m^G_{A,B} \circ (t_A \otimes' t_B) \qquad \textit{and} \qquad t_I \circ m^F_I = m^G_I.$$

Definition 5.3.3. *A* symmetric monoidal adjunction *is an adjunction $F \dashv G$ between symmetric monoidal functors F and G where the unit η and counit ϵ of the adjunction are symmetric monoidal natural transformations.*

Definition 5.3.4 (Benton, 1995)**.** *A linear/non-linear model consists of:*

1. *a symmetric monoidal closed category $\mathcal{L}$;*
2. *a cartesian closed category $\mathcal{C}$; and*
3. *functors* `Lift` $: \mathcal{L} \Rightarrow \mathcal{C}$ *and* `Lower` $: \mathcal{C} \Rightarrow \mathcal{L}$ *that form a symmetric monoidal adjunction* `Lower` $\dashv$ `Lift`*.*

5.4 Embedded meta-theory

Working in a richly-typed host language, we can now study some of the meta-theoretic properties of the linear embedded language inside the host language itself. In this chapter we establish that the linear embedded language forms a linear/non-linear model with the host language.

Consider the category $\mathcal{C}_0$ of host-language terms, whose objects are types α and whose morphisms $f : \mathcal{C}_0(\alpha, \beta)$ are host-language functions $f : \alpha \to \beta$. Let $\mathcal{L}_0$ be the category whose objects are linear type σ and whose morphisms $e : \mathcal{L}_0(\sigma, \tau)$ are closed linear expressions $e : \texttt{Lift}(\sigma \multimap \tau)$ quotiented by equivalence $e \sim e'$ of expressions. As a reminder, Figure 5.1 shows β and η equivalences of the system, written $e_1 \sim_\beta e_2$ and $e_1 \sim_\eta e_2$ respectively.

For convenience we will write $e : \sigma \multimap \tau$ for $e : \texttt{Lift}(\sigma \multimap \tau)$ and omit uses of `suspend` and `force`, when the meaning is clear from the context.

$$
\begin{aligned}
\texttt{let}\ x \coloneqq e\ \texttt{in}\ e' &\sim_\beta e'\{e/x\} \\
(\lambda x.e')\hat{\ } e &\sim_\beta e'\{e/x\} \\
\texttt{let}\ (x_1,x_2) \coloneqq (e_1,e_2)\ \texttt{in}\ e' &\sim_\beta e'\{e_1/x_1, e_2/x_2\} \\
\texttt{case}\ \iota_i e\ \texttt{of}\ (\iota_1 x_1 \to e_1 \mid \iota_2 x_2 \to e_2) &\sim_\beta e_i\{e/x_i\} \\
\pi_i[e_1,e_2] &\sim_\beta e_i \\
\texttt{put}\ a >!\ f &\sim_\beta f a \\
\texttt{force}(\texttt{suspend}\ e) &\sim_\beta e
\end{aligned}
$$

$$
\begin{array}{rll}
e & \sim_\eta (\lambda x.e)x & \text{for } \Delta \vdash e : \sigma \multimap \tau \\
e'\{e/x\} & \sim_\eta \texttt{let}\ (x_1,x_2) \coloneqq e\ \texttt{in}\ e'\{(x_1,x_2)/x\} & \text{for } \Delta, x : \sigma_1 \otimes \sigma_2 \vdash e' : \tau \\
e'\{e/x\} & \sim_\eta \texttt{case}\ e\ \texttt{of}\ (\iota_1 x_1 \to e'\{\iota_1 x_1/x\} \mid \iota_2 x_2 \to e'\{\iota_2 x_2/x\}) & \text{for } \Delta, x : \sigma_1 \oplus \sigma_2 \vdash e' : \tau \\
e & \sim_\eta [\pi_1 e, \pi_2 e] & \text{for } \Delta \vdash e : \sigma_1 \& \sigma_2 \\
e'\{e/x\} & \sim_\eta e >!\ \lambda a.e'\{\texttt{put}\ a/x\} & \text{for } \Delta, x : \texttt{Lower}\ \alpha \vdash e' : \tau
\end{array}
$$

Figure 5.1: β and η equivalence for the embedded linear lambda calculus.

From the cartesian product $\times$ in the host language and the multiplicative product $\otimes$ in the embedded language, it is easy to check that $\mathcal{C}_0$ is cartesian closed and $\mathcal{L}_0$ is symmetric monoidal closed.

Lemma 5.4.1. `Lower` *is a symmetric monoidal functor.*

Proof. Given a host function $f : \alpha \to \beta$ there is a linear expression $\texttt{fmap}_f : \texttt{Lower}\ \alpha \multimap \texttt{Lower}\ \beta$ defined as

$$\texttt{fmap}_f(e) \equiv \texttt{let}\ !a \coloneqq e\ \texttt{in}\ \texttt{put}(fa).$$

`Lower` is a functor because `fmap` satisfies the functor laws:

$$
\begin{aligned}
\texttt{fmap}_{\lambda a.a}(e) &\equiv \texttt{let}\ !a \coloneqq e\ \texttt{in}\ \texttt{put}\ a \sim_\eta e \\
\texttt{fmap}_{g\circ f}(e) &\equiv \texttt{let}\ !a \coloneqq e\ \texttt{in}\ \texttt{put}(g(fa)) \\
&\sim_\beta \texttt{let}\ !a \coloneqq e\ \texttt{in}\ \texttt{let}\ !a' \coloneqq \texttt{put}(fa)\ \texttt{in}\ \texttt{put}(ga') \\
&\sim_\eta \texttt{let}\ !a' \coloneqq (\texttt{let}\ !a \coloneqq e\ \texttt{in}\ \texttt{put}(fa))\ \texttt{in}\ \texttt{put}(ga') \\
&\equiv \texttt{fmap}_g(\texttt{fmap}_f(e))
\end{aligned}
$$

To check that `Lower` is symmetric monoidal we must exhibit the following morphisms:

$$
\begin{array}{ll}
m_I^{\texttt{Lower}} : \texttt{LUnit} \multimap \texttt{Lower Unit} & m_{\alpha,\beta}^{\texttt{Lower}} : \texttt{Lower}\ \alpha \otimes \texttt{Lower}\ \beta \multimap \texttt{Lower}(\alpha \times \beta) \\
m_I^{\texttt{Lower}} \equiv \hat{\lambda} x.\ \texttt{let}\ () \coloneqq x\ \texttt{in}\ \texttt{put}\ () & \qquad \equiv \hat{\lambda} z.\texttt{let}\ (!a, !b) \coloneqq z\ \texttt{in}\ \texttt{put}\ (a,b)
\end{array}
$$

Checking the monoidal conditions is straightforward, but tedious. □

Lemma 5.4.2. `Lift` *is a symmetric monoidal functor.*

Proof. For $f : \sigma \multimap \tau$ and $t : \texttt{Lift}\ \sigma$, define $\texttt{fmap}_f : \texttt{Lift}\ \sigma \to \texttt{Lift}\ \tau$ as

$$\texttt{fmap}_f(a) \equiv \texttt{suspend}(f\hat{}(\texttt{force}\ a)).$$

This function is functorial:

$$\begin{aligned}
\texttt{fmap}_{\hat{\lambda}x.x}(a) &= \texttt{suspend}((\hat{\lambda}x.x)\hat{}(\texttt{force}\ a)) \sim_\beta \texttt{suspend}(\texttt{force}\ a) = a \\
\texttt{fmap}_{g \circ f}(a) &= \texttt{suspend}(g\hat{}(f\hat{}(\texttt{force}\ a))) \\
&= \texttt{suspend}(g\hat{}(\texttt{force}(\texttt{suspend}(f\hat{}(\texttt{force}\ a))))) = \texttt{fmap}_g(\texttt{fmap}_f(a))
\end{aligned}$$

The monoidal components are exhibited by the following operations:

$$\begin{aligned}
&m_I^{\texttt{Lift}} : \texttt{Unit} \to \texttt{Lower Unit} && m_{\sigma,\tau}^{\texttt{Lift}} : \texttt{Lift}\ \sigma \times \texttt{Lift}\ \tau \to \texttt{Lift}(\sigma \otimes \tau) \\
&m_I^{\texttt{Lift}} \equiv \lambda_.\texttt{suspend}\ () && m_{\sigma,\tau}^{\texttt{Lift}} \equiv \lambda a.\texttt{suspend}\ (\texttt{force}(\pi_1 a), \texttt{force}(\pi_2 a))
\end{aligned}$$

□

Lemma 5.4.3. $\texttt{Lower} \dashv \texttt{Lift}$ *forms a symmetric monoidal adjunction.*

Proof. The adjunction $\texttt{Lower} \dashv \texttt{Lift}$ is given by the following natural transformations:

$$\begin{aligned}
\eta_\alpha &: \alpha \to \texttt{Lift}(\texttt{Lower}\ \alpha) && \epsilon_\sigma : \texttt{Lower}(\texttt{Lift}\ \sigma) \multimap \sigma \\
\eta_\alpha a &\equiv \texttt{suspend}(\texttt{put}\ a) && \epsilon_\sigma \equiv \hat{\lambda}x.\ x >!\ \texttt{force}
\end{aligned}$$

To show that $(\texttt{Lower}, \texttt{Lift}, \eta, \epsilon)$ really form an adjunction, we must show they satisfy two properties. First, that $\texttt{Lift}(\epsilon_\sigma) \circ \eta_{\texttt{Lift}\ \sigma} = \texttt{id}_{\texttt{Lift}\ \sigma}$:

$$\begin{aligned}
\texttt{Lift}(\epsilon_\sigma)\,(\eta_{\texttt{Lift}\ \sigma} a) &= \texttt{Lift}(\epsilon_\sigma)\,(\texttt{suspend}(\texttt{put}\ a)) \\
&= \texttt{suspend}(\epsilon_\sigma\hat{}\texttt{force}(\texttt{suspend}(\texttt{put}\ a))) \\
&\sim_\beta \texttt{suspend}(\epsilon_\sigma\hat{}\texttt{put}\ a) \\
&= \texttt{suspend}(\texttt{put}\ a >!\ \texttt{force}) \\
&\sim_\beta \texttt{suspend}(\texttt{force}\ a) \\
&\sim_\eta a
\end{aligned}$$

Next, we need to show that $\epsilon_{\texttt{Lower}\ \alpha} \circ \texttt{Lower}(\eta_\alpha) = \texttt{id}_{\texttt{Lower}\ \alpha}$:

$$\begin{aligned}
\epsilon_{\texttt{Lower}\ \alpha}(\texttt{Lower}(\eta_\alpha)(x)) &= \texttt{let}\ !b \coloneqq (\texttt{Lower}(\eta_\alpha)(x))\ \texttt{in force}\ b \\
&= \texttt{let}\ !b \coloneqq (\texttt{let}\ !a \coloneqq x\ \texttt{in put}(\eta_\alpha a))\ \texttt{in force}\ b \\
&\sim_\eta \texttt{let}\ !a \coloneqq x\ \texttt{in let}\ !b \coloneqq \texttt{put}(\eta_\alpha a)\ \texttt{in force}\ b \\
&\sim_\beta \texttt{let}\ !a \coloneqq x\ \texttt{in force}(\eta_\alpha a) \\
&= \texttt{let}\ !a \coloneqq x\ \texttt{in force}(\texttt{suspend}(\texttt{put}\ a)) \\
&\sim_\beta \texttt{let}\ !a \coloneqq x\ \texttt{in put}\ a \\
&\sim_\eta x
\end{aligned}$$

To show the adjunction is symmetric monoidal, it suffices to show that η and ϵ are

symmetric monoidal natural transformations—that is, that they commute with $m^{\texttt{Lift}\circ\texttt{Lower}}$ and $m^{\texttt{Lower}\circ\texttt{Lift}}$ respectively. These facts follow easily by unfolding definitions. □

Theorem 5.4.4. *The categories $\mathcal{C}_0$ and $\mathcal{L}_0$ form an LNL model.*

This result follows immediately from Lemma 5.4.3. It also follows from the discussion in Section 5.1.3 that the adjunction $\texttt{Lift} \dashv \texttt{Lower}$ gives rise to a comonad $\texttt{Lower}(\texttt{Lift}\ \sigma)$ on the linear category, which corresponds to the type operator $!\sigma$. In addition, $\texttt{Lin}\ \alpha = \texttt{Lift}(\texttt{Lower}\ \alpha)$ forms a monad and $\texttt{LinT}\ M\ \alpha = \texttt{Lift}(M(\texttt{Lower}\ \alpha))$ a monad transformer, formalizing the structures we programmed with in Section 3.4.

5.5 Conclusion

Establishing the category theory background of embedded LNL type systems does two things. First, it acts as a sanity check that our type system has a sound semantics and matches the semantics of other linear type systems. Second, it shows that the embedded language approach is strong enough to reason about its own type system.

The remainder of the dissertation will focus less on the high-level semantics and theory of embedded LNL type systems, and more on a particular case study—quantum computing. We will see how the needs of a particular DSL and the tools available from the host language can shape the structure and meta-theory of the embedded linear language, building on the foundation of linear/non-linear type theory.

Case study: Quantum computing

CHAPTER 6

A quantum/non-quantum type system

Quantum computing is an exciting upcoming area of computer science and physics. By harnessing the power of quantum mechanics, quantum computers have the potential to solve problems for which there is no known effective classical (non-quantum) algorithm. Such computers are still in their infancy, but the theory of quantum computing is in full swing. Shor's algorithm describes how to factor large numbers in polynomial time (Shor, 1999), Grover's algorithm describes how to search databases in logarithmic time (Grover, 1996), and several algorithms describe how to simulate other quantum systems, such as those found in chemistry and high-energy physics (Georgescu *et al.*, 2014).

Instead of bits, quantum computers operate on *qubits*—superpositions of classical bits of the form $c_0 |0\rangle + c_1 |1\rangle$ where $c_0, c_1 : \mathbb{C}$ and $|c_0|^2 + |c_1|^2 = 1$. A qubit can be measured, resulting in the bit 0 with probability $|c_0|^2$, or the bit 1 with probability $|c_1|^2$. Qubits e can also be manipulated by applying *unitary* transformations U, which we write $U \# e$. Quantum algorithms, therefore, need three domain-specific features: to initialize qubits, to apply unitary gates, and to measure qubits. However, they also need to be able to process the probabilistic, classical results of measurements, which means they also need good support for classical computing.

Programming with qubits presents many challenges, and programming languages are in a unique position to address them. The design of domain-specific quantum programming languages fall mainly into two camps.

Formal semantics and meta-theory. From the start of the field, researchers in the area of quantum programming languages have prioritized their formal semantics and meta-theory. The correctness of quantum algorithms can be subtle, and specifications of correctness are closely tied to the mathematical foundations of quantum computing. Without a clear semantics, it is impossible to reliably reason about the correctness of a quantum program.

Additionally, quantum languages use formal semantics to justify new programming abstractions when it might not be clear whether they are sound with respect to quantum computing. This has been the case for higher-order functions (Selinger and Valiron, 2009), recursion (Ying, 2014), and quantum conditionals (Ying *et al.*, 2014). Such computational features, though common for classical programming domains, are quite subtle quantum-mechanically, so special care must be taken to make sure they are valid in quantum languages.

Other innovative abstractions in quantum computing can be expressed with radically new programming abstractions. The ZX calculus is a graphical calculus with an elegant equational theory that is sound and complete with respect to quantum computing (Backens,

2015). However, the graphical nature of the calculus makes it somewhat alien from the perspective of classical programming. The measurement calculus is based on a computing model where only measurement is allowed, and not unitary transformations (Danos *et al.*, 2007). Algebraic quantum calculi allow algebraic reasoning about quantum programs which line up with their semantics (Altenkirch and Green, 2010; Vizzotto *et al.*, 2013).

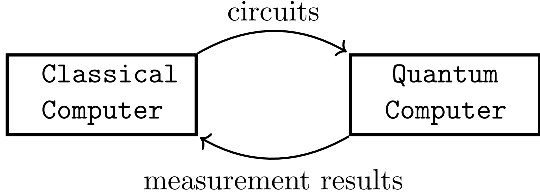
The most widely accepted programming model, however, is the quantum circuit model, where quantum programs consist of initialization, unitary transformations, and measurement, as well as classical features for manipulating the results of measurement.

For many of the languages following the quantum circuit model, linear type systems are a key factor that enables reasoning about the language's semantics. The need for linear types is closely tied to the mathematics of quantum computing in terms of linear transformations and linear algebra. More specifically, quantum physics abides by the so-called "no-cloning" theorem, which says that an arbitrary qubit's state cannot be duplicated. Naturally, linear types can be used to reject programs that would otherwise try to copy quantum data.

The quantum lambda calculus is a simple linear lambda calculus with a type of qubits, multiplicative products, and types for classical/non-linear data $!\sigma$ (van Tonder, 2004; Selinger and Valiron, 2009). As the name would suggest, it also provides higher-order linear functions. As we have seen throughout this dissertation, however, languages with non-linear data marked with the type $!\alpha$ are often impractical or difficult to use.

The QRAM model. Implementations of quantum programming languages have had to balance sound language abstractions like linear types against usability and accessibility for the wide range of computer scientists, mathematicians, and physicists who are developing quantum algorithms and building quantum computers. Languages like Quipper (Green *et al.*, 2013b), LIQ$Ui|\rangle$ (Wecker and Svore, 2014), and Q# (Svore *et al.*, 2018) provide high-level, modular programming abstractions for working with both quantum and classical data, though they lack the same degree of theory as found in *e.g.,* the quantum lambda calculus.

The relationship between quantum and classical data used in these high-level languages is known as the *QRAM model*, which describes how a quantum computer could work in tandem with a classical computer (Knill, 1996). In the QRAM model, the classical computer handles the majority of ordinary tasks, while the quantum computer performs specialized quantum operations. To communicate, the classical computer sends instructions to the quantum machine in the form of quantum circuits, and the quantum computer sends measurement results back to the classical computer as needed.



The philosophy of the QRAM model clearly has a lot in common with the embedded LNL framework described in this dissertation. Indeed, several state-of-the-art quantum circuit languages are implemented as embedded domain-specific languages.

Quipper is one such language embedded in Haskell (Green *et al.*, 2013b), and it takes advantage of many features from its host language, including monads, meta-programming

using Template Haskell (Sheard and Jones, 2002), type classes, and more. Quipper has been used to develop real-world quantum algorithms, and provides tools to visualize, simulate, and optimize quantum circuits. Other languages and toolkits follow a similar pattern: LIQ$Ui|\rangle$ is embedded in F# (Wecker and Svore, 2014); the Q language is embedded in C++ (Bettelli *et al.*, 2003); and Project Q[11], QISKit[12], and pyQuil[13] are all embedded in Python.

Unfortunately, these languages mostly lack formal meta-theory. For example, because Quipper is embedded in Haskell, it does not provide linear type checking and so is not type safe. In addition, Quipper does not have a formal semantics because giving such a semantics would require reasoning about all of Haskell, which itself does not have a formal semantics. The Proto-Quipper project is working to formalize the semantics of Quipper, but its target is limited to a standalone variant of Quipper, not the actual implementation in Haskell (Ross, 2015; Rios and Selinger, 2018).

The embedded linear/non-linear methodology developed in this dissertation gives us a way to retain linear types and a formal semantics while working inside a classical host language. In the next few chapters, we present a few variations of a quantum linear type system and its meta-theory developed in a dependently-typed host language. This case study shows how the linear/non-linear framework can be used to develop sound and expressive EDSLs in the domain of quantum computing.

In the remainder of this chapter we present a first-order embedded quantum lambda calculus that we call the *quantum/non-quantum* (QNQ) calculus. In Section 6.1 we discuss some relevant background on the mathematics of quantum computing, and in Section 6.2 we presents the calculus itself. We illustrate the calculus with a number of examples in Section 6.3, and give a denotational semantics in Section 6.4.

6.1 Quantum computing background

In this section we give some background on the mathematics of quantum computing. There are many excellent textbooks that provide a more comprehensive and nuanced perspective; we refer the interested reader to the standard text in the area by Nielsen and Chuang (2010). For the sake of this dissertation, we only expect the reader to be familiar with some basic concepts from linear algebra.

6.1.1 Pure states

A qubit is a vector in $\mathbb{C}^2$, the complex-valued two-dimensional vector space:

$$\begin{pmatrix} c_0 \\ c_1 \end{pmatrix}$$

where c_0 and c_1 are complex numbers such that $|c_0|^2 + |c_1|^2 = 1$. Recall that $|c|$ is the norm of a complex number c such that $|a + bi| \equiv \sqrt{a^2 + b^2}$.

We write $|0\rangle \equiv \left(\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right)$ and $|1\rangle \equiv \left(\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right)$ for a particular basis set of $\mathbb{C}^2$; we think of these states as the quantum analogue of bits 0 and 1. But qubits are not bits, since they exist in a *superposition* of $|0\rangle$ and $|1\rangle$. The general state $\left(\begin{smallmatrix} c_0 \\ c_1 \end{smallmatrix}\right)$ can be decomposed as $c_0 |0\rangle + c_1 |1\rangle$.

[11] `https://projectq.ch/`
[12] `https://github.com/QISKit`
[13] `https://github.com/rigetticomputing/pyQuil`

Measuring a qubit in the state $c_0 |0\rangle + c_1 |1\rangle$ along the basis $\{|0\rangle, |1\rangle\}$ is a probabilistic operation that results in a classical bit 0 with probability $|c_0|^2$, or 1 with probability $|c_1|^2$. Notice that $|c_0|^2$ and $|c_1|^2$ are both real numbers, and since $|c_0|^2 + |c_1|^2 = 1$, the result is a probability distribution.

Entanglement. An n-qubit state is a 2^n-dimensional vector with basis elements $|b_1\rangle \otimes \cdots \otimes |b_{n-1}\rangle$, where each $b_i \in \{0,1\}$ and $\otimes$ denotes the tensor product, also called the Kronecker product, of matrices. For convenience, we write $|b_1, \ldots, b_{n-1}\rangle$ for $|b_0\rangle \otimes \cdots \otimes |b_{n-1}\rangle$.

Measuring the ith qubit in an n-qubit system results in an $n-1$-qubit state. For simplicity, suppose we are measuring the first qubit in the system. We can always write the state of the system as $c_0 |0\rangle \otimes \varphi_0 + c_1 |1\rangle \otimes \varphi_1$ where each φ_i is an $n-1$-qubit system and $|c_0|^2 + |c_1|^2 = 1$. Then measuring that state will result in the state φ_0 with probability $|c_0|^2$ and φ_1 with probability $|c_1|^2$.

A *pure state* is an n-qubit state in $\mathbb{C}^{2^n}$, and a *mixed state* is a probability distribution over pure states.

If an n-qubit system can be decomposed into the tensor product of n individual qubits, such as $\varphi_1 \otimes \cdots \otimes \varphi_n$, then we call that system *separable*. However, not all systems are separable; consider for example the two-qubit system

$$\frac{1}{\sqrt{2}} |00\rangle + \frac{1}{\sqrt{2}} |11\rangle .$$

This state is known as the *Bell pair*, and it is interesting because measuring the first qubit collapses the state of the second qubit, and vice versa. That is, if measuring the first qubit results in a 0, then the second qubit will be in the classical state $|0\rangle$, and if measuring the first qubit results in a 1, then the second qubit will be in the classical state $|1\rangle$.

We say that two qubits are *entangled* if they are not separable.

Unitaries. Besides measurement, qubits can be transformed by applying *unitary transformations*, square matrices U such that U's conjugate transpose $U^\dagger$ is its own inverse. The conjugate transpose has entry $(U[j,i])^*$ at index $U^\dagger[i,j]$, where c^* is the complex conjugate of $c \in \mathbb{C}$: $(a+bi)^* \equiv a - bi$.

We write X for the "not" unitary transformation $\left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$, which sends $|0\rangle$ to $|1\rangle$ and $|1\rangle$ to $|0\rangle$, and we write H for the Hadamard matrix

$$H \equiv \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}$$

which sends $|0\rangle$ to $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|1\rangle$ to $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$.

For an n-qubit unitary U, the controlled unitary $\texttt{ctrl}\ U$ is an $n+1$-qubit unitary that is the identity on $|0\rangle \otimes \varphi$, but sends $|1\rangle \otimes \varphi$ to $|1\rangle \otimes U\varphi$. In other words, it applies U only when its control is $|1\rangle$. Some commonly used controlled unitaries include the controlled not operator $\texttt{CNOT} \equiv \texttt{ctrl}\ X$ and the Toffoli transformation $T \equiv \texttt{ctrl}(\texttt{ctrl}\ X)$.

The Bell state shown above can be derived by applying a Hadamard transformation to

the first qubit, followed by a controlled not gate:

$$\begin{aligned}
\texttt{CNOT}(H \otimes I)(|0\rangle \otimes |0\rangle) &= \texttt{CNOT}(\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |0\rangle) \\
&= \texttt{CNOT}(\frac{1}{\sqrt{2}}(|00\rangle + |10\rangle)) \\
&= \frac{1}{\sqrt{2}}(\texttt{CNOT}\,|00\rangle + \texttt{CNOT}\,|10\rangle) \\
&= \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)
\end{aligned}$$

The no-cloning theorem. In the introduction we referenced the no-cloning theorem as a motivation for linear types, and we can now make this intuition formal.

Theorem 6.1.1 (Nielsen and Chuang, 2010)**.** *There is no unitary transformation U such that for all qubits φ and classical state $|b\rangle$,*

$$U(\varphi \otimes |b\rangle) = \varphi \otimes \varphi.$$

6.1.2 Density matrices

In the formulation above, quantum programs containing measurement and unitary applications correspond to probabilistic transformations over pure states. That is, a state φ will be mapped to a probability distribution over pure states of the form $c_0\,|0\rangle + c_1\,|1\rangle$. *Density matrices* are matrix representations of probability distributions over pure states, which make it easier to reason about the behavior of quantum programs.

Formally, a density matrix is a positive Hermitian matrix whose trace sums to 1. Any pure state in column vector form $\varphi = \left(\begin{smallmatrix} c_0 \\ c_1 \end{smallmatrix}\right)$ can be transformed into a density matrix by taking its outer product with itself:

$$|\varphi\rangle\,\langle\varphi| = |\varphi\rangle\,|\varphi\rangle^\dagger = \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} \begin{pmatrix} c_0^* & c_1^* \end{pmatrix} = \begin{pmatrix} c_0 c_0^* & c_0 c_1^* \\ c_0^* c_1 & c_1 c_1^* \end{pmatrix}.$$

The state of an entangled Bell pair $\frac{1}{\sqrt{2}}\,|00\rangle + \frac{1}{\sqrt{2}}\,|11\rangle$ can be represented as the following density matrix:

$$\begin{pmatrix} 1/2 & 0 & 0 & 1/2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1/2 & 0 & 0 & 1/2 \end{pmatrix}$$

where the $1/2$ in the top left represents the probability of measuring two zeros, while the $1/2$ in the bottom right represents the probability of measuring two ones. After measuring this system, we would obtain the *mixed state* density matrix

$$\begin{pmatrix} 1/2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1/2 \end{pmatrix}$$

representing a probability distribution over $|00\rangle$ and $|11\rangle$.

Quantum computations can now be described as *superoperators*—completely positive maps that preserve the trace of their input matrix. The density matrix formulation means that there is no need to reason about probability distributions at the meta-level, since probability distributions are baked into the structure of density matrices.

Any matrix M can be lifted to a function $M^*(\rho) = M^\dagger \rho M$, and if M is unitary, then M^* is a superoperator.

Superoperators are subject to an additive structure, scalar multiplication, and a multiplicative structure $\otimes$. The construction of $\otimes$ is based on the fact that every superoperator S can be broken down into a sum of unitary transformations $\mathcal{U}^S$:

$$S = \sum_{U:\mathcal{U}^S} U^*.$$

Then $S \otimes T$ is defined as

$$S \otimes T \equiv \sum_{U:\mathcal{U}^S, V:\mathcal{U}^T} (U \otimes T)^*.$$

Formally, qubit initialization is defined as the superoperator $(|0\rangle)^*$, and qubit measurement is defined as $(|0\rangle\langle 0|)^* + (|1\rangle\langle 1|)^*$.

6.1.3 Category theory

Selinger (2004) formalizes the category **CPM** of density matrices and completely positive maps in a slightly different way from the presentation here. In the category **CPM**, objects do not correspond to density matrices, but rather tuples of density matrices such that qubits in different components cannot be entangled. This allows the category **CPM** to distinguish the domain of qubits from the domain of classical bits, since classical bits are always separable from the rest of the quantum state.

The category **CPM** is generalized by the theory of *dagger compact closed categories* (Abramsky and Coecke, 2004; Selinger, 2007), which are symmetric monoidal categories with a dagger involution $\sigma^\dagger$ on morphisms, satisfying certain properties. In earlier work, I showed how to extend linear/non-linear categories to models of classical linear logic (Paykin and Zdancewic, 2016); to extend them to dagger compact closed categories could use a similar technique, which we will not explore in this work.

Cho (2016) proposes a categorical semantics in terms of operator algebras, which admit recursive types and are the basis of a line of work for developing quantum domain theory (Rennela, 2014; Rennela and Staton, 2018). Staton (2015) describes an equational theory for quantum algebras (see Chapter 7) whose completeness result is based on the theory of operator algebras.

Several other categorical formulations have been proposed in order to accommodate higher-order functions, recursive types and programs, or other high-level programming abstractions (Malherbe, 2010; Hasuo and Hoshino, 2011; Pagani *et al.*, 2014). In this work we use the simple presentation of superoperators over density matrices as described above, where the type of bits and qubits are isomorphic. We propose, however, that the technique of embedded category theory for an embedded linear language is quite rich, and deserves further study.

6.2 The quantum/non-quantum (QNQ) calculus

This chapter will present an embedded linear lambda calculus based on Selinger and Valiron's quantum lambda calculus, which we call the *quantum/non-quantum* (QNQ) calculus. QNQ is an example of a linear/non-linear embedded DSL with first-order functions and primitives for quantum computing, and it illustrates how the LNL model can be highly expressive and also semantically sound. It will also serve as the basis for the calculi in Chapters 7 and 8, which explore different meta-theoretic aspects of quantum programming languages.

We start with an embedded language with multiplicative unit and pairs, and non-linear types `Lower` α. We choose to restrict our semantics to finite-dimensional vector spaces, so we add a restriction that α must be finite in order for it to be embedded in the linear language, as shown in Figure 6.1. The restriction to finite types is less restrictive than it seems, because we will encode data structures, typically represented as infinite types like lists, as dependent indexed data structures $n \otimes \sigma$. For any particular number n, an n-tuple $n \otimes \sigma$ of σ's is a finite type, but there are a countably infinite collection of such types available.

The basic β equivalences of linear/non-linear type systems hold for this fragment, and η equivalences hold for the multiplicative unit and product.

$$\texttt{let}\ x \coloneqq e\ \texttt{in}\ e' \sim_\beta e'\{e/x\} \qquad \texttt{let}\ (x_1, x_2) \coloneqq (e_1, e_2)\ \texttt{in}\ e' \sim_\beta e'\{e_1/x_1, e_2/x_2\}$$

$$\texttt{let}\ () \coloneqq ()\ \texttt{in}\ e' \sim_\beta e' \qquad \texttt{put}\ a >!\ f \sim_\beta f a$$

$$\frac{\Delta \vdash_{\text{QNQ}} e : \texttt{LUnit} \qquad \Delta', x : \texttt{LUnit} \vdash_{\text{QNQ}} e' : \tau}{e'\{e/x\} \sim_\eta \texttt{let}\ () \coloneqq e\ \texttt{in}\ e'\{()/x\}} \qquad \frac{\Delta \vdash_{\text{QNQ}} e : \sigma_1 \otimes \sigma_2 \qquad \Delta', x : \sigma_1 \otimes \sigma_2 \vdash_{\text{QNQ}} e' : \tau}{e'\{e/x\} \sim_\eta \texttt{let}\ (x_1, x_2) \coloneqq e\ \texttt{in}\ e'\{(x_1,x_2)/x\}}$$

Eta equivalence for the type `Lower` α is more subtle, however. Semantically, the operation >! corresponds to quantum measurement, so a traditional η rule that introduces a >! operator could be problematic. If the type system guarantees that all data of type `Lower` α is always in a classical state, then η equivalence could be allowed. However, Chapter 7 violates that restriction by taking the type of qubits to be equal to `Lower Bool`, so traditional η equivalence will not be allowed. However, we still want >! bindings to be able to commute inside a term; we call these *commuting conversions* (Girard *et al.*, 1989, Chapter 10). Thus, the following rule is admissible:

$$\frac{\Delta \vdash_{\text{QNQ}} e : \texttt{Lower}\ \alpha \qquad f : \alpha \to \texttt{LExp}_{\text{QNQ}}\ \Delta'\ \sigma \qquad \Delta'', x : \sigma \vdash_{\text{QNQ}} e' : \tau \qquad \Delta \perp \Delta' \perp \Delta''}{e'\{(e >!\ f)/x\} \sim_{cc} e >!\ \lambda a. e'\{f a/x\}}$$

Qubits. In addition to the standard linear types, QNQ has a type `Qubit` of qubits. Given a boolean b, we can initialize a qubit in the classical state b, written `init` b, and we can measure qubits, resulting in a lowered boolean value.

$$\frac{b : \texttt{Bool}}{\varnothing \vdash_{\text{QNQ}} \texttt{init}\ b : \texttt{Qubit}}\ \texttt{Qubit-I} \qquad \frac{\Delta \vdash_{\text{QNQ}} e : \texttt{Qubit}}{\Delta \vdash_{\text{QNQ}} \texttt{meas}\ e : \texttt{Lower Bool}}\ \texttt{Qubit-E}$$

$$\frac{}{x:\sigma \vdash_{\text{QNQ}} x:\sigma}\ \text{VAR} \qquad \frac{\Delta \vdash_{\text{QNQ}} e:\sigma \qquad \Delta', x:\sigma \vdash_{\text{QNQ}} e':\tau \qquad \Delta \bot \Delta'}{\Delta, \Delta' \vdash_{\text{QNQ}} \texttt{let}\ x \coloneqq e\ \texttt{in}\ e':\tau}\ \text{LET}$$

$$\frac{}{\varnothing \vdash_{\text{QNQ}} ():\texttt{LUnit}}\ \texttt{LUnit-I} \qquad \frac{\Delta \vdash_{\text{QNQ}} e:\texttt{LUnit} \qquad \Delta' \vdash_{\text{QNQ}} e':\tau \qquad \Delta \bot \Delta'}{\Delta, \Delta' \vdash_{\text{QNQ}} \texttt{let}\ () \coloneqq e\ \texttt{in}\ e':\tau}\ \texttt{LUnit-E}$$

$$\frac{\Delta_1 \vdash_{\text{QNQ}} e_1:\sigma_1 \qquad \Delta_2 \vdash_{\text{QNQ}} e_2:\sigma_2 \qquad \Delta_1 \bot \Delta_2}{\Delta_1, \Delta_2 \vdash_{\text{QNQ}} (e_1, e_2):\sigma_1 \otimes \sigma_2}\ \otimes\text{-I}$$

$$\frac{\Delta \vdash_{\text{QNQ}} e:\sigma_1 \otimes \sigma_2 \qquad \Delta', x_1:\sigma_1, x_2:\sigma_2 \vdash_{\text{QNQ}} e':\tau \qquad \Delta \bot \Delta'}{\Delta, \Delta' \vdash_{\text{QNQ}} \texttt{let}\ (x_1, x_2) \coloneqq e\ \texttt{in}\ e':\tau}\ \otimes\text{-E}$$

$$\frac{a:\alpha \qquad \alpha\ \text{finite}}{\varnothing \vdash_{\text{QNQ}} \texttt{put}\ a:\texttt{Lower}\ \alpha}\ \texttt{Lower-I}$$

$$\frac{\Delta \vdash_{\text{QNQ}} e:\texttt{Lower}\ \alpha \qquad f:\alpha \to \texttt{LExp}_{\text{QNQ}}\ \Delta'\ \tau \qquad \Delta \bot \Delta'}{\Delta, \Delta' \vdash_{\text{QNQ}} e >!\ f:\tau}\ \texttt{Lower-E}$$

Figure 6.1: Multiplicative exponential fragment of QNQ

The operational behavior states that measuring a newly initialized qubit will result in the original boolean value.

$$\texttt{meas}(\texttt{init}\ b) \rightsquigarrow_\beta \texttt{put}\ b$$

Unitary transformations. Unitary transformations, written $U:\mathcal{U}(\sigma,\tau)$, can be applied to linear expressions as follows:

$$\frac{U:\mathcal{U}(\sigma,\tau) \qquad \Delta \vdash_{\text{QNQ}} e:\sigma}{\Delta \vdash_{\text{QNQ}} U \# e:\tau}\ \text{U}$$

We expect several facts to hold about the behavior of unitary transformations, such as the fact that $U^\dagger \# U \# e$ is equivalent to e, and that $X \# \texttt{init}\ b$ is equivalent to $\texttt{init}(\neg b)$. We defer the discussion of these equivalences until Chapter 7, when we explore them in depth.

Functions. What is a quantum function? At first approximation, a function is just a procedure—a sequence of instructions applied to quantum data. Such procedures should be duplicable, in that a procedure does not get consumed when it is applied to an argument, and modular, in that they can be combined to build up layers of abstraction. Selinger (2004) shows that a quantum language with first-order procedures can be interpreted in the category of density matrices and completely positive maps.

The type $\sigma \multimap \tau$ of (possibly higher-order) linear functions offers more than just sim-

ple procedures, however, and treats higher-order functions as first class quantum data. The quantum lambda calculus demonstrates that higher-order functions are syntactically sound—it is possible to give an operational semantics to a quantum lambda calculus with higher-order functions (van Tonder, 2004; Selinger and Valiron, 2009). However, the denotational semantics (especially the quantum-mechanical interpretation) of higher-order quantum functions is not entirely settled. Selinger and Valiron (2008) show that higher-order functions are compatible with density matrices when bits and all other classical data is treated linearly in the program. Other denotational models have been proposed based on presheaves (Malherbe, 2010), geometry of interaction (Hasuo and Hoshino, 2011), and quantitative semantics (Pagani *et al.*, 2014), but it is still unclear how such functions would be implementable on a quantum computer. Quipper, a quantum circuit language based on the quantum lambda calculus, compiles away its higher-order functions while generating the circuits to be executed on a quantum computer (Green *et al.*, 2013b).

In the QNQ calculus, we choose to restrict functions to first-order procedures, which simplifies the meta-theory and is sufficient to express quantum algorithms (see Section 6.3). First-order procedures are also helpful when we adapt QNQ to a quantum circuit language in Chapter 8. However, QNQ could be extended with higher-order functions in the style of the quantum lambda calculus.

A first-order function in QNQ is called a *box*. A box of type $\texttt{Box}\ \sigma\ \tau$ holds a linear expression $x : \sigma \vdash_{\text{QNQ}} e : \tau$ that uses exactly one linear variable of type σ. A box is a generalization of the type $\texttt{Lift}\ \tau$; whereas $\texttt{Lift}\ \tau$ holds a linear expression using no linear variables, $\texttt{Box}\ \sigma\ \tau$ holds a linear expression using exactly one linear variable. We write $\texttt{box}\ x \Rightarrow e$ for the constructor of type $\texttt{Box}\ \sigma\ \tau$, and write $f\ \$\ e$ for function application.

$$\frac{x : \sigma \vdash_{\text{QNQ}} e : \tau}{\texttt{box}\ x \Rightarrow e : \texttt{Box}\ \sigma\ \tau}\ \textsc{box-I} \qquad \frac{f : \texttt{Box}\ \sigma\ \tau \qquad \Delta \vdash_{\text{QNQ}} e : \sigma}{\Delta \vdash_{\text{QNQ}} f\ \$\ e : \tau}\ \textsc{box-E}$$

The β- and η-equivalences for boxes are the same as those for ordinary λ abstraction.

$$(\texttt{box}\ x \Rightarrow e')\ \$\ e \sim_\beta e'\{e/x\} \qquad \frac{b : \texttt{Box}\ \sigma\ \tau}{b \sim_\eta \texttt{box}\ x \Rightarrow b\ \$\ x}$$

Boxed functions are composable: for $f : \texttt{Box}\ \sigma\ \tau$ and $g : \texttt{Box}\ \tau\ \rho$, we write $g \circ f : \texttt{Box}\ \sigma\ \rho$ for $\texttt{box}\ x \Rightarrow g\ \$\ (f\ \$\ x)$, and for $f_1 : \texttt{Box}\ \sigma_1\ \tau_1$ and $f_2 : \texttt{Box}\ \sigma_2\ \tau_2$, we write $f_1 \| f_2 : \texttt{Box}\ (\sigma_1 \otimes \sigma_2)\ \tau_1 \otimes \tau_2$ for the parallel composition $\texttt{box}\ (x_1, x_2) \Rightarrow (f_1\ \$\ x_1, f_2\ \$\ x_2)$. Every unitary transformation $U : \mathcal{U}(\sigma,\tau)$ can be coerced into a box of type $\texttt{Box}\ \sigma\ \tau$, and we sometimes just write $U : \texttt{Box}\ \sigma\ \tau$ for the coercion $\texttt{box}\ x \Rightarrow U \# x$.

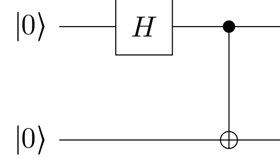
Note that we can derive $\texttt{Lift}\ \tau$ as $\texttt{Box}\ \texttt{LUnit}\ \tau$ and $\texttt{suspend}\ e$ as $\texttt{box}\ x \Rightarrow \texttt{let}\ () \coloneqq x\ \texttt{in}\ e$.

6.3 Examples

In this section we give a number of examples of common quantum programs in QNQ.

The Bell state described in Section 6.1 can be constructed by applying a hadamard unitary transformation to a newly initialized qubit (which puts the qubit into a superposition $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$) and then applying a controlled-not unitary on another newly initialized qubit. Intuitively, this program corresponds to the circuit diagram shown here.

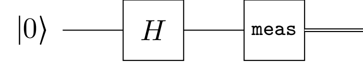
```
bell00 : Box LUnit (Qubit ⊗ Qubit)
bell00 ≡ box () ⇒ let a ≔ H # init 0 in
                  let b ≔ init 0 in
                  ctrl-X # (a,b)
```



We write box $(x_1, \ldots, x_n) \Rightarrow e$ for pattern matching against input to a box. For example, in the definition of `bell00` `box () ⇒ c` in is syntactic sugar for `box x ⇒ let () ≔ x in c`.

Coin flips. Measuring a qubit results in a lowered boolean value, and so, taking `Lin` α to be `Box LUnit` (`Lower` α), we can construct a quantum coin flip inside the linearity monad as follows:

```
flip : Lin Bool
flip ≡ box () ⇒ meas (H # init 0)
```



The program `flipN` flips up to n quantum coins until any of the coins returns `False`.

```
flipN : Nat → Lin Bool
flipN 0     ≡ return True
flipN (n+1) ≡ do b ← flip
                 if b then flipN n else return False
```

This program will return `True` with probability $\frac{1}{2^n}$ and `False` with probability $1 - \frac{1}{2^n}$.

Quantum natural numbers with dependent types. An n-qubit quantum natural number is a term of type $n \bigotimes$ `Qubit`, which we defined in Section 3.5 as follows:

$$0 \bigotimes \sigma \equiv \texttt{LUnit}$$
$$n+1 \bigotimes \sigma \equiv \sigma \otimes (n \bigotimes \sigma)$$

Notice that $n \bigotimes \sigma$ is a linear functor on `LType`'s; for $f :$ `Box` σ τ we can write `lfmap` f : $\prod_n$ `Box` $(n \bigotimes \sigma)$ $(n \bigotimes \tau)$ as follows:

```
lfmap : Box σ τ → Π n, Box (n ⨂ σ) (n ⨂ τ)
lfmap f 0     ≡ box ()     ⇒ ()
lfmap f (n+1) ≡ box (x,xs) ⇒ (f $ x, lfmap n $ xs)
```

We write `Tuple n` α for the corresponding host-language type of n-tuples. Because `Lower` distributes over ⊗, we can transform a linear n-tuple of (`Lower` α)'s into a lowered n-tuple of α's.

```
distrN : Π n. Box (n ⨂ Lower α) (Lower (Tuple n α))
distrN 0     ≡ box ()     ⇒ put ()
distrN (n+1) ≡ box (x,xs) ⇒ let !a ≔ x in
                            let !as ≔ distrN n $ xs in
                            put (a,as)
```

Now we can measure an n-qubit natural number into an n-tuple of boleans by combining `lfmap` and `distrN`.

```
measN : Π n. Box (n ⨂ Qubit) (Lower (Tuple n Bool))
measN n ≡ distrN ∘ lfmap meas
```

Quantum teleportation. The quantum version of `Hello world` is the quantum teleportation algorithm, which describes how Alice can transmit the state of a qubit to Bob at a remote location. Despite the name, teleportation does not imply faster-than-light communication, because, in order to transmit the information, Alice must send Bob two classical bits. Figure 6.2 shows the teleportation algorithm, broken up into three parts.

First, Alice and Bob receive two ends of a shared Bell state—this is their shared secret before moving to separate locations. At her secret location, Alice also has a qubit that she wishes to send to Bob. Alice entangles q with her end of the Bell state by applying a controlled X gate followed by a Hadamard gate. Notice that all three qubits are entangled at this point. Next, Alice measures her two qubits. Because of the no-cloning theorem, Alice cannot both send her qubit's state to Bob and also keep her own copy of the qubit, so measurement degrades her original qubit. However, the state of Bob's qubit has now changed as the result of the measurement.

Supposing that q was originally in the state $c_0 |0\rangle + c_1 |1\rangle$, we can now work out the state of the system after Alice is done with her measurement:

- If Alice measures $|0\rangle |0\rangle$, then Bob's qubit is in the state $c_0 |0\rangle + c_1 |1\rangle$—the state of the original qubit.
- If Alice measures $|0\rangle |1\rangle$, then Bob's qubit is in the state $c_0 |1\rangle + c_1 |0\rangle$—the negation of the original qubit.
- If Alice measures $|1\rangle |0\rangle$, then Bob's qubit is in the state $c_0 |0\rangle - c_1 |1\rangle$.
- If Alice measures $|1\rangle |1\rangle$, then Bob's qubit is in the state $c_0 |1\rangle - c_1 |0\rangle$.

If Bob knows the results of Alice's measurement, he can apply a particular sequence of unitary transformations to correct the state of his qubit to return it to the original position. For example, if Bob knows the result was $|0\rangle |1\rangle$, then he can apply an X unitary to transform the qubit back into the state $c_0 |0\rangle + c_1 |1\rangle$.

Thus, in the final part of the teleportation algorithm, Alice sends her two classical bits to Bob, who then applies his corrections depending on the state of those bits.

Quantum Fourier transform (QFT) The quantum Fourier transform (QFT) computes the discrete Fourier transform on n qubits in $O(n^2)$ time, and is a key component in many quantum algorithms, including Shor's factorization algorihtm. In comparison, the classical Fourier transform takes $O(n2^n)$.

The QFT is defined by induction on the number of input qubits, as shown in Figure 6.3. At the inductive step, the circuit calls out to the helper function `rotations`, which applies a sequence of controlled rotation gates $R\ m$, which are indexed by a natural number m. This example is adapted from an introduction to Quipper, to which we refer the reader for full details of the algorithm (Green *et al.*, 2013a). However, because Quipper does not have dependent quantum types, it cannot express the fact that the QFT has the same number of input and output qubits. Instead, Quipper uses lists and other data types for qubits, that must be instantiated at circuit generation time.

6.4 Denotational semantics

In this section we describe how to map quantum programs to superoperators over density matrices.

```
alice : Box (Qubit ⊗ Qubit) (Lower Bool ⊗ Lower Bool)
alice ≡ box (q,a) ⇒ let (q,a) ≔ CNOT # (q,a) in
                    (meas (H # q), meas a)

bob : Bool → Bool → Box Qubit Qubit
bob x y ≡ if x && y       then Z ∘ X
          else if x && ¬ y then X
          else if ¬x && y  then Z
          else id

teleport : Box Qubit Qubit
teleport ≡ box q ⇒ let (a,b)   ≔ bell00 $ () in
                   let (!x,!y) ≔ alice $ (q,a) in
                   bob x y $ b
```

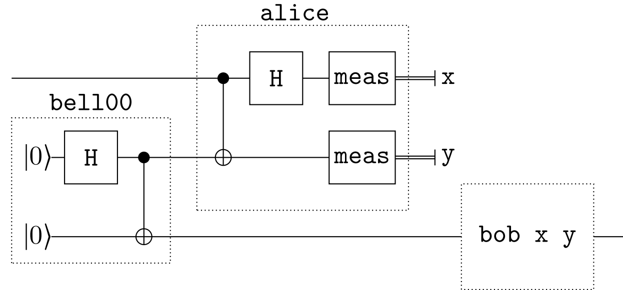


Figure 6.2: Quantum teleportation.

```
QFT : Π (n : Nat), Box (n ⊗ Qubit) (n ⊗ Qubit)
QFT 0      ≡ box ()     ⇒ ()
QFT 1      ≡ box (x,()) ⇒ (H # x, ())
QFT (n'+2) ≡ box (x,xs) ⇒ let xs ≔ QFT (n'+1) $ xs
                          let xs ≔ rotations n' (n'+1) $ (x,xs) in
                          (H # x, xs)

rotations : Π (n m : Nat), Box (n+2 ⊗ Qubit) (n+2 ⊗ Qubit)
rotations 0      ≡ box x ⇒ x
rotations 1      ≡ box x ⇒ x
rotations (n'+2) ≡ box (c,x,xs) ⇒ let (c,xs) ≔ rotations (n'+1) m $ (c,xs) in
                                  let (c,x)  ≔ ctrl (R (m-n'+1)) $ (c,x) in
                                  (c,x,xs)
```

Figure 6.3: Quantum Fourier transform in QNQ

We identify every type σ with a dimension $[\![\sigma]\!]$.

$$\begin{aligned}
[\![\texttt{LUnit}]\!] &\equiv 0 \\
[\![\texttt{Qubit}]\!] &\equiv 1 \\
[\![\sigma_1 \otimes \sigma_2]\!] &\equiv [\![\sigma_1]\!] + [\![\sigma_2]\!] \\
[\![\texttt{Lower}\ \alpha]\!] &\equiv |\alpha|
\end{aligned}$$

Because we restricted the constructor for `Lower` α to finite types, the size $|\alpha|$ is well-defined. For each type α, we also pick a canonical ordering on its elements, so that values $a : \alpha$ can be mapped to distinct vectors δ_a of dimension $2^{|\alpha|}$.

Each typing context is also associated with a dimension.

$$[\![\varnothing]\!] \equiv 0 \qquad [\![\Delta, x : \sigma]\!] \equiv [\![\Delta]\!] + [\![\sigma]\!]$$

We write `Density` σ for the type of $2^{[\![\sigma]\!]} \times 2^{[\![\sigma]\!]}$ density matrices and similarly for `Density` Δ. Our goal is to map expressions $\Delta \vdash_{\text{QNQ}} e : \sigma$ to functions `Density` $\Delta \to$ `Density` σ.

The category of density matrices is symmetric monoidal, which tells us how to interpret the multiplicative unit and product.

Every unitary $U : \mathcal{U}(\sigma, \tau)$ will be associated with a unitary matrix $[\![U]\!]$ of dimension $2^{[\![\sigma]\!]} \times 2^{[\![\tau]\!]}$. We can lift this to a superoperator $[\![U]\!]^* :$ `Density` $\sigma \to$ `Density` τ; recall that $U^*(\rho) = U^\dagger \rho U$.

$$[\![U \mathbin{\#} e]\!] \equiv [\![U]\!]^* \circ [\![e]\!]$$

For qubits, initialization and measurement are defined as follows:

$$\begin{aligned}
&[\![\texttt{init}\ b]\!] \equiv \langle b|^* \\
&[\![\texttt{meas}\ e]\!] \equiv ((|0\rangle\langle 0|)^* + (|1\rangle\langle 1|)^*) \circ [\![e]\!]
\end{aligned}$$

The semantics of `Lower` is a generalization of the semantics of qubits. Notice that the types `Qubit` and `Lower Bool` have the same representation as density matrices.

$$\begin{aligned}
&[\![\texttt{put}\ a]\!] \equiv (\delta_a^\dagger)^* \\
&[\![e \mathbin{>!} f]\!] \equiv \sum_{a:\alpha} [\![f a]\!] \circ (\delta_a \delta_a^\dagger)^* \circ [\![e]\!]
\end{aligned}$$

Finally, the interpretation of a quantum function $f :$ `Box` $\sigma\ \tau$ is a superoperator `Density` $\sigma \to$ `Density` τ, and applying f to an argument composes their denotations together.

$$\begin{aligned}
[\![\texttt{box}\ x \Rightarrow e]\!] &\equiv [\![e]\!] \\
[\![f \mathbin{\$} e]\!] &\equiv [\![f]\!] \circ [\![e]\!]
\end{aligned}$$

6.4.1 Soundness

Theorem 6.4.1 (Soundness of β-reduction)**.** *If $\Delta \vdash_{QNQ} e : \tau$ and $e \rightsquigarrow_\beta e'$, then $[\![e']\!] = [\![e]\!]$.*

Proof. The only interesting cases are those for qubits and the `Lower` type.

For qubits, we have $\texttt{meas(init}\ b) \leadsto_\beta \texttt{put}\ b$. Unfolding definitions, we have

$$[\![\texttt{meas(init}\ b)]\!] \equiv ((|0\rangle\langle 0|)^* + (|1\rangle\langle 1|)^*) \circ \langle b|^* . \tag{6.1}$$

Because $M^* \circ N^*$ is exactly $(NM)^*$, Equation (6.1) is equal to

$$(\langle b|\,|0\rangle\langle 0|)^* + (\langle b|\,|1\rangle\langle 1|)^*.$$

Notice that $|b\rangle\langle b'|$ is equal to the constant 1 if $b = b'$, or 0 if $b \neq b'$. Thus Equation (6.1) reduces to $\langle b|^*$.

For $\texttt{Lower}$, we have $\texttt{put}\ a >!\ f \leadsto_\beta fa$. Then:

$$\begin{aligned}
[\![\texttt{put}\ a >!\ f]\!] &= \sum_{b:\alpha} [\![fb]\!] \circ (\delta_b \delta_b^\dagger)^* [\![\texttt{put}\ a]\!] \\
&= \sum_{b:\alpha} [\![fb]\!] \circ (\delta_b \delta_b^\dagger)^* (\delta_a^\dagger)^* \\
&= \sum_{b:\alpha} [\![fb]\!] \circ (\delta_a^\dagger \delta_b \delta_b^\dagger)^*
\end{aligned}$$

As for qubits, it is the case that $\delta_a^\dagger \delta_b$ is 1 if $a = b$, and 0 otherwise. Therefore, the sum reduces to the single clause $[\![fa]\!]$. □

CHAPTER 7

Quantum equational theories in HoTT

The β- and η-equivalences of the QNQ calculus described in the previous section cover only a few of the equivalences we expect to hold from a quantum system. The operational semantics of unitary transformations cannot be expressed as equivalences—there is no simpler representation of the expression $H \mathbin{\#} \texttt{init}\ 0$, for example. Even so, there are several properties of unitary transformations that can be expressed equationally. Consider, for example the X unitary transformation, which maps $|0\rangle$ to $|1\rangle$ and $|1\rangle$ to $|0\rangle$. We expect $X \mathbin{\#} \texttt{init}\ b$ to be equivalent to $\texttt{init}(\neg b)$, and we also expect $\texttt{meas}(X \mathbin{\#} e)$ to be equivalent to $\texttt{meas}\ e >!\ \lambda b.\texttt{put}(\neg b)$.

Staton (2015) describes an equational theory for a small quantum programming language, given as an algebraic theory with quantum data and classical control. Staton's algebraic theory includes measurement-based branching, but it does not contain explicit classical data or features we would expect from a QRAM-style language, let alone access to an entire classical host language. In this chapter we adapt Staton's equational theory to the QNQ calculus, continuing to reason about the meta-theory of an embedded language inside its host language.

Other works propose axiomatizations of particular sets of unitary transformations (Amy *et al.*, 2018; Matsumoto and Amano, 2008; Nam *et al.*, 2018), but, like Staton, we focus on the relationships between quantum and classical features.

What features should our host language have in order to perform this meta-theory? To reason about program equivalences, we choose to work in a host language that specializes in equality—homotopy type theory (HoTT). In HoTT, proofs of equality, also called *paths*, can contain extra computational content. In the past few years a variety of applications have used these paths as a data structure for groupoids, such as containers (Abbott *et al.*, 2004), version control patches (Angiuli *et al.*, 2014), and SQL queries (Chu *et al.*, 2017).

The key contribution of this chapter is the observation that *unitary transformations form a groupoid*. We exploit this structure to encode unitaries in the paths between quantum types. With this structure, we can derive many of the structural rules from Staton's theory. We group the remaining axioms into two families of axioms describing the behavior of unitaries with respect to initialization and measurement, respectively.

The rest of this chapter is devoted to establishing the equational theory of QNQ in homotopy type theory. Section 7.1 gives some background of HoTT. Section 7.2 describes a slight variation of QNQ with additive sums and gives an overview of the equational principles we expect to hold. Section 7.3 shows how to derive many of the structural rules by encoding unitary transformations as higher inductive types. Section 7.4 establishes the remaining two axioms that cannot be derived from the HoTT meta-theory. Section 7.5

proves that these additional axioms are sound with respect to the denotational semantics of Section 6.4. Finally, Section 7.6 discusses some of the design decisions that went into this theory.

7.1 Background and main ideas

7.1.1 Homotopy type theory (HoTT)

Homotopy type theory is, in many ways, a type theory of equivalence. HoTT is based on the idea that proofs of equality $a = b$, called *paths*, may have computational content (Univalent Foundations Program, 2013). That is, there may be other proofs of equality besides the (trivial) reflexivity path $1_a : a = a$.

In homotopy type theory, we write the type of *propositional* proofs of equality as $a = b$; that is, $a = b$ is a type with a single constructor $1_a : a = a$. Propositional equality is distinguished from *judgmental* equality $a \equiv b$, which asserts that a and b are equal by definition. The judgment $a \equiv b$ is not a type; it is only valid in the meta-theory and has no computational content. For more intuition on the difference between propositional and judgmental equality, see the HoTT book (Univalent Foundations Program, 2013, Chapter 1).

Homotopy type theory was developed as a type-theoretic alternative to set theory, but it has applications in a wide variety of domains (Angiuli *et al.*, 2014; Chu *et al.*, 2017; Abbott *et al.*, 2004). When a domain is difficult to characterize equationally, but uses data in the shape of an equivalence relation or groupoid, HoTT can help.

Consider a type A that you want to quotient by a relation R. For every element $a : A$, the equivalence class of a is written $[a]_R : A/R$, and whenever $R(a, b)$, it should be the case that $[a]_R = [b]_R$. In set theory it is possible to define the equivalence class as a set $[a]_R = \{x : A \mid R(a, x)\}$, so that $[a]_R$ contains the same elements as $[b]_R$. However, in programming environments, where the representation of data structures matters, sets are often implemented as lists or arrays, so $[a]_R$ does not necessarily have the same representation as $[b]_R$.

Homotopy type theory addresses this discrepancy with *higher inductive types*, which are made up of both term constructors and also path constructors. For example:

Definition 7.1.1 (van Doorn *et al.* (2017))**.** *The quotient A/R of a type A by a relation $R : A \to A \to \mathcal{U}$ is a higher inductive type generated by the following constructors:*

- *for $a : A$, there is a term $[a]_R : A/R$; and*

- *for $a, b : A$ and $r : R\ a\ b$, there is a proof $[r]_R$ that $[a]_R = [b]_R$.*

Notice that if r_1 and r_2 are two different witnesses of $R\ a\ b$, then $[r_1]_R$ is different from $[r_2]_R$—the structure of the relation R is preserved in the paths A/R.

Despite the extra computational content, the usual properties of equality types still hold for paths generated by higher inductive types. The principle of path induction states that, given a property $P : \prod_{a,b:\alpha} a = b \to \texttt{Type}$ on paths, if P holds on the reflexivity path, then P holds on any path. That is, the induction principle for paths has the following type:

$$\texttt{path_ind}_P : \left(\prod(x : \alpha), P(1_x)\right) \to \prod(x\ y : \alpha)(p : x = y),\ P(p).$$

If $p : a = b$ for $a, b : \alpha$ and $x : P(a)$ for some property $P : \alpha \to \texttt{Type}$, then it is possible to *transport* the path p over x to obtain a proof $\texttt{transport}_P\ p\ x : P(b)$. If a and b are types and $x : a$, we write $\texttt{coerce}\ p\ x \equiv \texttt{transport}_{\lambda x.x}\ p\ x : b$.

In HoTT, functions $f : \alpha \to \beta$ are *functorial*, meaning that $p : a = b$ on α can be promoted to $\texttt{ap}_f\ p : f a = f b$.

The *univalence axiom* states that an equivalence $f : \alpha \cong \beta$ between types can be treated as a path $\texttt{univ}\ f : \alpha = \beta$, such that $\texttt{coerce}\ (\texttt{univ}\ f)\ a = f a$.

7.1.2 Unitaries as paths

The core idea of this chapter is to encode the unitary operators used in quantum computing in the higher inductive structure of quantum types. Unitary transformations $\mathcal{U}(\alpha, \beta)$ form a groupoid (a category whose morphisms are all invertible) whose objects are types, and so we take quantum types q, r to be members of $\texttt{QType} \equiv \texttt{Type}/\mathcal{U}$. This means the type of paths $[\alpha]_{\mathcal{U}} = [\beta]_{\mathcal{U}}$ contains members corresponding to unitary transformations $U : \mathcal{U}(\alpha, \beta)$.

Encoding unitaries as paths has two important benefits. First, there is no need for explicit syntax for applying a unitary transformation; it can be defined to be the result of transporting the path $U : q = r$ over a term $\Delta \vdash_{\text{QNQ}} e : q$, obtaining $U \# e = \texttt{transport}\ U\ e$. Second, many of the structural axioms on unitaries can now be proven by path induction. For example, consider the following statement:

Proposition 7.1.2. *Suppose $\Delta \vdash_{QNQ} e : q$. Then, for $U : q = r$ and $V : r = s$ we have*

$$V \# (U \# e) = (V \circ U) \# e.$$

Proof. By path induction over V. If V is the trivial path by reflexivity on r, written 1_r, then $V \circ U = U$. Furthermore, by definition of the $\texttt{transport}$, for all x we have $1 \# x = x$. So $1 \# (U \# e) = U \# e = (1 \circ U) \# e$. □

Crucially, it is *not* possible to prove the following false statement:

Proposition 7.1.3 (False)**.** *Let $\Delta \vdash_{QNQ} e : q$ and $U : q = q$. Then $U \# e = e$.*

Path induction only applies on proofs $a = b$ when at least one of a or b is a free variable, so it does not apply here. In fact, the statement is false–we can prove that $X \# |0\rangle = |1\rangle$, but it is not the case that $|0\rangle = |1\rangle$.

7.2 Equational theory of QNQ

In developing the equational theory, we make three changes to the QNQ calculus described in Chapter 7.

Qubits as `Lower Bool`. First, instead of taking `Qubit` to be a primitive type of qubits, we instead define `Qubit` to be `Lower Bool`, since qubits are represented by a two-dimensional vector space. We can define initialization and measurement as follows:

$$\texttt{init} : \texttt{Bool} \to \texttt{LExp}\ \varnothing\ \texttt{Qubit} \qquad \texttt{meas} : \texttt{Box Qubit}\ (\texttt{Lower Bool})$$
$$\texttt{init} \equiv \lambda b.\ \texttt{put}\ b \qquad \texttt{meas} \equiv \texttt{box}\ x \Rightarrow \texttt{let}\ !b \coloneqq x\ \texttt{in put}\ b$$

On first glance, these definitions appear to do nothing—`meas` in particular is just the η expansion of e. But the encoding of `Qubit` as `Lower Bool` highlights a critical semantic

fact of our system: *case analysis performs quantum measurement.* This has a number of consequences for the theory of the language, including the fact that η expansion is not sound in general: a measured qubit is *not* equivalent to an unmeasured one.

By choosing to encode measurement as case analysis, we open the door to a very expressive quantum theory. For example, the type $\texttt{Lower(Bool} \times \texttt{Bool)}$ is equivalent to the two-qubit system $\texttt{Qubit} \otimes \texttt{Qubit} \equiv \texttt{Lower Bool} \otimes \texttt{Lower Bool}$. We can also easily encode a qutrit (a base-3 quantum system) as $\texttt{Lower}(() + () + ())$.

We write $\sigma \Leftrightarrow \tau$ for an equivalence between σ and τ that arises from an isomorphism on basis sets. For example, $\texttt{swap} : \sigma \otimes \tau \Leftrightarrow \tau \otimes \sigma$ arises from symmetry and $\texttt{distr} : \sigma \otimes (\tau_1 \oplus \tau_2) \Leftrightarrow \sigma \otimes \tau_1 \oplus \sigma \otimes \tau_2$ reflects the distributivity of $\otimes$ over $\otimes$. We formalize the definition of such equivalences in Section 7.4.

Strengthened η equivalence for LUnit. As discussed in Section 6.2, we do allow the usual η equivalences for the tensor product and linear unit type, as follows:

$$\frac{\Delta \vdash_{\text{QNQ}} e : \texttt{LUnit} \qquad \Delta', x : \texttt{LUnit} \vdash_{\text{QNQ}} e' : \tau}{e'\{e/x\} \sim_\eta \texttt{let } () \coloneqq e \texttt{ in } e'\{()/x\}} \qquad \frac{\Delta \vdash_{\text{QNQ}} e : \sigma_1 \otimes \sigma_2 \qquad \Delta', x : \sigma_1 \otimes \sigma_2 \vdash_{\text{QNQ}} e' : \tau}{e'\{e/x\} \sim_\eta \texttt{let } (x_1, x_2) \coloneqq e \texttt{ in } e'\{(x_1, x_2)/x\}}$$

However, for units we actually expect a stronger η equivalence: LUnit is a terminal object in the category of density matrices. That is, every two terms of type LUnit are equivalent.

$$\frac{\Delta \vdash_{\text{QNQ}} e_1 : \texttt{LUnit} \qquad \Delta \vdash_{\text{QNQ}} e_2 : \texttt{LUnit}}{e_1 \sim_\eta e_2}$$

This is not a consequence of the usual linear η rule above, but is a consequence of the cartesian η equivalence, which states that every term of type unit is equal to the unit value.

Additive sums. Staton's equational theory relies on the fact that unitaries can be combined via the direct sum, which corresponds to the additive sum in the type theory. Therefore we extend QNQ with additive sums, written $\sigma_1 \oplus \sigma_2$. Although sums do not arise naturally in the context of quantum circuits, they do occur in several quantum programming languages.

$$\frac{\Delta \vdash_{\text{QNQ}} e_i : \sigma_i}{\Delta \vdash_{\text{QNQ}} \iota_i e_i : \sigma_1 \oplus \sigma_2} \qquad \frac{\Delta \vdash_{\text{QNQ}} e : \sigma_1 \oplus \sigma_2 \qquad \Delta', x_1 : \sigma_1 \vdash_{\text{QNQ}} e_1 : \tau \qquad \Delta', x_2 : \sigma_2 \vdash_{\text{QNQ}} e_2 : \tau}{\Delta, \Delta' \vdash_{\text{QNQ}} \texttt{case } e \texttt{ of } (\iota_1 x_1 \to e_1 \mid \iota_2 x_2 \to e_2) : \tau}$$

We allow β equivalences but not general η-equivalence for sums; like for the Lower type, case analysis for sums corresponds to measuring a quantum state. However, we do allow the commuting conversion rule.

$$\texttt{case } \iota_i e \texttt{ of } (\iota_1 x_1 \to e_1 \mid \iota_2 x_2 \to e_2) \sim_\beta e_i\{e/x_i\}$$

$$\frac{\Delta \vdash_{\text{QNQ}} e : \sigma_1 \oplus \sigma_2 \qquad \Delta_0, x_1 : \sigma_1 \vdash_{\text{QNQ}} e_1 : \sigma \qquad \Delta_0, x_2 : \sigma_2 \vdash_{\text{QNQ}} e_2 : \sigma \qquad \Delta', x : \sigma \vdash_{\text{QNQ}} e' : \tau}{e'\{(\texttt{case } e \texttt{ of } (\iota_1 x_1 \to e_1 \mid \iota_2 x_2 \to e_2))/x\} \sim_{cc} \texttt{case } e \texttt{ of } (\iota_1 x_1 \to e'\{e_1/x\} \mid \iota_2 x_2 \to e'\{e_1/x\})}$$

$$(U_1 \otimes U_2) \mathbin{\#} (e_1, e_2) \approx (U_1 \mathbin{\#} e_1, U_2 \mathbin{\#} e_2) \tag{U-⊗-intro}$$

$$\begin{aligned} &\texttt{let } (x_1, x_2) \coloneqq (U_1 \otimes U_2) \mathbin{\#} e \texttt{ in } e' \\ &\quad \approx \texttt{let } (y_1, y_2) \coloneqq e \texttt{ in } e'\{U_1 \mathbin{\#} y_1/x_1, U_2 \mathbin{\#} y_2/x_2\} \end{aligned} \tag{U-⊗-elim}$$

$$U \mathbin{\#} (\texttt{let } (x_1, x_2) \coloneqq e \texttt{ in } e') \approx \texttt{let } (x_1, x_2) \coloneqq e \texttt{ in } U \mathbin{\#} e' \tag{U-⊗-comm}$$

$$(U_0 \oplus U_1) \mathbin{\#} (\iota_0 e) \approx U_0 \mathbin{\#} e \tag{U-⊕-intro$_0$}$$

$$(U_0 \oplus U_1) \mathbin{\#} (\iota_1 e) \approx U_1 \mathbin{\#} e \tag{U-⊕-intro$_1$}$$

$$\begin{aligned} &\texttt{case } (U_0 \oplus U_1) \mathbin{\#} e \texttt{ of } (\iota_0 x_0 \to e_0 \mid \iota_1 x_1 \to e_1) \\ &\quad \approx \texttt{case } e \texttt{ of } (\iota_0 y_0 \to e_0\{U_0 \mathbin{\#} y_0/x_0\} \mid \iota_1 y_1 \to e_1\{U_1 \mathbin{\#} y_1/x_1\}) \end{aligned} \tag{U-⊕-elim}$$

$$U \mathbin{\#} (\texttt{case } e \texttt{ of } (\iota_0 x_0 \to e_0 \mid \iota_1 x_1 \to e_1)) \approx \texttt{case } e \texttt{ of } (\iota_0 x_0 \to U \mathbin{\#} e_0 \mid \iota_1 x_1 \to U \mathbin{\#} e_1) \tag{U-⊕-comm}$$

$$U \mathbin{\#} (e >! f) \approx e >! \lambda x \to U \mathbin{\#} (f x) \tag{U-Lower-comm}$$

$$U \mathbin{\#} e >! \lambda_.e' \approx e >! \lambda_.e' \tag{U-Lower-elim}$$

Figure 7.1: Structural axioms

$$U \mathbin{\#} (V \mathbin{\#} e) \approx (U \circ V) \mathbin{\#} e \tag{U-compose}$$

$$I \mathbin{\#} e \approx e \tag{U-I}$$

$$U^\dagger \mathbin{\#} U \mathbin{\#} e \approx e \tag{U-†}$$

Figure 7.2: Groupoid axioms

7.2.1 Unitary transformations.

Following Staton (2015), we focus on two main ways to combine unitaries: if $U : \mathcal{U}(q, r)$ and $V : \mathcal{U}(q', r')$, then $U \otimes V : \mathcal{U}(q \otimes q', r \otimes r')$ is the tensor product of U by V, and $U \oplus V : \mathcal{U}(q \oplus q', r \oplus r')$ is the direct sum.

Staton proves that all unitary matrices can be constructed from 1-qubit unitaries with the direct sum and tensor product. In our formulation, where unitaries are indexed by linear types, we must also account for equivalences between these types, such as associativity and distributivity. For example, the controlled-not unitary $\texttt{CNOT} : \mathcal{U}(\texttt{Qubit} \otimes \texttt{Qubit}, \texttt{Qubit} \otimes \texttt{Qubit})$, is exactly equal to $I \oplus X : \mathcal{U}(\texttt{Qubit} \oplus \texttt{Qubit}, \texttt{Qubit} \oplus \texttt{Qubit})$ when we account for the equivalence between $\texttt{Qubit} \otimes \texttt{Qubit}$ and $\texttt{Qubit} \oplus \texttt{Qubit}$.

The equational theory of unitaries is divided into three classes. First, the "structural" axioms, shown in Figure 7.1, characterize the how unitaries interact with syntactic forms of the language. For example, Equation (U-⊗-intro) describes how the tensor product $U_1 \otimes U_2$ distributes over pairs of expressions.

Second, the "groupoid" axioms in Figure 7.2 characterize that unitaries form a groupoid.

The third set of axioms describe how *unitary equivalences*—isomorphisms between the

$$X \mathbin{\#} \texttt{put}\ b \approx \texttt{put}\ \ \neg b \qquad \text{(X-INTRO)}$$

$$\texttt{let}\ !x \coloneqq \texttt{meas}\ \ X \mathbin{\#} e\ \texttt{in}\ e' \approx \texttt{let}\ !y \coloneqq \texttt{meas}\ e\ \texttt{in}\ e'\{\neg y/x\} \qquad \text{(X-ELIM)}$$

$$\texttt{SWAP} \mathbin{\#} (e_1, e_2) \approx (e_2, e_1) \qquad \text{(SWAP-INTRO)}$$

$$\texttt{let}\ (x,y) \coloneqq \texttt{SWAP} \mathbin{\#} e\ \texttt{in}\ e' \approx \texttt{let}\ (y,x) \coloneqq e\ \texttt{in}\ e' \qquad \text{(SWAP-ELIM)}$$

$$\texttt{DISTR} \mathbin{\#} (e, \iota_i e') \approx \iota_i(e, e') \qquad \text{(DISTR-INTRO)}$$

$$\texttt{case}(\texttt{DISTR} \mathbin{\#} e)\ \texttt{of}\ (\iota_0 z_0 \to e_0 \mid \iota_1 z_1 \to e_1) \approx \texttt{case}\ e\ \texttt{of} \begin{cases} (x, \iota_0 y_0) \to e_0\{(x,y_0)/z_0\} \\ (x, \iota_1 y_1) \to e_1\{(x,y_1)/z_1\} \end{cases} \qquad \text{(DISTR-ELIM)}$$

Figure 7.3: Unitary equivalence axioms for $X : \mathcal{U}(\texttt{Qubit}, \texttt{Qubit})$, $\texttt{SWAP} : \mathcal{U}(\sigma \otimes \tau, \tau \otimes \sigma)$, and $\texttt{DISTR} : \mathcal{U}(\sigma \otimes (\tau_1 \oplus \tau_2), (\sigma \otimes \tau_1) \oplus (\sigma \otimes \tau_2))$.

basis sets of linear types—interact with initialization and measurement. Every equivalence $f : \sigma \Longleftrightarrow \tau$ can be lifted to a unitary $\widetilde{f} : \mathcal{U}(\sigma, \tau)$. For example, for the equivalence `swap`, it should be the case that $\widetilde{\texttt{swap}} \mathbin{\#} (e_1, e_2) \approx (e_2, e_1)$. We call (e_1, e_2) the *partial initialization* of the quantum system $\sigma_1 \otimes \sigma_2$, reflected in the fact that `swap` quantifies over all types σ_1 and σ_2. Figure 7.3 shows a further selection of motivating examples.

The idea of partial initialization and its counterpart, partial measurement, provide a concise encapsulation of the behavior of unitary equivalences. Given an equivalence $f : \sigma \Longleftrightarrow \tau$, we have

$$\widetilde{f} \mathbin{\#} \texttt{init}_\sigma\ b \approx \texttt{init}_\tau(f b) \qquad \text{(U-INTRO)}$$

$$\texttt{match}_\tau\ (\widetilde{f} \mathbin{\#} e)\ \texttt{with}\ g \approx \texttt{match}_\sigma\ e\ \texttt{with}\ g \circ f \qquad \text{(U-ELIM)}$$

7.3 Deriving equational rules in homotopy type theory

Our goal in this section is to encode unitary transformations in the structure of linear types in order to minimize the number of axioms needed to recover the equational theory described in the previous section.

For finite types α and β, we write $\texttt{Matrix}(\alpha, \beta)$ for the type of $2^{|\alpha|} \times 2^{|\beta|}$ matrices, and we write $\texttt{UMatrix}(\alpha, \beta)$ for the restriction to unitary matrices. For an equivalence $f : \alpha \simeq \beta$ in the host language, we write $\hat{f} : \texttt{UMatrix}(\alpha, \beta)$ for $\texttt{transport}\ (\texttt{univ}\ f)\ I$.

A groupoid is a category whose morphisms are all invertible. By definition, every unitary U has an inverse $U^\dagger$, so `UMatrix` forms a groupoid. This groupoid structure forms the crux of the encoding of unitaries as higher paths in homotopy type theory.

7.3.1 Groupoid quotient as a higher inductive type

Definition 7.3.1 (Sojakova (2015))**.** *If G is a groupoid with objects α, then the* groupoid quotient *of G, written $\alpha/_1 G$, is a higher inductive 1-type with the following constructors:*

$$\begin{aligned} \texttt{point} &: \alpha \to \alpha/_1 G \\ \texttt{cell} &: G(a,b) \to \texttt{point}\ a = \texttt{point}\ b \\ \texttt{cell_compose} &: \prod_{f,g} \texttt{cell}\ \ g \circ f = \texttt{cell}\ g \circ \texttt{cell}\ f \end{aligned}$$

The fact that $\alpha/_1 G$ is a 1-type means that for any cells $f, g : x = y$ and paths $p, q : f = g$, it is the case that $p = q$.

The induction principle is as follows: for a predicate P on $\alpha/_1 G$, there is a proof $\texttt{ind}_P$ of $\prod x, Px$, provided:

- *For all x, Px is a 1-type;*

- *For all $a : \alpha$, there is a proof $\texttt{P_point}_a$ of $P(\texttt{point}\ a)$;*

- *For all $f : G(a,b)$, there is a proof $\texttt{P_cell}_f$ that $\texttt{transport}_P\ (\texttt{cell}\ f)\ (\texttt{P_point}_a) = \texttt{P_point}_b$; and*

- *For $f : G(a,b)$ and $g : G(b,c)$, the following diagram commutes:*

$$\begin{array}{ccc} \texttt{transport}_P\ (\texttt{cell}\ \ g \circ f)\ (\texttt{P_point}_a) & \overset{\texttt{P_cell}_{g \circ f}}{=\!=\!=\!=} & \texttt{P_point}_c \\ \texttt{ap}\ (\texttt{cell_compose}\ f\ g) \Big\| & & \Big\| \texttt{P_cell}_g \\ \texttt{transport}_P\ (\texttt{cell}\ g \circ \texttt{cell}\ f)\ (\texttt{P_point}_a) & & \texttt{transport}_P\ (\texttt{cell}\ g)\ (\texttt{P_point}_b) \\ & \searrow\!\!\!= \quad\quad =\!\!\!\swarrow\ \texttt{ap}\ \texttt{P_cell}_f & \\ & \texttt{transport}_P\ (\texttt{cell}\ g)\ (\texttt{transport}_P\ (\texttt{cell}\ f)\ (\texttt{P_point}_a)) & \end{array}$$

Furthermore, $\texttt{ind}_P$ must satisfy the following computation laws:

$$\texttt{ind}_P(\texttt{point}\ a) = \texttt{P_point}_a \qquad \textit{and} \qquad \texttt{apd}_{\texttt{ind}_P}(\texttt{cell}\ f) = \texttt{P_cell}_f$$

where, for $f : \prod(x : \alpha), P(x)$ and $p : a = b$ at type α, we have $\texttt{apd}_f(p) : \texttt{transport}_P\ p\ (fa) = fb$.

7.3.2 QType as a groupoid quotient

Define LType to be the groupoid quotient of UMatrix: $\texttt{LType} \equiv \texttt{Type}/_1\texttt{UMatrix}$. Then, for LTypes σ and τ, the type $\sigma = \tau$ corresponds to unitary transformations from σ to τ.

The groupoid quotient ensures that the identity and inverse of paths actually correspond to the appropriate operations on matrices.

Proposition 7.3.2. *Let I : $\texttt{UMatrix}(\alpha, \alpha)$ be the identity matrix on α. Then $\texttt{cell}\ I = 1_{\texttt{point}\ \alpha}$.*

Proof. Since $I = I \circ I$, by the compositionality of `cell` we know that $\texttt{cell}\ I = \texttt{cell}\ I \circ \texttt{cell}\ I$. But for any path $p : x = x$, if $p \circ p = p$ then p must be 1_x. □

Proposition 7.3.3. *Let* $U : \texttt{UMatrix}(\alpha, \beta)$*. Then* $(\texttt{cell}\ U)^{-1} = \texttt{cell}\ U^{\dagger}$*.*

Proof. By the compositionality of `cell`, we have that $\texttt{cell}\ U \circ \texttt{cell}\ U^{\dagger} = \texttt{cell}\ \ U \circ U^{\dagger} = \texttt{cell}\ I = 1$. □

Take $\texttt{Lower}\ \alpha$ to be the type $\texttt{point}\ \alpha$; the operations $\otimes$ and $\oplus$ are defined by quotient induction.

To define $\otimes : \texttt{LType} \to \texttt{LType} \to \texttt{LType}$, we apply a variant of the quotient recursion principle on two variables. It suffices to define how $\otimes$ acts on points and cells, and then show that it is bilinear. First, define $\texttt{point}\ \alpha_1 \otimes \texttt{point}\ \alpha_2 \equiv \texttt{point}*\ \alpha_1 \times \alpha_2$. If $U : \texttt{UMatrix}(\alpha, \alpha')$ and $V : \texttt{UMatrix}(\beta, \beta')$ then we have $\texttt{cell}\ \ U \otimes V : \texttt{point}\ \ \alpha \times \beta = \texttt{point}\ \ \alpha' \times \beta'$. The remaining condition is to show that

$$\texttt{cell}\ \ U_2 \otimes V_2 \circ \texttt{cell}\ \ U_1 \otimes V_1 = \texttt{cell}\ \ (U_2 \circ U_1) \otimes (V_2 \circ V_1).$$

This follows from the fact of linear algebra that $(U_2 \circ U_1) \otimes (V_2 \circ V_1) = (U_2 \otimes V_2) \circ (U_1 \otimes V_1)$.

For $U : q = r$ and $U' : q' = r'$, we lift the tensor product to $U \otimes U' \equiv \texttt{ap}_{\otimes}\ (U, U') : q \otimes q' = r \otimes r'$. The computation principle for $\otimes$ states that $\texttt{cell}\ U \otimes \texttt{cell}\ U' = \texttt{cell}\ \ U \otimes U'$.

A similar argument is used to define $\oplus : \texttt{LType} \to \texttt{LType} \to \texttt{LType}$.

7.3.3 Deriving the groupoid axioms

The fact that unitaries are paths means that the groupoid axioms of Figure 7.2 can be derived for free.

Proposition 7.3.4 (Equation (U-COMPOSE))**.** *Let* $V : q = r$ *and* $U : r = s$*. Then*

$$U \# (V \# e) = (U \circ V) \# e.$$

Proof. By path induction on V. Since $1 \# e \equiv e$ and $U \circ 1 = U$, both sides of the equation are equal to $U \# e$. □

Proposition 7.3.5 (Equation (U-I))**.** *If* $\Delta \vdash_{QNQ} e : q$*, then* $\texttt{cell}\ I \# e = e$*.*

Proof. Follows from Proposition 7.3.2 which states that $\texttt{cell}\ I = 1$. □

Proposition 7.3.6 (Equation (U-†))**.** *If* $U : q = r$ *and* $\Delta \vdash_{QNQ} e : q$ *then* $U^{\dagger} \# U \# e = e$*.*

Proof. Follows from Proposition 7.3.4 and the fact that, as matrices, $U^{\dagger} \circ U = I$. □

7.3.4 Deriving the structural axioms

Similarly, the structural axioms from Figure 7.1 are all trivial by path induction, with one exception:

Proposition 7.3.7. *For* $\Delta \vdash_{QNQ} e : \texttt{Qubit}$ *and* $U : \texttt{Qubit} = \texttt{Qubit}$*, then*

$$\texttt{let}\ !_ \coloneqq \texttt{meas}\ \ U \# e\ \texttt{in}\ e' \approx \texttt{let}\ !_ \coloneqq \texttt{meas}\ e\ \texttt{in}\ e'.$$

Proof. It is not possible to do induction on U here, since its endpoints are both fixed. However, Proposition 7.3.7 follows from the η rule for the unit type, which says, for any two terms $\Delta \vdash_{\text{QNQ}} e_1, e_2 : \texttt{Lower}\ ()$, that $e_1 \sim_\eta e_2$:

$$\begin{aligned} \texttt{let}\ !_ \coloneqq U \# e\ \texttt{in}\ e' &\sim_\eta \texttt{let}\ !_ \coloneqq (\texttt{let}\ !_ \coloneqq U \# e\ \texttt{in}\ \texttt{put}\ ())\ \texttt{in}\ e' \\ &\sim_\eta \texttt{let}\ !_ \coloneqq (\texttt{let}\ !_ \coloneqq e\ \texttt{in}\ \texttt{put}\ ())\ \texttt{in}\ e' \\ &\sim_\eta \texttt{let}\ !_ \coloneqq e\ \texttt{in}\ e' \end{aligned}$$

□

7.4 Equivalence of unitaries

This section addresses the equivalence axioms of Figure 7.3. For instance, consider the "not" unitary X.

Proposition 7.4.1 (Equations (X-INTRO) and (X-ELIM))**.**

$$\texttt{cell}\ X \# \texttt{put}\ b = \texttt{put}(\neg b) \qquad \textit{and} \qquad (\texttt{cell}\ X \# e) >!\ f = e >!\ \lambda b. f(\neg b)$$

The proof of this proposition relies on the following two lemmas, both easily proved by path induction:

Lemma 7.4.2. *For any $f : \alpha = \beta$ and $a : \alpha$:*

$$\texttt{ap}_{\texttt{Lower}}\ f \# \texttt{put}\ a = \texttt{put}(\texttt{coerce}\ f\ a) \quad \textit{and} \quad (\texttt{ap}_{\texttt{Lower}}\ f \# e) >!\ g = e >!\ \lambda x.\ g\,(\texttt{coerce}\ f\ x)\,.$$

Lemma 7.4.3. *If $U : \texttt{UMatrix}(\alpha_1, \alpha_2)$ and $H : \alpha_2 = \alpha_3$, then* $\texttt{cell}(\texttt{transport}\ H\ U) = \texttt{ap}_{\texttt{point}}\ H \circ \texttt{cell}\ U$.

Proof of Proposition 7.4.1. By instantiating Lemma 7.4.2 with $\texttt{univ}\ \neg$, it suffices to check that $\texttt{cell}\ X = \texttt{ap}_{\texttt{point}}(\texttt{univ}\ \neg)$. Observe that the unitary matrix X is equal to the matrix $\texttt{transport}_{\texttt{UMatrix(Bool,-)}}\ (\texttt{univ}\ \neg)\ I$. Then

$$\begin{aligned} \texttt{cell}(\texttt{transport}_{\texttt{UMatrix(Bool,-)}}\ (\texttt{univ}\ \neg)\ I) &= \texttt{ap}_{\texttt{point}}(\texttt{univ}\ \neg) \circ \texttt{cell}\ I && \text{by Lemma 7.4.3} \\ &= \texttt{ap}_{\texttt{point}}(\texttt{univ}\ \neg) && \text{by Proposition 7.3.2.} \end{aligned}$$

□

This technique does not extend to other equivalences such as $\texttt{swap} : \prod \alpha\beta, \alpha \times \beta \to \beta \times \alpha$. Lemma 7.4.2 tells us how $\widetilde{\texttt{swap}}$ behaves on classical states: $\widetilde{\texttt{swap}} \# \texttt{put}\ (a, b) = \texttt{put}\ (b, a)$. But Equation (SWAP-INTRO) is even stronger, stating that for any e_1 and e_2, $\widetilde{\texttt{swap}} \# (e_1, e_2) \sim_q (e_2, e_1)$. Similarly, the elimination form of Lemma 7.4.2 tells us measuring both components of $\widetilde{\texttt{swap}} \# e$, where e is a pair of qubits, is the same as measuring e and then swapping its arguments. However, Equation (SWAP-ELIM) doesn't ask that we measure both qubits, only that we eliminate the pair:

$$\texttt{let}\ (x, y) \coloneqq \widetilde{\texttt{swap}} \# e\ \texttt{in}\ e' \sim_q \texttt{let}\ (x, y) \coloneqq e\ \texttt{in}\ e'.$$

$$[X]^m \equiv mX$$
$$[\texttt{Lower}\ \alpha]^m \equiv \alpha$$
$$[\sigma_1 \otimes \sigma_2]^m \equiv [\sigma_1]^m \times [\sigma_2]^m$$
$$[\sigma_1 \oplus \sigma_2]^m \equiv [\sigma_1]^m + [\sigma_2]^m$$

$$\gamma^m_X(x : \texttt{Var}) \equiv x : \texttt{point}(mX)$$
$$\gamma^m_{\texttt{Lower}\ \alpha}(a : \alpha) \equiv \varnothing$$
$$\gamma^m_{\sigma_1 \otimes \sigma_2}(b_1, b_2) \equiv \gamma^m_{\sigma_1}(b_1), \gamma^m_{\sigma_2}(b_2)$$
$$\gamma^m_{\sigma_1 \oplus \sigma_2}(\texttt{inl}\ b_1) \equiv \gamma_{\sigma_1}(b_1)$$
$$\gamma^m_{\sigma_1 \oplus \sigma_2}(\texttt{inr}\ b_2) \equiv \gamma_{\sigma_2}(b_2)$$

$$\texttt{init}^m_X\ x \equiv x$$
$$\texttt{init}^m_{\texttt{Lower}\ \alpha}\ a \equiv \texttt{put}\ a$$
$$\texttt{init}^m_{\sigma_1 \otimes \sigma_2}\ (b_1, b_2) \equiv \left(\texttt{init}^m_{\sigma_1}\ b_1, \texttt{init}^m_{\sigma_2}\ b_2\right)$$
$$\texttt{init}^m_{\sigma_0 \oplus \sigma_1}(\texttt{inl}\ b_0) \equiv \iota_0(\texttt{init}^m_{\sigma_0}\ b_0)$$
$$\texttt{init}^m_{\sigma_0 \oplus \sigma_1}(\texttt{inr}\ b_1) \equiv \iota_1(\texttt{init}^m_{\sigma_1}\ b_1)$$

$$\begin{aligned}
\texttt{match}_X\ e\ \texttt{with}\ bs \equiv{}& bs\ x\{e/x\} \qquad \text{where } x \text{ is fresh} \\
\texttt{match}_{\texttt{Lower}\ \alpha}\ e\ \texttt{with}\ bs \equiv{}& e >!\ bs \\
\texttt{match}_{\sigma_1 \otimes \sigma_2}\ e\ \texttt{with}\ bs \equiv{}& \texttt{let}\ (x_1, x_2) \coloneqq e\ \texttt{in} \\
& \texttt{match}_{\sigma_1}\ x_1\ \texttt{with}\ \lambda b_1.\ \texttt{match}_{\sigma_2}\ x_2\ \texttt{with}\ \lambda b_2.\ bs(b_1, b_2) \\
\texttt{match}_{\sigma_0 \oplus \sigma_1}\ e\ \texttt{with}\ bs \equiv{}& \texttt{case}\ e\ \texttt{of} \begin{cases} \iota_0 x_0 \to \texttt{match}_{\sigma_0}\ x_0\ \texttt{with}\ \lambda b_0.\ bs(\texttt{inl}\ b_0) \\ \iota_1 x_1 \to \texttt{match}_{\sigma_1}\ x_1\ \texttt{with}\ \lambda b_1.\ bs(\texttt{inr}\ b_1) \end{cases}
\end{aligned}$$

Figure 7.4: Operations on open linear types

We can think of swap's behavior as acting on a state whose structure is only *partially* known, corresponding to the parametric polymorphism of its underlying function swap. Our solution, then, is to define a sort of *partial initialization* and *partial measurement* that generalizes this notion for swap and other polymorphic functions.

7.4.1 Partial initialization and measurement

Consider linear types with the addition of type variables $X : \texttt{TVar}$:

$$\sigma ::= X \mid \texttt{Lower}\ \alpha \mid \sigma_1 \otimes \sigma_2 \mid \sigma_1 \oplus \sigma_2.$$

We call these *open linear types*. Given a map $m : \texttt{TVar} \to \texttt{Type}$, we can define a basis set corresponding to σ, written $[\sigma]^m$, as shown in Figure 7.4.

Let $m : \texttt{TVar} \to \texttt{Type}$ and let $\underline{\texttt{Var}}$ be the constant map $\lambda_.\texttt{Var}$. Then every $b : [\sigma]^{\underline{\texttt{Var}}}$ corresponds to a typing context $\gamma^m_\sigma(b)$, as well as a term using these variables: if $\Delta = \gamma^m_\sigma(b)$ then $\Delta \vdash_{\text{QNQ}} \texttt{init}^m_\sigma\ b : \texttt{point}\ [\sigma]^m$ is called *generalized initialization*, as defined in Figure 7.4.

Open linear types also describe a way to eliminate terms of type $\texttt{point}\ [\sigma]^m$:

$$\frac{\Delta \vdash_{\text{QNQ}} e : \texttt{point}\ [\sigma]^m \qquad bs : \prod_{b:[\sigma]^{\underline{\texttt{Var}}}} \gamma^m_\sigma(b), \Delta' \vdash_{\text{QNQ}} - : q}{\Delta, \Delta' \vdash_{\text{QNQ}} \texttt{match}_\sigma\ e\ \texttt{with}\ bs : q}$$

$$\frac{}{\sigma \approx \sigma}\ \text{REFL} \qquad \frac{\sigma_1 \approx \sigma_2}{\sigma_2 \approx \sigma_1}\ \text{SYMM} \qquad \frac{\sigma_1 \approx \sigma_2 \qquad \sigma_2 \approx \sigma_3}{\sigma_1 \approx \sigma_3}\ \text{TRANS}$$

$$\frac{\sigma_1 \approx \sigma_2 \qquad \tau_1 \approx \tau_2}{\sigma_1 \otimes \tau_1 \approx \sigma_2 \otimes \tau_2}\ \text{CONG}_\otimes \qquad \frac{\sigma_1 \approx \sigma_2 \qquad \tau_1 \approx \tau_2}{\sigma_1 \oplus \tau_1 \approx \sigma_2 \oplus \tau_2}\ \text{CONG}_\oplus$$

$$\sigma_1 \otimes \sigma_2 \approx \sigma_2 \otimes \sigma_1 \qquad (\mathtt{SWAP}_\otimes)$$
$$\sigma_1 \oplus \sigma_2 \approx \sigma_2 \oplus \sigma_1 \qquad (\mathtt{SWAP}_\oplus)$$
$$\sigma_1 \otimes (\sigma_2 \otimes \sigma_3) \approx (\sigma_1 \otimes \sigma_2) \otimes \sigma_3 \qquad (\mathtt{ASSOC}_\otimes)$$
$$\sigma_1 \oplus (\sigma_2 \oplus \sigma_3) \approx (\sigma_1 \oplus \sigma_2) \oplus \sigma_3 \qquad (\mathtt{ASSOC}_\oplus)$$
$$\sigma_1 \otimes (\sigma_2 \oplus \sigma_3) \approx (\sigma_1 \otimes \sigma_2) \oplus (\sigma_1 \otimes \sigma_3) \qquad (\mathtt{DISTR})$$
$$\mathtt{Lower}\ \alpha_1 \otimes \mathtt{Lower}\ \alpha_2 \approx \mathtt{Lower}\ \ \alpha_1 \times \alpha_2 \qquad (\mathtt{Lower}_\otimes)$$
$$\mathtt{Lower}\ \alpha_1 \oplus \mathtt{Lower}\ \alpha_2 \approx \mathtt{Lower}\ \ \alpha_1 + \alpha_2 \qquad (\mathtt{Lower}_\oplus)$$
$$\mathtt{Lower}\ () \otimes \sigma \approx \sigma \qquad (\mathtt{lunit}_\otimes)$$
$$\mathtt{Lower}\ \mathtt{Void} \oplus \sigma \approx \sigma \qquad (\mathtt{lunit}_\oplus)$$
$$\mathtt{Lower}\ \mathtt{Void} \otimes \sigma \approx \mathtt{Lower}\ \mathtt{Void} \qquad (\mathtt{LZERO})$$

Figure 7.5: Inductive presentation of open type equivalence.

The relation $\sigma \Leftrightarrow \tau$ of equivalence of linear types can be extended to open linear types to mean that $[\sigma]^m \cong [\tau]^m$ for every m. For example, the equivalence $X \otimes Y \Leftrightarrow Y \otimes X$ is given by $\lambda m.\ \lambda(x,y).(y,x)$.

Whenever $f : \sigma \Leftrightarrow \tau$ and $b : [\sigma]^{\underline{\mathtt{Var}}}$, the typing contexts $\gamma^m_\tau(fb)$ and $\gamma^m_\sigma(b)$ are identical; the following section develops the proof of this proposition.

7.4.2 Equivalence of typing contexts for equivalent open linear types.

This section develops a proof of the following property of open linear types:

Lemma 7.4.4. *If $f : \sigma \Leftrightarrow \tau$ then for every $b : [\sigma]^{\underline{\mathtt{Var}}}$ there is a path $\gamma^m_\tau(fb) = \gamma^m_\sigma(b)$.*

The proof depends on the observation that open type equivalence $\sigma \Leftrightarrow \tau$ is equivalent to the inductively defined relation $\sigma \approx \tau$ presented in Figure 7.5. It is easy to check that every proof $f : \sigma \approx \tau$ corresponds to an equivalence $\hat{f} : \sigma \Leftrightarrow \tau$, and it is also easy to check that Lemma 7.4.4 follows for inductively-generated equivalences $f : \sigma \approx \tau$.

Lemma 7.4.5. *If $f : \sigma_1 \approx \sigma_2$ and $b : [\sigma]^{\underline{\mathtt{Var}}}$, then $\gamma^m_\tau(\hat{f}b) = \gamma^m_\sigma(b)$.*

Proof. By induction on f. □

To complete the proof of Lemma 7.4.4 we need to show that $\sigma \Leftrightarrow \tau$ implies $\sigma \approx \tau$, which is not trivial. The argument proceeds in two steps:

1. Every open linear type σ corresponds to one in a normal form N_σ such that $\sigma \approx N_\sigma$.

2. If $N_\sigma \Leftrightarrow N_\tau$ then $N_\sigma \approx N_\tau$.

Thus if $\sigma \leftrightsquigarrow \tau$ then by (1) it is the case that $\sigma \approx N_\sigma$ and $\tau \approx N_\tau$. This implies $\sigma \leftrightsquigarrow N_\sigma$ and $\tau \leftrightsquigarrow N_\tau$, and so $N_\sigma \leftrightsquigarrow \sigma \leftrightsquigarrow \tau \leftrightsquigarrow N_\tau$. By (2) we can conclude $\sigma \approx N_\sigma \approx N_\tau \approx \tau$.

Normal linear types N have the following structure:

$$\big(\texttt{Lower}\ \alpha_1 \otimes X_1^1 \otimes X_2^1 \otimes \cdots \otimes X_{n_1}^1\big) \oplus \cdots \oplus \big(\texttt{Lower}\ \alpha_m \otimes X_1^m \otimes \cdots \otimes X_{n_m}^m\big)$$

Proposition 7.4.6. *For every σ, there is a normal linear type N_σ such that $\sigma \approx N_\sigma$.*

Proof. By induction on σ. □

Now, let $f : N \leftrightsquigarrow N'$ where

$$N = \bigoplus_{1 \le i \le n} (\texttt{Lower}\ \alpha_i \otimes Xs_i) \qquad \text{and} \qquad N' = \bigoplus_{1 \le j \le n'} (\texttt{Lower}\ \beta_j \otimes Ys_j)$$

where each Xs_i and Ys_j are $\otimes$-separated sequences of type variables. In particular, that means f has the form

$$f : \Pi(m : \texttt{TVar} \to \texttt{Type}),\ \ \Sigma(i : \mathbb{N}_n), \alpha_i \times m(Xs_i) \cong \Sigma(j : \mathbb{N}_{n'}), \beta_j \times m(Ys_j).$$

Let $\mathcal{R}_f \subseteq \mathcal{P}(\mathbb{N}_n \times \mathbb{N}_{n'})$ be a relation defined as follows:

$$(i,j) \in \mathcal{R}_f \quad \leftrightarrow \quad \Sigma(a : \alpha_i)(b : \beta_j), f_{(\lambda_.())}(i,a) = (j,b)$$

That is, $f_{\lambda_.()}$ has type $\Sigma i, \alpha_i \cong \Sigma j, \beta_j$ and $(i,j) \in \mathcal{R}_f$ says that there is some $a : \alpha_i$ that f maps to some $b : \beta_j$.

Importantly, this implies a broader property by a parametricity argument:

Proposition 7.4.7. *For any m_1 and m_2 of type* $\texttt{TVar} \to \texttt{Type}$*, and for $a : \alpha_i$, $x_1 : m_1(Xs_i)$, and $x_2 : m_2(Xs_j)$,*

$$\pi_1(f_{m_1}(i,a,x_1)) = \pi_1(f_{m_2}(i,a,x_2)) \qquad \textit{and} \qquad \pi_2(f_{m_1}(i,a,x_1)) = \pi_2(f_{m_2}(i,a,x_2)).$$

Proof. Follows from the abstraction theorem (Uemura, 2017). □

Lemma 7.4.8. *If $(i,j) \in \mathcal{R}_f$ then $Xs_i \approx Ys_j$.*

Proof. First, observe that $Xs_i \leftrightsquigarrow Ys_i$. For a fixed m, let $x : m(Xs_i)$. Now, take a to be the element of α_i witnessed by $(i,j) \in \mathcal{R}_f$. Then by Proposition 7.4.7 we know that there exists some (unique) $b : \beta_j$ and $y : m(Ys_j)$ such that $f_m(i,a,x) = (j,b,y)$. The map $x \mapsto y$ is in fact an equivalence.

It is easy to see, then, that $Xs_i \approx Ys_j$, by induction on the sizes of Xs_i and Ys_j. □

Finally, we can prove the main property of this section.

Lemma 7.4.4. The proof is by induction on $n + n'$. We consider five cases: either $\mathcal{R}_f$ is an isomorphism, or it is either not functional, not well-defined on all input, not injective, or not surjective.

1. Suppose $\mathcal{R}_f$ is an isomorphism. Then, observe that whenever $(i, j) \in \mathcal{R}_f$, we have $\alpha_i \cong \beta_j$. This isomorphism is witnessed by the map $a \mapsto \pi_2(f_{\lambda_.()}(i, a))$; since $\mathcal{R}_f$ is injective, we can be sure that this value is in β_j. Then, applying this fact as well as Lemma 7.4.8 we have that

$$\begin{aligned} N &= \bigoplus_{1 \le i \le n} (\texttt{Lower}\ \alpha_i \otimes Xs_i) \\ &= \bigoplus_{1 \le i \le n} \left(\texttt{Lower}\ \beta_{\mathcal{R}_f(i)} \otimes Xs_i\right) \\ &\approx \bigoplus_{1 \le i \le n} \left(\texttt{Lower}\ \beta_{\mathcal{R}_f(i)} \otimes Ys_{\mathcal{R}_f(i)}\right) \approx N' \end{aligned}$$

2. Suppose $\mathcal{R}_f$ is not functional, meaning that there exits some $(i, j_1) \in \mathcal{R}_f$ and $(i, j_2) \in \mathcal{R}_f$ with $j_1 \neq j_2$. We know $Ys_{j_1} \approx Xs_i \approx Ys_{j_2}$ by Lemma 7.4.8, so we have that

$$\begin{aligned} N' &\approx (\texttt{Lower}\ \beta_{j_1} \otimes Ys_{j_1}) \oplus (\texttt{Lower}\ \beta_{j_2} \otimes Ys_{j_2}) \oplus \bigoplus_{j \neq j_1, j_2} (\texttt{Lower}\ \beta_j \otimes Ys_j) \\ &\approx (\texttt{Lower}\ \ \beta_{j_1} + \beta_{j_2} \otimes Ys_{j_1}) \oplus \bigoplus_{j \neq j_1, j_2} (\texttt{Lower}\ \beta_j \otimes Ys_j) \end{aligned}$$

Call this new normal type N''. We still have $N'' \leftrightsquigarrow N$, but the number of clauses of N'' is smaller than that of N, so we can invoke the induction hypothesis to show $N'' \approx N$ and thus by transitivity, $N \approx N'$.

3. If $\mathcal{R}_f$ is not injective, we invoke a similar argument to the case that $\mathcal{R}_f$ is not functional by reducing the number of clauses of N instead of N'.

4. Suppose $\mathcal{R}_f$ is not well-defined on its domain, meaning that there is some i_0 not in the domain of $\mathcal{R}_f$. Observe first that α_{i_0} must be equal to the empty type, `Void`. If not, then there is some $a : \alpha_{i_0}$, and let $j = \pi_1(f(i_0, a))$; we have $(i_0, j) \in \mathcal{R}_f$, a contradiction. Thus

$$\begin{aligned} N &\approx (\texttt{Lower}\ \alpha_{i_0} \otimes Xs_{i_0}) \oplus \bigoplus_{i \neq i_0} (\texttt{Lower}\ \alpha_i \otimes Xs_i) \\ &\approx (\texttt{Lower Void} \otimes Xs_{i_0}) \oplus \bigoplus_{i \neq i_0} (\texttt{Lower}\ \alpha_i \otimes Xs_i) \\ &\approx \texttt{Lower Void} \oplus \bigoplus_{i \neq i_0} (\texttt{Lower}\ \alpha_i \otimes Xs_i) \\ &\approx \bigoplus_{i \neq i_0} (\texttt{Lower}\ \alpha_i \otimes Xs_i) \end{aligned}$$

Again, call this new type N'', then by the induction hypothesis we have that $N'' \approx N'$, and by transitivity $N \approx N'$.

5. If $\mathcal{R}_f$ is not surjective, the proof follows parallel to the case that $\mathcal{R}_f$ is not well-defined.

□

$$\begin{aligned}\texttt{SWAP} \# (x,y) &\equiv \texttt{SWAP} \# \texttt{init}_{X\otimes Y}\ (x,y) \sim_q \texttt{init}_{Y\otimes X}(\texttt{swap}(x,y)) \\ &\equiv \texttt{init}_{Y\otimes X}\ (y,x) \\ &\equiv (y,x)\end{aligned}$$

$$\begin{aligned}&\texttt{let}\ (y,x) \coloneqq \texttt{SWAP} \# e\ \texttt{in}\ e' \equiv \texttt{match}_{Y\otimes X}\ (\texttt{SWAP} \# e)\ \texttt{with}\ \lambda(y,x).\ e' \\ &\sim_q \texttt{match}_{X\otimes Y}\ e\ \texttt{with}\ \lambda(x',y').\ (\lambda(y,x).\ e')(\texttt{swap}(x',y')) \\ &\equiv \texttt{match}_{X\otimes Y}\ e\ \texttt{with}\ \lambda(x,y).\ e' \\ &\equiv \texttt{let}\ (x,y) \coloneqq e\ \texttt{in}\ e'\end{aligned}$$

Figure 7.6: Proofs of Equations SWAP-INTRO and SWAP-ELIM.

7.4.3 Axioms of partial initialization and measurement.

Recall that for $f : \alpha \cong \beta$ we write $\widetilde{f}$ for $\texttt{ap}_{\texttt{point}}(\texttt{univ}(f))$ of type $(\texttt{point}\ \alpha = \texttt{point}\ \beta)$. The following two axioms describe how unitaries of this form interact with partial initialization and measurement, completing the equational theory described in Section 7.2:

Axiom 7.4.9. *Let* $f : \sigma \leftrightsquigarrow \tau$, *and let* $b : [\sigma]^{\underline{\texttt{Var}}}$, $\Delta \vdash_{QNQ} e : \texttt{point}\ [\sigma]^m$, *and* $bs : \prod_{b':[\tau]^{\underline{\texttt{Var}}}} \Delta', \gamma_\tau^m(b') \vdash_{QNQ} - : q$. *Then*

$$\widetilde{f_m} \# \texttt{init}_\sigma^m\ b \quad \sim_q \quad \texttt{init}_\tau^m(f_{\underline{\texttt{Var}}} b) \qquad \text{(U-INTRO)}$$

$$\texttt{match}_\tau\ (\widetilde{f_m} \# e)\ \texttt{with}\ bs \quad \sim_q \quad \texttt{match}_\sigma\ e\ \texttt{with}\ bs \circ f_{\underline{\texttt{Var}}} \qquad \text{(U-ELIM)}$$

Definition 7.4.10. *We define the relation* $e_1 \approx_q e_2$ *on linear expressions as*

$$\approx_q \quad \equiv \quad \sim_\alpha \cup \sim_\beta \cup \sim_\eta \cup \sim_{cc} \cup \sim_q$$

We write $e_1 \approx e_2$ *for equality modulo* $\approx_q$, i.e., *the type* $[e_1]_{\approx_q} = [e_2]_{\approx_q}$.

7.4.4 Instances of equational axioms

Proposition 7.4.11 (Equations (SWAP-INTRO) and (SWAP-ELIM))**.** *Let* SWAP *be the unitary* $\widetilde{\texttt{swap}}$, *where* swap *is the equivalence* $\lambda(x,y).(y,x)$ *of type* $X \otimes Y \leftrightsquigarrow Y \otimes X$. *Then*

$$\texttt{SWAP} \# (e_1,e_2) \approx (e_2,e_1) \qquad \text{(SWAP-INTRO)} \quad \textit{and}$$

$$\texttt{let}\ (y,x) \coloneqq \texttt{SWAP} \# e\ \texttt{in}\ e' \approx \texttt{let}\ (x,y) \coloneqq e\ \texttt{in}\ e' \qquad \text{(SWAP-ELIM)}.$$

Proof. Figure 7.6. □

Proposition 7.4.12. *Let* CNOT *be the unitary* $\widetilde{\texttt{cnot}}$, *where* cnot *is the equivalence*

$$\lambda(b,b').\ (b, \texttt{if}\ b\ \texttt{then}\ \neg b'\ \texttt{else}\ b')$$

of type $\texttt{Bool} \times \texttt{Bool} \cong \texttt{Bool} \times \texttt{Bool}$*. Then:*

$$\texttt{CNOT} \# (\texttt{put}\ b, e) \approx (\texttt{put}\ b, \texttt{if}\ b\ \texttt{then}\ X \# e\ \texttt{else}\ e) \qquad (\texttt{CNOT}\text{-INTRO})$$

$$\texttt{let}\ (!_, y) \coloneqq \texttt{CNOT} \# e\ \texttt{in}\ e' \approx \texttt{let}\ (!b, y') \coloneqq e\ \texttt{in}\ \texttt{if}\ b\ \texttt{then}\ e'\{X \# y'/y\}\ \texttt{else}\ e'\{y'/y\} \qquad (\texttt{CNOT}\text{-ELIM})$$

Proof. Let $\texttt{DISTR} : \texttt{Lower Bool} \otimes X \Leftrightarrow X \oplus X$ be defined by the equivalence

$$\lambda(b, x).\ \texttt{if}\ b\ \texttt{then}\ \texttt{inr}\ x\ \texttt{else}\ \texttt{inl}\ x.$$

From Equation (U-INTRO) we can derive that for any boolean b and expression e, we have

$$\texttt{DISTR} \# (\texttt{put}\ b, e) \sim_q \texttt{if}\ b\ \texttt{then}\ \iota_1\ e\ \texttt{else}\ \iota_0\ e \quad \text{and} \quad \begin{array}{l}\texttt{DISTR}^{-1} \# (\iota_0 e) \sim_q (\texttt{put false}, e)\\ \texttt{DISTR}^{-1} \# (\iota_1 e) \sim_q (\texttt{put true}, e)\end{array}$$

As a matrix, $\texttt{CNOT}$ is equal to $\texttt{DISTR}^{-1} \circ (I \oplus X) \circ \texttt{DISTR}$. Thus,

$$\begin{aligned}
&\texttt{CNOT} \# (\texttt{put}\ b, e)\\
&= \texttt{DISTR}^{-1} \# (I \oplus X) \# \texttt{DISTR} \# (\texttt{put}\ b, e) && \text{(U-COMPOSE)}\\
&\approx \texttt{DISTR}^{-1} \# (I \oplus X) \# \texttt{if}\ b\ \texttt{then}\ \iota_1\ e\ \texttt{else}\ \iota_0\ e && \text{(U-INTRO)}\\
&= \texttt{if}\ b\ \texttt{then}\ \big(\texttt{DISTR}^{-1} \# (I \oplus X) \# \iota_1\ e\big)\ \texttt{else}\ \big(\texttt{DISTR}^{-1} \# (I \oplus X) \# \iota_0\ e\big)\\
&= \texttt{if}\ b\ \texttt{then}\ \big(\texttt{DISTR}^{-1} \# \iota_1(X \# e)\big)\ \texttt{else}\ \big(\texttt{DISTR}^{-1} \# \iota_0(I \# e)\big) && \text{(U-}\oplus\text{-INTRO)}\\
&\approx \texttt{if}\ b\ \texttt{then}\ (\texttt{put true}, X \# e)\ \texttt{else}\ (\texttt{put false}, e) && \text{(U-INTRO)}\\
&= (\texttt{put}\ b, \texttt{if}\ b\ \texttt{then}\ X \# e\ \texttt{else}\ e)
\end{aligned}$$

The proof of Equation ($\texttt{CNOT}$-ELIM) follows similarly. □

7.5 Denotational Semantics

To extend the denotational semantics of Section 6.4 to the language described in this chapter, it suffices to give a semantics for sums.

The semantics of $\Delta \vdash_{\text{QNQ}} \iota_1 e : \sigma_1 \oplus \sigma_2$, where $\Delta \vdash_{\text{QNQ}} e : \sigma_1$, is given by a superoperator from $\texttt{Density}\ \Delta$ to $\texttt{Density}(\sigma_1 \oplus \sigma_2)$:

$$[\![\iota_1 e]\!] \equiv (I \oplus \mathbf{0})^* \circ [\![e]\!]$$

where $\mathbf{0}$ is the zero matrix, and similarly

$$[\![\iota_2 e]\!] \equiv (\mathbf{0} \oplus I)^* \circ [\![e]\!]$$

Next, consider $\Delta, \Delta' \vdash_{\text{QNQ}} \texttt{case}\ e\ \texttt{of}\ (\iota_1 x_1 \to e_1 \mid \iota_2 x_2 \to e_2) : r$ where $\Delta \vdash_{\text{QNQ}} e : q_1 \oplus q_2$, $\Delta', x_1 : q_1 \vdash_{\text{QNQ}} e_1 : r$, and $\Delta', x_2 : q_2 \vdash_{\text{QNQ}} e_2 : r$. Note that for $\rho : [\![r_1 \oplus r_2]\!]$ and $f_1 : [\![r_1]\!] \to [\![r']\!]$ and $f_2 : [\![r_2]\!] \to [\![r']\!]$, there is a density matrix $(f_1 \oplus f_2)(\rho) : [\![r']\!]$ given by $f_1(\iota_1^\dagger \rho \iota_1) + f_2(\iota_2^\dagger \rho \iota_2)$. Notice that $[\![e]\!] \otimes I^*$ is a superoperator from $\texttt{Density}(\Delta \otimes \Delta')$ to $\texttt{Density}((\sigma_1 \oplus \sigma_2) \otimes \Delta')$ Applying the unitary $\texttt{DISTR} : \texttt{UMatrix}((\tau_1 \oplus \tau_2) \otimes \tau, (\tau_1 \otimes \tau) \oplus (\tau_2 \otimes \tau))$ leads to a density

matrix in $\mathtt{Density}((\sigma_1 \otimes \Delta') \oplus (\sigma_2 \otimes \Delta'))$, and from there we can apply $[\![e_1]\!] \oplus [\![e_2]\!]$. Thus:

$$[\![\mathtt{case}\ e\ \mathtt{of}\ (\iota_1 x_1 \to e_1 \mid \iota_2 x_2 \to e_2)]\!] \equiv ([\![e_1]\!] \oplus [\![e_2]\!]) \circ \mathtt{DISTR}^* \circ ([\![e]\!] \otimes I)$$

Theorem 7.5.1 (Soundness of Axiom 7.4.9)**.** *Let $f : \sigma \Longleftrightarrow \tau$ and $b : [\sigma]^{\underline{\mathtt{Var}}}$; then*

$$[\![\widetilde{f_m} \# \mathtt{init}^m_\sigma\ b]\!] = [\![\mathtt{init}^m_\tau(f_{\underline{\mathtt{Var}}} b)]\!]$$

and for $\Delta \vdash_{QNQ} e : \mathtt{point}\ [\sigma]^m$ and $bs : \prod_{bs:[\tau]^{\underline{\mathtt{Var}}}} \gamma^m_\tau(b), \Delta' \vdash_{QNQ} - : q$, then

$$[\![\mathtt{match}_\tau\ (\widetilde{f_m} \# e)\ \mathtt{with}\ bs]\!] = [\![\mathtt{match}_\sigma\ e\ \mathtt{with}\ bs \circ f_{\underline{\mathtt{Var}}}]\!].$$

Proof. Section 7.4.2 introduces an inductively-defined relation $\sigma \approxeq \tau$ that holds exactly when $\sigma \sim \tau$. Thus, it suffices to prove this property with respect to $f : \sigma \approxeq \tau$. First we check the properties with respect to reflexivity, symmetry, transitivity, and congruence.

Reflexivity and symmetry follow directly from Proposition 7.3.5 and Proposition 7.3.4 respectively, and congruence follows from the congruence of density matrices.

For symmetry, we have:

$$\begin{aligned} [\![\widetilde{f^{-1}_m} \# \mathtt{init}^m_\tau\ b]\!] &= [\![\widetilde{f^{-1}_m} \# \mathtt{init}^m_\tau\ f(f^{-1}b)]\!] \\ &= [\![\widetilde{f_m}^{-1} \# \widetilde{f_m} \# \mathtt{init}^m_\sigma\ f^{-1}b]\!] && \text{(Induction Hyp.)} \\ &= [\![\mathtt{init}^m_\sigma\ f^{-1}b]\!] \end{aligned}$$

$$\begin{aligned} [\![\mathtt{match}_\tau\ \widetilde{f^{-1}_m} \# e\ \mathtt{with}\ bs]\!] &= [\![\mathtt{match}_\tau(\widetilde{f^{-1}_m} \# e)\ \mathtt{with}\ bs \circ f^{-1} \circ f]\!] \\ &= [\![\mathtt{match}_\sigma(\widetilde{f_m} \# \widetilde{f^{-1}_m} \# e)\ \mathtt{with}\ bs \circ f^{-1}]\!] && \text{(Induction Hyp.)} \\ &= [\![\mathtt{match}_\sigma\ e\ \mathtt{with}\ bs \circ f^{-1}]\!] \end{aligned}$$

Next we check the behavior of the ten specific unitaries. For initialization we have:

$$\begin{aligned} [\![\mathtt{SWAP}_\otimes \# (e_1, e_2)]\!] &= [\![(e_2, e_1)]\!] \\ [\![\mathtt{SWAP}_\oplus \# \iota_i e]\!] &= [\![\iota_{\neg i} e]\!] \\ [\![\mathtt{ASSOC}_\otimes \# (e_1, (e_2, e_3))]\!] &= [\![((e_1, e_2), e_3)]\!] \\ [\![\mathtt{ASSOC}_\oplus \# \iota_0 e]\!] &= [\![\iota_0(\iota_0 e)]\!] \\ [\![\mathtt{ASSOC}_\oplus \# \iota_1(\iota_0 e)]\!] &= [\![\iota_0(\iota_1 e)]\!] \\ [\![\mathtt{ASSOC}_\oplus \# \iota_1(\iota_1 e)]\!] &= [\![\iota_1 e]\!] \\ [\![\mathtt{DISTR} \# (e_1, \iota_i e_2)]\!] &= [\![\iota_i(e_1, e_2)]\!] \\ [\![\mathtt{Lower}_\otimes \# (\mathtt{put}\ a_1, \mathtt{put}\ a_2)]\!] &= [\![\mathtt{put}\ (a_1, a_2)]\!] \\ [\![\mathtt{Lower}_\oplus \# \iota_i(\mathtt{put}\ a)]\!] &= [\![\mathtt{put}\ \ \mathtt{in}_i a]\!] \\ [\![\mathtt{lunit}_\otimes \# (\mathtt{put}\ (), e)]\!] &= [\![e]\!] \\ [\![\mathtt{lunit}_\oplus(\iota_0(\mathtt{put}\ \ a : \mathtt{Void}))]\!] &= [\![\mathtt{init}^m_\sigma(\mathtt{case}\ a\ \mathtt{of}\ ())]\!] \\ [\![\mathtt{lunit}_\oplus(\iota_1 e)]\!] &= [\![e]\!] \\ [\![\mathtt{LZERO} \# (\mathtt{put}\ \ a : \mathtt{Void}, e)]\!] &= [\![\mathtt{put}\ a]\!] \end{aligned}$$

Notice that the two equations containing values $a : \texttt{Void}$ are vacuously true.

For measurement:

$$\begin{aligned}
[\![\texttt{let } (x_2,x_1) \coloneqq \texttt{SWAP}_\otimes \# e \texttt{ in } e']\!] &= [\![\texttt{let } (x_1,x_2) \coloneqq e \texttt{ in } e']\!] \\
[\![\texttt{case}(\texttt{SWAP}_\oplus \# e) \texttt{ of } (\iota_0 x_0 \to e_0 \mid \iota_1 x_1 \to e_1)]\!] &= [\![\texttt{case } e \texttt{ of } (\iota_0 x_1 \to e_1 \mid \iota_1 x_0 \to e_0)]\!] \\
[\![\texttt{let } ((x_1,x_2),x_3) \coloneqq (\texttt{ASSOC}_\otimes \# e) \texttt{ in } e']\!] &= [\![\texttt{let } (x_1,(x_2,x_3)) \coloneqq e \texttt{ in } e']\!] \\
[\![\texttt{match}_{X_1 \oplus (X_2 \oplus X_3)}(\texttt{ASSOC}_\oplus \# e) \texttt{ with } bs]\!] &= [\![\texttt{match}_{(X_1 \oplus X_2) \oplus X_3}\ e \texttt{ with } bs \circ \texttt{ASSOC}_\oplus]\!] \\
[\![\texttt{let } !(a,b) \coloneqq (\texttt{Lower}_\otimes \# e) \texttt{ in } e']\!] &= [\![\texttt{let } (!a,!b) \coloneqq e \texttt{ in } e']\!]
\end{aligned}$$

$$\begin{aligned}
&[\![\texttt{case}(\texttt{DISTR} \# e) \texttt{ of } (\iota_0(x,y_0) \to e_0 \mid \iota_1(x,y_1) \to e_1)]\!] \\
&\qquad = [\![\texttt{let } (x,y) \coloneqq e \texttt{ in case } y \texttt{ of } (\iota_0 y_0 \to e_0 \mid \iota_1 y_1 \to e_1)]\!] \\
&[\![\texttt{Lower}_\oplus \# e >!\ f]\!] \\
&\qquad = [\![\texttt{case } e \texttt{ of } (\iota_0 x_0 \to x_0 >!\ f \circ \texttt{inl} \mid \iota_1 x_1 \to x_1 >!\ f \circ \texttt{inr})]\!]
\end{aligned}$$

$$\begin{aligned}
[\![\texttt{let } (!(),x) \coloneqq \texttt{lunit}_\otimes \# e \texttt{ in } e']\!] &= [\![\texttt{let } x \coloneqq e \texttt{ in } e']\!] \\
[\![\texttt{lunit}_\oplus e]\!] &= [\![\texttt{case } e \texttt{ of } (\iota_0 !(a : \texttt{Void}) \to _ \mid \iota_1 x \to x)]\!] \\
[\![\texttt{LZERO} \# e >!\ f]\!] &= [\![\texttt{let } (!a, _) \coloneqq e \texttt{ in } f a]\!]
\end{aligned}$$

□

Theorem 7.5.2 (Soundness of $\texttt{LUnit}$). *$\Delta \vdash_{QNQ} e, e' : \texttt{LUnit}$, we have $[\![e]\!] = [\![e']\!]$.*

Proof. The type $\texttt{Density(LUnit)}$ is the set of 1×1 density matrices, so it has only one element–the identity matrix. □

Theorem 7.5.3 (Soundness). *If $e_1 \approx e_2$ in the equational theory, then $[\![e_1]\!] = [\![e_2]\!]$.*

7.6 Discussion

In this section we discuss a few of the non-essential design decisions made in this work.

Axiom schemes. Our equational theory prioritizes equations based on the structure of the language, such as β, η, and commuting conversion rules. Such rules do not depend on any quantum-specific principles, and their meta-theories are well-understood. In addition, we prioritize collecting many axioms into a single axiom scheme, as we do for commuting conversions and the equational axioms U-INTRO and U-ELIM. This approach gives concise axioms that highlight the important structure, but requires more overhead to express.

Our axiom schemes are also somewhat redundant—for example, we proved the equations for the "not" unitary X in Proposition 7.4.1, but they are a consequence of the U-INTRO and U-ELIM axioms.

Unitaries. In this chapter we did not axiomatize unitary transformations, in order to focus on the relationship between quantum and non-quantum data. However, axiomatizations based on universal (or even non-universal) sets of unitaries, such as those by Matsumoto and Amano (2008) or Amy *et al.* (2018), could be incorporated with a higher-inductive type (HIT). As a first approximation, we could define $\texttt{QType}$ as a HIT that axiomatizes only the behavior of the Hadamard gate H, with the following constructors: a type $\texttt{Qubit} : \texttt{QType}$; a path $\texttt{H} : (\texttt{Qubit} = \texttt{Qubit})$, and a higher path expressing that $H^\dagger = H$. Since unitaries are still encoded in the path type of quantum types, the aspects of the equational

theory we derived by path induction would still hold, and it would allow finer control over the ways by which unitaries approximate each other. On the other hand, working with higher inductive types with many constructors can quickly become unwieldy.

7.7 Conclusion

In this chapter we have explored the QNQ calculus and its equational theory through the lens of homotopy type theory. By using HoTT-specific features like higher inductive types and univalence, we were able to simplify the theory and identify the unitary equivalence axioms as the core of the equational theory.

In the next chapter we take a step back to look at QNQ more explicitly as a quantum circuit language. We also implement this circuit language in Coq, and examine techniques to embed a linear DSL in Coq.

CHAPTER 8

$\mathcal{Q}$wire: Quantum circuits in Coq

Many state-of-the-art quantum programming languages are actually *circuit description languages*, where quantum programs are compiled to circuits, intended to be executed on a quantum machine. This chapter describes $\mathcal{Q}$WIRE, a variation of the quantum/non-quantum lambda calculus described in the previous chapters, tailored specifically for describing quantum circuits. $\mathcal{Q}$WIRE is implemented in the Coq proof assistant,[14] and ongoing work by Rand, Paykin, and Zdancewic (2017) explores how to use the implementation to formally verify the correctness of quantum circuits.

$\mathcal{Q}$WIRE differs from the QNQ calculus described in Chapter 6 in several key ways.

- While QNQ can apply arbitrary unitary transformations to qubits, $\mathcal{Q}$WIRE can apply both unitary and non-unitary gates. In particular, initialization and measurement will be implemented as gates instead of as primitive operations, so that gate application is the only domain-specific component of $\mathcal{Q}$WIRE. The set of $\mathcal{Q}$WIRE gates is shown in Figure 8.1.

- At its core, a $\mathcal{Q}$WIRE circuit is just a sequence of gate applications, with the addition of >! bindings to interact with the host language. In the QRAM model of quantum computing, a >! operation pauses execution of the circuit on the quantum computer, sends measurement results to the classical computer, and resumes execution of the circuit once the classical computer has processed the results.

- While QNQ can embed arbitrary host-language data in the linear language using the `put` constructor, such data has no place on a quantum circuit. $\mathcal{Q}$WIRE does not include the `put` constructor, and the only way to construct values of type `Lower` α is by applying initialization or measurement gates.

- $\mathcal{Q}$WIRE is implemented in Coq, which has powerful automation techniques that we harness to check linear typing judgments. For the sake of this chapter, we assume the reader is familiar with the basic syntax of Coq and interactive theorem proving; for more background we refer the reader to Pierce *et al.* (2016).

- The rest of this dissertation uses intrinsic typing judgments for expressions, where the type of expressions is `LExp` Δ τ. In comparison, the Coq implementation uses an extrinsic typing judgment `Typed_Circ` Δ τ `c` to characterize that $\Delta \vdash c : \tau$ for a weakly-typed circuit c. The extrinsic judgment makes it easier to develop the meta-theory and semantics of $\mathcal{Q}$WIRE in Coq because of the way Coq handles dependent pattern matching.

[14] `https://github.com/jpaykin/QWIRE`

```
Inductive Unitary : WType → Set :=
| H              : Unitary Qubit                      (* Hadamard gate *)
| X              : Unitary Qubit                      (* Not gate *)
| …                                         (* Other single qubit unitaries *)
| ctrl {σ}       : Unitary σ → Unitary (Qubit ⊗ σ) (* quantum control *)
| bit-ctrl {σ}   : Unitary σ → Unitary (Bit ⊗ σ)    (* classical control *)
| transpose {σ} : Unitary σ → Unitary σ             (* conjugate transpose *).

Inductive Gate : WType → WType → Set :=
| unitary {σ} : Unitary σ → Gate σ σ
| init         : Bool → Gate LUnit Qubit (* qubit initialization *)
| new          : Bool → Gate LUnit Bit    (* bit initialization *)
| meas         : Gate Qubit Bit           (* measurement *)
| discard      : Gate Bit LUnit.          (* discard a classical bit *)

Coercion U : Unitary ↣ Gate.
```

Figure 8.1: Unitary and non-unitary gates in $\mathcal{Q}$WIRE. Different gate sets could have been chosen, for example by picking a different universal set of unitary gates, or by allowing arbitrary circuits to be frozen as gates, which is a feature allowed by many practical circuit languages. Rennela and Staton (2018) propose some extensions to $\mathcal{Q}$WIRE that expand the gate set to add sums and recursive data types.

The $\mathcal{Q}$WIRE project illustrates two important points about the LNL embedded programming model. It serves as a case study for implementing the LNL framework in Coq, and it shows how variations of the basic model—in this case, eliminating `put` and allowing only first-order functions—are still expressive enough and practical enough for domain-specific applications.

In the remainder of this chapter, we describe the implementation of $\mathcal{Q}$WIRE in Coq. Section 8.1 presents the syntax and typing rules of $\mathcal{Q}$WIRE, and Section 8.2 describes the details of linear type checking in Coq. Section 8.3 formally establishes the relationship between $\mathcal{Q}$WIRE and QNQ, showing how to define composition and function application on top of $\mathcal{Q}$WIRE. Finally, Section 8.4 discusses some design decisions and related work.

8.1 The $\mathcal{Q}$wire circuit language

In this section we describe the $\mathcal{Q}$WIRE circuit language.

Linear types are defined as an inductive data type.

```
Inductive LType := Qubit | Lower α | LUnit | Tensor : LType → LType → LType.
Notation "σ1 ⊗ σ2" := (Tensor σ1 σ2).
Definition Bit := Lower Bool.
```

We use notations like $\sigma_1 \otimes \sigma_2$ liberally in the development to make code more legible. In addition, we write `Bit` to refer to the type `Lower Bool`. Note that in the Coq development online, we refer to linear types as *wire types*, and include only `Bit` instead of `Lower` α. To keep types consistent between $\mathcal{Q}$WIRE and QNQ, we use the more general type `Lower` α in this presentation.

8.1.1 Patterns, and extrinsic typing judgments

A $\mathcal{Q}$WIRE circuit is a sequence of gates applied to patterns of wire variables, where a pattern is a nested tuple of bit- and qubit-valued variables. The type of patterns of type σ is written `Pat` σ.

```
Inductive Pat : LType → Set :=
| unit         : Pat LUnit
| qubit        : Var → Pat Qubit
| bit          : Var → Pat Bit
| pair {σ1 σ2} : Pat σ1 → Pat σ2 → Pat (σ1 ⊗ σ2).
```

Notice that the type of patterns is *not* indexed by a typing context. Instead, we check the linearity of typing contexts using an extrinsic predicate, written $\Delta \Rrightarrow_{\mathcal{Q}} \mathtt{p} : \sigma$.

```
Inductive Types_Pat : Ctx → ∀ σ, Pat σ → Set :=
| types_unit        : ∅ ⇛Q unit : LUnit
| types_qubit {x Δ} : Δ = singleton x Qubit → Δ ⇛Q qubit x : Qubit
| types_bit   {x Δ} : Δ = singleton x Bit   → Δ ⇛Q bit x : Bit
| types_pair  {Δ1 Δ2 Δ σ1 σ2} p1 p2 :
              Δ == Δ1 • Δ2 →
              Δ1 ⇛Q p1 : σ1 → Δ2 ⇛Q p2 : σ2 →
              Δ ⊢Q pair p1 p2 : σ1 ⊗ σ2
where "Δ ⇛Q p : σ" := (Types_Pat Δ σ p).
```

In Section 8.2 we describe the implementation of typing contexts Δ : `Ctx`. For now, it suffices to know that `singleton x` σ is the singleton typing context $x : \sigma$, and the judgment Δ `==` $\Delta_1 \bullet \Delta_2$ encodes the fact that Δ_1 is disjoint from Δ_2 and $\Delta = \Delta_1, \Delta_2$.

8.1.2 Circuits and boxes

A circuit in $\mathcal{Q}$WIRE is either an output pattern of wires, a gate application, or a >! binding, which here we call *dynamic lifting*, following the quantum programming literature (Green *et al.*, 2013b). We use higher-order abstract syntax for variable binding in gate applications, but using Coq notations we are able to write `gate_` $\mathtt{p}_2 \leftarrow$ `g #` $\mathtt{p}_1$`; c'` for `gate g` $\mathtt{p}_1$ (`fun` $\mathtt{p}_2 \Rightarrow$ `c'`).

```
Inductive Circuit (σ : LType) :=
| output       : Pat σ → Circuit σ
| gate {σ1 σ2} : Gate σ1 σ2 →  Pat σ1 → (Pat σ2 → Circuit σ) → Circuit σ
| lift         : Pat Bit → (Bool → Circuit σ) → Circuit σ.
Notation "gate_ p2 ← g # p1 ; c'" := (gate g p1 (fun p2 ⇒ c')).
Notation "p >! f" := (lift p f).
```

The only patterns of type `Lower` α are for `Bit` = `Lower Bool`, so for simplicity we restrict the `lift` constructor to bits.

The linear typing judgment for circuits is written $\Delta \vdash_{\mathcal{Q}} \mathtt{C} : \sigma$.

```
Inductive Types_Circuit : Ctx → ∀ σ, Circuit σ → Set :=
| types_output {Δ σ} {p : Pat σ} : Δ ⇛Q p : σ → Δ ⊢Q output p : σ
| types_gate {Δ Δ1 Δ1' σ1 σ2 σ}
             {f : Pat σ2 → Circuit σ} {p1 : Pat σ1} {g : Gate σ1 σ2} :
              Δ1 ⇛Q p1 : σ1    →
              Δ ⊢Q f :Fun      →
              Δ1' == Δ1 • Δ   →
```

```
              Δ₁' ⊢Q gate g p₁ f : σ
| types_lift {Δ₁ Δ₂ Δ σ} {p : Pat Bit} {f : bool → Circuit σ} :
              Δ₁ ⇒Q p : Bit        →
              (∀ b, Δ₂ ⊢Q f b : σ) →
              Δ == Δ₁ ∙ Δ₂         →
              Δ ⊢Q lift p f : σ
where "Δ ⊢Q C : σ" := (Types_Circuit Δ σ C)
where "Δ ⊢Q f :Fun"  := (forall Δ₀ Δ₀' p₀, Δ₀' == Δ₀ ∙ Δ →
                                            Δ₀ ⇒Q p₀ : _ →
                                            Δ₀' ⊢Q f p₀ : _).
```

If $\texttt{f} : \texttt{Pat}\ \sigma \to \texttt{Circuit}\ \tau$, then the notation $\Delta \vdash_Q \texttt{f :Fun}$ means that f is a well-typed function: given a pattern $\Delta_0 \Rightarrow$_q $\texttt{p} : \sigma$ such that $\Delta \bot \Delta_0$, then $\Delta,\Delta_0 \vdash_Q \texttt{f p} : \tau$.

Boxed circuits in $\mathcal{Q}$WIRE correspond to the first-order functions in QNQ, and are represented as a function from patterns to circuits.

```
Inductive Box σ₁ σ₂ := box : (Pat σ₁ → Circuit σ₂) → Box σ₁ σ₂.
Notation "box_ p ⇒ c" := (box (fun p ⇒ c)).
```

For $\texttt{b} \equiv \texttt{box f} : \texttt{Box}\ \sigma_1\ \sigma_2$, we write `unbox b p` for `f p`.

We say a box is well-typed if it uses only the variables introduced by its function, and the underlying circuit is also well-typed.

```
Definition Typed_Box {σ₁ σ₂ : LType} (b : Box σ₁ σ₂) :=
  ∀ Δ (p : Pat σ₁), Δ ⇒Q p : σ₁ → Δ ⊢Q unbox b p : σ₂
```

8.2 Linear type checking in Coq

One of the hardest parts of implementing a linear embedded language is checking the linearity constraints. As we saw in Chapter 4, linear type checking is multi-directional, as some typing contexts must be checked and others must be inferred in any particular typing derivation.

For example, consider type checking the judgment $\Delta_1' \vdash_Q \texttt{gate}_{_}\ p_2 \leftarrow g\ \#\ p_1;\ c : \sigma$. Although we know the value of the typing context Δ_1', in order to apply the `types_gate` constructor, we have to guess the values of the typing contexts Δ and Δ_1. From the value of p_1, we will be able to infer the value of Δ_1; we can then use the judgment $\Delta_1' == \Delta_1 \bullet \Delta$ to infer the value of Δ.

In Coq we can facilitate this sort of bidirectional type checking using `EVars`, unknown variables that will be filled in over the course of a proof with concrete values.

In order to solve goals of linear type constraints, Robert Rand and I developed a standalone library for automatically solving linearity constraints, available on GitHub.[15] The library contains a theory of linear typing contexts and automation techniques for discharging goals of linearity constraints in Coq.

8.2.1 Partial commutative monoids

The library is based on the theory of typing contexts as a *partial commutative monoids* (PCM) (Wehrung, 2017). A PCM is a commutative monoid with a zero *undefined* element satisfying certain laws.

[15] `https://github.com/jpaykin/LinearTypingContexts`

Definition 8.2.1. *A* commutative monoid $(\alpha, \top, \bullet)$ *consists of a type* α *with an element* $\top : \alpha$ *and a binary operation* $\bullet : \alpha \to \alpha \to \alpha$ *satisfying:*

$$\top \bullet a = a \qquad (\bullet\text{-}\top)$$

$$a \bullet (b \bullet c) = (a \bullet b) \bullet c \qquad (\bullet\text{-ASSOC})$$

$$a \bullet b = b \bullet a \qquad (\bullet\text{-COMMUTE})$$

A partial *commutative monoid* $(\alpha, \top, \bot, \bullet)$ *consists of a commutative monoid* $(\alpha, \top, \bullet)$*, along with an element* $\bot : \alpha$ *representing undefined values, such that:*

$$a \bullet \bot = \bot \qquad (\bullet\text{-}\bot)$$

Example 8.2.2. *Let* `IdxMap` *be the type* `list (option WType)`*, representing a partial map from indices* i *in the list* Δ *to wire types* $\Delta[i]$*. We define the partial merge operation* `mergeIdxMap` *on* `IdxMap` *as follows:*

```
Fixpoint mergeIdxMap (i₁ i₂ : IdxMap) : option IdxMap :=
  match i₁, i₂ with
  | [],              _              ⇒ Some i₂
  | _,               []             ⇒ Some i₁
  | None    :: i₁', None :: i₂'     ⇒ consOption None (mergeIdxMap i₁' i₂')
  | Some σ₁ :: i₁', None :: i₂'     ⇒ consOption (Some σ₁) (mergeIdxMap i₁' i₂')
  | None    :: i₁', Some σ₂ :: i₂' ⇒ consOption (Some σ₂) (mergeIdxMap i₁' i₂')
  | Some _  :: _,   Some _ :: _     ⇒ Nothing
  end.
```

where `consOption a ls` *maps* `(cons a)` *over* `ls : option (list` α`)`.

We write `IdxCtx` *for* `option IdxMap`*, which forms a partial commutative monoid with the following components:*

```
⊤       := Some []
⊥       := Nothing
Δ₁ • Δ₂ := match Δ₁, Δ₂ with
           | Some i₁, Some i₂ ⇒ mergeIdxMap i₁ i₂
           | _, _ ⇒ Nothing
           end
```

The structure of a PCM tells us how to solve goals of the form $a = b$, where $a, b : \alpha$ are permutations and associations of the same set of elements. The Coq tactic `monoid` solves goals of this form, such as:

```
Lemma PCM_abc : ∀ a b c, a • b • c = c • a • ⊤ • b.
Proof. intros. monoid. Qed.
```

Because we will sometimes be reasoning about unknown contexts represented as `EVar`s, the `monoid` tactic can solve goals with at most one `EVar`. Here, the tactic `eexists` introduces an `EVar`, which the `monoid` tactic fills in with b.

```
Lemma PCM_evar : ∀ a b, exists unknown, unknown • a = a • b.
Proof. intros. eexists. monoid. Qed.
```

The `monoid` tactic is based on Chlipala's reflection technique (Chlipala, 2013, Chapter 15), and we have extended it to partial commutative monoids that may also contain `EVars`.

8.2.2 Validity

We say that a typing context is *valid* if it is not equal to the undefined element $\bot$. Instead of reasoning directly about the negative assertion $\Delta \neq \bot$, we reformulate this judgment as a positive assertion `is_valid` Δ.

```
Definition is_valid Δ := Δ <> ⊥.
```

Validity satisfies the following properties:

- $\top$ is valid; and
- $\Delta_1 \bullet \Delta_2 \bullet \Delta_3$ is valid if and only if $(\Delta_1 \bullet \Delta_2)$, $(\Delta_1 \bullet \Delta_3)$, and $(\Delta_2 \bullet \Delta_3)$ are all valid.

Notice that the merge of Δ_1 and Δ_2 should always be valid provided their domains are disjoint. To capture this fact, we introduce two additional properties that describe the validity of singleton typing contexts of the form `singleton x` τ.

- every singleton context `singleton x` σ is valid; and
- `singleton x` σ $\bullet$ `singleton y` τ is valid if and only if $x \neq y$.

We say that a kind of typing contexts is well-formed if it satisfies these four rules.

Example 8.2.3. *The kind* `IdxCtx ≡ option IdxMap` *is well-formed, where a singleton context is defined as*

```
singleton 0     σ ≡ Some [Some σ]
singleton (n+1) σ ≡ consOption None (singleton n σ)
```

The tactic `validate` solves goals of the form `is_valid` Δ. Unlike `monoid`, `validate` cannot handle goals that contain an `EVar`, which might be filled in with the undefined element $\bot$.

```
Lemma valid_test : ∀ x y z a b c, x <> y → y <> z → x <> z →
      is_valid (singleton x a • singleton y b • singleton z c).
Proof. validate. Qed.
```

In the typing rules, we often combine judgments of the form $\Delta = \Delta_1 \bullet \Delta_2$ and `is_valid` Δ, so we introduce notation for such goals.

```
Notation "Δ == Δ₁ • Δ₂" := (is_valid Δ ∧ Δ = Δ₁ • Δ₂) (at level 75).
```

The tactic `solve_ctx` solves goals of this form.

```
Ltac solve_ctx := split; [validate | monoid].
```

8.2.3 The `type_check` tactic.

The `type_check` tactic uses `solve_ctx` and other tactics to discharge goals of the form $\Delta \Rightarrow_Q$ `p` $: \sigma$, $\Delta \vdash_Q$ `c` $: \tau$, $\Delta \vdash_Q$ `f` `:Fun`, and `Typed_Box b`. It does this by repeatedly calling `econstructor`, which automatically applies a constructor from the appropriate inductively-defined data type, introducing `EVars` for the values of unknown typing contexts. The `type_check` tactic will not apply induction or other high-level proof techniques, but if a circuit is concrete, it can discharge the goal automatically in most cases.

8.3 Surface language

In this section we show an equivalence between QNQ expressions and $\mathcal{Q}$WIRE circuits. The translation from $\mathcal{Q}$WIRE to QNQ endows $\mathcal{Q}$WIRE programs with a denotational semantics, and the translation from QNQ to $\mathcal{Q}$WIRE provides a surface language to $\mathcal{Q}$WIRE that is compositional and easy to use.

8.3.1 $\mathcal{Q}$wire to QNQ translation

To reason about the correctness of $\mathcal{Q}$WIRE, we start by translating $\mathcal{Q}$WIRE patterns $\Delta \Rightarrow_{\mathrm{Q}} p : \sigma$ into QNQ expressions $\Delta \vdash_{\mathrm{QNQ}} p^+ : \sigma$ as follows:

$$\begin{aligned} x^+ &\equiv x \\ ()^+ &\equiv () \\ (p_1, p_2)^+ &\equiv (p_1^+, p_2^+) \end{aligned}$$

Since QNQ only has unitary gates, classical gates in $\mathcal{Q}$WIRE will be translated to QNQ boxes.

$$\begin{aligned} (\texttt{unitary}\ U)^+ &\equiv \texttt{box}\ x \Rightarrow U \# x \\ (\texttt{init}\ b)^+ &\equiv \texttt{box}\ () \Rightarrow \texttt{init}\ b \\ (\texttt{meas})^+ &\equiv \texttt{box}\ x \Rightarrow \texttt{meas}\ x \\ (\texttt{discard})^+ &\equiv \texttt{box}\ x \Rightarrow x >!\ \lambda_.() \end{aligned}$$

For convenience, we define a generalized let binding $\texttt{let}\ p \coloneqq e\ \texttt{in}\ e'$ on QNQ expressions by induction on p such that

$$\texttt{let}\ (p_1, p_2) \coloneqq e\ \texttt{in}\ e' \qquad \equiv \qquad \texttt{let}\ (x_1, x_2) \coloneqq e\ \texttt{in}\ \texttt{let}\ p_1 \coloneqq x_1\ \texttt{in}\ \texttt{let}\ p_2 \coloneqq x_2\ \texttt{in}\ e'.$$

$\mathcal{Q}$WIRE circuits $\Delta \vdash_{\mathrm{Q}} c : \sigma$ and boxes $b : \texttt{Box}\ \sigma\ \tau$ are translated to QNQ expressions $\Delta \vdash_{\mathrm{QNQ}} c^+ : \sigma$ and boxes $b^+ : \texttt{Box}\ \sigma\ \tau$ respectively.

$$\begin{aligned} (\texttt{output}\ p)^+ &\equiv p^+ \\ (\texttt{gate}_\ p_2 \leftarrow g \# p_1;\ c')^+ &\equiv \texttt{let}\ p_2 \coloneqq g^+ \$ p_1^+\ \texttt{in}\ (c')^+ \\ (p >!\ f)^+ &\equiv p^+ >!\ \lambda a.(f a)^+ \\ (\texttt{box}_\ p \Rightarrow c)^+ &\equiv \texttt{box}\ x \Rightarrow \texttt{let}\ p \coloneqq x\ \texttt{in}\ c^+ \end{aligned}$$

Formally, composition is defined in Coq as follows:

8.3.2 QNQ to $\mathcal{Q}$wire translation

In the other direction, every QNQ expression $\Delta \vdash_{\mathrm{QNQ}} e : \sigma$ can encoded as a $\mathcal{Q}$WIRE circuit $\Delta \vdash_{\mathrm{Q}} e^- : \sigma$.

We start by defining the composition of two $\mathcal{Q}$WIRE circuits. Given $\Delta \vdash_{\mathrm{Q}} c : \sigma$ and

$\Delta', x : \sigma \vdash_{Q} c' : \tau$, we define $\Delta, \Delta' \vdash_{Q}$ $\texttt{let}_{-}\ x \leftarrow c;\ c' : \tau$ as follows:

$$\texttt{let}_{-}\ x \leftarrow \texttt{output}\ p;\ c' \equiv c'\{p/x\}$$
$$\texttt{let}_{-}\ x \leftarrow (\texttt{gate}_{-}\ p_2 \leftarrow g \mathbin{\#} p_1;\ c_0);\ c' \equiv \texttt{gate}_{-}\ p_2 \leftarrow g \mathbin{\#} p_1;\ \texttt{let}_{-}\ x \leftarrow c_0;\ c'$$
$$\texttt{let}_{-}\ x \leftarrow (p >!\ f);\ c' \equiv p >!\ \lambda a.\texttt{let}_{-}\ x \leftarrow f a;\ c'$$

Formally, composition is defined as a coq function.

```
Fixpoint compose {σ τ} (c : Circuit σ) (f : Pat σ → Circuit τ) : Circuit τ :=
  match c with
  | output p     ⇒ f p
  | gate g p c'  ⇒ gate g p (fun p' ⇒ compose (c' p') f)
  | lift p c'    ⇒ lift p (fun b ⇒ compose (c' b) f)
  end.
Notation "let_ p ← c ; c'" := (compose c (fun p ⇒ c')).
```

We can verify that composition is well-typed with the help of the `solve_ctx` tactic.

```
Lemma compose_typing : forall Δ1 W (c : Circuit W),
        Δ1 ⊢ c :Circ →
        forall Δ1' Δ W' (f : Pat W → Circuit W'),
        Δ ⊢ f :Fun →
        Δ1' ==  Δ1 • Δ →
        Δ1' ⊢ compose c f :Circ.
Proof.
  intros Δ1 W c types_c.
  induction types_c as [ Δ0 Δ0' p W
                       | Δ' Δ0 Δ0' w1 w2 w h p1 g
                       | Δ1 Δ2 Δ0 w p h ];
    intros Δ1' Δ W' f types_f pf_merge.
  - (* c = output p *)
    subst. eapply types_f; eauto.
  - (* c = gate g p c' *)
    simpl. eapply types_gate; eauto; try solve_ctx. (* constructor *)
    intros. eapply H; try eauto; solve_ctx; auto.   (* induction hypothesis *)
  - eapply types_lift_bit; eauto; solve_ctx.        (* constructor *)
    intros. eapply H; eauto; solve_ctx.             (* induction hypothesis *)
Qed.
```

Figure 8.2 shows the negative translation on boxes and expressions, excluding those of the form $\texttt{put}\ a$. Figure 8.3 shows how we can formally implement this translation by defining syntactic sugar for QNQ syntax in $\mathcal{Q}$WIRE.

8.3.3 Soundness

The QNQ translations are sound if they preserve equivalence in QNQ. We start with the correctness of composition.

Lemma 8.3.1. *If $\Delta \vdash_{Q} c : \sigma$ and $\Delta', x : \sigma \vdash_{Q} c' : \tau$, then*

$$(\texttt{let}_{-}\ x \leftarrow c;\ c')^{+} \sim \texttt{let}\ x \coloneqq c^{+}\ \texttt{in}\ (c')^{+}.$$

$$
\begin{aligned}
x^{-} &\equiv \texttt{output}\ x \\
(\texttt{let}\ x \coloneqq e\ \texttt{in}\ e')^{-} &\equiv \texttt{let_}\ x \leftarrow e^{-};\ (e')^{-} \\
()^{-} &\equiv \texttt{output}\ () \\
(\texttt{let}\ () \coloneqq e\ \texttt{in}\ e')^{-} &\equiv \texttt{let_}\ () \leftarrow e^{-};\ (e')^{-} \\
(e_1, e_2)^{-} &\equiv \texttt{let_}\ x_1 \leftarrow e_1^{-};\ \texttt{let_}\ x_2 \leftarrow e_2^{-};\ \texttt{output}\ (x_1, x_2) \\
(\texttt{let}\ (x_1, x_2) \coloneqq e\ \texttt{in}\ e')^{-} &\equiv \texttt{let_}\ (x_1, x_2) \leftarrow e^{-};\ (e')^{-} \\
(U \mathbin{\#} e)^{-} &\equiv \texttt{let_}\ x \leftarrow e^{-};\ \texttt{gate_}\ y \leftarrow U \mathbin{\#} x;\ \texttt{output}\ y \\
(\texttt{init}\ b)^{-} &\equiv \texttt{gate_}\ x \leftarrow \texttt{init}\ b \mathbin{\#} ();\ \texttt{output}\ x \\
(\texttt{meas}\ e)^{-} &\equiv \texttt{let_}\ x \leftarrow e^{-};\ \texttt{gate_}\ y \leftarrow \texttt{meas} \mathbin{\#} x;\ \texttt{output}\ y \\
(f \mathbin{\$} e)^{-} &\equiv \texttt{let_}\ x \leftarrow e^{-};\ \texttt{unbox}\ f^{-}\ x \\
(e \mathbin{>!} f)^{-} &\equiv \texttt{let_}\ x \leftarrow e^{-};\ \texttt{lift}\ x\ (\lambda a.(f a)^{-}) \\
(\texttt{box}\ x \Rightarrow e)^{-} &\equiv \texttt{box_}\ x \Rightarrow e^{-}
\end{aligned}
$$

Figure 8.2: Translation of QNQ expressions and boxes to $\mathcal{Q}$WIRE circuits and boxes.

```
(* apply a box to the output of a circuit *)
Definition apply_box {σ τ} (b : Box σ τ) (c : Circuit σ) : Circuit τ :=
  match b with
  | box f ⇒ let_ p ← c; f p
  end.
Notation "b $ c" := (apply_box b c).
Lemma apply_box_WT : forall σ τ (b : Box σ τ) (c : Circuit σ) Δ,
      Typed_Box b → Δ ⊢ c :Circ → Δ ⊢ b $ c :Circ.

(* coerce a gate to a box, so gates can be used wherever boxes are used *)
Definition boxed_gate {σ τ} (g : Gate σ τ) : Box σ τ :=
  box_ p ⇒ gate g p output.
Lemma boxed_gate_WT {σ τ} (g : Gate σ τ) : Typed_Box (boxed_gate g).
Proof. type_check. Qed.
Coercion boxed_gate : Gate ↣ Box.

(* Tuples of circuits *)
Definition pair_circ {σ1 σ2} (c1 : Circuit σ1) (c2 : Circuit σ2)
                     : Circuit (σ1 ⊗ σ2) :=
  let_ p1 ← c1; let_ p2 ← c2; output (p1,p2)
Notation "( x , y , .. , z )" := (pair_circ .. (pair_circ x y) .. z).

(* let! against arbitrary circuits *)
Definition lift_circ c f := let_ p ← c; lift p f.
Notation "c >! f" := (lift_circ c f).
Notation "lift_ a ← c ; c'" := (c >! fun a ⇒ c').
```

Figure 8.3: Implementing QNQ syntax in $\mathcal{Q}$WIRE.

Proof. By induction on c. For $c = \texttt{output}\ p$, we have

$$\begin{aligned}(\texttt{let}_{_}\ x \leftarrow \texttt{output}\ p;\ c')^{+} &\equiv (c'\{p/x\})^{+} \\ &\sim_{\beta} \texttt{let}\ x \coloneqq p\ \texttt{in}\ (c')^{+}.\end{aligned}$$

For $c = \texttt{gate}_{_}\ p_2 \leftarrow g \# p_1;\ c_0$, we have

$$\begin{aligned}(\texttt{let}_{_}\ x \leftarrow \texttt{gate}_{_}\ p_2 \leftarrow g \# p_1;\ c_0;\ c')^{+} &\equiv (\texttt{gate}_{_}\ p_2 \leftarrow g \# p_1;\ \texttt{let}_{_}\ x \leftarrow c_0;\ c')^{+} \\ &\equiv \texttt{let}\ p_2 \coloneqq g^{+}\ \$\ p_1\ \texttt{in}\ (\texttt{let}_{_}\ x \leftarrow c_0;\ c')^{+}.\end{aligned}$$

By the induction hypothesis, this is equivalent to $\texttt{let}\ p_2 \coloneqq g^{+}\ \$\ p_1\ \texttt{in let}\ x \coloneqq c_0^{+}\ \texttt{in}\ (c')^{+}$. By η equivalence for `let` bindings, this is equivalent to

$$\texttt{let}\ x \coloneqq \texttt{let}\ p_2 \coloneqq g^{+}\ \$\ p_1\ \texttt{in}\ c_0^{+}\ \texttt{in}\ (c')^{+} \equiv \texttt{let}\ x \coloneqq (\texttt{gate}_{_}\ p_2 \leftarrow g \# p_1;\ c_0)^{+}\ \texttt{in}\ (c')^{+}.$$

Finally, for $c = p >!\ f$, we have

$$\begin{aligned}(\texttt{let}_{_}\ x \leftarrow p >!\ f;\ c')^{+} &\equiv (p >!\ \lambda a.\texttt{let}_{_}\ x \leftarrow fa;\ c')^{+} \\ &\equiv p^{+} >!\ \lambda a.(\texttt{let}_{_}\ x \leftarrow fa;\ c')^{+} \\ &\sim p^{+} >!\ \lambda a.\texttt{let}\ x \coloneqq (fa)^{+}\ \texttt{in}\ (c')^{+} \\ &\sim \texttt{let}\ x \coloneqq p^{+} >!\ \lambda a.(fa)^{+}\ \texttt{in}\ (c')^{+} \\ &\equiv \texttt{let}\ x \coloneqq (p >!\ f)^{+}\ \texttt{in}\ (c')^{+}.\end{aligned}$$

□

Theorem 8.3.2. *For any QNQ expression* $\Delta \vdash_{QNQ} e : \sigma$*, we have* $(e^{-})^{+} \sim e$*.*

Proof. By induction on e, using Lemma 8.3.1. The proof is straightforward by unfolding definitions, so we only illustrate two cases here.

For $e = (e_1, e_2)$, we unfold definitions to obtain

$$(e_1, e_2)^{-+} \equiv (\texttt{let}_{_}\ x_1 \leftarrow e_1^{-};\ \texttt{let}_{_}\ x_2 \leftarrow e_2^{-};\ \texttt{output}\ (x_1, x_2))^{+}.$$

Applying Lemma 8.3.1, we see this is equivalent to

$$\texttt{let}\ x_1 \coloneqq e_1^{-+}\ \texttt{in let}\ x_2 \coloneqq e_2^{-+}\ \texttt{in}\ (x_1, x_2).$$

By the induction hypothesis and β equivalences, this expression is equivalent to (e_1, e_2), as expected.

For $e = U \# e_0$, we have

$$\begin{aligned}
(U \# e_0)^{-+} &\equiv (\texttt{let}_{-}\ x \leftarrow e_0^{-};\ \texttt{gate}_{-}\ y \leftarrow U \# x;\ \texttt{output}\ y)^{+} \\
&\equiv \texttt{let}\ x \coloneqq e_0^{-+}\ \texttt{in}\ (\texttt{gate}_{-}\ y \leftarrow U \# x;\ \texttt{output}\ y)^{+} \\
&\sim \texttt{let}\ x \coloneqq e_0\ \texttt{in}\ (\texttt{gate}_{-}\ y \leftarrow U \# x;\ \texttt{output}\ y)^{+} \\
&\sim \texttt{let}\ x \coloneqq_0\ \ \texttt{in}\ \texttt{let}\ y \coloneqq U^{+}\ \$\ x\ \texttt{in}\ y \\
&\equiv (\texttt{box}\ x \Rightarrow U \# x)\ \$\ e_0 \\
&\sim U \# e_0.
\end{aligned}$$

□

8.4 Discussion

Compared to other quantum circuit languages like Quipper, LIQ$Ui|\rangle$, and Q#, $\mathcal{Q}$WIRE has a number of advantages, which we discuss in this section.

Linear types. The embedded linear programming model provides strong static guarantees about the correctness of $\mathcal{Q}$WIRE circuits, as well as flexible access to host-language data, for all the reasons discussed in this dissertation. The embedded linear framework also allows us to formalize the semantics of $\mathcal{Q}$WIRE inside the host language itself. The $\mathcal{Q}$WIRE implementation in Coq provides such a formalization (Rand *et al.*, 2017), though we do not discuss the details in this work.

Verification. The existence of a formal semantics in Coq enables the formal verification of quantum properties. In the Coq development we have used the semantics to reason about the state of prepared quantum systems like the Bell state or a coin flip, as well as the equivalence of two circuits. We have also verified the soundness of several of the equational rules described in Chapter 7.

Although formal verification of algorithms is time-consuming, in the case of quantum computing the cost is likely worthwhile: quantum computing resources will be expensive for the foreseeable future, debugging is doubly difficult in a quantum setting, and testing using simulations is not scalable. The verified reversible compiler ReVerC (Amy *et al.*, 2017) was recently adapted for quantum circuits in $\mathcal{Q}$WIRE by Rand *et al.* (2018). Other verification techniques are based on Hoare logic (D'Hondt and Panangaden, 2006; Kakutani, 2009; Ying, 2011). LIQ$Ui|\rangle$ makes it possible to reason about the state of a quantum system, because qubits in LIQ$Ui|\rangle$ are represented as pure state vectors. However, there does not seem to be any formal verification or semantics of LIQ$Ui|\rangle$ programs; the language primarily focuses on the optimization and simulation of large (up to 30 qubits) quantum systems.

Data structures. In $\mathcal{Q}$WIRE we have seen how to organize linear data into length-indexed lists $n \otimes \sigma$ using dependent types, and we can imagine other dependently-indexed data structures being designed in the same way. By virtue of the dependently typed host language, $\mathcal{Q}$WIRE inherits such dependent types for free. In comparison, other quantum circuit languages do not support dependent types, and they instead expose lists or other data structures to facilitate reasoning about families of circuits. In Quipper, for example, families of circuits must be instantiated at a particular size at circuit generation time. In $\mathcal{Q}$WIRE this instantiation is type directed—a family of circuits is given by the type $\prod_n \mathsf{Box}\ (n \otimes \sigma)\ (n \otimes \sigma)$, and instantiating the size of the circuit is done by applying the function to an argument.

In LIQ$Ui|\rangle$, all programs operate over lists of qubits, so there is no way to specify a

single or two-qubit operation at the type level. Like $\mathcal{Q}$WIRE, Q# can work with single qubits as well as tuples or lists of qubits. Unlike $\mathcal{Q}$WIRE, Q# cannot freely initialize and discard qubits. Instead, qubits can be brought into scope using a `borrowing` statement, where new qubits can be used inside the scope of the statement, but are freed at the end.

With respect to functions, Quipper allows higher-order functions that are eventually compiled out of top-level circuits. However, they are harder to account for semantically, as the standard semantic interpretations of quantum computing do not admit them. $\mathcal{Q}$WIRE, LIQ$Ui|\rangle$, and Q# allow only first-order functions, which we have found to be sufficient in practice.

Bits versus Booleans. In $\mathcal{Q}$WIRE we allow quantum circuits to contain bits of the type `Lower Bool`, which can be coerced to Coq booleans using the >! operation. Quipper makes a similar distinction between bits and booleans, where bits can be manipulated by the quantum computer, but booleans must be processed by the accompanying classical computer. Because the communication time between the classical and quantum computers will be high, on-circuit operations like applying a bit-controlled unitary will be much more efficient than using >! to extract the underlying boolean value and compute the value of the resulting circuit. On the other hand, it is more convenient for programmers to work directly with booleans. For example, the Quantum IO monad is a computational model where measurement produces a monadic boolean value (Altenkirch and Green, 2010); we hypothesize that this monad is exactly the linearity state monad `LStateT` [`Qubit`] α.

Linear embedded types for quantum computing. In general, the techniques for embedded quantum programming languages described in the last three chapters are intended to compliment the literature of existing quantum programming languages. The framework for linear/non-linear EDSLs should extend to any number of quantum programming languages that use linear or substructural type systems. The framework should both make linear types more accessible to existing embedded languages like Quipper, and also make toy languages like the linear lambda calculus more practical.

CHAPTER 9

Future work

This dissertation has proposed linear/non-linear type theory as a framework in which to develop, program, and reason about embedded domain-specific languages. We have developed several applications and implementations of the embedded LNL framework, but there are also rich areas for future work.

9.1 Adapting LNL to other substructural type systems.

Many programming applications are based on substructural type systems that are variations of traditional linear logic. Substructural type systems form a lattice based on which structural rules they allow. For example, linear type systems do not allow weakening or contraction, but they do allow exchange—that variables in a type system can be used in any order. The lattice of substructural type systems is shown in Figure 9.1.

The LNL model extends naturally to the lattice of substructural type systems. Reed (2009) developed adjoint logic which, given a preorder of modes $(M, \leq)$ each corresponding to different structural rules, assigns to each pair $m \leq n$ a pair of adjoint operators $U_{m \leq n} \dashv F_{n \geq m}$, where $U_{m \leq n}$ corresponds to `Lower` and F_n corresponds to `Lift`. Pfenning and Griffith (2015) adapts adjoint logic with a type theory for polarized session types with both linear and affine components, and Licata and Shulman (2016) extend adjoint type theory to systems described by not just a pre-order, but a 2-category of modes. The richer mode structure lets Licata *et al.* (2017) use adjoint operators for an even wider range of substructural and modal type operators.

The embedded LNL type theory discussed in this dissertation should extend naturally to an embedded adjoint framework. For one, if the linear EDSL were replaced with an affine, relevant, or ordered EDSL, the usual `Lift` and `Lower` operators could still expose useful host language data, libraries, and tools to the embedded language since the intuitionistic mode lies above every other mode in Figure 9.1.

Future work could also explore how more complex structures of modes, such as those introduced by Licata *et al.* (2017), could fit into an embedded framework. For example, if multiple developers produced different substructural EDSLs, an embedded adjoint type theory could provide a uniform way in which different languages interact.

Modal and temporal type systems. Adjoint type theory encompasses not just substructural type systems, but also modal and temporal type systems (Reed, 2009). In S4 modal logic, the (co-)modality $\Delta \vdash e : \Box\sigma$ is a proof of "necessarily σ", *i.e., e* is a proof that constructively exhibits a proof of σ. The rules of $\Box\sigma$ mimic the rules of $!\sigma$, and so can be partitioned into an adjunction between the fragment of necessary propositions and

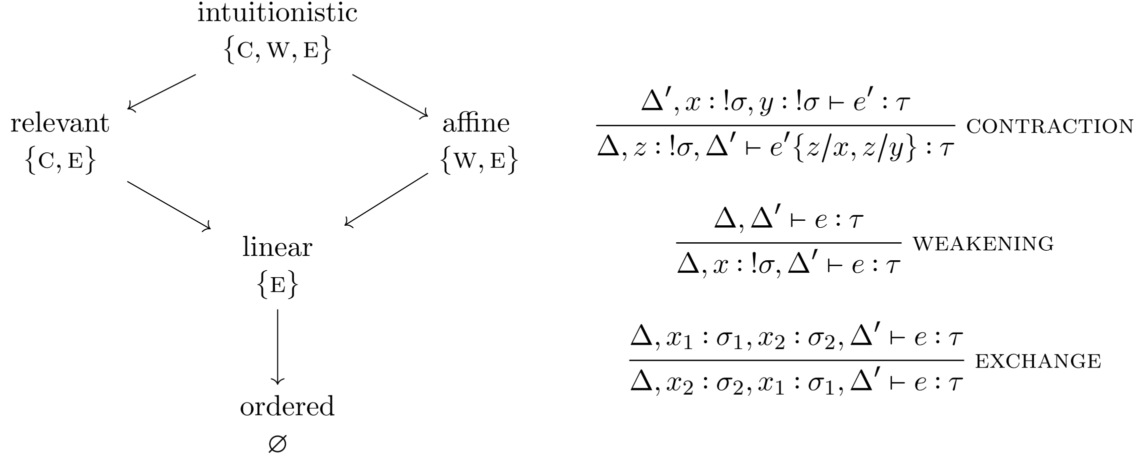


Figure 9.1: Lattice of substructural type systems

the fragment of merely true propositions. A similar decomposition applies to the operator "possibly σ", written $\diamond\sigma$, as well as the temporal operators "eventually σ" and "always σ".

Substructural types for mutable state. Over the years, several substructural type systems have arisen that adapt linear or affine type systems for particular programming domains, and thus depart from the strict correspondence with substructural logic. For example, Rust has an affine type system with the addition of *borrowing*, which lets mutable references be temporarily shared when the relevant parties only access the data safely and eventually relinquish ownership (Matsakis and Klock, 2014). Other ownership type systems allow partial or full ownership of data to be transferred between owners (Clarke *et al.*, 1998).

Uniqueness types, employed by languages such as Clean (Brus *et al.*, 1987) and Idris (Idris Community, 2017), enforce that there is at most one reference to a piece of data at any given time; any unique type can be trivially promoted to a non-unique type. Uniqueness logic, developed by Harrington (2001), formalizes the type system as a linear type system that replaces $!\sigma$ with σ^*, subject to weakening and contraction like !, but whose typing rules are dual to !:

$$\frac{\Delta \vdash e : \sigma}{\Delta \vdash e^* : \sigma^*}\ *\text{-I} \qquad \frac{\Delta_1 \vdash e : \sigma^* \qquad \Delta_2, x : \sigma \vdash e' : \tau^* \qquad \Delta_1 \bot \Delta_2}{\Delta_1, \Delta_2 \vdash \texttt{let}\ x^* \coloneqq e\ \texttt{in}\ e' : \tau^*}\ *\text{-E}$$

The type operator $(-)^*$ is a monad, not a comonad like !. Although Harrington (2001) writes briefly about the relationship between linear and uniqueness logic, the relationship is subtle and often a source of confusion. Benton (1995) cites the relationship between uniqueness and linear types as a motivation for developing linear/non-linear logic, but that goal has not entirely materialized. Perhaps an embedded uniqueness type system that uses techniques from this thesis could provide a formal meta-theory in which to compare the two systems.

Reasoning about mutable state. Through the Curry-Howard isomorphism, linear and substructural type systems correspond closely with linear and substructural logics. These logics have, in their own right, proven useful for reasoning about domain-specific languages

that may not themselves use linear types. Separation logic (Reynolds, 2002) is a popular extension of Floyd/Hoare logic that enables modular reasoning about the state of a heap, and the Linear Logical Framework (LLF) (Cervesato and Pfenning, 1996) is an extension of LF (Harper *et al.*, 1993) used for reasoning about systems with mutable state.

The embedded LNL framework shares many properties with LLF, such as the ability to reason about effectful DSLs and the ability to use types that depend on non-linear values. Future work could perhaps formalize an LLF-like system for an embedded LNL language.

9.2 Formalizing the theory of embedded languages.

In this thesis we worked inside the host language to reason about embedded linear languages, but it is another question to formalize the theory of the embedded and host languages from an external perspective. For example, every linear expression $\Delta \vdash e : \tau$ is also a host-language term $e : \texttt{LExp}\ \Delta\ \tau$. In the context of $\mathcal{Q}$WIRE (actually, a variation of $\mathcal{Q}$WIRE called EWire), Rennela and Staton (2018) proposes enriched category theory as a setting for this type of analysis, and there is lots of space for additional study of embedded and host language systems.

9.3 Variations to the structure of LNL

In the QNQ calculus of Chapter 6 we made two changes to the structure of `Lift` and `Lower` in order to satisfy constraints of the domain spece. For example, in QNQ `Lift` is not a free data type, but is derived as an instance of `Box` $\sigma\ \tau$. In addition, `Lower` is restricted to only finite types. We anticipate that other domains may require similar variations of the basic LNL adjunction.

For example, Benton's linear dependent type theory replaces the type `Lower` α with a dependent binder $F\ (x : \alpha).\tau$, which can be read as the dependent pair of $a : \alpha$ and $e : \tau\{a/x\}$. The type `Lower` α can be encoded as $F\ (_ : \alpha).\texttt{LUnit}$.

9.4 Drawing on the host language

Embedded LNL is more than just an implementation strategy for linear/non-linear logic. As we have seen in several places, the embedded structure allows us to use features of the host language—monadic programming, dependent types, higher-inductive types—to do meta-programming and meta-theory about the embedded language. Implementations of linear EDSLs in other dependently-typed host languages could introduce powerful new abstractions. For example, Idris is a dependently-typed language with uniqueness types (Idris Community, 2017); perhaps its capacity for substructural types could give an elegant embedding for full linear types. Scala provides limited support for dependent types (Odersky *et al.*, 2004); an LNL language embedded in Scala could take advantage of object-oriented paradigms and target the JVM. OCaml also has support for GADTs, and provides a rich module system and mutable state.

9.5 Shortcomings and outstanding problems

Embedded languages are not always the right solution for domain-specific programming languages. EDSLs are often less efficient than native languages, and are often more awkward to use.

Recently, more and more programming languages have begun to integrate some substructural features into the built-in type checking of the language. Rust's borrowing types are perhaps the most widely-used example (Matsakis and Klock, 2014), which enforce memory safety in Rust. In addition to its uniqueness types, Idris has an experimental language extension for full linear types based on McBride's (2016) resource-based presentation of linear logic.[16] A similar extension for Haskell is also in developement (Bernardy *et al.*, 2017). However, these language extensions, inspired by resource-based linear logic, require substantial changes to the base language and are not easily ported to new languages. They are also fixed to a specific presentation of substructural types and cannot be adapted by the user to fit their specific needs. Embedded languages, on the other hand, require no changes to the host language, and can support a variety of different presentations of linear or substructural types.

9.6 Conclusion

The strength of the embedded LNL framework is its ability to seamlessly integrate linear and non-linear data. In this dissertation we have demonstrated the quality of this programming model through a number of examples, including mutable state, IO, session types, and quantum computing. The domain of quantum computing in particular is an excellent case study because prior work has so far failed to integrate three crucial components: linear types, expressive embedded languages, and sound reasoning about the language's meta-theory.

Being able to reason about the meta-theory of embedded linear languages is a crucial contribution of the embedded LNL framework. When working in a dependently-typed host language, we can reason about the meta-theory of an embedded language without having to reason about the entirety of the host language. We have developed several meta-theoretic analyses of linear embedded languages—a categorical semantics in Chapter 5, a denotational semantics for quantum computing in Chapter 6, and an equational theory for quantum programs in Chapter 7.

We implemented the embedded LNL framework in Haskell (Chapter 4) and in Coq (Chapter 8). These implementations demonstrate how different host languages affect the tools available to the embedded language. For example, in Haskell we use the existing type class inference mechanism to check linearity constraints, while in Coq we develop a custom solver using user-defined proof tactics. In addition, features of the host language like general recursion, Π-types, and axioms like univalence or higher inductive types affect user experience in the embedded language.

Linear/non-linear type theory has proven itself to be a powerful framework for defining linear EDSLs, and we hope the techniques demonstrated in this dissertation make linear types more accessible and useful to programmers in years to come.

[16] https://www.idris-lang.org/idris-1-2-0-released/

BIBLIOGRAPHY

Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. *Constructing polymorphic programs with quotient types*. In Mathematics of Program Construction, edited by Dexter Kozen, pp. 2–15. Springer Berlin Heidelberg, Berlin, Heidelberg [2004]. ISBN 978-3-540-27764-4.

S. Abramsky and B. Coecke. *A categorical semantics of quantum protocols*. In Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004., pp. 415–425 [2004]. doi:10.1109/LICS.2004.1319636.

Samson Abramsky. *Computational interpretations of linear logic*. *Theoretical Computer Science*, volume 111(1):3 – 57 [1993]. doi:10.1016/0304-3975(93)90181-R.

Thorsten Altenkirch and Alexander S Green. *The quantum IO monad*. *Semantic Techniques in Quantum Computation*, pp. 173–205 [2010]. doi:10.1017/cbo9781139193313.006.

Matthew Amy, Jianxin Chen, and Neil J. Ross. *A finite presentation of CNOT-dihedral operators*. *Electronic Proceedings in Theoretical Computer Science*, volume 266:84–97 [2018]. doi:10.4204/eptcs.266.5. URL `https://doi.org/10.4204%2Feptcs.266.5`.

Matthew Amy, Martin Roetteler, and Krysta M. Svore. *Verified compilation of space-efficient reversible circuits*. In Computer Aided Verification, edited by Rupak Majumdar and Viktor Kunčak, pp. 3–21. Springer International Publishing [2017]. ISBN 978-3-319-63390-9.

Carlo Angiuli, Edward Morehouse, Daniel R. Licata, and Robert Harper. *Homotopical patch theory*. In Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming, ICFP '14, pp. 243–256. ACM, New York, NY, USA [2014]. doi:10.1145/2628136.2628158.

David Aspinall and Martin Hofmann. *Dependent types*. In Advanced Topics in Types and Programming Languages, edited by Benjamin C. Pierce, chapter 1, pp. 45–86. MIT Press [2005].

Steve Awodey. Category Theory. Oxford Logic Guides. OUP Oxford [2010]. ISBN 9780199587360.

Miriam Backens. Completeness and the ZX-calculus. Ph.D. thesis, University of Oxford [2015].

Andrew Barber. *Dual intuitionistic linear logic*. Technical Report ECS-LFCS-96-347 [1996].

Nick Benton. *A mixed linear and non-linear logic: Proofs, terms and models.* In Computer Science Logic, *Lecture Notes in Computer Science*, volume 933, edited by Leszek Pacholski and Jerzy Tiuryn, pp. 121–135. Springer Berlin Heidelberg [1995]. doi:10.1007/BFb0022251.

Nick Benton, Gavin Bierman, Valeria de Paiva, and Martin Hyland. *A term calculus for intuitionistic linear logic.* In Typed Lambda Calculi and Applications, *Lecture Notes in Computer Science*, volume 664, edited by Marc Bezem and JanFriso Groote, pp. 75–90. Springer Berlin Heidelberg [1993]. doi:10.1007/BFb0037099.

Nick Benton and Philip Wadler. *Linear logic, monads and the lambda calculus.* In Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science, 1996. LICS '96., pp. 420–431 [1996]. doi:10.1109/LICS.1996.561458.

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.*, volume 2(POPL) [2017]. doi:10.1145/3158093.

Stefano Bettelli, Tommaso Calarco, and Luciano Serafini. *Toward an architecture for quantum programming. The European Physical Journal D*, volume 25(2):181–200 [2003].

G.M. Bierman. *What is a categorical model of intuitionistic linear logic?* In Typed Lambda Calculi and Applications, *Lecture Notes in Computer Science*, volume 902, edited by Mariangiola Dezani-Ciancaglini and Gordon Plotkin, pp. 78–93. Springer Berlin Heidelberg [1995]. doi:10.1007/BFb0014046.

Edwin Brady and Kevin Hammond. *Resource-safe systems programming with embedded domain specific languages.* In Practical Aspects of Declarative Languages, edited by Claudio Russo and Neng-Fa Zhou, pp. 242–257. Springer Berlin Heidelberg, Berlin, Heidelberg [2012]. ISBN 978-3-642-27694-1.

Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. *A core quantitative coeffect calculus.* In Programming Languages and Systems, edited by Zhong Shao, pp. 351–370. Springer Berlin Heidelberg, Berlin, Heidelberg [2014]. doi:10.1007/978-3-642-54833-8_19.

T. H. Brus, M. C. J. D. van Eekelen, M. O. van Leer, and M. J. Plasmeijer. *Clean — a language for functional graph rewriting.* In Functional Programming Languages and Computer Architecture, edited by Gilles Kahn, pp. 364–384. Springer Berlin Heidelberg, Berlin, Heidelberg [1987]. ISBN 978-3-540-47879-9.

Luís Caires and Frank Pfenning. *Session types as intuitionistic linear propositions.* In CONCUR 2010 - Concurrency Theory, *Lecture Notes in Computer Science*, volume 6269, edited by Paul Gastin and François Laroussinie, pp. 222–236. Springer Berlin Heidelberg [2010]. doi:10.1007/978-3-642-15375-4_16.

Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. *Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. Journal of Functional Programming*, volume 19(5):509–543 [2009]. doi:10.1017/S0956796809007205.

Iliano Cervesato and Frank Pfenning. *A linear logical framework*. In Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science, pp. 264–275 [1996]. doi: 10.1109/LICS.1996.561339.

Chih-Ping Chen and Paul Hudak. *Rolling your own mutable ADT—a connection between linear types and monads*. In Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '97. Association for Computing Machinery (ACM) [1997]. doi:10.1145/263699.263708.

Adam Chlipala. Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant. The MIT Press [2013]. ISBN 0262026651.

Kenta Cho. *Semantics for a quantum programming language by operator algebras*. *New Generation Computing*, volume 34(1):25–68 [2016]. doi:10.1007/s00354-016-0204-3.

Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. *HoTTSQL: Proving query rewrites with univalent SQL semantics*. *SIGPLAN Not.*, volume 52(6):510–524 [2017]. doi:10.1145/3140587.3062348.

David G. Clarke, John M. Potter, and James Noble. *Ownership types for flexible alias protection*. In Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '98, pp. 48–64. ACM, New York, NY, USA [1998]. doi:10.1145/286936.286947.

Vincent Danos, Elham Kashefi, and Prakash Panangaden. *The measurement calculus*. *J. ACM*, volume 54(2) [2007]. doi:10.1145/1219092.1219096.

Ellie D'Hondt and Prakash Panangaden. *Quantum weakest preconditions*. *Mathematical Structures in Computer Science*, volume 16(03):429–451 [2006].

Richard Eisenberg, Benoıt Valiron, and Steve Zdancewic. *Typechecking linear data: Quantum computation in Haskell* [2012]. URL `http://www.monoidal.net/papers/qhaskell.pdf`.

Richard A. Eisenberg and Jan Stolarek. *Promoting functions to type families in Haskell*. In Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell, Haskell '14, pp. 95–106. ACM, New York, NY, USA [2014]. doi:10.1145/2633357.2633361.

Richard A. Eisenberg and Stephanie Weirich. *Dependently typed programming with singletons*. In Proceedings of the 2012 Haskell Symposium, Haskell '12, pp. 117–130. ACM, New York, NY, USA [2012]. doi:10.1145/2364506.2364522.

Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan G. Ahmed. *Visible type application*. In Programming Languages and Systems: 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, April 2–8, 2016, edited by Peter Thiemann, pp. 229–254. Springer Berlin Heidelberg, Berlin, Heidelberg [2016]. doi:10.1007/978-3-662-49498-1_10.

Matthew Fluet, Greg Morrisett, and Amal Ahmed. *Linear regions are all you need*. In Programming Languages and Systems, pp. 7–21. Springer Science + Business Media [2006]. doi:10.1007/11693024_2.

Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. *Linear dependent types for differential privacy*. In Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, pp. 357–370. ACM, New York, NY, USA [2013]. doi:10.1145/2429069.2429113.

Simon J. Gay and Vasco T. Vasconcelos. *Linear type theory for asynchronous session types*. *Journal of Functional Programming*, volume 20:19–50 [2010]. doi:10.1017/S0956796809990268.

I. M. Georgescu, S. Ashhab, and Franco Nori. *Quantum simulation*. *Rev. Mod. Phys.*, volume 86:153–185 [2014]. doi:10.1103/RevModPhys.86.153.

Jean-Yves Girard. *Une extension de l'interpretation de Gödel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types*. In Proceedings of the Second Scandinavian Logic Symposium, *Studies in Logic and the Foundations of Mathematics*, volume 63, edited by J.E. Fenstad, pp. 63 – 92. Elsevier [1971]. doi:10.1016/S0049-237X(08)70843-7.

Jean-Yves Girard. *The system F of variable types, fifteen years later*. *Theoretical Computer Science*, volume 45:159 – 192 [1986]. doi:10.1016/0304-3975(86)90044-7.

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

Jean-Yves Girard. *A new constructive logic: classic logic*. *Mathematical Structures in Computer Science*, volume 1:255–296 [1991]. doi:10.1017/S0960129500001328.

Jean-Yves Girard. *On the unity of logic*. *Annals of pure and applied logic*, volume 59(3):201–217 [1993].

Jean-Yves Girard. *Light linear logic*. *Information and Computation*, volume 143(2):175 – 204 [1998]. ISSN 0890-5401. doi:10.1006/inco.1998.2700.

Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. *Bounded linear logic: a modular approach to polynomial-time computability*. *Theoretical Computer Science*, volume 97(1):1 – 66 [1992]. doi:https://doi.org/10.1016/0304-3975(92)90386-T.

Jean-Yves Girard, Paul Taylor, and Yves Lafont. Proofs and types, volume 7. Cambridge University Press Cambridge [1989].

Alexander Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. *An introduction to quantum programming in Quipper*. In Proceedings of the 5th International Conference on Reversible Computation, *Lecture Notes in Computer Science*, volume 7948, pp. 110–124 [2013a]. ISBN 978-3-642-38985-6.

Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. *Quipper: A scalable quantum programming language*. In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, pp. 333–342. ACM, New York, NY, USA [2013b]. doi:10.1145/2491956.2462177.

Lov K. Grover. *A fast quantum mechanical algorithm for database search.* In Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing, STOC '96, pp. 212–219. ACM [1996]. doi:10.1145/237814.237866.

Robert Harper, Furio Honsell, and Gordon Plotkin. *A framework for defining logics. J. ACM*, volume 40(1):143–184 [1993]. doi:10.1145/138027.138060.

Dana G. Harrington. A type system for destructive updates in declarative programming languages. Ph.D. thesis, University of Calgary [2001]. doi:10.5072/PRISM/14286.

Ichiro Hasuo and Naohiko Hoshino. *Semantics of higher-order quantum computation via geometry of interaction.* In Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, pp. 237–246 [2011].

Kohei Honda. *Types for dyadic interaction.* In CONCUR'93, pp. 509–523 [1993]. doi:10.1007/3-540-57208-2_35.

The Idris Community. *Uniqueness types* [2017]. URL `http://docs.idris-lang.org/en/latest/reference/uniqueness-types.html`.

Yoshihiko Kakutani. *A logic for formal verification of quantum programs.* In Advances in Computer Science-ASIAN 2009. Information Security and Privacy, pp. 79–93. Springer [2009].

Oleg Kiselyov. Typed Tagless Final Interpreters, pp. 130–174. Springer Berlin Heidelberg, Berlin, Heidelberg [2012]. doi:10.1007/978-3-642-32202-0_3.

Emmanuel H. Knill. *Conventions for quantum pseudocode.* Technical Report LAUR-96-2724, Los Alamos National Laboratory [1996]. doi:10.2172/366453.

Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. *Linearity and the π-calculus. ACM Transactions on Programming Languages and Systems*, volume 21(5):914–947 [1999]. doi:10.1145/330249.330251.

Neelakantan R Krishnaswami and Nick Benton. *A semantic model for graphical user interfaces.* In ACM SIGPLAN Notices, *ICFP '11*, volume 46, pp. 45–57. ACM, New York, NY, USA [2011]. doi:10.1145/2034773.2034782.

Neelakantan R Krishnaswami, Nick Benton, and Jan Hoffmann. *Higher-order functional reactive programming in bounded space. ACM SIGPLAN Notices*, volume 47(1):45–58 [2012].

Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. *Integrating linear and dependent types.* In Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '15, pp. 17–30. ACM, New York, NY, USA [2015]. doi:10.1145/2676726.2676969.

John Launchbury and Simon L Peyton Jones. *State in Haskell. LISP and Symbolic Computation*, volume 8(4):293–341 [1995]. doi:10.1007/bf01018827.

Olivier Laurent. Étude de la polarisation en logique. Theses, Université de la Méditerranée - Aix-Marseille II [2002].

Paul Blain Levy. *Call-by-push-value: A subsuming paradigm.* In Call-By-Push-Value, pp. 27–47. Springer Science Business Media [2003]. doi:10.1007/978-94-007-0954-6_2.

Daniel R. Licata and Michael Shulman. *Adjoint logic with a 2-category of modes.* pp. 219–235 [2016]. doi:10.1007/978-3-319-27683-0_16. URL http://dx.doi.org/10.1007/978-3-319-27683-0_16.

Daniel R Licata, Michael Shulman, and Mitchell Riley. *A fibrational framework for substructural and modal logics (extended version)* [2017]. Draft, URL http://dlicata.web.wesleyan.edu/pubs/lsr17multi/lsr17multi-ex.pdf.

Sam Lindley and J. Garrett Morris. *A semantics for propositions as sessions.* In Proceedings of Programming Languages and Systems, 24th European Symposium on Programming, ESOP 2015, volume 9032, edited by Jan Vitek, pp. 560–584. Springer Berlin Heidelberg, London, UK [2015]. doi:10.1007/978-3-662-46669-8_23.

Octavio Malherbe. Categorical Models of Computation: Partially Traced Categories and Presheaf Models of Quantum Computation. Ph.D. thesis, University of Ottawa [2010].

Nicholas D. Matsakis and Felix S. Klock, II. *The rust language.* In Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology, HILT '14, pp. 103–104. ACM, New York, NY, USA [2014]. doi:10.1145/2663171.2663188.

Ken Matsumoto and Kazuyuki Amano. *Representation of quantum circuits with Clifford and pi/8 gates* [2008]. arXiv:quant-ph/0806.3834.

Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. *Lightweight linear types in system F°.* In Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation - TLDI '10. Association for Computing Machinery (ACM) [2010]. doi:10.1145/1708016.1708027.

Conor McBride. I Got Plenty o' Nuttin', pp. 207–233. Springer International Publishing [2016]. doi:10.1007/978-3-319-30936-1_12.

Paul André Melliès. *Categorical models of linear logic revisited.* Technical Report 22, Laboratoire PPS a la Université Paris Denis Diderot [2003]. URL https://hal.archives-ouvertes.fr/hal-00154229.

Robin Milner. Communicating and mobile systems: the pi calculus. Cambridge university press [1999].

Eugenio Moggi. *Computational lambda-calculus and monads.* In Proceedings of the 4th Annual Symposium on Logic in Computer Science, pp. 14–23. IEEE Press [1989].

J. Garrett Morris. *The best of both worlds: Linear functional programming without compromise.* In Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, pp. 448–461. New York, NY, USA [2016]. doi:10.1145/2951913.2951925.

Yunseong Nam, Neil J. Ross, Yuan Su, Andrew M. Childs, and Dmitri Maslov. *Automated optimization of large quantum circuits with continuous parameters. npj Quantum Information*, volume 4(1) [2018]. doi:10.1038/s41534-018-0072-4.

Michael A Nielsen and Isaac L Chuang. Quantum computation and quantum information. Cambridge university press [2010]. ISBN 9781139495486.

Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. *An overview of the scala programming language.* Technical Report LAMP-REPORT-2004-006 [2004].

Michele Pagani, Peter Selinger, and Benoît Valiron. *Applying quantitative semantics to higher-order quantum computing.* In Proceedings of the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, pp. 647–658 [2014]. doi:10.1145/2535838.2535879.

Jennifer Paykin, Robert Rand, and Steve Zdancewic. *QWIRE: A core language for quantum circuits.* In Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, pp. 846–858. ACM, New York, NY, USA [2017]. doi:10.1145/3009837.3009894.

Jennifer Paykin and Steve Zdancewic. *A linear/producer/consumer model of classical linear logic. Mathematical Structures in Computer Science*, p. 1–26 [2016]. doi: 10.1017/S0960129516000347.

Jennifer Paykin and Steve Zdancewic. *The linearity monad.* In Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Haskell 2017, pp. 117–132. ACM, New York, NY, USA [2017]. doi:10.1145/3122955.3122965.

Tomas Petricek, Dominic Orchard, and Alan Mycroft. *Coeffects: A calculus of context-dependent computation.* In Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming, ICFP '14, pp. 123–135. ACM, New York, NY, USA [2014]. doi:10.1145/2628136.2628160.

F. Pfenning and C. Elliott. *Higher-order abstract syntax.* In Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, PLDI '88, pp. 199–208. ACM, New York, NY, USA [1988]. doi:10.1145/53990.54010.

Frank Pfenning and Dennis Griffith. *Polarized substructural session types.* In Foundations of Software Science and Computation Structures, *Lecture Notes in Computer Science*, volume 9034, edited by Andrew Pitts, pp. 3–22. Springer Berlin Heidelberg [2015]. doi: 10.1007/978-3-662-46678-0_1.

Benjamin C Pierce. Basic category theory for computer scientists. MIT press [1991].

Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. Software Foundations. Electronic textbook [2016]. Version 4.0. `http://www.cis.upenn.edu/~bcpierce/sf`.

Jeff Polakow. *Embedding a full linear lambda calculus in Haskell*. In Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015, pp. 177–188 [2015]. doi:10.1145/2804302.2804309.

François Pottier and Jonathan Protzenko. *Programming with permissions in Mezzo*. *ACM SIGPLAN Notices*, volume 48(9):173–184 [2013]. doi:10.1145/2544174.2500598.

Robert Rand, Jennifer Paykin, Dong-Ho Lee, and Steve Zdancewic. *ReQWIRE: Reasoning about reversible quantum circuits*. In Proceedings of the 15th International Conference on Quantum Physics and Logic, QPL 2018, Halifax, Nova Scotia, 3-7 June 2018 [2018].

Robert Rand, Jennifer Paykin, and Steve Zdancewic. *QWIRE practice: Formal verification of quantum circuits in Coq*. In Proceedings 14th International Conference on Quantum Physics and Logic, QPL 2017, Nijmegen, The Netherlands, 3-7 July 2017., pp. 119–132 [2017]. doi:10.4204/EPTCS.266.8.

Jason Reed. *A judgmental deconstruction of modal logic* [2009]. URL `http://jcreed.org/papers/jdml.pdf`.

Jason Reed and Benjamin C. Pierce. *Distance makes the types grow stronger: A calculus for differential privacy*. In Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP '10, pp. 157–168. ACM, New York, NY, USA [2010]. doi:10.1145/1863543.1863568.

Mathys Rennela. *Towards a quantum domain theory: Order-enrichment and fixpoints in W*-algebras*. *Electronic Notes in Theoretical Computer Science (MFPS XXX)*, volume 308:289 – 307 [2014]. doi:0.1016/j.entcs.2014.10.016. Proceedings of the 30th Conference on the Mathematical Foundations of Programming Semantics.

Mathys Rennela and Sam Staton. *Classical control and quantum circuits in enriched category theory*. *Electronic Notes in Theoretical Computer Science*, volume 336:257 – 279 [2018]. ISSN 1571-0661. doi:10.1016/j.entcs.2018.03.027. The Thirty-third Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIII).

John C. Reynolds. *Separation logic: a logic for shared mutable data structures*. In Proceedings 17th Annual IEEE Symposium on Logic in Computer Science, pp. 55–74 [2002]. doi:10.1109/LICS.2002.1029817.

Francisco Rios and Peter Selinger. *A categorical model for a quantum circuit description language (extended abstract)*. *Electronic Proceedings in Theoretical Computer Science*, volume 266:164–178 [2018]. doi:10.4204/eptcs.266.11.

Neil J. Ross. Algebraic and Logical Methods in Quantum Computation. Ph.D. thesis, Dalhousie University [2015].

Peter Selinger. *Towards a quantum programming language*. *Mathematical Structures in Computer Science*, volume 14(4):527–586 [2004]. doi:10.1017/S0960129504004256.

Peter Selinger. *Dagger compact closed categories and completely positive maps.* In Proceedings of the 3rd International Workshop on Quantum Programming Languages, QPL 2005, Electronic Notes in Theoretical Computer Science 170, pp. 139–163. Elsevier Science [2007].

Peter Selinger and Benoît Valiron. *On a fully abstract model for a quantum linear functional language.* In Proceedings of the 4th International Workshop on Quantum Programming Languages, QPL 2006, Electronic Notes in Theoretical Computer Science 210, pp. 123–137. Elsevier [2008].

Peter Selinger and Benoît Valiron. *Quantum lambda calculus.* In Semantic Techniques in Quantum Computation, edited by Simon Gay and Ian Mackie, pp. 135–172. Cambridge University Press [2009]. doi:10.1017/cbo9781139193313.005.

Tim Sheard and Simon Peyton Jones. *Template meta-programming for Haskell.* In Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell, Haskell '02, pp. 1–16. ACM, New York, NY, USA [2002]. doi:10.1145/581690.581691.

P. W. Shor. *Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer.* *SIAM Review*, volume 41:303–332 [1999]. doi:10.1137/S0036144598347011.

Kristina Sojakova. *Higher inductive types as homotopy-initial algebras.* In Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '15, pp. 31–42. ACM, New York, NY, USA [2015]. doi:10.1145/2676726.2676983.

Sam Staton. *Algebraic effects, linearity, and quantum programming languages.* In Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '15, pp. 395–406. ACM, New York, NY, USA [2015]. doi:10.1145/2676726.2676999.

Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. *Q#: Enabling scalable quantum computing and development with a high-level dsl.* In Proceedings of the Real World Domain Specific Languages Workshop 2018, RWDSL '18, pp. 7:1–7:10. ACM, New York, NY, USA [2018]. doi:10.1145/3183895.3183901.

Jesse A. Tov and Riccardo Pucella. *Practical affine types.* In Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '11, pp. 447–458. ACM, New York, NY, USA [2011]. doi:10.1145/1926385.1926436.

Taichi Uemura. *Homotopies for free!* [2017]. `arXiv:cs.LO/1701.07937`.

The Univalent Foundations Program. Homotopy Type Theory: Univalent Foundations of Mathematics. `http://homotopytypetheory.org/book`, Institute for Advanced Study [2013].

M. Vákár. *Syntax and semantics of linear dependent types* [2014]. `arXiv:cs.LO/1405.0033`.

Floris van Doorn, Jakob von Raumer, and Ulrik Buchholtz. *Homotopy type theory in Lean*. In Interactive Theorem Proving, edited by Mauricio Ayala-Rincón and César A. Muñoz, pp. 479–495. Springer International Publishing, Cham [2017].

André van Tonder. *A lambda calculus for quantum computation. SIAM Journal of Computation*, volume 33(5):1109–1135 [2004].

Juliana Kaizer Vizzotto, Bruno Crestani Calegaro, and Eduardo Kessler Piveta. *A double effect λ-calculus for quantum computation*. In Programming Languages, *Lecture Notes in Computer Science*, volume 8129, edited by André Rauber Du Bois and Phil Trinder, pp. 61–74. Springer Berlin Heidelberg, Berlin, Heidelberg [2013]. doi:10.1007/978-3-642-40922-6_5.

Philip Wadler. *Linear types can change the world!* In IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel. North Holland [1990].

Philip Wadler. *There's no substitute for linear logic*. In 8th International Workshop on the Mathematical Foundations of Programming Semantics [1992].

Philip Wadler. *A syntax for linear logic*. In Mathematical Foundations of Programming Semantics, edited by Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, pp. 513–529. Springer Berlin Heidelberg, Berlin, Heidelberg [1994]. ISBN 978-3-540-48419-6.

Philip Wadler. *Propositions as sessions*. *Journal of Functional Programming*, volume 24:384–418 [2014]. ISSN 1469-7653. doi:10.1017/S095679681400001X.

David Walker. *Substructural type systems*. In Advanced Topics in Types and Programming Languages, edited by Benjamin C. Pierce, chapter 1, pp. 3–44. MIT Press [2005].

Dave Wecker and Krysta M Svore. *LIQUiD: A software design architecture and domain-specific language for quantum computing* [2014]. URL `https://www.microsoft.com/en-us/research/publication/liqui-a-software-design-architecture-and-domain-specific-language-for-quantum-computing/`.

Friedrich Wehrung. Refinement Monoids, Equidecomposability Types, and Boolean Inverse Semigroups. Springer International Publishing [2017]. doi:10.1007/978-3-319-61599-8.

Mingsheng Ying. *Floyd–Hoare logic for quantum programs. ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 33(6):19 [2011].

Mingsheng Ying. *Quantum recursion and second quantisation* [2014]. `arXiv:quant-ph/1405.4443`.

Mingsheng Ying, Nengkun Yu, and Yuan Feng. *Alternation in quantum programming: From superposition of data to superposition of programs* [2014]. `arXiv:cs.PL/1402.5172`.

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. *Giving Haskell a promotion*. In Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation, TLDI '12, pp. 53–66. ACM, New York, NY, USA [2012]. doi:10.1145/2103786.2103795.

Noam Zeilberger. *On the unity of duality*. *Annals of Pure and Applied Logic*, volume 153(1–3):66 – 96 [2008]. doi:10.1016/j.apal.2008.01.001. Special Issue: Classical Logic and Computation (2006).