Master's Thesis

A Basis for Event-Driven Programming

Mohammad-Ali A'râbi

Adviser: Prof. Dr. Peter Thiemann
Examiner: Prof. Dr. Andreas Podelski

Albert-Ludwigs-University Freiburg
Faculty of Engineering
Department of Computer Science
Chair for Programming Languages

May 2021

Writing period

25. 11. 2020 – 25. 05. 2021

Examiner

Prof. Dr. Andreas Podelski

Advisers

Prof. Dr. Peter Thiemann

Masterarbeit

Beiträge zur Begründung der ereignisgesteuerten Programmierung

Mohammad-Ali A'râbi

Betreuer: Prof. Dr. Peter Thiemann
Gutachter: Prof. Dr. Andreas Podelski

Albert-Ludwigs-Universität Freiburg
Technische Fakultät
Institut für Informatik
Lehrstuhl für Programmiersprachen

Mai 2021

Bearbeitungszeit

25. 11. 2020 – 25. 05. 2021

Gutachter

Prof. Dr. Andreas Podelski

Betreuer

Prof. Dr. Peter Thiemann

Declaration

I hereby declare that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I hereby also declare that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

_____ _____

Place, Date Signature

Abstract

Event-driven programming as a paradigm is gaining more and more popularity with introducing new technologies such as Rx and the single-threaded asynchronous Node.js. Although many mainstream languages have no support for linear or affine types, the nature of this new event-driven approach is somewhat linear: The program's logic is wrapped into a linear pipeline of callbacks. Furthermore, the monadic style of events and callbacks has a linear nature. Types like promises and observables model variables that are not immediately available, but will become available eventually. These types are a natural resemblance of the linear-time temporal logic's *future* modality. This dissertation proposes a mixture of linear/non-linear logic and linear-time temporal logic to model the type system in event-driven programming. The practicality of this framework is examined via a number of examples. A Coq implementation of the typing rules is there to verify the soundness of the system. And a TypeScript implementation tries to set infrastructure to discuss the suitability of the framework for practical use.

Zusammenfassung

Die ereignisgesteuerte Programmierung als Paradigma gewinnt mit der Einführung neuer Technologien wie Rx und der asynchronen Single-Thread-Node.js immer mehr an Beliebtheit. Obwohl viele Mainstream-Sprachen keine Unterstützung für lineare oder affine Typen bieten, ist die Natur dieses neuen ereignisgesteuerten Ansatzes etwas linear: Die Logik des Programms ist in eine lineare Pipeline von Rückrufen eingebunden. Darüber hinaus ist der monadische Stil von Ereignissen und Rückrufen linear. Typen wie Versprechen und beobachtbare Modellvariablen, die nicht sofort verfügbar sind, aber irgendwann verfügbar sein werden. Diese Typen sind eine natürliche Ähnlichkeit mit der *irgendwann*-Modalität der zeitlichen Logik der linearen Zeit.

Diese Dissertation schlägt eine Mischung aus linearer / nichtlinearer Logik und zeitlich linearer Logik vor, um das Typsystem in der ereignisgesteuerten Programmierung zu modellieren. Die Praktikabilität dieses Rahmens wird anhand einer Reihe von Beispielen untersucht. Eine Coq-Implementierung der Typisierungsregeln dient dazu, die Solidität des Systems zu überprüfen. Eine TypeScript-Implementierung versucht, eine Infrastruktur einzurichten, um die Eignung des Frameworks für den praktischen Gebrauch zu erörtern.

Acknowledgments

First and foremost, I would like to thank my wife and friend, Farnaz, who was there for me every step of the way and helped me out especially with the category theory parts (which didn't end up in the theatrical cut anyway).
I would also thank my adviser Peter Thiemann who beard with me throughout the years. And I would like to thank Andreas Podelski for reading my work: Thanks, man, you're probably the only other person reading it.
Other people I would like to thank are the following:

- Leonhard Euler, for inspiring the American Math Society to create such a beautiful font.
- Hieronymus Formschneider, for inventing Fraktur.
- David Hilbert, for using Fraktur for logical formulae in his *Grundzüge der theoretischen Logik*.
- Georg Cantor, for no specific reason.
- Google Translate, for translating my abstract into German.

Also, the following people claimed they have read my work and made helpful comments: Hayk Gevorgyan.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

1.1 Problem Definition

Event-driven programming is becoming the standard paradigm for programming in almost every domain: backend, frontend, GUI, etc. All of the framework and library vendors are now offering *reactive*, *i.e.* event-driven interfaces for their products. Spring framework came up with WebFlux, Angular is all wrapped in RxJS's observables, and Node.js is async by nature. Reactive drivers are being implemented for databases, to support an end-to-end reactive spectrum.

Unfortunately, event-driven programming can be difficult, as it is a combination of two seemingly contradictory elements: callbacks and states. When callbacks change the state of a program, we say they have *side effects*, a very unpopular term among developers (A'râbi [2020]).

To overcome these problems, many different and somewhat ad hoc approaches were taken. Scala has only 3 different built-in abstractions for event types: `Promise`, `Future`, and `Signal` (Maier et al. [2010]), putting RxScala aside. And of course we have access to Java libraries in Scala, adding Reactor and similar abstractions to the bunch.

In such a world, laying a basis for event-driven programming is necessary more than ever.

This work aims to study event-driven programming by identifying structures common to it with a natural perspective from temporal logic. Under this cor-

respondence, an event (specially in the Rx or `Promise` sense) is computation that can eventually return a value, and can be identified with the temporal logic's *eventually*/*future* modality $\Diamond\mathfrak{A}$.

A correspondence between linear-time temporal logic (LTL) and event-driven programming is presented in terms of Curry–Howard correspondence, and then is tried to be verified with an implementation in the proof-assistant Coq language (A'râbi [2021]). A novel reformulation of the theory behind LTL and Curry–Howard correspondence is also presented. And finally a TypeScript implementation tries to relate the theory to real-life production code written in JavaScript/TypeScript.

1.1.1 Verifying LTTT

Paykin et al. [2016] introduced a type system based on linear/non-linear logic of Benton [1994] and linear-time temporal logic of Pnueli [1977] and inspired by the linear logic of Girard [1987]. We will refer to this type system and the theory around it as linear-time temporal type theory, or LTTT for short. The Coq implementation aims to verify the soundness of LTTT.

1.1.2 Generic Store

Typing contexts lie at the core of type inference relations, storing all the hypotheses of a sequent. Having the right data model for the context is especially important as it will affect the typing rules as well as the proofs about them.

Among his libraries for the locally nameless infrastructure, Arthur Charguéraud [2012] built a generic model for environments, and later Emmanuel Polonowski [2011] rewrote his generic environments model in Coq 8.3, publishing it as a standalone Coq library, namely `generic-environments`[1].

The initial problem with `generic-environments` was incompatibility with

[1] `https://github.com/coq-community/generic-environments`

newer versions of Coq. After discussions with the library's author and Théo Zimmermann, the man behind `coq-community`, the library was moved to `coq-community` and then published for the newer versions of Coq by the author of the current thesis.

Still, the implementation of `generic-environments` made use of Coq features that had breaking changes in the newer versions, hence a new simpler model was implemented.

1.1.3 TypeScript Embedding

Linear and affine types are not supported by many mainstream programming languages. Paykin [2018] worked on embedding linear types in non-linear host languages. She particularly implemented an embedding for Coq and Haskell.

To study event-driven programming in action, we chose JavaScript for multiple reasons. To name one, the server-side JS is async by nature, and hence most of the logic written in Node.JS are wrapped into callbacks or async types such as `Promise` and `Observable`.

TypeScript is currently the typed version of JavaScript and hence was used for our study. But, there are fundamental differences between TypeScript and Haskell or Coq. Types in TypeScript are parts of the metalanguage and are not shipped to the executable code. Hence, one has very limited access to types in the runtime and particularly cannot store them in a key-value store.

So, to embed linearity into TypeScript/JavaScript, one should employ logic that has similar behavior.

1.2 Thesis Statement

Linear-time temporal type theory is a natural model for event-driven programming. The theory is sound and has practical applications in a variety of programming languages and libraries.

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

- Chapter 2 contains a tutorial and novel formulation of theoretical background for setting a basis for event-driven programming. This includes:
  - A crash course on intuitionistic logic, simply typed lambda calculus, linear lambda calculus, linear logic, and modal logic,
  - Stating Curry–Howard correspondence,
  - A review and reformulation of (linear-time) temporal logic.
- Chapter 3 formulates the linear-time temporal type theory, LTTT, based on work of Paykin et al. [2016], and discusses its connection to the real-world event-driven programming paradigm.
- Chapter 4 presents the implementation of LTTT in Coq, to verify that the theory's soundness. This chapter in part contains details about implementation of the generic store that is later used for proving statements about the typing rules.
- Chapter 5 presents an implementation of LTTT in TypeScript, to demonstrate LTTT in practice, and to set an infrastructure for further discussion. The TypeScript implementation is published as an NPM library and can be used in production code.

1.3 Audience

This dissertation can be read by undergraduate students of Computer Science and Mathematics. Prior familiarity with logic and type systems is helpful but not required. Chapter 2 tries to define everything required to understand the LTTT's formulation later given in Chapter 3. For more background in the logic used in the current work, we refer the reader to the book of Ben-Ari [2012b] and its chapter on linear-time temporal logic, Ben-Ari [2012a]. For more background in type systems, we refer the reader to the book of Pierce [2004] and especially its first chapter by Walker [2004] on linear type systems. For more background in the theoretical part, work of Girard [1987] on linear logic is suggested. The Backus–Naur form (BNF) was used extensively to define formal languages (see Backus [1959]).

Chapter 4 discusses a Coq implementation of the theory, so familiarity with the syntax of Coq is helpful. Similarly, familiarity with syntax of TypeScript is helpful for Chapter 5.

To describe the future types, we make examples from the ReactiveX[2] and especially RxJS[3], as well as Promise/A specification (Zyp [2010]) and its implementation in JavaScript (Bershanskiy et al. [2020]).

1.4 Conventions

Throughout the thesis, small Greek letters are used as metavariables in the logics: $\alpha, \beta, \varphi, \psi$. In the type systems, Fraktur letters are used instead: small letters for terms and expressions in the language being typed, e.g. $\mathfrak{t}, \mathfrak{e}$, and capital letters for the types: $\mathfrak{A}, \mathfrak{B}, \mathfrak{T}$. One reason for this separation is because some small Greek letters have meanings in standard formulation of programming languages and so are used in the toy languages: λ for the

[2] `http://reactivex.io/`

[3] `https://github.com/ReactiveX/rxjs`

Example	Name	Meaning
$\alpha\beta\gamma$	Greek small	metavariables in logics
$\Gamma\Delta$	Greek capital	(typing) contexts
$\mathfrak{abc}$	Fraktur small	metavariables ranging over programming languages in type systems
$\mathfrak{ABC}$	Fraktur capital	metavariables ranging over types in type systems
`abc`	monospace	code

Table 1: Different typefaces and their meanings

lambda expression, ι for injection, and π for projection. Furthermore, most capital Greek letters are indistinguishable from their Latin descendants, e.g. $\mathrm{A}, \mathrm{B}, \mathrm{T}$. The use of Fraktur letters for logical expressions is inspired by Hilbert and Ackermann [1928]. In Chapter 4, the Fraktur letters represent types in the metalanguage (in this case Coq):

```
Inductive type : 𝔊 -> t -> 𝔗 -> Prop := ...
```

Here `type` is the typing relation represented as a sequent in Chapter 3: $\Gamma \vdash t : \mathfrak{T}$.

2 Background

Propositional logic is about judgments of the form "α holds". In contrast, a type system is about judgments of the form "t is of type $\mathfrak{T}$". Although the judgments are different, to derive them one should employ similar techniques. A good example is depicted in Table 2.

Intuitionistic Logic	Lambda Calculus
$\dfrac{}{\Gamma_1, \alpha, \Gamma_2 \vdash \alpha}$ Ax	$\dfrac{}{\Gamma_1, x : \alpha, \Gamma_2 \vdash x : \alpha}$
$\dfrac{\Gamma, \alpha \vdash \beta}{\Gamma \vdash \alpha \to \beta} \to$-I	$\dfrac{\Gamma, x : \alpha \vdash t : \beta}{\Gamma \vdash \lambda x.t : \alpha \to \beta} \to$-I
$\dfrac{\Gamma \vdash \alpha \quad \Gamma \vdash \alpha \to \beta}{\Gamma \vdash \beta} \to$-E	$\dfrac{\Gamma \vdash x : \alpha \quad \Gamma \vdash f : \alpha \to \beta}{\Gamma \vdash fx : \beta} \to$-E

Table 2: Correspondence between intuitionistic implicational natural deduction and lambda calculus type assignment rules, as stated by W. H. Howard [1969]

As one can see from Table 2, logic and type theory are closely related. We will investigate their relation thoroughly later in this chapter. To do so, we need to define a few things first.

2.1 Intuitionistic Logic

The *intuitionistic logic* is known to differ from the classical logic by rejecting the *law of excluded middle* (Latin: *principium tertii exclusi*, aka *tertium non datur*). As a result, the two following classical tautologies are not valid in the

intuitionistic logic: $p \vee \neg p$ (*tertium non datur*) and $\neg\neg p \to p$ (double-negation elimination).

2.1.1 Language of Propositional Intuitionistic Logic

The language of intuitionistic propositional logic, IPC (C for calculus) or IPC($\bot, \to, \vee, \wedge$), is defined with the following grammar:

$$\alpha ::= \bot \mid x \mid \alpha \to \alpha \mid \alpha \vee \alpha \mid \alpha \wedge \alpha.$$

Here $\bot$ is constant (falsity or contradiction), x ranges over the propositional variables, and the connectives are: implication $\to$, disjunction $\vee$, and conjunction $\wedge$. Negation $\neg\alpha$ is defined by $\neg\alpha = \alpha \to \bot$.

2.1.2 Sequent and Natural Deduction

A *context* is a set of α's, usually denoted by Γ. We write Γ, Δ for $\Gamma \cup \Delta$, and also Γ, α for $\Gamma \cup \{\alpha\}$.

Definition 2.1 (Sequent)**.** The relation $\Gamma \vdash \alpha$ is defined syntactically by Figure 1, but can be though of as "α holds if all Γ holds". The relation $\vdash$ is called *sequent*, the elements of Γ are called *antecedents* or *hypotheses*, and α is called *succedents* or *consequents*. When $\Gamma = \emptyset$, we write $\vdash \alpha$, and say that α is a *theorem*. $\Diamond$

The proof system which consists of the sequent relation together with the deduction rules, is called *natural deduction*.

The subset of IPC that only deals with $\alpha ::= x \mid \alpha \to \alpha$, with three rules of Ax, $\to$-I, and -E, is called IPC($\to$).

$$\frac{}{\Gamma, \alpha \vdash \alpha}\ \text{Ax} \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash \beta}\ \bot\text{-E}$$

$$\frac{\Gamma \vdash \alpha_1 \quad \Gamma \vdash \alpha_2}{\Gamma \vdash \alpha_1 \wedge \alpha_2}\ \wedge\text{-I} \qquad \frac{\Gamma \vdash \alpha_1 \wedge \alpha_2}{\Gamma \vdash \alpha_i}\ \wedge\text{-E}$$

$$\frac{\Gamma \vdash \alpha_i}{\Gamma \vdash \alpha_1 \vee \alpha_2}\ \vee\text{-I} \qquad \frac{\Gamma \vdash \alpha_1 \vee \alpha_2 \quad \Gamma, \alpha_1 \vdash \beta \quad \Gamma, \alpha_2 \vdash \beta}{\Gamma \vdash \beta}\ \vee\text{-E}$$

$$\frac{\Gamma, \alpha \vdash \beta}{\Gamma \vdash \alpha \to \beta}\ \to\text{-I} \qquad \frac{\Gamma \vdash \alpha \quad \Gamma \vdash \alpha \to \beta}{\Gamma \vdash \beta}\ \to\text{-E}$$

Figure 1: Natural deduction rules of the propositional intuitionistic logic

2.2 Simply Typed Lambda Calculus

The language of simply typed lambda calculus $\lambda(\to)$ is defined by the following grammar:

$$\mathfrak{T} ::= \mathsf{T} \mid \mathfrak{T} \to \mathfrak{T}$$

$$t ::= x \mid \lambda x.t \mid t_1 t_2$$

Here, T designates constant types (take `Boolean` | `String` for example) and x designates variables. The t's are called *terms* and the $\mathfrak{T}$'s are called *types*.

2.2.1 Sequent and Natural Deduction

A *context* here is a set of ordered pairs $\langle t, \mathfrak{T} \rangle$, usually denoted by Γ. We additionally assume that there are no duplicate keys, making contexts *functions* (in the set-theoretic sense). The set of keys in Γ is denoted by $\iota_1[\Gamma]$ and the values $\iota_2[\Gamma]$. When $\iota_1[\Gamma] \cap \iota_1[\Delta] = \emptyset$, we write Γ, Δ for $\Gamma \cup \Delta$. We also write $\Gamma, t : \mathfrak{T}$ for $\Gamma, \{\langle t, \mathfrak{T} \rangle\}$.

$$\frac{}{\Gamma, x : \mathfrak{T} \vdash x : \mathfrak{T}}\ \text{Var}$$

$$\frac{\Gamma, x : \mathfrak{T} \vdash t : \mathfrak{S}}{\Gamma \vdash \lambda x.t : \mathfrak{T} \to \mathfrak{S}} \to\text{-I} \qquad \frac{\Gamma \vdash t_1 : \mathfrak{T} \to \mathfrak{S} \quad \Gamma \vdash t_2 : \mathfrak{T}}{\Gamma \vdash t_1 t_2 : \mathfrak{S}} \to\text{-E}$$

Figure 2: Natural deduction rules of the simply typed lambda calculus

Definition 2.2. The relation $\Gamma \vdash t : \mathfrak{T}$ for $\lambda(\to)$ is defined syntactically by Figure 2. $\Diamond$

2.3 Curry–Howard Correspondence

The following theorem, states the Curry–Howard correspondence between implicational fragment of intuitionistic propositional logic IPC($\to$) and the simply typed lambda calculus $\lambda(\to)$.

Theorem 2.3. *The types in* $\lambda(\to)$ *form an* IPC($\to$)*. Furthermore:*

- *If* $\Gamma \vdash t : \alpha$*, then* $\iota_2[\Gamma] \vdash \alpha$*.*
- *If* $\Delta \vdash \alpha$*, then there is a* Γ *and a* t*, such that* $\Delta = \iota_2[\Gamma]$ *and* $\Gamma \vdash t : \alpha$*.*

An interesting case is when $\Gamma = \emptyset$.

Corollary 2.3.1. *A formula* φ *in* IPC($\to$) *is a true theorem, iff it's inhabited by a* $\lambda(\to)$*-term* t*, that is* $\vdash t : \varphi$*.*

An example is the formula $(\beta \to \alpha) \to (\gamma \to \beta) \to \gamma \to \alpha$. Only the fact that the $\lambda(\to)$-term $\lambda a.\lambda b.\lambda c.a(bc)$ inhabits this type, proves it. Figure 3 demonstrate a proof of it by natural deduction. For neatness, we named $\Gamma := a : \beta \to \alpha, b : \gamma \to \beta, c : \gamma$.

A case of Corollary 2.3.1 is $\varphi = \bot$. If any term of the language inhabits the contradiction, then the underlying logic on the types is inconsistent.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash a : \beta \to \alpha} \quad \dfrac{\dfrac{}{\Gamma \vdash b : \gamma \to \beta} \quad \dfrac{}{\Gamma \vdash c : \gamma}}{\Gamma \vdash bc : \beta}}{a : \beta \to \alpha, b : \gamma \to \beta, c : \gamma \vdash a(bc) : \alpha}}{a : \beta \to \alpha, b : \gamma \to \beta \vdash \lambda c.a(bc) : \gamma \to \alpha}}{a : \beta \to \alpha \vdash \lambda b.\lambda c.a(bc) : (\gamma \to \beta) \to \gamma \to \alpha}}{\vdash \lambda a.\lambda b.\lambda c.a(bc) : (\beta \to \alpha) \to (\gamma \to \beta) \to \gamma \to \alpha}$$

Figure 3: Proof by natural deduction that $\vdash \lambda a.\lambda b.\lambda c.a(bc) : (\beta \to \alpha) \to (\gamma \to \beta) \to \gamma \to \alpha$

2.4 Structural Rules

In natural deduction, *structural rules* are inference rules that do not refer to any logical connective, but instead operates on the sequents directly.

Three common structural rules are:

- **Weakening**. Which states that hypotheses of a sequent could be extended.

$$\dfrac{\Gamma_1, \Gamma_2 \vdash \beta}{\Gamma_1, \alpha, \Gamma_2 \vdash \beta}\ \text{Weak} \qquad \dfrac{\Gamma_1, \Gamma_2 \vdash t : \mathfrak{B}}{\Gamma_1, x : \mathfrak{A}, \Gamma_2 \vdash t : \mathfrak{B}}\ \text{Weak}$$

- **Contraction**. Which states that two similar (or unifiable) hypothesis can be replaced by a single hypothesis.

$$\dfrac{\Gamma_1, \alpha, \alpha, \Gamma_2 \vdash \beta}{\Gamma_1, \alpha, \Gamma_2 \vdash \beta}\ \text{Contr} \qquad \dfrac{\Gamma_1, x : \mathfrak{A}, x : \mathfrak{A}, \Gamma_2 \vdash t : \mathfrak{B}}{\Gamma_1, x : \mathfrak{A}, \Gamma_2 \vdash t : \mathfrak{B}}\ \text{Contr}$$

  Another version of the weakening rule is proposed by Walker [2004] for type systems:

$$\dfrac{\Gamma_1, x_2 : \mathfrak{A}, x_3 : \mathfrak{A}, \Gamma_2 \vdash t : \mathfrak{B}}{\Gamma_1, x_1 : \mathfrak{A}, \Gamma_2 \vdash t[x_2 \mapsto x_1, x_3 \mapsto x_1] : \mathfrak{B}}\ \text{Contr}$$

- **Exchange**. Which states that two hypotheses may be swapped.

$$\frac{\Gamma_1, \alpha_1, \alpha_2, \Gamma_2 \vdash \beta}{\Gamma_1, \alpha_2, \alpha_1, \Gamma_2 \vdash \beta}\ \text{Exch} \qquad \frac{\Gamma_1, x_1 : \mathfrak{A}_1, x_2 : \mathfrak{A}_2, \Gamma_2 \vdash t : \mathfrak{B}}{\Gamma_1, x_2 : \mathfrak{A}_2, x_1 : \mathfrak{A}_1 \Gamma_2 \vdash t : \mathfrak{B}}\ \text{Exch}$$

Any logic (or type system) that lacks any of these structural rules is called a substructural logic (or type system, respectively).

	W	C	E
Unrestricted	✓	✓	✓
Affine	✓		✓
Relevant		✓	✓
Linear			✓
Ordered			

Table 3: Substructural logics / type systems and the structural rules they have

To understand how the structural rules for the type systems affect programming languages they are used for, a list of substructural type systems is presented with the restrictions they have:

- **Affine**. Every variable can be used at most once.
- **Relevant**. Every variable must be used at least once.
- **Linear**. Every variable must be used exactly once.
- **Ordered**. Every variable must be used exactly once and in the order in which it is introduced.

2.5 A Taste of Linear Logic

Linear type systems correspond to or are inspired by the linear logic of Girard [1987]. In ordinary logic, we think of $\psi \vdash \varphi$ as "φ is true whenever ψ is", and when the truth of something is proven, it is true for the eternity.

That is not the case with the linear logic. Members of the logic are not facts about the nature, rather resources that are consumed.
A good example is the following. In ordinary logic (take intuitionistic for example), one can prove

$$\alpha \to \beta, \alpha \to \gamma, \alpha \vdash \beta \wedge \gamma.$$

But this is not the case for the linear logic:

$$\alpha \multimap \beta, \alpha \multimap \gamma, \alpha \nvdash \beta \otimes \gamma.$$

Once the hypothesis α is used to derive β, it's not available any more. We need another α to derive γ:

$$\alpha \multimap \beta, \alpha \multimap \gamma, \alpha, \alpha \vdash \beta \otimes \gamma.$$

As Wadler [1993] puts it: "Traditional logic encourages reckless use of resources. Contraction profligately duplicates assumptions, Weakening foolishly discards assumptions. This makes sense for logic, where truth is free; and it makes sense for some programming languages, where copying a value is as cheap as copying a pointer. But it is not always sensible."
The language of linear logic, LL($0, \multimap, \oplus, \otimes, !$) or simply LL, is defined with the following grammar:

$$\alpha ::= 0 \mid \alpha \multimap \alpha \mid \alpha \oplus \alpha \mid \alpha \otimes \alpha \mid !\alpha.$$

Here 0 is a constant and the linear version of $\bot$, the modality ! is called *of course* or *bang* and is similar to the modality $\Box$, the connective $\multimap$ is called *lollipop* which is the bilinear version of implication, $\oplus$ is called *additive disjunction* is the bilinear version of or, and $\otimes$ is called *multiplicative conjunction* or *tensor* is the bilinear version of and.

The linear logic of Girard [1987] is more extensive and has a richer inventory of connectives and modalities. An honorable mention is the modality ?, that is called *why not* or *par*, and is the DeMorgan dual of !. As you have already guessed, ? is the linear version of $\diamond$. The two modalities ! and ? are called *exponential* modalities.

2.6 Linear/Non-Linear Logic

It is usually not enough to have only linear members in the logic or type system. In many domains, when using the linear types, they are mixed with unrestricted ones. As an example, Walker [2004] prefixes his types with `lin` or `un` and treats them differently.

In the linear logic, the modality ! is used for addressing this shortcoming. Members of the form $!\alpha$ can be duplicated or discarded. But it's not practical to be used in a type system, especially when embedding a linear type system in an unrestricted host language (see Paykin [2018]). The linear/non-linear logic of Benton [1994] offers a more practical modeling of linear and non-linear logics mixed together.

The language of the linear/non-linear logic, LNL, is defined with the following grammar:

$$\tau ::= \top \mid \mathsf{T} \mid \tau \times \tau \mid \tau + \tau \mid \tau \to \tau \mid \lceil \alpha \rceil ,$$
$$\alpha ::= 1 \mid \mathsf{A} \mid \alpha \otimes \alpha \mid \alpha \oplus \alpha \mid \alpha \multimap \alpha \mid \lfloor \tau \rfloor .$$

Here A (resp. T) ranges over a set of linear (resp. non-linear) atomic propositions / types, the functor $\lceil \cdot \rceil$ is called *lift*, and the functor $\lfloor \cdot \rfloor$ is called *lower*. These two functors are in charge of moving between the linear and the non-linear worlds. The two connectives $\times$ and $+$ denote conjunction and disjunction.

2.7 Modal Logic

Modal logic is a collection of formal systems initially created to reason about necessity and possibility. These two are denoted by two modalities $\diamond\varphi$, that reads "possibly φ", and $\Box\varphi$, that reads "necessarily φ".
So, the language of the modal logic is defined by the following grammar:

$$\alpha ::= \bot \mid \chi \mid \alpha \to \alpha \mid \alpha \vee \alpha \mid \alpha \wedge \alpha \mid \diamond\alpha \mid \Box\alpha.$$

Here the modalities $\diamond$ and $\Box$ are De Morgan duals:

- $\diamond\varphi$ "possibly φ" is equivalent to $\neg\Box\neg\varphi$ "not necessarily not φ",
- $\Box\varphi$ "necessarily φ" is equivalent to $\neg \diamond \neg\varphi$ "not possibly not φ".

The following are a few axioms in modal logic:

- **N**. $\vdash \varphi$ then $\vdash \Box\varphi$.
- **K**. $\Box(\varphi \to \psi) \to (\Box\varphi \to \Box\psi)$.
- **T**. $\Box\varphi \to \varphi$.
- **4**. $\Box\varphi \to \Box\Box\varphi$.
- **5**. $\diamond\varphi \to \Box \diamond \varphi$.
- **B**. $\varphi \to \Box \diamond \varphi$.
- **D**. $\Box\varphi \to \diamond\varphi$.

2.8 Temporal Logic

Temporal logic as an umbrella term is the study of reasoning about propositions qualified in terms of time. Formulating temporal logic as a modal

logic was first done by Prior [1957, 1967, 1968] and then ported to computer science by Pnueli [1977].
Prior's 4 tense operators were the following (see Goranko and Rumberg [2020]):

- $\mathsf{P}\varphi$, "It has at some time been the case that φ".
- $\mathsf{F}\varphi$, "It will at some time be the case that φ".
- $\mathsf{H}\varphi$, "It has always been the case that φ".
- $\mathsf{G}\varphi$, "It will always be the case that φ".

As we are not interested in the past, we will omit P and H, and show the other two, F and G, with $\diamond$ and $\Box$, respectively. Note that F and G form a modal logic, and e.g. $\neg\mathsf{F}\neg\varphi$ "not eventually not φ" is equivalent to $\mathsf{G}\varphi$ "always φ". Here G is always in the sense of "from now on".
The language of temporal logic is then the same as the language of modal logic, but one can add more operators:

- Next. $\mathsf{N}\varphi$, or $\circ\varphi$, "φ holds in the next stage".
- Until. $\varphi\ \mathsf{U}\ \psi$, or $\varphi \spadesuit \psi$, "φ (holds at least) until ψ (and ψ has to hold eventually)".
- Release. $\varphi\ \mathsf{R}\ \psi$, or $\varphi \clubsuit \psi$, "φ releases ψ", or "ψ is true until and including the point where φ becomes true (if φ never becomes true, ψ has to stay true forever)".

2.9 Linear-Time Temporal Logic

Linear-time temporal logic (LTL) is a modal temporal logic reasoning about the linear time, in contrast to the branching-time temporal logics such as computation tree logic (CTL). LTL was introduced by Pnueli [1977].
LTL adds two more operators to the temporal logic:

- Weak until. $\varphi \, W \, \psi$, or $\varphi \heartsuit \psi$, "φ (holds at least) until ψ (and φ holds forever if ψ never becomes true)".
- Strong release. $\varphi \, M \, \psi$, or $\varphi \diamondsuit \psi$, "$\psi$ is true until and including the point where φ becomes true (and φ has to hold eventually)".

The following equivalences hold in LTL:

- Next is self-dual: $\neg \circ \neg\varphi \equiv \circ\varphi$.
- Future/eventually and always are dual:

$$\begin{aligned} \neg \diamond \neg\varphi &\equiv \Box\varphi, \\ \neg\Box\neg\varphi &\equiv \diamond\varphi. \end{aligned}$$

- Until and release are dual:

$$\begin{aligned} \neg(\neg\varphi \spadesuit \neg\psi) &\equiv \varphi \clubsuit \psi, \\ \neg(\neg\varphi \clubsuit \neg\psi) &\equiv \varphi \spadesuit \psi. \end{aligned}$$

- Weak until and strong release are dual:

$$\begin{aligned} \neg(\neg\varphi \heartsuit \neg\psi) &\equiv \varphi \diamondsuit \psi, \\ \neg(\neg\varphi \diamondsuit \neg\psi) &\equiv \varphi \heartsuit \psi. \end{aligned}$$

Some special temporal axioms in LTL are the following:

- $\Box\varphi \rightarrow \Box\Box\varphi$ (4).
- $\Box\Box\varphi \rightarrow \Box\varphi$ (follows from T).
- $\diamond\varphi \rightarrow \diamond\diamond\varphi$ (follows from 5 and D).
- $\diamond\diamond\varphi \rightarrow \diamond\varphi$.

3 Linear-Time Temporal Type Theory

The type system introduced in this chapter is based on the work of Paykin et al. [2016].

3.1 Event-Based Toy Language and Types

Throughout this chapter, a toy language together with a type system is used to discuss using temporal logic in event-driven programming. The type system is presented in the work of Paykin et al. [2016]. The types are based on the linear/non-linear logic of Benton [1994] with an addition of $\diamond$ modality based on linear-time temporal logic of Pnueli [1977].
The language and its type are defined in Figure 4. Non-linear types are designated with $\mathfrak{T}$ and linear types are designated with $\mathfrak{A}$. Terms, denoted by $\mathfrak{t}$, form the non-linear fragment of the language. On the other hand, expressions, denoted by $\mathfrak{e}$, form the linear fragment.

- `Void` (resp. 0) represents a type that is inhabited by no term (resp. expression).
- `Unit` (resp. 1) represents a type that is inhabited by exactly one term (resp. expression), namely () (resp. ()). We won't differentiate the term () and the expression () in notation, as it should be apparent from the context which one we are talking about.
- The multiplicative connectives $\times$ and $\otimes$ represent type of pairs, in the

$$\mathfrak{T} ::= \texttt{Unit} \mid \texttt{Void} \mid \mathfrak{T} \times \mathfrak{T} \mid \mathfrak{T} + \mathfrak{T} \mid \mathfrak{T} \to \mathfrak{T} \mid \lceil \mathfrak{A} \rceil$$
$$\mathfrak{A} ::= 1 \mid 0 \mid \mathfrak{A} \otimes \mathfrak{A} \mid \mathfrak{A} \oplus \mathfrak{A} \mid \mathfrak{A} \multimap \mathfrak{A} \mid \diamond \mathfrak{A} \mid \lfloor \mathfrak{T} \rfloor$$

$$\begin{aligned} t ::= {} & x \mid () \mid \texttt{case}\ t\ \texttt{of}\ () \mid (t_1, t_2) \mid \pi_i t \mid \texttt{in}_i t \mid \texttt{case}\ t\ \texttt{of}\ (\texttt{in}_1 x_1 \to t_1 \mid \texttt{in}_2 x_2 \to t_2) \\ & \mid \lambda x.t \mid t_1 t_2 \mid \texttt{suspend}\ e \\ e ::= {} & x \mid () \mid \texttt{case}\ e\ \texttt{of}\ () \mid (e_1, e_2) \mid \texttt{in}_i e \mid \texttt{case}\ t\ \texttt{of}\ (\texttt{in}_1 x_1 \to e_1 \mid \texttt{in}_2 x_2 \to e_2) \\ & \mid \lambda x.e \mid e_1 e_2 \mid \texttt{force}\ t \mid \texttt{return}\ e \mid \texttt{bind}\ x = e_1\ \texttt{in}\ e_2 \mid \lfloor t \rfloor \\ & \mid \texttt{let}\ () = e_1\ \texttt{in}\ e_2 \mid \texttt{let}\ (x_1, x_2) = e_1\ \texttt{in}\ e_2 \mid \texttt{let}\ \lfloor x \rfloor = e_1\ \texttt{in}\ e_2 \end{aligned}$$

Figure 4: Syntax of linear and non-linear types, terms, and expressions

$$\frac{}{\Gamma, x : \mathfrak{T} \vdash x : \mathfrak{T}}\ \text{Var} \qquad \frac{}{\Gamma \vdash () : \texttt{Unit}}\ \texttt{Unit-I} \qquad \frac{\Gamma \vdash t : \texttt{Void}}{\Gamma \vdash \texttt{case}\ t\ \texttt{of}\ () : \mathfrak{S}}\ \texttt{Void-E}$$

$$\frac{\Gamma \vdash t_1 : \mathfrak{T}_1 \quad \Gamma \vdash t_2 : \mathfrak{T}_2}{\Gamma \vdash (t_1, t_2) : \mathfrak{T}_1 \times \mathfrak{T}_2}\ \times\text{-I} \qquad \frac{\Gamma \vdash t : \mathfrak{T}_1 \times \mathfrak{T}_2}{\Gamma \vdash \pi_i t : \mathfrak{T}_i}\ \times\text{-E}$$

$$\frac{\Gamma \vdash t : \mathfrak{T}_i}{\Gamma \vdash \texttt{in}_i t : \mathfrak{T}_1 + \mathfrak{T}_2}\ +\text{-I}$$

$$\frac{\Gamma \vdash t : \mathfrak{T}_1 + \mathfrak{T}_2 \quad \Gamma, x_1 : \mathfrak{T}_1 \vdash t_1 : \mathfrak{S} \quad \Gamma, x_2 : \mathfrak{T}_2 \vdash t_2 : \mathfrak{S}}{\Gamma \vdash \texttt{case}\ t\ \texttt{of}\ (\texttt{in}_1 x_1 \to t_1 \mid \texttt{in}_2 x_2 \to t_2) : \mathfrak{S}}\ +\text{-E}$$

$$\frac{\Gamma, x : \mathfrak{T} \vdash t : \mathfrak{S}}{\Gamma \vdash \lambda x.t : \mathfrak{T} \to \mathfrak{S}}\ \to\text{-I} \qquad \frac{\Gamma \vdash t_1 : \mathfrak{T} \to \mathfrak{S} \quad \Gamma \vdash t_2 : \mathfrak{T}}{\Gamma \vdash t_1 t_2 : \mathfrak{S}}\ \to\text{-E}$$

Figure 5: Typing rules of the linear types

$$\frac{}{\Gamma, x : \mathfrak{A} \vdash x : \mathfrak{A}}\ \text{Var} \qquad \frac{\Gamma; \Delta \vdash e : 0}{\Gamma; \Delta \vdash \texttt{case}\ e\ \texttt{of}\ () : \mathfrak{B}}\ 0\text{-E}$$

$$\frac{}{\Gamma; \cdot \vdash () : 1}\ 1\text{-I} \qquad \frac{\Gamma; \Delta_1 \vdash e_1 : 1 \quad \Gamma; \Delta_2 \vdash e_2 : \mathfrak{B}}{\Gamma; \Delta_1, \Delta_2 \vdash \texttt{let}\ () = e_1\ \texttt{in}\ e_2 : \mathfrak{B}}\ 1\text{-E}$$

$$\frac{\Gamma; \Delta_1 \vdash e_1 : \mathfrak{A}_1 \quad \Gamma; \Delta_2 \vdash e_2 : \mathfrak{A}_2}{\Gamma; \Delta_1, \Delta_2 \vdash (e_1, e_2)e : \mathfrak{A}_1 \otimes \mathfrak{A}_2}\ \otimes\text{-I}$$

$$\frac{\Gamma; \Delta_1 \vdash e_1 : \mathfrak{A}_1 \otimes \mathfrak{A}_2 \quad \Gamma; \Delta_2, x_1 : \mathfrak{A}_1, x_2 : \mathfrak{A}_2 \vdash e_2 : \mathfrak{B}}{\Gamma; \Delta_1, \Delta_2 \vdash \texttt{let}\ (x_1, x_2) = e_1\ \texttt{in}\ e_2 : \mathfrak{B}}\ \otimes\text{-E}$$

$$\frac{\Gamma; \Delta \vdash e : \mathfrak{A}_i}{\Gamma; \Delta \vdash \texttt{in}_i e : \mathfrak{A}_1 \oplus \mathfrak{A}_2}\ \oplus\text{-I}$$

$$\frac{\Gamma; \Delta_1 \vdash e : \mathfrak{A}_1 \oplus \mathfrak{A}_2 \quad \Gamma; \Delta_2, x_1 : \mathfrak{A}_1 \vdash e_1 : \mathfrak{B} \quad \Gamma; \Delta_2, x_2 : \mathfrak{A}_2 \vdash e_2 : \mathfrak{B}}{\Gamma; \Delta_1, \Delta_2 \vdash \texttt{case}\ t\ \texttt{of}\ (\texttt{in}_1 x_1 \to e_1 \mid \texttt{in}_2 x_2 \to e_2) : \mathfrak{B}}\ \oplus\text{-E}$$

$$\frac{\Gamma; \Delta, x : \mathfrak{A} \vdash e : \mathfrak{B}}{\Gamma; \Delta \vdash \lambda x.e : \mathfrak{A} \multimap \mathfrak{B}}\ \multimap\text{-I} \qquad \frac{\Gamma; \Delta_1 \vdash e_1 : \mathfrak{A} \multimap \mathfrak{B} \quad \Gamma; \Delta_2 \vdash e_2 : \mathfrak{A}}{\Gamma; \Delta_1, \Delta_2 \vdash e_1 e_2 : \mathfrak{B}}\ \multimap\text{-E}$$

$$\frac{\Gamma; \Delta \vdash e : \mathfrak{A}}{\Gamma; \Delta \vdash \texttt{return}\ e : \diamond\mathfrak{A}}\ \diamond\text{-I} \qquad \frac{\Gamma; \Delta_1 \vdash e_1 : \diamond\mathfrak{A} \quad \Gamma; \Delta_2, x : \mathfrak{A} \vdash e_2 : \diamond\mathfrak{B}}{\Gamma; \Delta_1, \Delta_2 \vdash \texttt{bind}\ x = e_1\ \texttt{in}\ e_2 : \diamond\mathfrak{B}}\ \diamond\text{-E}$$

Figure 6: Typing rules of the non-linear types

$$\frac{\Gamma; \cdot \vdash e : \mathfrak{A}}{\Gamma \vdash \texttt{suspend}\ e : \lceil \mathfrak{A} \rceil}\ \lceil\cdot\rceil\text{-I} \qquad \frac{\Gamma \vdash t : \lceil \mathfrak{A} \rceil}{\Gamma; \cdot \vdash \texttt{force}\ t : \mathfrak{A}}\ \lceil\cdot\rceil\text{-E}$$

$$\frac{\Gamma \vdash t : \mathfrak{T}}{\Gamma; \cdot \vdash \lfloor t \rfloor : \lfloor \mathfrak{T} \rfloor}\ \lfloor\cdot\rfloor\text{-I} \qquad \frac{\Gamma; \Delta_1 \vdash e_1 : \lfloor \mathfrak{T} \rfloor \quad \Gamma, x : \mathfrak{T}; \Delta_2 \vdash e_2 : \mathfrak{B}}{\Gamma; \Delta_1, \Delta_2 \vdash \texttt{let}\ \lfloor x \rfloor = e_1\ \texttt{in}\ e_2 : \mathfrak{B}}\ \lfloor\cdot\rfloor\text{-E}$$

Figure 7: Typing rules for moving between linear and non-linear types

non-linear and linear domains, respectively.

- The additive connectives + and $\oplus$ are used for branching.
- The connectives $\rightarrow$ and $\multimap$ represent the type of functions.
- The modality $\diamond$ represent the type of a variable that will eventually become available.

Definition 3.1. Two sequent relations are introduced as follows:

- The linear sequent $\Gamma; \Delta \vdash \mathfrak{e} : \mathfrak{A}$ which is defined through Figure 6,
- The non-linear sequent $\Gamma \vdash \mathfrak{t} : \mathfrak{T}$ which is defined through Figure 5,
- The relation between the two sequents are given by Figure 7.

Here, Γ denotes the unrestricted typing context, and Δ the linear context. The non-linear sequent has access only to the unrestricted context, although the linear sequent has access to both. $\Diamond$

3.2 The Future Type

The future modality $\diamond$ denotes the type of a variable that will become available eventually. Think of $\diamond$ as `Promise` in the sense of Promise/A specification, or `Observable` in the Rx, or the `Future` in Scala/C++.
The `return` function in the toy language is a function that turns an already available variable into a future variable that becomes available immediately. Type of `return` can be represented as $\Box\mathfrak{A} \multimap \diamond\mathfrak{A}$ which corresponds to axiom D of modal logic.
The expression `bind` x `=` $\mathfrak{e}_1$ `in` $\mathfrak{e}_2$ shows the monadic nature of future types. The `then` method of promises is very similar to the `bind` operator. In JavaScript, the function passed to a `then` can return an immediate value or a promise. This is not the case with Rx that ones needs to specify whether its a

`map` transformation or a `flatMap`. The bind operator of our language is more similar to a `flatMap` and has the type $\diamond\mathfrak{A} \otimes (\mathfrak{A} \multimap \diamond\mathfrak{B}) \multimap \diamond\mathfrak{B}$. A `map` operator in contrast has the type $\diamond\mathfrak{A} \otimes (\mathfrak{A} \multimap \mathfrak{B}) \multimap \diamond\mathfrak{B}$.

4 Coq Implementation of LTTT

In this chapter we describe the Coq implementation[1] of the type system introduced in Chapter 3. It has multiple parts that we discuss in the following sections:

- A generic key-value store to be used as the typing context.
- Implementation of the language types $\mathfrak{T}$ and $\mathfrak{A}$.
- Implementation of terms and expressions, $\mathfrak{t}$ and $\mathfrak{e}$, respectively.
- Declarative typing.
- Operational semantics.

4.1 Generic Store

From now on, we say *generic store* or simply *store*, when talking about the data model we implemented in Coq, and *context* when we refer to its role in the type system. The store can be used for purposes other than a typing context.

Definition 4.1. The generic stores are denoted by S in this chapter, which are unordered lists of keys k and values v, defined inductively as follows:

$$S ::= \emptyset \mid S :: \langle k, v \rangle.$$

[1] `https://github.com/aerabi/lttt`

The empty store is denoted by $\emptyset$ and :: is the append operator. $\Diamond$

We would need decidablity of equality on the keys and values of the store. So, we assume the keys are chosen from a class **K** equipped with a reflexive relation $\asymp$ with the following property:

- If $k_1 \asymp k_2$, then $k_1 = k_2$.

The axiom of extentionality is also required for the class of values, **V**.

Definition 4.2. Concatenation of two stores are defined as follows:

$$S \circ S' = \begin{cases} S, & S' = \emptyset, \\ (S \circ S'') :: \langle k, v \rangle, & S' = S'' :: \langle k, v \rangle. \end{cases}$$

$\Diamond$

Lemma 4.3 (Associativity)**.** *For stores* S_1, S_2, S_3*, we have* $(S_1 \circ S_2) \circ S_3 = S_1 \circ (S_2 \circ S_3)$.

Proof. By induction on S_2. $\square$

Definition 4.4 (Membership)**.** By definition, stores are lists of ordered pairs. Membership is defined recursively as follows:

- If $S = S' :: \langle k, v \rangle$, then $\langle k, v \rangle \in S$,
- If $\langle k', v' \rangle \in S'$ and $k \neq k'$, then $\langle k', v' \rangle \in S :: \langle k, v \rangle$. $\Diamond$

The membership is defined so, so that appending a pair with the same key and a different value would overwrite the old record, in a sense:

- $\langle k, v \rangle \in \emptyset :: \langle k, v \rangle$,
- $\langle k, v \rangle \notin \emptyset :: \langle k, v \rangle :: \langle k, v' \rangle$.

Definition 4.5. A key k is said to be a key in the store S, denoted as $\langle k, \cdot \rangle \in S$, iff for some value v we have $\langle k, v \rangle \in S$. $\Diamond$

Lemma 4.6. *Let* B *and* C *be stores,* k *a key, and* v *a value. Then:*

- *If* $\langle \mathsf{k}, v \rangle \in \mathsf{C}$, *then* $\langle \mathsf{k}, v \rangle \in \mathsf{B} \circ \mathsf{C}$,
- *If* $\langle \mathsf{k}, v \rangle \in \mathsf{B}$ *and* $\langle \mathsf{k}, \cdot \rangle \notin \mathsf{C}$, *then* $\langle \mathsf{k}, v \rangle \in \mathsf{B} \circ \mathsf{C}$,
- *If* $\langle \mathsf{k}, v \rangle \in \mathsf{B} \circ \mathsf{C}$, *then either* $\langle \mathsf{k}, v \rangle \in \mathsf{B}$ *or* $\langle \mathsf{k}, v \rangle \in \mathsf{C}$.

Definition 4.7 (Equivalence)**.** We say the store S is a subset of the store S', writing $S \sqsubseteq S'$, iff for all $\langle \mathsf{k}, v \rangle \in S$ we have $\langle \mathsf{k}, v \rangle \in S'$. We say the two stores are equivalent, writing $S \equiv S'$, when $S \sqsubseteq S'$ and $S' \sqsubseteq S$. $\Diamond$

Lemma 4.8. *The subset relation* $\sqsubseteq$ *on the stores is reflexive and transitive. Furthermore, the equivalence relation* $\equiv$ *is reflexive, transitive, and symmetric.*

Definition 4.9. We say a store S is *duplicated*, if it has duplicated keys, recursively defined as follows:

- If $S = S' :: \langle \mathsf{k}, v \rangle$ and $\langle \mathsf{k}, \cdot \rangle \in S'$, then S is duplicated,
- If $S = S' :: \langle \mathsf{k}, v \rangle$ and S' is duplicated, so is S.

If S is not duplicated, we call it *unique-paired*. $\Diamond$

From now on, we would like to call two stores *equal* if they are *equivalent*.

Axiom 4.10 (Extensionality)**.** *Let* B *and* C *be stores. Then* $\mathsf{B} = \mathsf{C}$ *if* $\mathsf{B} \equiv \mathsf{C}$.

Corollary 4.10.1. *Let* B *and* C *be stores. Then* $\mathsf{B} = \mathsf{C}$ *iff* $\mathsf{B} \equiv \mathsf{C}$.

Proof. Follows from reflexiveness of $\equiv$. $\square$

Theorem 4.11 (Commute)**.** *Let* B *and* C *be stores. If* $\mathsf{B} \circ \mathsf{C}$ *is unique-paired, then* $\mathsf{B} \circ \mathsf{C} = \mathsf{C} \circ \mathsf{B}$.

Corollary 4.11.1 (Exchange)**.** *For stores* $\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}$, *if* $\mathsf{B} \circ \mathsf{C}$ *is unique-paired, then* $\mathsf{A} \circ \mathsf{B} \circ \mathsf{C} \circ \mathsf{D} = \mathsf{A} \circ \mathsf{C} \circ \mathsf{B} \circ \mathsf{D}$.

Corollary 4.11.1 shows that the store we implemented can only be used with type systems that support the structural rule of exchange (hence, all except ordered ones).

Lemma 4.12 (Append Commute)**.** *For any store* S*, keys* k, k'*, and values* v, v'*, if* $k \neq k'$*, then* $S :: \langle k, v\rangle :: \langle k', v'\rangle = S :: \langle k', v'\rangle :: \langle k, v\rangle$.

Proof. By writing $S :: \langle k, v\rangle :: \langle k', v'\rangle$ as $S \circ (\emptyset :: \langle k, v\rangle) \circ (\emptyset :: \langle k', v'\rangle)$, realizing that $(\emptyset :: \langle k, v\rangle) \circ (\emptyset :: \langle k', v'\rangle)$ is unique-paired, and then using Corollary 4.11.1 (exchange). □

There are many more lemmas and propositions stated and proved in the Coq implementation. One can find them in `ListCtx.v`.

4.2 Types and Their Reflexivity

The Coq types $\mathfrak{T}$ and $\mathfrak{A}$ that represent non-linear and linear types of the language, implemented based on Figure 4. The listing is presented in Listings 1.

Two fixpoint functions $\mathfrak{T}$`_eq` and $\mathfrak{A}$`_eq` are also implemented to check equality among $\mathfrak{T}$ and $\mathfrak{A}$ types. The definitions of these two functions are also presented in Listing 2. These two functions are essentials for membership check in the store. To verify their soundness, the following lemma is created:

Lemma 4.13. *Both relations* $\mathfrak{T}$`_eq` *and* $\mathfrak{A}$`_eq` *are reflexive.*

```
Lemma 𝔗𝔄_eq_refl :
(forall T, 𝔗_eq T T = true) /\ (forall A, 𝔄_eq A A = true).
```

To prove the lemma, a combined induction scheme is required:

```
Scheme 𝔗_ind := Induction for 𝔗 Sort Prop
with   𝔄_ind := Induction for 𝔄 Sort Prop.
```

```
Inductive 𝔗 : Type :=
  | Unit : 𝔗
  | Void : 𝔗
  | 𝔗mult : 𝔗 -> 𝔗 -> 𝔗
  | 𝔗plus : 𝔗 -> 𝔗 -> 𝔗
  | 𝔗impl : 𝔗 -> 𝔗 -> 𝔗
  | 𝔗ceil : 𝔄 -> 𝔗
with 𝔄 : Type :=
  | 𝔄1 : 𝔄
  | 𝔄0 : 𝔄
  | 𝔄mult : 𝔄 -> 𝔄 -> 𝔄
  | 𝔄plus : 𝔄 -> 𝔄 -> 𝔄
  | 𝔄impl : 𝔄 -> 𝔄 -> 𝔄
  | 𝔄flor : 𝔄 -> 𝔄
  | 𝔄diam : 𝔄 -> 𝔄.
```

Listing 1: Inductive definition of non-linear and linear types in Coq

```
Combined Scheme 𝔗𝔄_ind from 𝔗_ind, 𝔄_ind.
```

With the combined scheme, the proof of Lemma 4.13 is straightforward:

```
Proof.
  apply 𝔗𝔄_ind; simpl; auto; intros;
  try rewrite -> andb_true_iff; try split; auto.
Qed.
```

And hence we get two corollaries:

```
Corollary 𝔗_eq_refl : forall T, 𝔗_eq T T = true.
Proof. apply 𝔗𝔄_eq_refl. Qed.

Corollary 𝔄_eq_refl : forall A, 𝔄_eq A A = true.
Proof. apply 𝔗𝔄_eq_refl. Qed.
```

To use the types $\mathfrak{T}$ and $\mathfrak{A}$ as the value types of the store, modules `module`$\mathfrak{T}$ and `module`$\mathfrak{A}$ are created, inheriting from `ValModuleType` (see Listing 3).

```
Fixpoint 𝔗_eq ( T1 : 𝔗 ) ( T2 : 𝔗 ) : bool :=
  match T1, T2 with
  | Unit, Unit => true
  | Void, Void => true
  | 𝔗mult T1' T1'', 𝔗mult T2' T2'' => andb (𝔗_eq T1' T2') (𝔗_eq T1'' T2'')
  | 𝔗plus T1' T1'', 𝔗plus T2' T2'' => andb (𝔗_eq T1' T2') (𝔗_eq T1'' T2'')
  | 𝔗impl T1' T1'', 𝔗impl T2' T2'' => andb (𝔗_eq T1' T2') (𝔗_eq T1'' T2'')
  | 𝔗ceil T1', 𝔗ceil T2' => 𝔄_eq T1' T2'
  | _, _ => false
  end
with 𝔄_eq ( A1 : 𝔄 ) ( A2 : 𝔄 ) : bool :=
  match A1, A2 with
  | 𝔄0, 𝔄0 => true
  | 𝔄1, 𝔄1 => true
  | 𝔄mult A1' A1'', 𝔄mult A2' A2'' => andb (𝔄_eq A1' A2') (𝔄_eq A1'' A2'')
  | 𝔄plus A1' A1'', 𝔄plus A2' A2'' => andb (𝔄_eq A1' A2') (𝔄_eq A1'' A2'')
  | 𝔄impl A1' A1'', 𝔄impl A2' A2'' => andb (𝔄_eq A1' A2') (𝔄_eq A1'' A2'')
  | 𝔄flor A1', 𝔄flor A2' => 𝔗_eq A1' A2'
  | 𝔄diam A1', 𝔄diam A2' => 𝔄_eq A1' A2'
  | _, _ => false
  end.
```

Listing 2: Implementation of equality relation for non-linear and linear types in Coq

```
Module m𝔊 := ListCtx.ListCtx moduleId module𝔗.
Module m𝔇 := ListCtx.ListCtx moduleId module𝔄.
```

The modules `m𝔊` and `m𝔇` hold the contexts Γ and Δ, respectively, named $\mathfrak{G}$ and $\mathfrak{D}$ in the Coq code.

```
Definition 𝔊 : Type := m𝔊.T.
Definition 𝔇 : Type := m𝔇.T.
```

4.3 Declarative Typing

In this section, we are going to describe declarative typing of the type system implemented in Coq. Coq types `t` and `e` represent terms and expressions, based on Figure 4. Also,

```
Module moduleT <: ValModuleType.

  Definition T := T.
  Definition equal : T -> T -> bool := T_eq.

  Definition eq_refl : forall t, equal t t = true.
  Proof.
    intros. apply T_eq_refl.
  Qed.

End moduleT.

Module moduleU <: ValModuleType.

  Definition T := U.
  Definition equal : T -> T -> bool := U_eq.

  Definition eq_refl : forall t, equal t t = true.
  Proof.
    intros. apply U_eq_refl.
  Qed.

End moduleU.
```

Listing 3: Implementation of non-linear and linear as value module type, to be used in the store as value types

- `DeclarativeTyping`$\mathfrak{T}$ is the implementation of typing rules for terms, based on Figure 5 and represented in Listing 4,
- `DeclarativeTyping`$\mathfrak{U}$ is the implementation of typing rules for expressions, based on Figure 6,
- `DeclarativeTyping`$\mathfrak{TU}$ is the implementation of typing rules featuring lift and lower, to move between non-linear and linear types, based on Figure 7.

The first obvious result of the implementation follows:

Lemma 4.14. *If* $\langle x, \mathfrak{T} \rangle \in \Gamma$, *then* $\Gamma \vdash x : \mathfrak{T}$.

```
Module DeclarativeTypingℑ.

Reserved Notation "G '⊢' t '|' T" (at level 60).

Inductive type : 𝔊 -> 𝔱 -> ℑ -> Prop :=
  | Var : forall Γ x T, (m𝔊.append Γ x T) ⊢ (tid x) | T
  | Unit_I : forall Γ, Γ ⊢ thole | Unit
  | Void_E : forall Γ t T, Γ ⊢ t | Void -> Γ ⊢ tholecase t | T
  | ℑmult_I : forall Γ t1 t2 T1 T2, Γ ⊢ t1 | T1 -> Γ ⊢ t2 | T2 ->
    Γ ⊢ tpair t1 t2 | ℑmult T1 T2
  | ℑmult_E1 : forall Γ t T1 T2, Γ ⊢ t | ℑmult T1 T2 -> Γ ⊢ tprj 1 t | T1
  | ℑmult_E2 : forall Γ t T1 T2, Γ ⊢ t | ℑmult T1 T2 -> Γ ⊢ tprj 2 t | T2
  | ℑplus_I1 : forall Γ t T1 T2, Γ ⊢ t | T1 -> Γ ⊢ tinj 1 t | ℑplus T1 T2
  | ℑplus_I2 : forall Γ t T1 T2, Γ ⊢ t | T2 -> Γ ⊢ tinj 2 t | ℑplus T1 T2
  | ℑplus_E : forall Γ t t1 t2 x1 x2 T1 T2 S, Γ ⊢ t | ℑplus T1 T2 ->
    (m𝔊.append Γ x1 T1) ⊢ t1 | S -> (m𝔊.append Γ x2 T2) ⊢ t2 | S ->
    Γ ⊢ tcase t x1 t1 x2 t2 | S
  | ℑimpl_I : forall Γ t x T S, (m𝔊.append Γ x T) ⊢ t | S ->
    Γ ⊢ tlambda x t | ℑimpl T S
  | ℑimpl_E : forall Γ t1 t2 T S, Γ ⊢ t1 | ℑimpl T S -> Γ ⊢ t2 | T ->
    Γ ⊢ tapp t1 t2 | S
where "G '⊢' t '|' T" := (type G t T).

End DeclarativeTypingℑ.
```

Listing 4: Implementation of non-linear typing rules

Proof. By induction on the definition of membership (Definition 4.4), either one of the following statements are true:

- $\Gamma = \Gamma' :: \langle x, \mathfrak{T} \rangle$, in which case the statement follows from the `Var` rule of Listing 4.

- $\Gamma = \Gamma' :: \langle x', \mathfrak{T}' \rangle$ and $\langle x, \mathfrak{T} \rangle \in \Gamma'$ and $x \neq x'$. In this case, by induction we have $\Gamma' \vdash x : \mathfrak{T}$, which means $\Gamma' = \Gamma'' :: \langle x, \mathfrak{T} \rangle$. Using the append commute lemma (Lemma 4.12), we have

$$\Gamma = \Gamma'' :: \langle x, \mathfrak{T} \rangle :: \langle x', \mathfrak{T}' \rangle = \Gamma'' :: \langle x', \mathfrak{T}' \rangle :: \langle x, \mathfrak{T} \rangle ,$$

and so $\Gamma \vdash x : \mathfrak{T}$ follows once more from `Var`. □

The Coq implementation of this lemma is much less verbose:

```
Lemma contains_𝔗_var : forall Γ x T,
  m𝔊.contains Γ x T -> Γ ⊢ ŧid x | T.
Proof.
  intros. induction H.
  - subst s. apply T𝔗.Var.
  - inversion IHcontains. rewrite -> m𝔊.append_commut_weak.
    apply T𝔗.Var. auto.
Qed.
```

5 LTTT in TypeScript

5.1 The `Promise` Type

The `Promise` type is probably the most natural embodiment of $\diamond\mathfrak{A}$. A promise basically exposes a `next` API, which is identical to the `bind` operator in our toy language. This whole binding business is very linear:

```
const record: Promise<Record> = ...
record.next(r => f(r));
```

After the record is resolved, it is passed into a callback function `f` exactly once. It may be duplicated or discarded inside `f`, but that's another story. But, there are aspects of promises that make them less linear:

- The promises are eager rather than lazy. This results in computations that may be discarded.
- A promise, when calculated, is persisted and can be used many times:

  ```
  record.next(r => f(r));  // first binding, OK
  record.next(r => g(r));  // second binding, also OK
  ```

Although the method chaining interface is linear, the underlying data structure is not.

5.2 The `Observable` Type

With introduction of Rx, the way we coded was changed forever. Nowadays, we wrap all of the logic in the code, in chains of callbacks called transformations. The main type in the Rx implementations is called `Observable`, named after the observable pattern, although it has many fundamental differences. The observables wrap a stream of events, rather than just one, but they contain only one event most of the times. This is why RxJava has created multiple observable types depending on the number of events they contains:

- `Single`. Contains one event.
- `Maybe`. Contains zero or one events.
- `Completable`. Is used to model computations that do not emit any values.

These types are not present in RxJS implementation, yet, but one can enforce an observable there to have exactly one element:

```
const record$: Observable<Record> = ...
record$.pipe(single());  // fails if != 1 item is emitted
```

An observable that emits exactly one element, is a closer resemblance of the type $\diamond\mathfrak{A}$:

- As the observables are lazy, the variables are not computed if not used. So, one cannot discard them.
- By subscribing twice to an observable, the whole pipeline of computation is replayed from the very beginning. This is very similar to linear logic's duplicate entries in the context:

$$\alpha \multimap \beta, \alpha \multimap \gamma, \alpha, \alpha \vdash \beta \otimes \gamma.$$

5.3 Embedding Linear Types in TypeScript

The `Observable` type of RxJS is already a very good model to study linear events, but it is too complicated. We need a more limited and controlled version of an `Observable`. In particular, we need our new monadic type to satisfy the following criteria:

- Wrap exactly one event (just like `Promise` or `Single` of RxJava).
- Expose one binding interface (similar to `Promise`'s `next`), rather than dozens like in `Observable`.
- Be linear (unlike `Promise` that can be used indefinitely).

The key ingredients to create such a type system are the following:

- A class `Lin<`$\mathfrak{T}$`>` that implements $\lfloor \mathfrak{T} \rfloor$, to create an embedded linear type system into TypeScript, that lacks it.
- A class that implements $\diamond \mathfrak{A}$, based on a `Lin`.

The explanations that follow are based on the TypeScript implementation of `Lin`, `LPromise`, and `Single`[1].

5.4 Linear Type of `Lin`

Type `Lin` is implemented as a TypeScript class. One can create a linear variable from a non-linear value by calling on the constructor of `Lin`:

```
const lin: Lin<number> = new Lin(12);
```

And to read the value later, one needs to call on the class method `read`:

```
console.log(lin.read());
```

[1] `https://github.com/aerabi/lttt-typescript`

The `read` method cannot be called more than once; it will through an error on the second attempt. And code linters such as TSLint and ESLint can guard against the variable not being used.
The types `LUnit` and `LZero` are special cases of the type `Lin`:

```
export type LUnit = Lin<undefined>;
export type LZero = Lin<void>;
```

5.5 Eager Type of `LPromise`

As presented in Chapter 3, the modal $\diamond$ exists in the linear fragment of the logic. In our setting, we need to use `Lin` to convert an unrestricted type into a linear one, and then apply the modality on it.
This section explains the type `LPromise<`$\mathfrak{T}$`>`, which is an implementation of $\diamond \lfloor \mathfrak{T} \rfloor$ (so the generic unrestricted type is converted into a linear type under the hood):

```
let lpromise: LPromise<string>;
```

The `LPromise` type wraps a linear value that is not necessarily available immediately. One can create a single from a value or a `Promise`:

```
const lp1 = LPromise.fromLinearValue(lin);
const lp2 = LPromise.fromValue(12);
const lp3 = LPromise.fromPromise(Promise.resolve(12));
```

The values are wrapped into a linear variable under the hood, so all of the 3 introduced variables (`lp1` and `lp2` and `lp3`) are equal.
There is a `bind` method that will read the value of the eventual linear variable and pass it to the callback function:

```
const lpromise = LPromise.fromValue(12);
lpromise.bind(x => expect(x).toEqual(12));
```

And, in contrast to `Observables` and `Promises`, a `Single` cannot be bound twice:

```
lpromise.bind(x => ...);  // first time, OK
lpromise.bind(x => ...);  // second time, error
```

5.6 Lazy Type of `Single`

A `Single` is very similar to an `LPromise`, except for its laziness: The callbacks are only computed when the `exec` method is called.

```
const single = Single.fromValue(12);
single.bind(console.log);  // nothing happens...
single.exec(() => {});     // now, it prints to console
```

5.7 Linear Embedded Language

One can describe the language of the embedded linear types as follows:

$$\mathfrak{A} ::= \texttt{LUnit} \mid \texttt{LZero} \mid \texttt{LPair<}\mathfrak{A}, \mathfrak{A}\texttt{>} \mid \texttt{LFun<}\mathfrak{A}, \mathfrak{A}\texttt{>} \mid \texttt{Lin<}\mathfrak{T}\texttt{>} \mid \texttt{Single<}\mathfrak{A}\texttt{>}.$$

6 Conclusion

6.1 Generic Store

The choice of data model representing the typing context is very important. In an earlier implementation of linear lambda calculus[1] based on the work of Walker [2004], we learned that most of the proofs consist of reasoning about the context rather than the rules themselves.

As an example, Walker had no assumption on the typing context, so the rules were formulated like $\Gamma_1, x : \mathfrak{T}, \Gamma_2 \vdash x : \mathfrak{T}$. By assuming that the elements of context can commute, the rule can be reformulated in an easier form: $\Gamma, x : \mathfrak{T} \vdash x : \mathfrak{T}$.

We used a commutative store and formulated the rules in the latter form to make proofs easier. The commutativity (Theorem 4.11) was achieved with a membership relation (Definition 4.4) used together with an axiom of extentionality (Axiom 4.10).

6.2 QuickChick

We attempted to check validity of the presented type system by QuickChick (Paraskevopoulou et al. [2015]), which is an automated testing tool for Coq. Although all of the necessary infrastructure for using QuickChick (e.g. generators for different types, string representations of them, computable proofs

[1] `https://github.com/aerabi/llc`

of their decidability, etc.) was implemented, our attempt to use QuickChick finally failed as Coq (as of version 8.12) was unable to infer instances for mutually inductive types. The types $\mathfrak{T}$ and $\mathfrak{A}$ are mutually inductive types, defined together. This is also the case for $\mathfrak{t}$ and $\mathfrak{e}$. QuickChick uses Coq's instance inference to infer that a type is `Checkable` if a certain criteria is met. This part was broken in our case.
Instead, we started proving different statements about the type system directly. These statements were later translated into TypeScript unit tests and were tested against the newly introduced async types.

6.3 LTTT in TypeScript

The TypeScript implementation is published as an NPM package called `lttt`[2]. The package exports the following types:

- `Lin`. The linear type that become unavailable after first usage.
- `LPromise`. Analogous to `Promise<Lin<`$\mathfrak{T}$`>>`, it's a type for values that become available eventually, and then can be used only once.
- `Single`. Analogous to `Observable<Lin<`$\mathfrak{T}$`>>` and the `Single` type of RxJava, it's a lazy type for a single value that become available eventually, and then can be used only once.

The type `Single` is specially very useful in practice. After further contribution, the `Single` type will be added to the RxJSx[3] package.
The key difference between `Single` and `Observable` is the the fact that the former wraps only one element, rather than many. This shows how close the $\diamond\mathfrak{A}$ type is to the `Observable`.

[2] `https://www.npmjs.com/package/lttt`
[3] `https://github.com/rxjsx/rxjsx`

7 Future Work

This dissertation has presented the linear-time temporal type theory for event-driven programming. A Coq version of the theory was implemented consisting of the typing rules and an operational semantics. A generic store was implemented in Coq to provide a means of storing typing context and reasoning about it. Some TypeScript types were implemented to represent the linear and future types of the system. So, there are many ways that one can extend this work.

7.1 Extend the Theory to Model Streams

As discussed earlier, the diamond modality $\diamond\mathfrak{A}$ models a single future value, rather than an stream of values. That was the reason the types `LPromise` and `Single` were introduced. But to model async stream of values such as `Observable`, one should extend the notion of diamond modality.

Just creating an stream out of the diamond modalities won't be sufficient as the elements are ordered:

$$\diamond\mathfrak{A}_0 \otimes \diamond\mathfrak{A}_1 \otimes \diamond\mathfrak{A}_2 \otimes \cdots .$$

Here, $\diamond\mathfrak{A}_0$ should resolve before $\diamond\mathfrak{A}_1$, $\diamond\mathfrak{A}_1$ before $\diamond\mathfrak{A}_2$, and so on.

For the sake of example, let's assume the stream consists of two elements.

One can type the stream as follows:

$$\diamond\mathfrak{A}_0 \otimes (\mathfrak{A}_0 \multimap \diamond\mathfrak{A}_1).$$

This reads "$\mathfrak{A}_1$ becomes available eventually after $\mathfrak{A}_0$ became available".
Another approach would be using the binary operators of linear-time temporal logic, e.g. release (see Pnueli [1977]). Let's denote release by $\clubsuit$, then $\psi \clubsuit \varphi$ reads "φ has to be true until and including the point where ψ first becomes true". So $\alpha_0 \clubsuit \neg\alpha_1$ means "α_1 can resolve only after α_0 is resolved".

7.2 Add Compile-Time Typing

TypeScript transpiles into JavaScript, hence the type information are lost in the runtime. The new TypeScript runtime, Deno, might change that, but it's still too soon to bet. Also, TypeScript has no plan of supporting linear/affine types (see Gozalishvili [2017]).
Still, one can access the type data in the compile time. The type system can then use the already available types of `Lin` and `Single/LPromise` and treat them differently.

7.3 Prove Preservation

The preservation lemma is introduced in the Coq implementation as a conjunction, and no proof is presented for it. The proof would require a mutual induction, hence a double preservation lemma for both $\mathfrak{T}$ and $\mathfrak{A}$ should be proved first, and then each of the separate lemmas be derived from it. The proof will consist of case analysis, dealing with the context, and reasoning about substitutions.

Bibliography

Mohammad-Ali A'râbi. aerabi/lttt: DOI initial release, v0.1.0, May 2021. URL https://doi.org/10.5281/zenodo.4749276.

Mohammad-Ali A'râbi. Keeping original value when transforming in RxJS, Feb 2020. URL https://itnext.io/keeping-original-value-when-transforming-in-rxjs-f4650e12c4cf.

John W. Backus. The syntax and semantics of the proposed international algebraic language of the zurich ACM-GAMM conference. In *Information Processing, Proceedings of the 1st International Conference on Information Processing, UNESCO, Paris 15-20 June 1959*, pages 125–131. UNESCO (Paris), 1959.

Mordechai Ben-Ari. *Temporal Logic: Formulas, Models, Tableaux*, pages 231–262. Springer London, London, 2012a. ISBN 978-1-4471-4129-7. doi: 10.1007/978-1-4471-4129-7_13. URL https://doi.org/10.1007/978-1-4471-4129-7_13.

Mordechai Ben-Ari. *Mathematical Logic for Computer Science*. Springer London, 3rd edition, 2012b. ISBN 978-1-4471-4129-7.

P. N. Benton. A mixed linear and non-linear logic: Proofs, terms and models (extended abstract). In Leszek Pacholski and Jerzy Tiuryn, editors, *Computer Science Logic, 8th International Workshop, CSL '94, Kazimierz, Poland, September 25-30, 1994, Selected Papers*, volume 933 of *Lecture Notes in Com-*

puter Science, pages 121–135. Springer, 1994. doi: 10.1007/BFb0022251. URL https://doi.org/10.1007/BFb0022251.

Anton Bershanskiy, Chris Mills, Hamish Willee, and Bruno Ribarić. Promise, Sep 2020. URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise.

Arthur Charguéraud. The locally nameless representation. *J. Autom. Reason.*, 49(3):363–408, 2012. doi: 10.1007/s10817-011-9225-2. URL https://doi.org/10.1007/s10817-011-9225-2.

Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987. doi: 10.1016/0304-3975(87)90045-4. URL https://doi.org/10.1016/0304-3975(87)90045-4.

Valentin Goranko and Antje Rumberg. Temporal Logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, summer 2020 edition, 2020.

Irakli Gozalishvili. Affine types / ownership system · issue 16148 · microsoft/typescript, 2017. URL https://github.com/microsoft/TypeScript/issues/16148.

D. Hilbert and W. Ackermann. *Grundzüge der theoretischen Logik*. Grundlehren der mathematischen Wissenschaften in Einzeldarstellungen mit besonderer Berücksichtigung der Anwendungsgebiete. J. Springer, 1928. URL https://books.google.de/books?id=0UDOAAAAMAAJ.

W. Howard. The formulae-as-types notion of construction. 1969.

Ingo Maier, Tiark Rompf, and Martin Odersky. Deprecating the observer pattern. 01 2010.

Zoe Paraskevopoulou, Catalin Hritcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. Foundational property-based testing. In

Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, volume 9236 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2015. doi: 10.1007/978-3-319-22102-1_22. URL https://doi.org/10.1007/978-3-319-22102-1_22.

Jennifer Paykin. *Linear/non-Linear Types For Embedded Domain-Specific Languages*. dissertation, University of Pennsylvania, 2018. URL https://repository.upenn.edu/edissertations/2752/.

Jennifer Paykin, N. Krishnaswami, and S. Zdancewic. The essence of event-driven programming. 2016.

Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004. ISBN 0262162288.

Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977. doi: 10.1109/SFCS.1977.32. URL https://doi.org/10.1109/SFCS.1977.32.

Emmanuel Polonowski. Generic environments in coq. *CoRR*, abs/1112.1316, 2011. URL http://arxiv.org/abs/1112.1316.

A. N. Prior. *Time and Modality*. Oxford University Press, 1957.

A. N. Prior. *Past, Present, and Future*. Oxford University Press, 1967.

A.N. Prior. *Papers on Time and Tense*. Clarendon P., 1968. ISBN 9780198243229. URL https://books.google.de/books?id=cGOmAAAAIAAJ.

Philip Wadler. A taste of linear logic. In Andrzej M. Borzyszkowski and Stefan Sokolowski, editors, *Mathematical Foundations of Computer Science 1993, 18th International Symposium, MFCS'93, Gdansk, Poland, August 30 - September 3, 1993, Proceedings*, volume 711 of *Lecture Notes in Computer*

Science, pages 185–210. Springer, 1993. doi: 10.1007/3-540-57182-5_12. URL https://doi.org/10.1007/3-540-57182-5_12.

David Walker. *Advanced Topics in Types and Programming Languages*, chapter Substructural Type Systems. In Pierce [2004], 2004. ISBN 0262162288.

Kris Zyp. Promises/a, Feb 2010. URL http://wiki.commonjs.org/wiki/Promises/A.