OLIVER BRAČEVAC

EVENT CORRELATION WITH ALGEBRAIC EFFECTS

EVENT CORRELATION
with
ALGEBRAIC EFFECTS

Theory, Design and Implementation

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

DISSERTATION

zur Erlangung des akademischen Grades eines

DOKTOR-INGENIEURS (DR.-ING.)

vorgelegt von

M.SC. OLIVER BRAČEVAC

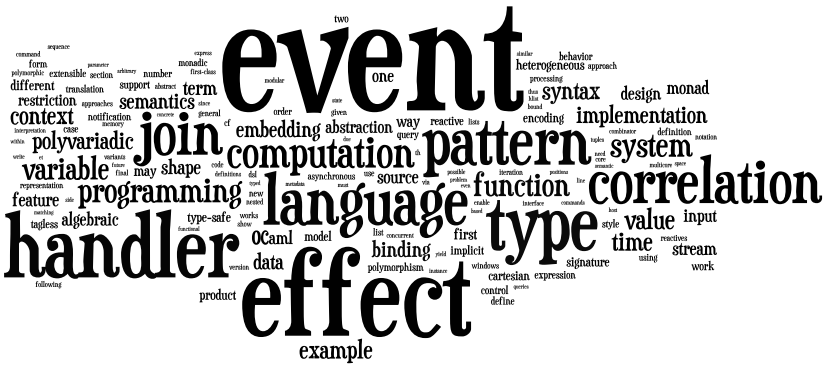
geboren in Offenbach am Main

GUTACHTER:

Prof. Dr.-Ing. Mira Mezini
Technische Universität Darmstadt

Dr. Sam Lindley
The University of Edinburgh, UK

Darmstadt 2019



Oliver Bračevac: *Event Correlation with Algebraic Effects,* Theory, Design and Implementation
Darmstadt, Technische Universität Darmstadt

JAHR DER VERÖFFENTLICHUNG DER DISSERTATION AUF TUPRINTS: 2019

TAG DER MÜNDLICHEN PRÜFUNG: 13.09.2019

ABSTRACT

Modern software systems are increasingly becoming distributed, reactive, and data intensive. In this context, events are an essential abstraction for representing and communicating interesting situations, enabling loosely-coupled and scalable systems. An integral part of these systems is event correlation, which is about computing relations between observed event notifications in space and time, e.g., to programmatically reason about the state of the system, for reacting to changes, for synchronization, for coordination or data processing.

The importance of event correlation is witnessed by the considerable attention it has received from different research communities, under different names. This has led to the emergence of different "families" of event correlation approaches with a sheer staggering amount of specialized correlation semantics and software abstractions. We distinguish between four families: complex event processing, stream processing, reactive programming, and concurrent programming.

We face a rather dire state of affairs: Existing event correlation approaches lack clarity, and most of them are not designed to be customizable or composable. Software systems evolve and their requirements change over time. Yet, no single approach is sufficiently adaptable to meet all requirements. We lack appropriate conceptual models and software abstractions that enable modular, extensible and cross-cutting compositions of features among the different event correlation families, fostering customizability and adaptability.

It is time for a new generation of unifying, highly adaptable and customizable reactive software systems, by means of event correlation "à la carte". That is, we need language designs for event correlation that can express features from all families in a uniform, extensible and freely composable manner.

This thesis makes event correlation "à la carte" a reality. We solve this issue by principled application of functional programming, in particular algebraic effects and effect handlers, which are a modern take on integrating side effects into functional languages in a modular and extensible way. The main statement of the thesis is that algebraic effects and effect handlers are a good programming abstraction for obtaining principled, versatile event correlation systems.

To validate the thesis, we develop Cartesius, the first computational model that captures the essence of event correlation. It has an extensible and customizable semantics based on algebraic effects and effect handlers. We view instances of event correlation as cartesian products over streams of events, where user-defined effect handlers locally customize the control flow of the computation to obtain semantic variants that behave differently. Handlers compose freely and new kinds of correlation features can be added by introducing new user-defined side effects.

Furthermore, we complement Cartesius with PolyJoin, which is an extensible programming language integration of declarative frontend syntax for event correlation systems. It can be deployed independently of Cartesius to provide a type-safe frontend for other event correlation systems in a programming language and subsumes mainstream language-integrated query techniques.

To evaluate our approach, we conduct a survey, comparing the expressivity of Cartesius against representative systems from the above event correlation families. Our findings demonstrate that Cartesius captures the essence of other system's features and is fully customizable and adaptable, to a degree none of the surveyed event correlation approaches offer. Additionally, we evaluate the performance of our approach in terms of synthetic microbenchmarks on common event correlation variants. The benchmarks demonstrate that despite the modular decomposition into an expensive cartesian product and user-defined extensions, Cartesius' design is practical and achieves characteristic time and space complexity of the considered event correlation variants.

Our "à la carte" approach based on algebraic effects and handlers exhibits a degree of unification, fine-grained customization and hybridization that no previous work on event correlation has attained.

ZUSAMMENFASSUNG

Moderne Softwaresysteme werden immer verteilter, reaktiver und datenintensiver. In diesem Zusammenhang sind Ereignisse eine wesentliche Abstraktion für die Repräsentation und Kommunikation von interessanten Situationen, die lose gekoppelte und skalierbare Systeme ermöglichen. Ein integraler Bestandteil dieser Systeme ist die Ereigniskorrelation, d. h., die Berechnung von Beziehungen zwischen Ereignisbenachrichtigungen in Raum und Zeit. Zum Beispiel wird dies zur programmatischen Inferenz des Systemzustands, zur Reaktion auf Änderungen, zur Synchronisation, zur Koordination oder zur Datenverarbeitung genutzt.

Die Bedeutung der Ereigniskorrelation wird durch die beachtliche Aufmerksamkeit belegt, die sie bisher durch unterschiedliche Forschungsgemeinschaften erhielt, teilweise unter anderen Namen geführt. Dies hatte zufolge, dass verschiedene Arten von "Familien" von Ereigniskorrelationsansätzen enstanden sind, jeweils mit einer schier unglaublichen Menge an spezialisierten Korrelationssemantiken und Softwareabstraktionen. Wir unterscheiden zwischen vier Familien: komplexe Ereignisverarbeitung, Datenstrom-Verarbeitung, reaktive Programmierung und nebenläufige Programmierung.

Der Ist-Zustand im Bereich der Ereigniskorrelation ist inakzeptabel: Bestehende Ansätze zur Ereigniskorrelation haben eine unklare Semantik, und die meisten von ihnen sind weder auf Anpassbarkeit noch Komponierbarkeit ausgelegt. Softwaresysteme entwickeln sich ständig weiter und ändern ihre Anforderungen im Laufe der Zeit. Jedoch ist kein einziger Ansatz ausreichend anpassungsfähig, um alle Anforderungen zu erfüllen. Wir haben keine geeigneten mentalen Modelle und Softwareabstraktionen für modulare, erweiterbare und familienübergreifende Kompositionen von Funktionalitäten und Merkmalen aus den bestehenden Familien der Ereigniskorrelation. Dieser Zustand erschwert die Anpassbarkeit und Veränderbarkeit von reaktiven Softwaresystemen.

Es ist Zeit für eine neue Generation vereinheitlichter, hoch anpassungsfähiger und veränderbarer reaktiver Softwaresysteme. Sie muss Ereigniskorrelation "à la carte"' beherrschen können. Das heißt, wir brauchen Programmiersprachenkonzepte, die Funktionalität und Merkmale aus allen Familien der Ereigniskorrelation in einer einheitlichen, erweiterbaren und frei komponierbaren Form ausdrücken können.

Mit der vorliegenden Arbeit wird die Ereigniskorrelation "à la carte"' zur Realität. Wir lösen dieses Problem mithilfe von Prinzipien der funktionalen Programmierung, insbesondere algebraischer Effekte und Effektbehandler. Diese sind eine moderne Form der Modellierung und Integration von Seiteneffekten in funktionale Programmierprachen und stellen hierfür einen modularen und erweiterbaren Ansatz bereit. Die Hauptthese dieser Arbeit ist, dass algebraische Effekte und Effektbehandler eine gute Programmierabstraktion sind, mit deren Hilfe fundierte und vielseitige Ereigniskorrelationssysteme konzipiert werden können.

Um die These zu validieren, entwickeln wir Cartesius, das erste Berechnungsmodell, das die Essenz der Ereigniskorrelation erfasst. Es verfügt über eine erweiterbare und anpassbare Semantik, die auf algebraischen Effekten und Effektbehandlern basiert. Wir betrachten Ereigniskorrelationen als kartesische Produkte über Datenströme von Ereignissen, bei denen benutzerdefinierte Effektbehandler den Kontrollfluss der Berechnung lokal anpassen, um semantische Varianten zu erhalten, die sich anders verhalten. Effektbehandler sind enschränkungslos komponierbar und neuartige Korrelationsfunktionalität kann durch die Einführung weiterer benutzerdefinierter Seiteneffekte hinzugefügt werden.

Darüber hinaus ergänzen wir Cartesius mit PolyJoin, einer erweiterbaren Programmiersprachenintegration von deklarativer Spezifikationssyntax für Ereigniskorrelationssysteme. PolyJoin kann unabhängig von Cartesius verwendet werden für eine typsichere Programmiersprachenintegration anderer Ereigniskorrelationssysteme. PolyJoin subsumiert vorherrschende Techniken zur Programmiersprachenintegration.

Um unseren Ansatz zu evaluieren, führen wir eine Erhebung durch, in der wir die Ausdrucksstärke von Cartesius durch Vergleichen mit repräsentativen Systemen aus den oben genannten Ereigniskorrelationsfamilien untersuchen. Unsere Ergebnisse zeigen, dass Cartesius die Essenz der Funktionalitäten aller anderen Ereigniskorrelationsansätze adäquat erfasst. Es ist wesentlich veränderbarer und anpassbarer als alle untersuchten Ansätze. Zusätzlich evaluieren wir die Leistungsfähigkeit unseres Ansatzes mithilfe synthetischer Mikrobenchmarks, in denen wir häufig genutzte Ereigniskorrelationsvarianten in Cartesius modellieren und messen. Die Messresultate zeigen, dass der Entwurf von Cartesius praktikabel ist, trotz der modularen Zerlegung in ein rechenintensives kartesisches Produkt und benutzerdefinierte Erweiterungen. Wir erreichen eine charakteristische zeitliche und räumliche Komplexität der betrachteten Ereigniskorrelationsvarianten.

Unser "à la carte" Ansatz, der auf algebraischen Effekten und Effektbehandlern basiert, weist einen hohen Grad an Uniformität, Anpassungsfähigkeit und Hybridisierung auf, die keine vorherige Arbeit zur Ereigniskorrelation in dieser Form erreicht hat.

PUBLICATIONS

Some contributions of this thesis appeared previously in the following publications at peer-reviewed conferences and workshops, while others are unpublished manuscripts. Parts of them are used verbatim.

[Bra+19] Oliver Bračevac, Guido Salvaneschi, Sebastian Erdweg, and Mira Mezini. "Type-safe, polyvariadic event correlation." In: *CoRR* abs/1907.02990 (2019). arXiv: 1907.02990. URL: http://arxiv.org/abs/1907.02990.

[Bra+18] Oliver Bračevac, Nada Amin, Guido Salvaneschi, Sebastian Erdweg, Patrick Eugster, and Mira Mezini. "Versatile event correlation with algebraic effects." In: *Proceedings of the ACM on Programming Languages (PACMPL)* 2.International Conference on Functional Programming (ICFP) (2018).

[Bra+16] Oliver Bračevac, Sebastian Erdweg, Guido Salvaneschi, and Mira Mezini. "CPL: A core language for cloud computing." In: *Proceedings of Conference on Modularity (MODULARITY)*. 2016.

[Bra15] Oliver Bračevac. "Temporal correlation patterns – intersecting joins, streams, events and reactive programming." In: *Workshop on Reactive and Event-based Languages & Systems (REBLS) at SPLASH '15*. 2015.

I furthermore (co)authored the following publications, which are *not* part of this thesis:

[Bra+18] Oliver Bračevac, Richard Gay, Sylvia Grewe, Heiko Mantel, Henning Sudbrock, and Markus Tasch. "An Isabelle/HOL formalization of the modular assembly kit for security properties." In: *Archive of Formal Proofs* (May 2018). http://isa-afp.org/entries/Modular_Assembly_Kit_Security.html, Formal proof development. ISSN: 2150-914x.

[Kuc+17] Edlira Kuci, Sebastian Erdweg, Oliver Bračevac, Andi Bejleri, and Mira Mezini. "A co-contextual type checker for Featherweight Java." In: *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*. 2017.

[Erd+15] Sebastian Erdweg, Oliver Bračevac, Edlira Kuci, Matthias Krebs, and Mira Mezini. "A co-contextual formulation of type rules and its application to incremental type checking." In: *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2015.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF DEFINITIONS

LIST OF THEOREMS

PREFACE

If you would be a real seeker after truth, it is necessary that at least once in your life you doubt, as far as possible, all things.

— René Descartes

Nothing could have ever prepared me for this dark, grotesque and sinister maze, which bears the innocuous name PhD (or rather: "Doktor-Ing." in a harsh German voice). It swallowed me whole, tore me apart, etched the soft parts away and reassembled me in new ways, again, and again, and again. I experienced ridicule, praise, loss, reward, hate, love, disappointment, excitement, betrayal, friendship, abuse, kindness and so on. Shrouded in loneliness and despair, no stranger to failure, surrounded by false gods and pretense, a little spark would occasionally shine the way and lead me to the most delightful and rewarding intellectual treasures. At last, I emerge, and you, fellow poor soul, hold in your hands what I have learned during my academic endeavours. So that you may embark on your own journey through the maze and perhaps feel the same delight as I did. The journey is never truly finished, though: My knowledge is always incomplete; anyone who claims otherwise is a fool or a con artist. Bon courage! May our paths cross back in the maze, for there is more treasure waiting to be unearthed. I would never miss the chance to go look for more, again, and again, and again…

Never would I have made it out of the maze without the support of a few people. My heartfelt gratitude goes to

- Mira Mezini my advisor, with a sheer infinite amount of patience. Visionary, pragmatist and speedy writer. I am glad that my time at her group has *woven* some *aspects* of her qualities into my fabric.
- Nada Amin, who is absolutely brilliant, a hacker par excellence and the kindest person I know. You are a true hare (with *DOT*ted fur) and I am just a turtle. See you at the finish line!
- Sebastian Erdweg, a true master of paper writing, debate and strategy. We surely had some heated and intense debates, as if I am from Mars and you are from *Pluto*, but nothing that *incremental* doses of *sugar* and *CoCo*nut ice cream can't cool down.
- Guido Salvaneschi, the madman, who dragged me into the maze. He always lends an ear for my woes and ideas and *reacts* with helpful advice, even if he and I are sometimes distributed among different *loci*.
- Patrick Eugster, who always gives helpful pointers and advice around all things related to event processing, distributed systems and academia. Any words of wisdom he *publish*es, I always *subscribe* to and cherish.

I also would like to thank Sam Lindley, for his encouraging feedback, the invitation to Edinburgh, the introduction to the marvellous algebraic effects community, and for agreeing to be my second examiner.

Furthermore, I would like to thank the remaining members of my PhD committee: Carsten Binnig, Matthias Hollick, and Andy Schürr.

Especially, for feedback and discussions on my work, I thank Jeremy Gibbons, Philipp Haller, Sven Keidel, Oleg Kiselyov, Neel Krishnaswami, Tomas Petricek, Matija Pretnar, KC Sivaramakrishnan, Jeremy Yallop and Philip Wadler. Of course, the list would not be complete without Daan Leijen, with whom I had several enlightening discussions on algebraic effects, Koka and Japanese culture.

I thank KC, Daan, Matija and Tom Schrijvers for inviting me to Dagstuhl Seminar 18172. I also thank Oleg, Aggelos Biboudis and Martin Odersky for inviting me to Shonan Meeting 136. Moreover, I thank Sam, Oleg, Gordon Plotkin and Nicolas Wu for the invitation to Shonan Meeting 146. These were all unique and enlightening experiences, it was truly an honor and a privilege to be part of them.

I thank Neelk, for hosting me at Cambridge and giving me helpful pointers, despite begin so incredibly busy. Special thanks to William Byrd and Daniel Friedman, for whom I had the pleasure to review a draft of the 2nd edition of the Reasoned Schemer [Fri+18]. In short, for all of the above, I can proudly say: "Achievement Unlocked".

Moreover, I thank: my family for being always there for me, even if some of you don't approve of my academic career path, Natalia for enduring me for a great deal of my PhD, my friends (a.k.a. "ihr Vögel"), who unfortunately didn't get to see a lot of me, my former and current coworkers from the Software Technology Group (especially Joscha and Sylvia for reading my thesis draft and providing feedback), our beloved secretary Gudrun Harris, who encouraged me to grow a beard, and the many nice folks from the scientific community I met at conferences and seminars.

Last, but not least, I am grateful for all the constructive, quality reviews my papers received from the anonymous reviewers at conferences. Part of the academic experience is that quality reviews are not always a given, and sometimes hard to not take personally. But overall, I think the peer-reviewing system works and made my work better.

Part of my work is supported by the European Research Council (ERC, Advanced Grant No. 321217 and Consolidator Grant No. 617805) and by the German Research Foundation (DFG, SFB 1053 and SA 2918/2-1).

MY CONTRIBUTION

While the work presented here is my own, research is ultimately a collaborative effort: I had countless discussions and exchanges with my coauthors and colleagues and received valuable feedback from anonymous reviewers at conferences. All of them helped shaping me and my work. To honor their influence, and to engage the reader in the dialogue, I use the royal "we" throughout this thesis.

All of my coauthors had an advisory and editorial role, while the ideas and their execution are my own. Furthermore, during her brief time at my group, Nada Amin helped me working on what eventually became our ICFP 2018 paper [Bra+18], for which I am deeply grateful. We were exploring design alternatives for the Cartesius language in hands-on programming sessions. We reached an important milestone in the form of an early version of the language design. After Nada left, I continued working on the language design on my own. It kept changing and evolving, until it eventually reached its final form presented in the paper and in this thesis. Conceptually and design-wise, this version is vastly different. However, I owe it to Nada that I could reach this point.

Part I

PROLOGUE

Streaming raises difficulties of its own & the question of which technique leads to more modular/easily maintained systems remains open.

— Abelson and Sussman [AS96]

1 INTRODUCTION

Events are related in various ways, by cause, by timing, and by membership.

— *Luckham [Luc01]*

1.1 MOTIVATION

Event correlation is an emergent field which is concerned with the design, theory, and application of computing systems that continuously process unbounded information flows from diverse sources [Luc01]. *Events* are discrete units of information representing incidents that happen in some environment. *Event sources* are the parts of the environment that produce event notifications, e.g., sensors, input devices, or network hosts. *Event correlation* is then *computing relations* between observed event notifications in space and time, for reasoning about the state of the environment and for reacting to changes. Usually, the time gap (*latency*) between the occurrence of events and the reaction should be minimal, ideally close to real-time processing.

Modern software systems are increasingly becoming distributed and event-driven, performing event correlation. These systems profoundly impact our daily lives and come in many guises. For example, by correlating batches or streams of events, computer systems drive cars, trade stocks, give recommendations, compute credit scores, track goods and people. They sift through and make sense of staggering amounts of data, ultimately making sense of the world. This does not only happen on a large scale, but also in the small, in our very hands: consumers operate *billions* of smartphone and tablet devices by touch gestures, which at the software level are correlations of input events reported by the device hardware.

Due to its broad applicability, event correlation has been attracting a lot of attention from different research communities, under different names. This has lead to what we colloquially might call the "tower of babel syndrome": each community approaches the problem with its own terminology and conceptual tools, "reinventing the wheel" over again. These circumstances (1) hinder the exchange of ideas, slowing down progress in the field, and (2) significantly complicate the development of event-driven applications.

A good measure of the tower of babel syndrome are recent works by Cugola and Margara [CM12] and Bainomugisha et al. [Bai+13] that give comprehensive, albeit incomplete surveys of specific research communities which contribute techniques that enable event correlation. For the purpose of this introduction, we give a simplified, coarse overview of existing research communities that have contributed event correlation approaches ("families"). We provide more background in Chapter 2.

1.1.1 *State of the Art Event Correlation Approaches*

COMPLEX EVENT AND STREAM PROCESSING Cugola and Margara [CM12] survey systems papers belonging to (1) the *complex event processing (CEP)* and (2) the *stream processing* family. (1) features sequence patterns, aggregations and timing constraints over unbounded event sequences, e.g., as in the SASE (Diao, Immerman, and Gyllstrom [DIG07]), Cayuga (Demers et al. [Dem+06]) and

Esper [Esp06] systems. CEP systems express event correlation by declarative patterns, which are executed by monolithic, non-programmable runtime systems, i.e., the semantics of patterns is fixed and cannot be adapted by clients. The semantics of CEP languages is based on ad-hoc variants of automata theory [HMU06], extended as needed to suit specific CEP systems.

Furthermore, family (2) describes a class of systems based on *stream-relational algebra*, having query languages that descended from database query languages, e.g., CQL (Arasu, Babu, and Widom [ABW06; ABW04] and Arasu and Widom [AW04]). These systems have a semantics based on infinite streams as time-indexed relations and notions of time windows (Krämer and Seeger [KS09]). Event correlation is specified by joins in the familiar relational algebra notation. The distinction between CEP and stream processing is not crisp and hybrid languages exist, i.e., embeddings of sequence patterns into CQL. However, until now, such hybridization efforts were still an open research problem [CM12]. As we will show, part of this thesis is about enabling such hybridized event correlation approaches in a well-behaved manner.

REACTIVE PROGRAMMING Bainomugisha et al. [Bai+13] focus on programming languages and libraries for *reactive programming*. For example, these include Fran (Elliott and Hudak [EH97]), FrTime (Cooper and Krishnamurthi [CK06] and Cooper [Coo08]), Flapjax (Meyerovich et al. [Mey+09]), Scala.react (Maier, Rompf, and Odersky [MRO10] and Maier [Mai13]), Rx (Meijer [Mei12] and [Rea11]), Elm (Czaplicki and Chong [CC13]) and ReScala (Salvaneschi, Hintz, and Mezini [SHM14]). These approaches are usually embedded in a general purpose programming language and feature a notion of first-class data flow, which can be specified and composed by combinator functions. We argue that reactive languages implement a specialized form of event correlation.

CONCURRENT PROGRAMMING To obtain a wholesome overview, we argue for including a fourth family, which the two surveys above do not consider: *concurrent programming languages*. For example, these include CML (Reppy [Rep91]), Manticore (Fluet et al. [Flu+08]), JoCaml (Conchon and Le Fessant [CL99]), Asynchronous C♯ (Benton, Cardelli, and Fournet [BCF04]), JEScala (Ham et al. [Ham+14]) and Scala Joins (Haller and Cutsem [HC08]). These languages have high-level programming abstractions and pattern notations for *synchronizing* and *coordinating* concurrently running processes. The pattern notation is uncannily similar to some of the event correlation approaches above. Thus, we view synchronization as an instance of event correlation.

We also consider *message passing concurrency* to be part of this family, specifically actor systems and languages (Hewitt, Bishop, and Steiger [HBS73] and Agha [Agh90]), because correlation of messages is just as relevant there. We observe that also within this family, there is a desire for hybridization efforts, e.g., JErlang (Plociniczak and Eisenbach [PE10]) combines actors and join patterns. Similarly, our own work on CPL (Bračevac et al. [Bra+16]), combines message passing and join patterns for programming and deploying cloud services.

1.1.2 *Issues with the State of the Art*

SEMANTIC VARIABILITY AND LACK OF CLARITY The four families above are all different expressions of event correlation. Each family has a staggering number of variants and features, as witnessed by the need to conduct surveys and the size of the surveys. Some of the families even lack clarity in their semantics. For example, the semantics of time windows, which are a characteristic feature of stream processing systems, varies greatly between systems (cf. Dindar et al. [Din+13]). How, then, are developers supposed to have confidence and trust that a given system meets the requirements of an application?

LACK OF COMPOSABILITY WITHIN AND ACROSS FAMILIES Each family of event correlation approaches and each member within a family has specialized abstractions and traits that make it unique. Since their abstractions cannot be easily changed, some applications may not even find any adequate language/system to meet their requirements. So far, no existing approach is flexible enough to support composability and customizability of features from the whole design spectrum of event correlation approaches. This especially holds for composing features *across* families, e.g., it is not straightforward to have synchronization patterns that take timing of processes into account.

ISSUES WITH EVENT CORRELATION SYNTAX We observe that a rather neglected aspect in the field of event correlation concerns *syntax design* for specifying event correlations and the *integration* of event correlation syntax into programming languages. Existing event correlation systems either have no language integration or rely on language-integrated query techniques originally developed for database systems. An example in the first category is Esper [Esp06], which only supports formulating event patterns/queries as strings, which are parsed and compiled at runtime, possibly yielding runtime errors. Examples in the second category are Trill [Cha+14] and Rx.Net [Rea11], which employ the hugely popular LINQ (Meijer, Beckman, and Bierman [MBB06] and Cheney, Lindley, and Wadler [CLW13]) and express event patterns in database join notation.

The reliance on database query notation for correlation patterns has historical and pragmatic reasons: (1) some event correlation approaches have their roots in the database community, featuring similar declarative query syntax, e.g., the CQL language [ABW06]. (2) intuitively, one may indeed think of event correlations as a generalized variant of join queries. (3) language-integrated query techniques for databases are mature and readily usable for implementations of event correlation systems. However, we will show that state of the art technologies for embedding database syntax are inappropriate for general event correlation computations and will develop novel, alternative embedding approaches.

1.1.3 *Conclusion*

The state of the art in event correlation technology imposes uncalled-for limitations on how we may design and implement event-driven applications. We lack appropriate conceptual models and software abstractions that (1) enable cross-cutting compositions of features among the different event correlation

families and at the same time (2) allow for modular and extensible application design. Finally, (3) we lack accompanying declarative frontend syntax for specifying semantically diverse event correlation computations. We need new ways of designing and implementing event correlation systems.

1.2 EXTENSIBLE LANGUAGE DESIGN

The limits of my language means the limits of my world.

— Ludwig Wittgenstein

We view the above problems with state of the art event correlation systems as a language design issue. Language is *the* tool by which we communicate our ideas. The beauty of programming languages lies in being able to communicate ideas so that they become executable. Thus, fundamentally, how we think about and build event correlation systems is a matter of programming language design. The abstractions and linguistic concepts that a programming language offers profoundly influence which ideas are expressible and how easily they are expressible. Nonetheless important are linguistic concepts that support large-scale developments, separation of concerns, reuse, composability, uniformity, extensibility, modularity and variability management.

1.2.1 *Functional Programming*

Functional programming gives us executable mathematical specifications and thus principled and well-behaved software systems.

We address the above concerns by principled use of *functional programming*. It stems from the formal study of mathematical functions, based on the well-known λ-calculus (cf. Church [Chu36] and Barendregt [Bar84]), which expresses computation with function abstraction and function application, capturing the essence of programming languages having first-class, higher-order functions. De-facto, programming languages and their features are studied in formalisms derived from the λ-calculus and this thesis is no different.

Among the hallmarks of functional programming are (1) *purity* and *referential transparency* and along with them (2) the *principled separation of impurity/side effects*. By (1) we mean that function values in the programming language represent mathematical functions, which are free of side effects, thus yielding always the same answer for the same input in any usage context. By (2) we mean semantic models that reconcile language features having side effects (e.g., input/output (I/O), nondeterminism, concurrency, state, exceptions) with the pure mathematical formalism and resulting structured programming abstractions.

1.2.2 *Effects in Functional Programming – An Incomplete History*

Terminology: We use *language feature* and *effect* interchangably.

Research on effects and research on language design are intertwined, since many useful programming language features induce side effects. Mathematicians and computer scientist have been striving for ways of uniformly modeling and structuring the semantics of programming languages. Thus, one is well-advised to study approaches for modeling effects in order to design a language.

The success of monads illustrates how a single abstraction may *uniformly* describe disparate notions of computation.

Perhaps the most influential approach for embedding side effects in functional programs are *monads*, which come from category theory [Awo10]. Moggi [Mog89; Mog91] discovered that many kinds of (effectful) programming language features can be *uniformly* given a denotational semantics in terms of some monad. That is, monads are a good mathematical tool to structure denotational specifications of languages. Shortly afterwards, Wadler [Wad92] would propose using monads as programming abstractions for structuring functional programs.

For example, the Haskell programming language [Has19] deeply integrates monads to include side effects in the sense of proper "world-changing" effects (I/O) as well as custom *user-defined effects*, e.g., threading of state and exceptions. Monads give us a uniform way to embed different kinds of new language features and computational effects into a functional language.

The accomplishments of monads are especially inspiring for this thesis: we face a similar problem of uniformly describing disparate event correlation systems, as well as their features and their composition. Furthermore, event correlation inherently has computational effects, e.g., event sources can asynchronously and concurrently send event notifications. It seems justified and certainly it is possible to model event correlation systems in terms of monads. However, as this thesis will show, traditional monadic design approaches in functional programming are in general inadequate for event correlation (though monads still play an important role).

Monads do not compose well.

The success of monads also spawned controversy, partly because they force a certain programming style and partly because they do not compose well, e.g., it is not straightforward to compose a monad for I/O with a monad for state into a monad exhibiting both effects. Composition of monads is an active area of research and one particularly popular approach are monad transformers (Liang, Hudak, and Jones [LHJ95]), but they have a reputation of being rather unergonomic to use for programmers. This is another point against using monads for event correlation. We argue for simpler-to-use mechanisms that enable cross-cutting compositions among the different event correlation families.

1.2.3 *Extensible Denotational Language Specifications*

Shortly before the publication of monad transformers, Cartwright and Felleisen [CF94] proposed an alternative approach to structuring and composing the denotational semantics of languages and effects, but it did not become as popular as monads and monad transfomers. However, the promise of their work is intriguing and provides superior composability of effects compared to monad transformers:

> ***A complete program is thought of as an agent that interacts with the outside world**, e.g., a file system, and that affects global resources, e.g., the store. A central authority administers these resources. The meaning of a program phrase is a **computation**, which may be a **value** or an **effect**. If the meaning of a program phrase is an **effect**, it is propagated to the central authority. The propagation process adds a function to the effect package such that the **central authority** can **resume the suspended calculation**.* (Cartwright and Felleisen [CF94], emphasis added)

Event correlation computations interact with their environment.

Cartwright and Felleisen give us the important insight that programs are agents interacting with the outside world. This is in agreement with our view that event correlation computations (agents) relate event notifications from external event sources (outside world).

The innovation of their work is that a denotation of a programming language feature (e.g., numbers and arithmetic) can be *modularly composed* with denotations of another language feature (e.g., mutable references on a store/heap) to yield the denotation of a language having both features (arithmetic and references), *without changes* to the two denotations (e.g., dependence on a heap

does not pollute the signature of the arithmetic denotation). Furthermore, their approach enables that expressions written in a given language continue to be compatible in the context of an extended language (e.g., the expression 1 + 2 in the arithmetic language is also an expression in the language for arithmetic and references). Programs interact with a central authority, which gives effect invocations a semantics. Control flow shifts back and forth between the program and the central authority. This is a useful design approach for defining and composing languages.

1.2.4 *Algebraic Effects and Effect Handlers*

Terminology: We interchangably use the terms *operation*, *command* and *effect constructor* for syntactic entities in a programming language that trigger side effects, e.g., **print**, **throw**.

Whereas Cartwright and Felleisen's 1994 paper was overshadowed by the popularity of monads and monad transformers, variants of their ideas would later re-emerge, this time gaining more traction. About nine years later, Plotkin and Power [PP03] introduced *algebraic effects*, investigating how the monadic effects that Moggi considered (a *semantic* account) arise in a programming language by ways of operations (a *syntactic* account) and associated equational theories, leading to a view of effects and computations in terms of universal algebra.

Six years later, Plotkin and Pretnar [PP09] would complement this research, proposing *effect handlers* (or simply handlers) as structured programming abstractions for defining the semantics of effects. While syntactic operations *construct* effects, handlers *deconstruct* effects. Handlers are generalizations of exception handlers that may resume back into the program. Importantly, the algebraic view on effects naturally lends itself to typed *effect interfaces*, enabling *modular abstraction* over language features, and handlers enable *modular instantiation* of the semantics of a language feature (cf. Kammar, Lindley, and Oury [KLO13]).

The interactions between a handler and a computation are similar to the Cartwright and Felleisen approach, where a program interacts with the central authority. Importantly, handlers are decisively *decentral* and apply *locally*, thus being more modular than the monolithic central authority. Effects/language features can thus have a different semantics in different places of a program. These traits are in part reminiscent of Aspect-Oriented Programming (AOP), where effects correspond to joinpoints and handlers to aspect instances (cf. Kiczales and Mezini [KM05]).

$$\frac{\textit{effects+handlers}}{\textit{delimited continuations}} = \frac{\textit{while}}{\textit{goto}}$$

— Andrej Bauer, reported in [KLO13]

Due to their modularity and composability, and their utility as "more structured" forms of delimited control, effects and handlers have been gaining considerable attention. They can be embedded in terms of free monads into a programming language [Swi08; KI15], essentially obtaining an interpreter over an abstract syntax tree (AST) of computations. Importantly, they can also be implemented as native programming language abstractions in compiled languages. For example, these include Eff (Bauer and Pretnar [BP15]), Koka (Leijen [Lei17b]), Frank (Lindley, McBride, and McLaughlin [LMM17]), multicore OCaml (Dolan et al. [Dol+17]), Links (Hillerström and Lindley [HL18]), Pyro (Bingham et al. [Bin+18]) and Helium (Biernacki et al. [Bie+19]). This makes effects and handlers a practical programming abstraction for extensible language design, because integration into the programming language enables compiler optimizations and avoids the abstraction overhead of naïve AST interpreters. We provide more background on algebraic effects and handlers in Chapter 2.

1.3 THE THESIS IN A NUTSHELL

The objective of this thesis is to tame the high complexity and high variability in the event correlation domain by means of extensible language design. Ideally, it should be possible to define event correlation "à la carte", i.e., having a language design that enables free composition of features from all event correlation families and the extension with new features.

THESIS STATEMENT

> *Algebraic effects and effect handlers are a good programming abstraction for* principled *and* versatile *event correlation systems.*

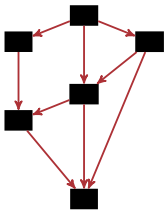
That means, effects and handlers can express language features from all the event correlation families in a structured, uniform, extensible, modular, safe and freely composable manner. We can thus (1) design and implement event correlation systems by defining effect interfaces for language features, (2) write pieces of denotations for the language features via effect handlers and (3) compose these pieces to obtain a system. This includes *novel* compositions *across families* of event correlation approaches.

VALIDATION OF THE THESIS We develop CARTESIUS, the first computational model that captures the essence of event correlation, having an extensible *semantics*. CARTESIUS is based on algebraic effects and effect handlers and contributes a practical programming framework for event correlation system *backends*. To evaluate CARTESIUS, we conduct a survey, comparing the expressivity of CARTESIUS against systems from all event correlation families and measure its performance.

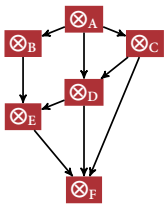
As a secondary contribution, we complement CARTESIUS with POLYJOIN, which is an extensible programming language integration of *declarative frontend syntax* for event correlation systems. It can be deployed independently of CARTESIUS to provide a type-safe frontend for other event correlation systems in a programming language and subsumes mainstream language-integrated query techniques, e.g., LINQ. The deployment of POLYJOIN for CARTESIUS demonstrates the effectiveness of POLYJOIN in adequately supporting a broad range of event correlation systems, given the versatility of CARTESIUS.

1.3.1 *Cartesius: Versatile, à la Carte Event Correlation Semantics*

CARTESIUS captures the essence of the different event correlation families due to a rather unorthodox view. Traditional modeling approaches in the field view event correlation computations as forms of data flow over a graph, where information is processed by nodes and transmitted over the edges. Usually, these models emphasize the edges of the graphs, treating the nodes as black boxes (e.g., the Brooklet calculus for stream processing by Soulé et al. [Sou+10]). We deviate from this view, by emphasizing what happens *inside of the nodes*, i.e., how incoming event data is computationally *joined*.



Traditional view: Emphasis on data flow *topology* (red), nodes are black box abstractions.



Dually, CARTESIUS emphasises nodes and their behavior. Nodes compute variations of cartesian products (⊗).

We propose n-way joins as fundamental units of computation, where $n \in \mathbb{N}$ is the number of incoming edges (input dependencies) of a node in the data flow graph. Cartesius is the *first* extensible language design to uniformly express semantic variants of n-way joins. We argue that the families and individual family members in event correlation differ in the way they join data. Thus, by understanding and mastering joins, we attain a unified view on event correlation and defeat the tower of babel syndrome.

The difference in join behavior is only one half of the story, which leads to a second unorthodox view (giving Cartesius its name): We consider all n-way joins as "degenerate" variants of the n-way *cartesian product*. Our rationale is simple: the cartesian product is a join that is maximally unrestricted, yielding all possible combinations of n-tuples from inputs. Any other join produces fewer combinations of n-tuples.

Conceptually, we may view joins as a filter applied to the cartesian product, in the sense of relational algebra. However, *computationally*, it seems impractical to divide a join into these stages. Cartesian products are not efficient in space and time and one must consider that the join occurs over infinite streams of event notifications, making this division hopelessly impractical. The computation requires storing incoming event notifications forever. Furthermore, some forms of "filtering" can only be explained by means of computational effects, and not by simple boolean tests on data. For example, the way the Join Calculus [FG96] consumes notifications induces a nondeterministic outcome for joins. Thus a more accurate characterization of joins is *cartesian products with filters on data and computational effects*.

The last observation is important, because it provides a way to make the conceptual decomposition of joins practical, which is our key innovation: Computationally, Cartesius decomposes joins into a generic, naïve enumeration procedure of the cartesian product, plus variant-specific "effects/control flow deformations". The point is to avoid exhaustive generation and testing of all the n-tuple combinations *a priori* and let the computation generate/materialize only the combinations that are characteristic of the specific join variant.

We realize this computational interpretation of joins in terms of algebraic effects and handlers, in a pure functional language. We exploit that effect handlers are structured, first-class control abstractions. The overall idea is that *event notifications* are effects, *event sources* are computations inducing the effects, and effect handlers are *event observers*. Programming effort focuses on defining event-observing effect handlers that integrate into a cartesian product computation. Overall, different variants of joins can coexist in an application and handlers can be freely composed, yielding semantic variants of event correlation computations as desired.

Our microbenchmarks validate that this extensible design indeed avoids needless materialization and is practical. Furthermore, alongside a formal semantics for joining and prototypes in Koka and multicore OCaml, we contribute a systematic comparison of the covered domains and features.

1.3.2 *PolyJoin: Lightweight, Portable, Safe, Extensible Event Correlation Syntax*

Whereas Cartesius is primarily concerned with uniform and extensible event correlation *semantics*, PolyJoin is concerned with matters of uniform and extensible *syntax* for *declarative event correlation patterns* and their *integration* into programming languages. Such syntax integrations can be realized by adapting a language specification and compiler, which is non-trivial. Alternatively, integrating syntax can be achieved by *embedding*, which is the approach we take in this thesis. That is, we model event correlation patterns purely in terms of the linguistic concepts of a programming language, an idea popularized by Hudak [Hud96].

Event correlation systems adopted the well-known and intuitive join notations from database query languages, along with the techniques for typed embedding, e.g., LINQ [CLW13; MBB06]. However, these approaches are essentially descendants of the work on monad comprehensions, which are not well-suited for event correlation patterns: The monadic semantics of variable binding in joins induces a statically predetermined selection order among the event sources, which is incompatible with the concurrency/data flow semantics of variable bindings in some event correlation systems. Thus, mainstream embeddings cannot accommodate the entire design space for event correlation systems. Especially, they are unsuitable for Cartesius.

We close this gap with PolyJoin, a novel approach to type-safe embedding of event patterns that retains the familiar join notation, but permits a fully customizable semantics of variable bindings. We effectively embed *pattern matching* into a typed host language with first-class functions, with nothing but combinators, *as if the language had no built-in pattern matching*. We accomplish this by building on the concept of *tagless interpreters* proposed by Carette, Kiselyov, and Shan [CKS09]: By means of a module system or interfaces, we separate pattern combinators and their type signatures (abstract syntax) from their implementation (semantics). This separation makes pattern specifications completely abstract and we can freely re-interpret the meaning of patterns by module implementations. Furthermore, we exploit the type system of the underlying host language to check and enforce that patterns are well-formulated and well-typed. In addition, the pattern syntax is extensible with new syntax forms as needed.

Terminology: We say *object/embedded language* for the language to be embedded, e.g., the language of event patterns. We refer to the target of the embedding as the *meta/host language*.

We model the variables in patterns by reusing the host language's mechanisms for variable bindings as provided by first-class functions. This idea is known as higher-order abstract syntax (HOAS) (cf. Huet and Lang [HL78] and Pfenning and Elliott [PE88]). Especially, we are not burdened with implementing the complicated and tedious machinery of variables, scope and substitutions for the object language, since the host programming language already includes these facilities.

The key idea underlying PolyJoin is viewing the join syntax as a compound binding construct for multiple variables. That means, the join syntax should allow *arbitrary numbers* of variable bindings (any finite number of event sources can be joined) which are at the same time *heterogeneously typed* (input sources may each produce values of different types). The simultaneous binding of multiple variables permits more general interpretations of the join syntax compared to LINQ. It can accommodate the variable bindings of data flow and concurrency languages.

Consequently, the HOAS representation necessitates a function representation of such a "multi-binding" construct in the host language. Such functions are called *polyvariadic* [Kis15], i.e., functions that accept a variable number of heterogeneously-typed arguments. These functions are notoriously difficult to encode in typed mainstream programming languages, because the function types are irregular and require type-level calculations that often push the limits of the type system. An encoding approach must make a trade-off between (1) the sophistication of required type-level calculations, (2) ease of use, and (3) portability to other languages. For example, dependently-typed languages (e.g., Agda [Nor07]) can express polyvariadic functions with ease (e.g., [WC10b; WC10a]), but these solutions are not portable to OCaml, which employs a Hindley-Milner (HM) type system.

We contribute a portable encoding of polyvariadic functions that exploits the syntax structure of the join notation. The essential information to keep track of at the type level is the *shape* of the context of bound variables, i.e., the number of variable bindings and their types. This information can be automatically calculated from a given join pattern without any annotation burden for end users.

Our motto is: More power to system programmers, and less burden for compiler implementers!

Event patterns in PolyJoin neither require dependent types, nor code generation, nor dedicated compiler support. Our approach is practical, type-safe, extensible, lightweight and portable. An implementation in multicore OCaml (Dolan et al. [Dol+17]) is readily usable. However, the ideas are applicable in many modern programming languages that have support for bounded polymorphism and type constructor polymorphism (cf. Pierce [Pie02]).

A generative effect is an effect that is declared at runtime. For example, mutable references are generative effects.

Finally, with the aid of PolyJoin, we contribute a fully polyvariadic implementation of Cartesius with a proper embedded pattern syntax. Cartesius makes heavy use of generative effects, which are still an open research issue in the area of algebraic effects. We provide a practical solution derived from tracking the pattern's variable context shape at the type level. This results in programming abstractions for safe, reusable and modular handlers of polyvariadic generic effects. One particular problem that we successfully solve this way is of more general interest for implementing asynchrony and concurrency libraries, as well as programming language embeddings of process calculi: synthesizing type-safe callbacks for joining n heterogeneous communication partners in the presence of external choice.

1.4 CONTRIBUTIONS OF THE THESIS

The major contributions of this thesis are the design, implementation and evaluation of Cartesius and the PolyJoin embedding. Our specific contributions are:

- The design of Cartesius for "à la carte" event correlation semantics, formulated in a λ-calculus with algebraic effects and handlers, data types, recursion, parametric polymorphism and effect polymorphism.
  - We contribute novel uses of algebraic effects and handlers. Event correlation computations are represented in terms of effect handlers that act as elimination forms of interleavings over infinitary streams of event notifications. Furthermore, we encode control abstractions that enable fine-grained coordination and suspension/resumption of parallel asynchronous computations.

- The PolyJoin embedding for event correlation pattern syntax into mainstream programming languages, which is polyvariadic, statically type-safe, polymorphic, extensible, and modular. We provide its formalization and its encoding in pure OCaml.
  - The systematic analysis of why mainstream techniques for language-integrated query and comprehensions are inadequate for embedding event correlation patterns.
  - The embedding of Cartesius with PolyJoin, yielding a type-safe and fully polyvariadic implementation in multicore OCaml.
  - The evaluation of the PolyJoin version of Cartesius, comparing it against an initial, non-polyvariadic prototype. Our version adds declarative pattern syntax, has exponential savings in code size, supports any arity and significantly reduces programmer effort when defining extensions.
- The evaluation of Cartesius' design and implementation.
  - Evaluation of its expressivity, through a systematic feature comparison with works surveyed across CEP/streaming engines, reactive and concurrent programming languages.
  - Performance evaluations in terms of microbenchmarks (1) microbenchmarks quantifying the effectiveness of the computational cartesian product decomposition. (2) Comparison against Strymonas (Kiselyov et al. [Kis+17]), a state of the art, highly optimizing library implementation of demand-based streams, using multi-stage programming [Kis14].

1.5 STRUCTURE OF THE THESIS

We visualize the thesis structure and chapter dependencies along with a brief summary of chapter contents in Figure 1.1. We recommend having read the chapter dependencies before reading a given chapter.

Chapter 2 gives a programming-language-centric background and overview on the different families of event correlation, the origins/foundations of streams and events as programming abstractions, and algebraic effects and handlers. Chapter 3 presents Cartesius, Chapter 4 presents PolyJoin, Chapter 5 presents the fully polyvariadic multicore OCaml implementation and embedding of Cartesius with PolyJoin, Chapters 6 and 7 present our evaluations, and Chapter 8 discusses conclusions and future research directions.

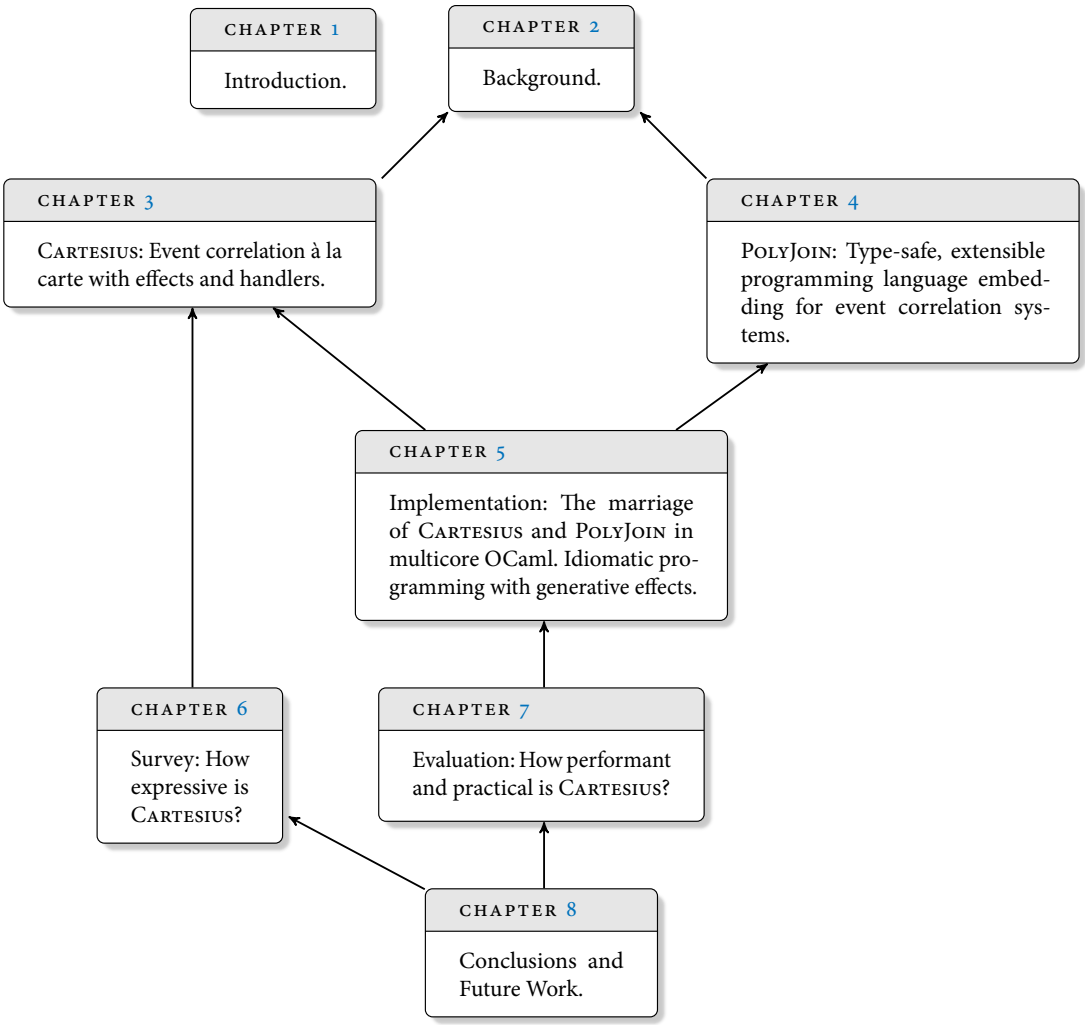


Figure 1.1: Thesis Structure and Overview with Chapter Dependencies.

BACKGROUND

2

SYNOPSIS This chapter gives background information and context on the topic of this thesis. First, we will review foundations and fundamental programming language concepts related to streams and events. The terms "stream" and "event" are overloaded, i.e., they have different interpretations in different contexts, which can make understanding and exchange of ideas quite difficult. Then, we give a high-level overview of the different families of event correlation, followed by an example-driven introduction to algebraic effects and effect handlers. Finally, we introduce tagless interpreters, which are a dependency of the PolyJoin embedding for declarative event correlation syntax (Chapter 4 and 5).

2.1 MODULAR PROGRAMMING WITH INFINITY

Event correlation systems process event notifications sent by event sources. The usual requirement is that they continue to function practically "ad infinitum", i.e., they are always responsive and productive, for any arbitrary, but finite amount of execution time. In mathematics and programming, it is common to abstract over execution times, instead describing such computations by infinite structures that are representable with finite, discrete computers. In this section, we give a brief overview of programming abstractions for infinite structures, e.g., streams, and their relation to events.

2.1.1 *What are Events?*

Events and event-driven programming originate in part from research on input/output (I/O) systems in computer architecture and operating system design. The earliest computer featuring I/O interrupts was the NBS DYSEAC computer in 1954 (cf. Smotherman [Smo89; Smo13]). External input from devices, e.g., mouse or keyboard, would interrupt the execution of a program, triggering an interrupt handler that implements a reaction to the interruption. An alternative to interruption is polling, which was first implemented in the CDC 6600 mainframe computer system in 1964 [Smo89; Smo13]. Polling means that the computer system actively samples external devices. Higher-level software abstractions would be built around these lower-level I/O systems in operating systems and software libraries, e.g., signals and signal handlers, as well as event loops and event handlers/callbacks. These abstractions underlie interactive and communicating applications.

However, the etymology of "event" in computer science is unclear to the best of our knowledge. One must be aware that academics and practitioners conflate with the term "event" the following related, yet different notions:

EVENT Signifies a happening, e.g., a sensor triggers, or a key is pressed. There could be attributes (e.g., a temperature sensor reads 50°C) attached as well as space or time information (metadata), e.g., today at 5am, the temperature sensor read 50°C. Events are a concept, in the same sense that numbers are a concept.

EVENT VALUE A representation/evidence of an event (the concept). We are concerned with programming language representations, e.g., a data type pairing the temperature sensor value and time stamp.

EVENT NOTIFICATION/OCCURRENCE The act of communicating an event value, e.g., invoking an event handler/callback with the event value for reacting to the represented event. We also refer to the communicated event value as the notification.

COMPLEX EVENT A term used in the event processing literature [CM12] for events that are the result of correlating other events, e.g., readings of smoke and temperature events indicate a fire alarm event.

EVENTUALITY A programming abstraction that models the *possibility* of an event, respectively the potential to yield an event value at some point in the future. This is the event notion in concurrent programming languages, e.g., Concurrent ML (CML, Reppy [Rep91; Rep93]). Furthermore, this includes Futures (Baker and Hewitt [BH77]) and Promises (Friedman and Wise [FW78] and Liskov and Shrira [LS88]) for asynchronous programming, e.g., in Scala [EPF13], C♯ [Bie+12], F♯ [SPL11], Dart [G0011], Python [Pyt00], JavaScript. These notions of events are related to the "eventually" modality $\diamond E$ in temporal logics (cf. Paykin, Krishnaswami, and Zdancewic [PKZ16] and Cave et al. [Cav+14]).

For simplicity, we will continue to conflate the first four notions throughout this thesis. We will be careful to distinguish against the last one, because of the focus on the past versus focus on the future.

2.1.2 *Asynchronous and Event-Driven Programming*

Don't call us, we'll call you (Hollywood's Law).

— Sweet [Swe85]

A common implementation technique for asynchronous and event-driven programming libraries are callbacks (a.k.a. continuations/event handlers), essentially non-returning functions that are invoked in case of an event notification from an event source (e.g., a system interrupt). The well-known observer design pattern (cf. Gamma et al. [Gam+94]) is a way to implement this idea in object-oriented (OO) languages. Event sources correspond to "subjects" and callbacks to "observers".

Callbacks and Inversion of Control

Programming with callbacks exhibits *inversion of control.*, i.e., the decision to continue a client computation awaiting events is externalized into the event sources, which execute independently and produce event notifications at their own pace. This makes events a good abstraction for large-scale, distributed systems (e.g., publish/subscribe systems [Eug+03]), because programming with callbacks loosely couples components and works well in conjunction with message passing for remote event notifications.

The following (overly simplified) example in JavaScript sketches a typical use of callback functions to fetch a number of remote resources asynchronously:

```
1 var fetchImages = new Request("http://foo.bar?images")
2 //Outer callback after fetching images
3 fetchImages.ondone = function () {
4   var fetchArticles = new Request("http://baz?articles")
5   //Inner callback after fetching news articles
6   fetchArticles.ondone = function () {  applicationLogic()  }
7   fetchArticles.send()
8 }
9 fetchImages.send()
```

In the first step, we fetch images from a remote host, registering the outer callback that triggers when this step finishes. Then, we fetch news articles from another host, registering the inner callback. Finally, upon completion of the remote fetch, we continue with some further application logic, in the body of the inner callback.

Eventuality with Futures and Promises

A related notion for programming with asynchronous computations as well as parallel threads are first-class representations of *eventuality* (cf. Section 2.1), i.e., data types that represent the result of a concurrent computation, which at some point in the future will be completed. They are well-known under the names Futures (Baker and Hewitt [BH77]) and Promises (Friedman and Wise [FW78] and Liskov and Shrira [LS88]). The following example shows a variation of the previous one, using futures in the Scala language [EPF04]:

```
1 val images: Future[Images] =
2   Future { fetchImage("http://foo.bar?images") }
3 val articles: Future[Articles] =
4   Future { fetchArticles("http://baz?articles") }
5 val render: Future[Unit] =
6   images.map { i => articles.map { a => display(i,a) }
7 otherTask()
8 Await.result(render, Duration.Inf)
9 applicationLogic()
```

Compared to the previous example, this version fetches images and articles in parallel. Lines 1-6 specify concurrent future values, which implicitly spawn threads that execute the code with the `Future { }` delimiter. Importantly, these are non-blocking, and concurrently executed with the task in Line 7. Futures relieve programmers from the burden of manual synchronization and scheduling of tasks. For instance, the `images` and `articles` futures may be executed in parallel, whereas the `render` future (Lines 5-6) depends on the previous two which is specified as a transformation using the combinator `map` on futures. The point of futures is to relieve programmers from the burden of manual synchronization of the concurrent tasks. Synchronization and scheduling of successive tasks is implicitly done in the combinators on futures. At Line 8, the main thread imperatively synchronizes on the `render` future, before continuing with the application logic.

Out of the Callback-Hell with Direct-Style

Programming with callbacks and futures tends to scatter the control flow of the application logic and makes larger code bases hard to comprehend, leading into the infamous "callback hell" (Edwards [Edw09]):

> *An analysis [...] of Adobe's desktop applications indicated that event handling logic comprised a third of the code and contained half of the reported bugs.* (Edwards [Edw09])

Higher-level abstractions have been developed to facilitate developing asynchronous and event-driven programs in "direct style". That means, we can write equivalent programs that appear to have un-scattered, sequential control flow, where the inversion of control is hidden. Many programming languages support direct style with the **`async`**/**`await`** syntax, e.g., C♯ [Bie+12], Kotlin [Fou11], Dart [Goo11], Python [Pyt00], Rust [Fou10] and Scala [EPF13]. For example, we may reformulate the above example in Dart much more cleanly:

```
1 main() async {
2   await fetchImages();
3   await fetchArticles();
4   applicationLogic()
5 }
```

The **`async`** delimiter indicates that the computation is asynchronous. Within its scope, the **`await`** keyword indicates to wait and suspend on the given asynchronous sub-computation and continue upon its completion.

Behind the scenes, the necessary callback logic is automatically calculated. The compiler may mechanically obtain the callback version of a direct-style program, such as the Dart program above, by applying what is essentially the well-known continuation-passing style (CPS) translation (Fischer [Fis72], Reynolds [Rey72], Plotkin [Plo75], and Danvy and Filinski [DF92]), obtaining a representation of the program which essentially looks like the JavaScript version with callbacks. Alternatively, languages supporting forms of first-class continuations enable implementations of **`async`**/**`await`** in terms of combinators, making special transformations by the compiler superfluous. For example, one can define them with algebraic effects and effect handlers (Leijen [Lei17a] and Dolan et al. [Dol+17]).

2.1.3 *Origins of Streams and Coroutines*

The origin of streams can be traced back to the 1960s and is tied to coroutines and later, generators/iterators in the 1970s. Below, we discuss a few relevant developments which happened in short succession within these decades.

In 1963, Conway [Con63] would propose *coroutines* for separable program organization, in the context of writing a compiler for COBOL. Coroutines are essentially a modular abstraction for interacting computations that suspend/resume, transferring control back and forth, e.g., for pairing producers and consumers of data, or general implementations of cooperative multitasking. His characterization of separable program organization foreshadows how dataflow and reactive computations are structured in the modern day:

> *A program organization is separable if it is broken up into processing modules which communicate with each other according to the following restrictions: (1) the only communication between modules is in the form of discrete items of information; (2) the flow of each of these items is along fixed, one-way paths; (3) the entire program can be laid out so that the input is at the left extreme, the output is at the right extreme, and everywhere in between all information items flowing between modules have a component of motion to the right.* (Conway [Con63])

In 1964, McIlroy at Bell Labs suggested what would eventually become Unix pipes for composing programs into execution pipelines, where one program's output is the input to the next:

> *We should have some ways of coupling programs like garden hose–screw in another segment when it becomes [...] necessary to massage data in another way.* (McIlroy, Bell Labs [Bel99])

Pipes are still widely used in contemporary descendants of Unix operating systems. For example, the shell command

```
cat log.txt | grep -i warning | wc -l
```

connects three programs with the pipe symbol (|) in left to right order, first outputting a logfile, then filtering it for lines containing warning messages, and then counting the total number of lines with warnings.

In 1965, Landin [Lan65] would model Algol 60 in terms of Church's λ-calculus [Chu36], where he introduces the term *stream* for infinite lists, which he encodes by a combination of first-class functions and binary pair values. He notes:

> *It appears that in stream-transformers we have a functional analogue of what Conway [...] calls "co-routines" [sic].* (Landin [Lan65])

Here, "stream-transformers" refers to a functions from streams to streams.

Coroutines and streams in the above sense are related to *generators* in the Alphard language (Shaw, Wulf, and London [SWL77]), respectively *iterators* in CLU (Liskov et al. [Lis+77]). Both concepts appeared in the middle of the 1970s. They are restricted forms of coroutines for resumable, on-demand enumerations of collections, where the enumerating computation is subordinate to the demanding computation. It sometimes takes a few decades before good ideas catch on: Fast-forwarding to today, generators/iterators are part of modern programming languages, e.g., Python, C♯, Ruby, and Dart. For example,

```
def fibs():
    a, b = 0, 1        # initial seeds
    while True:
        yield a        # suspend and return to the caller
        a, b = b, a+b # simultaneous assignment of next seeds
```

defines a generator in Python for the infinite sequence of Fibonacci numbers. While the computation loops forever, it is not divergent, because the **`yield`** line suspends the computation and passes control back to the caller, who may decide whether to proceed or not. For example, the program

```
# prints 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89
for i in fibs():
    if i <= 100:
        print(i)
    else:
        break
```

iterates over `fibs` and terminates after printing the first 12 Fibonacci numbers. The caller/consumer of `fibs` is in control of when and how many values it should produce (we say that the whole computation is *demand-driven*). The state of the enumeration is encapsulated in the generator instance, and its life time is tied to the caller's life time. The point is to show that generators are a modular abstraction that separates the strategy of generating data from traversal/consumption strategy in a clean manner.

Generators/iterators have cheaper implementation costs (they share the stack with the consumer) than coroutines, at the expense of expressivity. For example, generators allow nesting of iterations, while it would take proper coroutines to do side-by-side iteration (cf. Section 3.10 of Liskov [Lis93]), where there is no subordination. We will keep returning to coroutines throughout this thesis in the algebraic effects setting, but an in-depth discussion of all their aspects is beyond our scope. We refer to Moura and Ierusalimschy [MI09] for a more thorough treatment of the subject.

Finally, while coroutines and generators are imperative, impure programming concepts, they have a purely functional counterpart in Landin's notion of stream-transformers, i.e., functions from streams to streams. Both have in common that they decouple the generation principle of a potentially infinite structure from its consumption. Consumers drive/demand the generation, as much as they need. We will make Landin's notion more precise further below. But first, we shall see that there are other kinds of streams, which work in the opposite way, in the next section.

2.1.4 *Push versus Pull Streams – External versus Internal Choice*

We called the notion of streams in the previous section *demand-driven*, because the consumer controls and drives the production of the stream's elements. This is also known as *pull-based stream*, or simply *pull stream*. It seems misleading to call such an abstraction a stream, because it might evoke associations with real-world phenomena that do not behave in such a manner. E.g., a river flows of its own accord, irrespective of someone tapping into it to consume some water.

Perhaps unsurprisingly, there is a dual notion to demand-driven/pull streams, called *data-driven/push streams*, where control of the stream lies with the producer. Push streams are closely related to asynchronous and event-driven programming (Section 2.1.2), and the shift from consumer to producer is an instance of inversion of control. Similarly to the case of events, the etymology of "push" and "pull" is unclear, but the terms seem to have been in common use at the latest in the mid-1990s (Franklin and Zdonik [FZ97]).

In essence, push streams generalize asynchronous and event-driven programming from single events to unbounded event sequences. The producer/event source repeatedly invokes registered callbacks/observers with new event notifications, an unbounded number of times. A push stream is a value that repeatedly communicates event notifications, given a subscription/callback. For example, the Reactive Extensions (Rx) library (Meijer [Mei12] and ReactiveX [Rea11]) is built around this idea, and generalizes the OO observer pattern to push streams.

Similar problems related to "callback-hell" (Section 2.1.2) occur in this generalized setting, which is why there have been efforts to extend **async**/**await** to direct-style programs on push streams. For example, the following Dart function implements the higher-order `map` function on push streams:

```
map(stream, f) async* {
  await for (var x in stream) {
    yield f(x);
} }
```

The function definition looks very similar to how one would define `map` in an iterative program over demand-based collections. Analogously to the singular case, the starred **async*** delimiter indicates an asynchronous push stream computation, and the **await for** syntax specifies repeated awaiting and reacting to the next event notification. In this case, we apply the given function parameter `f` to each observed event and emit the result with **yield**. Iterations can be freely nested, e.g.,

```
zipWith(s1, s2, f) async* {
  await for (var v1 in s1) {
    await for (var v2 in s2) {
      yield f(v1,v2);
} } }
```

zips and transforms two push streams by a given function.

However, in contrast to generators/coroutines from the previous section, **yield** *does not* transfer control of the stream iteration to the consumer. For more details, we refer to Haller and Miller [HM19], who formalize a similar extension of **async**/**await** for push streams in the context of the Scala language.

Push streams are better-suited than pull streams for low-latency computations, because of the relative immediacy of reactions to event notifications, whereas pull streams require regular sampling, which might increase latency. On the other hand, pull streams can be more resource efficient, because consumers demand only the data they need. Without further interaction mechanisms between consumer and producer for controlling the flow of data, push streams may overburden a system if their production rate is too high, leading to "backpressure".

Push resp. pull are related to *external choice* resp. *internal choice* in the terminology of process calculi, e.g., CSP (Hoare [Hoa85]). In the former case, the environment of the consumer computation controls when it proceeds. In the latter case, progression is under the control of the consumer. The point is that a mechanism (CPS or continuations, cf. Section 2.1.2), enables a systematic "flip" from internal to external choice, which is why the above examples look like direct-style, iterative programs.

In this thesis, we regard event correlation as push-based computations, which are subject to external choice by their environment. A typical assumption is that event sources (e.g., sensors, network hosts, input devices) act independently and concurrently, in the manner of the "flowing river" analogy from the beginning of this section. The issue of direct-style specifications is just as important in our setting, and we will investigate pathways for obtaining them in a programming language.

2.1.5 *Lazy versus Eager Evaluation*

The examples we thus far considered stateful programming abstractions suitable for integrating notions of streams into imperative programming languages. Here, we consider how functional programming languages integrate them.

Firstly, in the design and implementation of programming languages, one should be aware that there are multiple possible evaluation strategies for applying a function to an argument expression: $f\ e$. Three strategies are in use today: call-by-value (a.k.a. *eager evaluation*), call-by-name (Plotkin [Plo75]), and call-by-need (Ariola et al. [Ari+95]), a refinement of call-by-name. The latter two are synonymous with *lazy evaluation*. Call-by-value fully reduces the argument e to a value, before invoking the function f. Call-by-name invokes f with e unevaluated, so that f may decide whether to demand the argument or not. This strategy is suitable for defining control abstractions as ordinary functions. For example

```
1 ifte :: Bool -> a -> a -> a
2 ifte True  e1 e2 = e1
3 ifte False e1 e2 = e2
```

defines a lazy, three-ary function which implements if-then-else, in Haskell [Has19]. The function does not evaluate the branch parameters, in contrast to the call-by-value strategy. Call-by-need enriches call-by-name with sharing/memoization, so that once evaluated sub-expressions do not require re-evaluation if they occur in other places of a program.

Hughes [Hug89] compares functional programming to *structured programming*:

> *[…] [S]tructured programs are designed in a modular way. Modular design brings with it great productivity improvements. First of all, small modules can be coded quickly and easily. Second, general-purpose modules can be reused, leading to faster development of subsequent programs. Third, the modules of a program can be tested independently, helping to reduce the time spent debugging.* (Hughes [Hug89])

He argues that functional programming languages with higher-order functions and lazy evaluation add new ways of "glueing" software components to the repertoire of programmers, compared to structured programming. Particularly relevant to our discussion is "glueing" stream computations. For example, consider functions from pull streams to pull streams (what Landin called stream transformers, Section 2.1.3):

$$producer : \mathsf{Stream}_{\mathsf{pull}}[A] \to \mathsf{Stream}_{\mathsf{pull}}[B]$$

and

$$consumer : \mathsf{Stream}_{\mathsf{pull}}[B] \to \mathsf{Stream}_{\mathsf{pull}}[C],$$

then function composition naturally connects stream computations

$$consumer \circ producer : \mathsf{Stream}_{\mathsf{pull}}[A] \to \mathsf{Stream}_{\mathsf{pull}}[C],$$

where function composition is defined as

$$consumer \circ producer := \lambda input.\ consumer\ (producer\ input)$$

in terms of first-class functions, in the notation of the λ-calculus [Bar84].

Here, we can clearly see what Landin meant by stream transformers being the "functional analogue" of coroutines (Section 2.1.3). It is lazy evaluation that prevents the uncontrolled unraveling of the possibly infinite *producer input* stream expression in the argument position of *consumer*. The latter demands only as much as it needs from its argument. What had to be ensured by an elaborate, imperative control transfer mechanism, comes for free in a lazy language.

Haskell pervasively incorporates laziness into its design and perhaps most succinctly brings Hughes' point across. For example, the Fibonacci sequence can be defined in Haskell in one line, as a recursive list definition, which is well-known:

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

and the following stream transformation terminates without further ado

```
take 12 (map (\x -> x + 1) fibs)
-- returns [1,2,2,3,4,6,9,14,22,35,56,90]
```

A common argument against laziness is that the evaluation order of expressions can become unpredictable, which badly interacts with language features having side effects. This is why most other programming languages implement an eager evaluation strategy. Haskell does not exhibit these issues, because it strictly separates purity and side effects via monads. Lazy evaluation can be encoded in call-by-value languages, albeit resulting in much less concise programs. One uses a combination of first-class functions for delaying expressions (called thunks) and mutable state for memoization.

As much as laziness yields elegant and concise stream computations, it is tied to demand-based computation. It raises the question whether the same conciseness can be achieved with push-based computations, including event correlation. Due to external choice from their environment (Section 2.1.4), such computations exhibit asynchrony and concurrency effects, where the expectation is that reactions are instantaneous. It seems that eager evaluation is a more suitable evaluation strategy for this domain. However, the next section shows that push and pull may be combined.

2.1.6 *Combining Push and Pull*

Pull-stream computations are driven by consumers and dually, push-stream computations are driven by producers. However, there are hybrid approaches, where neither is the main driver of the computation. A few recent programming language developments have attempted combining push and pull into a single stream type ("push/pull streams"), i.e., the Lula language (Sperber [Spe01b; Spe01a], 2001), Push/Pull FRP (Elliott [Ell09], 2009), and the asynchronous

programming library of F♯ (Petricek and Syme [PS14], 2014). The following is F♯'s definition of a push/pull stream [PS14]:

```
type AsyncSeq τ = Async(AsyncSeqInner τ)
and  AsyncSeqInner τ = Nil | Cons of τ × AsyncSeq τ
```

Thus, push/pull streams are lists enclosed in an eventuality/future, designated by the `Async()` type constructor application in the first line. This type constructor represents an asynchrony effect. The stream tail is delayed similarly to the lazy, demand-driven case, but accessing the tail works differently: the consumer suspends and control is relinquished, until the tail's value becomes available. The tail is computed asynchronously and eventually, the result is pushed, resuming the consumer. Thus, accessing the tail flips from internal to external choice.

The push/pull construction effectively achieves a decoupling of producers and consumers, where both can consume resp. produce at their own pace. Thus, it is possible to have push-based producers and pull-based consumers, which can be written in direct style. Already demanded and resolved prefixes of a push/pull stream are automatically cached, similar to call-by-need.

The enclosing of the stream in an eventuality allows some design flexibility on its laziness resp. eagerness, which depends on the semantics of the eventuality. For example, a lazy future implementation would not start calculating the tail until a consumer demands it, in the fashion of the streams-as-lazy-lists encoding from the previous section. On the other hand, an eager variant of futures would let the producer generate subsequent tails in the background in advance of consumer demand. We refer to Rossberg [Ros07], Chapter 6 for a more detailed discussion of variants of futures.

2.1.7 *Foundations*

Event and stream notions have been studied by mathematicians in the areas of logic, type theory, category theory and (co)algebra, e.g., Jacobs [Jac16], Mendler [Men87], Wadler [Wad90d], Paykin, Krishnaswami, and Zdancewic [PKZ16], Jeffrey [Jef12], Cave et al. [Cav+14], and Hancock, Pattinson, and Ghani [HPG09]. While we will not dive too deeply into these areas, it is useful to consider their findings, since they grant a clean, uniform view on the matter, stripped off from the context and noise of concrete language implementations. For this purpose, we will examine the type structure of events (in the sense of eventuality, Section 2.1.1), as well as push and pull streams in the following. In particular, there are interesting correspondences between their type structure and temporal logics, as well as algebraic effects, upon which we build "à la carte" event correlation systems in this thesis.

Parametric Polymorphism Encoding with (Co)algebras

First, we consider an encoding of push and pull streams in the second-order λ-calculus (System F, Reynolds [Rey74] and Girard [Gir72]), which is a foundational model for programming languages with parametric polymorphism and related notions, e.g., Java generics. These encodings in terms of polymorphism are well-known (cf. Wadler [Wad90d]). They yield a practical way to embed push and pull

streams into programming languages having forms of parametric polymorphism, e.g., ML, Haskell, Scala, Java, and C♯. Recently, these encodings received renewed attention by the programming languages community, in the context of optimizing stream pipelines in OCaml (Kiselyov et al. [Kis+17]).

PUSH STREAMS AS ALGEBRAS/FOLDS The embedding of push streams into System F stems from the embedding of finite lists, which we consider first. Lists yielding values of type T have polymorphic type signatures of the form

$$\mathsf{List}[T] := \forall\alpha.(S_T[\alpha] \to \alpha) \to \alpha,$$

where we write $S_T[\cdot]$ for a type expression with one parameter:

$$S_T[X] := 1 + T \times X,$$

where 1 designates the unit type, which is inhabited by exactly one value, $\langle\rangle$ (zero-ary product), $+$ designates a binary sum (a.k.a. coproduct), and $\times$ a binary product type. $S_T[X]$ describes the *shape* of lists in terms of sums (constructors) and products (constructor parameters), where "X marks the spot" for recursive occurrences of lists, i.e., the right component of the list tail. Category theorists would call $S_T[X]$ a *functor* [Awo10], or more precisely: the encoding of an *algebraic signature* in terms of a functor. We will keep using the intuitive term "shape".

To understand the construction, we expand the definition and apply some isomorphisms (written $\cong$) on types.[1] We obtain

$$\begin{aligned}
\mathsf{List}[T] &= \forall\alpha.((1 + T \times \alpha) \to \alpha) \to \alpha \\
&\cong \forall\alpha.((1 \to \alpha) \times (T \times \alpha \to \alpha)) \to \alpha && \text{by } (A+B)\to C \cong (A\to C)\times(B\to C) \\
&\cong \forall\alpha.(1 \to \alpha) \to (T \times \alpha \to \alpha) \to \alpha && \text{by } (A\times B)\to C \cong (A\to B\to C) \\
&\cong \forall\alpha.\alpha \to (T \times \alpha \to \alpha) \to \alpha. && \text{by } (1\to A)\cong A
\end{aligned}$$

Functional programmers know this type signature as the *fold right* function, resp. the *induction principle* over lists (cf. Hutton [Hut99])! If we know how to fold the empty list (induction basis, α), and assuming we have already folded the tail of the list, we know how to fold head and tail (inductive hypothesis and step, $T \times \alpha \to \alpha$), then we know how to fold all lists into α. That means, lists are encoded in terms of their folds (i.e., algebra over the signature $S_T[X]$) in System F. The construction works for other kinds of data structures, such as trees. All we need to do is adapt the definition of the shape, e.g.,

$$S'_T[X] := T + X \times X$$

defines shapes of binary trees carrying labels of type T in their leaves. The above algebraic encoding of data types is equivalent to the well-known Church encoding [Chu41] in untyped λ-calculi, respectively the encoding by Böhm and Berarducci [BB85] in typed languages, and a typed CPS translation (cf. Meyer and Wand [MW85]) with polymorphic answer type.

What about push streams? They have the same shape $S_T[X]$ as finite lists, and they are also represented in terms of folds, e.g., Kiselyov et al. [Kis+17] propose the *same* algebraic encoding of lists for push streams:

1 These are valid due to the System F types representing formulas in second-order logic, where unit is "truth", function arrows are implication, sum is disjunction and product is conjunction.

$$\mathsf{Stream}_{\mathsf{push}}[T] := \forall \alpha.(S_T[\alpha] \to \alpha) \to \alpha.$$

This might seem puzzling at first, since it is not apparent where the unboundedness of streams appears in the type. For instance, System F is strongly normalizing (cf. Girard, Taylor, and Lafont [GTL89]), i.e., all well-typed terms reduce to a value, which asserts that all folds on lists terminate in this system. The difference is that Kiselyov et al. consider the encoding in a practical programming language, OCaml [Oca19], which has general recursion. The consequence is that the above encoding in terms of universal types does not any longer encode the induction principle over finite lists. That is, there can be values of that type implementing infinite lists, a coinductive interpretation of the list shape. Furthermore, function types are impure in OCaml, so that function calls may induce side effects, e.g., function calls may implicitly interact with a concurrency and asynchrony runtime that sets up continuous processing over infinite lists.

We may intuitively read the type of a push stream $s : \mathsf{Stream}_{\mathsf{push}}[T]$ as follows: for any answer type α, given a callback that accepts the list shape and returns α, s will describe itself by repeated invocation of the callback, eventually producing an α value. Polymorphism ensures that the implementation of s is oblivious to the concrete choice of α, and it is the client supplying the callback that determines the result type.

As stated above, the type of push streams/list folds is essentially a typed CPS translation of lists, which is polymorphic in the answer type α. There are other versions of the CPS translation, which have a global, designated answer type. In that setting, push streams would have the monomorphic type

$$\mathsf{Stream}'_{\mathsf{push}}[T] := (S_T[\mathsf{Answer}] \to \mathsf{Answer}) \to \mathsf{Answer},$$

for some type Answer.

A global answer type enables abstracting over a backend implementation that manages the details of concurrency, asynchrony and continuous processing of streams, even in a language with pure functions. For example, Claessen [Cla99] exploits a CPS translation with global answer type to integrate a cooperative concurrency model into a purely functional language, by setting the answer type to a monad. We could in principle implement a purely functional runtime for push streams on top of his construction (cf. Paykin, Krishnaswami, and Zdancewic [PKZ16]). It is beyond the scope of this thesis to discuss the detailed differences and merits of the design variations of answer types, and we refer to Thielecke [Thi03] for a more thorough treatment.

The Reactive Extensions library [Mei12; Rea11] essentially implements the fold-based encoding for push streams described here in object-oriented languages, with the unit type as the global answer type. We sketch the correspondence of their framework to the construction described here, in the following. In Rx, push streams accept objects implementing the generic $\mathsf{Observer}[T]$ interface, which represents consumers of a stream's event notifications. We define a simplified version of this interface, in terms of a record type:

$$\mathsf{Observer}[T] := \{\mathsf{onDone} : 1 \to 1;\ \mathsf{onNext} : T \to 1\}.$$

That means, observers are objects with methods that receive either the next event notification of type T, via method onNext, or an end of stream notification, via onDone. Push streams in Rx are then objects of the interface

$$\mathsf{Observable}[T] := \{\mathsf{subscribe} : \mathsf{Observer}[T] \to 1\},$$

which can be subscribed to by observers. Since record types are isomorphic to product types, we can derive the correspondence of Rx observables to the fold-based push stream encoding, having the unit answer type:

$$\begin{aligned}
\mathsf{Observable}[T] &:= \{\mathsf{subscribe} : \mathsf{Observer}[T] \to 1\} \\
&\cong \mathsf{Observer}[T] \to 1 \\
&= \{\mathsf{onDone} : 1 \to 1;\ \mathsf{onNext} : T \to 1\} \to 1 \\
&\cong ((1 \to 1) \times (T \to 1)) \to 1 \\
&\cong ((1 + T) \to 1) \to 1 \\
&\cong ((1 + T \times 1) \to 1) \to 1 \\
&\cong (S_T[1] \to 1) \to 1.
\end{aligned}$$

PULL STREAM AS COALGEBRAS/UNFOLDS The above algebraic encoding of push streams in terms of polymorphism and folding is nuanced, because some care had to be taken to reconcile an induction principle over finite lists with the unboundedness of streams. In contrast, the corresponding encoding of pull streams is pleasantly simple, and the dual of the fold/induction principle over finite lists in System F, resulting in the unfold/coinduction principle over finite and infinite lists (cf. Wadler [Wad90d]). The encoding expresses pull streams in terms of existential types:

$$\mathsf{Stream}_{\mathsf{pull}}[T] := \exists \alpha.\alpha \times (\alpha \to S_T[\alpha]),$$

over the same shape for lists, $S_T[X] := 1 + T \times X$, which we used before. Thus, a pull stream consist of some state value (seed) of some type α and an unfold/observation function on the state, where the shape $S_T[\alpha]$ defines the type of possible observations. We say that α and $\alpha \to S_T[\alpha]$ form a coalgebra over the signature encoded by $S_T[\alpha]$. Let us again inline the shape definition:

$$\mathsf{Stream}_{\mathsf{pull}}[T] := \exists \alpha.\alpha \times (\alpha \to (1 + T \times \alpha)).$$

With the list shape, either the stream is terminated (unit case), or it has a head element of type T and a tail of type α, which is corecursively a stream. The existential quantification keeps the internal implementation α of the stream abstract, enforcing an abstraction barrier (cf. Mitchell and Plotkin [MP85]). We can only use state values for passing them to the observation function (i.e., public interface), potentially obtaining further states, and nothing else.

By adapting the shape, we may define different kinds of infinite, demand-based structures. For example, we could have infinite streams that never terminate, by leaving out the unit case:

$$\mathsf{Stream}'_{\mathsf{pull}}[T] := \exists \alpha.\alpha \times (\alpha \to T \times \alpha),$$

or we could have infinite binary trees with labels of type T:

$$\mathsf{CoTree}[T] := \exists \alpha.\alpha \times (\alpha \to T \times \alpha \times \alpha).$$

Existential types and coinduction capture the essence of pull streams within System F. Compared to coroutines and generators, which are stateful (Section 2.1.3), the encoding is purely functional and immutable, i.e., the current state α will not change, if applied to the observation function $\alpha \to S_T[\alpha]$, and state values

resulting from observation are separate copies of the abstract stream state advanced to the next state. Existential types can be represented in a number of ways in real-world programming languages. For example, in OCaml [Oca19], they are representable in term of modules or GADTs [JG08]. Alternatively, they can be represented in terms of objects in OO languages. Indeed, the coalgebraic view on streams formalizes the well-known iterator design pattern in OO languages [Gam+94], which is found in the collection libraries of many programming languages, e.g., Java's iterators [Ora93].

ARE PUSH AND PULL STREAMS DUAL? The encodings of folds and unfolds in terms of universal types and existential types are known to be exact opposites (duals), in the sense of category theory (Wadler [Wad90d]). However, whether push and pull streams are categorical duals is not clear. We observe an asymmetry: push streams in terms of folds require the context of an ambient effect, because we expect that the streams/folds might never terminate, respectively, the environment controls for how long an external source provides data. Otherwise, in the absence of side effects, push streams collapse into a single point, by the induction principle over lists. We do not require such an assumption for pull streams in terms of unfolds. We consider investigating whether there is a categorical duality between push and pull streams an interesting direction for future work.

Least and Greatest Fixpoints – Data and Codata

The above System F encodings of folds/induction and unfolds/coinduction in terms universal resp. existential types are (co)algebraic representations of recursive type structure. We may also have a more direct representation of recursive structure by means of recursive types, i.e., least fixed points and greatest fixed points of types, also known as inductive and coinductive types (Wadler [Wad90d] and Geuvers [Geu92; Geu15]).

Recall the list shape $S_T[X] = 1 + T \times X$. We may characterize finite lists in terms of an inductive type, which is the least fixed point of S_T:

$$\mathsf{List}[T] := \mu X.S_T[X] = \mu X.1 + T \times X.$$

The least fixed point $\mu X.S_T[X]$ is a type U that represents the smallest solution to the recursion over X in $S_T[X]$, so that

$$U \cong S_T[U].$$

Thus, $U = \mu X.S_T[X]$ represents the type of finite lists with elements of type T, and the set of values this type describes is isomorphic to the smallest set of values, which is closed under the following two rules:

$$\mathsf{nil} \in \mathsf{List}[T] \qquad \frac{t \in T \qquad tl \in \mathsf{List}[T]}{\mathsf{cons}\ t\ tl \in \mathsf{List}[T]}$$

Hence, one may model data type definitions in programming languages in a λ-calculus with inductive types, which are equivalent to the algebraic representation with universal types from before. Inductive types made from sums, products and function types model well-founded trees, i.e., trees where all paths from the root to the leaves are finite.

The dual of least fixed points $\mu X.T$ are greatest fixed points, written $\nu Y.T$. They define the greatest solution to the type recursion. Pull streams are the greatest fixed point over the list shape:

$$\mathsf{Stream}_{\mathsf{pull}}[T] := \nu Y.S_T[Y] = \nu Y.1 + T \times Y.$$

This type contains finite and infinite list values and it is equivalent to the coalgebraic formulation with existential types from before. The greatest fixed point representation hides the internal state and observation function of the coalgebraic formulation. Instead, it characterizes streams purely in terms of sequences of possible observations (type T), which can be infinite. Dually to inductive types, coinductive types permit non-well-founded trees.

Inductive and coinductive types can be integrated into typed λ-calculi, while preserving strong normalization (cf. Harper [Har16]). Thus, they are useful in interactive proof assistants (e.g., Coq [Fou19] and Agda [Nor07]) which are based on the Curry-Howard Correspondence (a.k.a. propositions-as-types, cf. Wadler [Wad15]), because strong normalization is a necessary precondition so that the type system forms a consistent logic. (Co)inductive types enable (co)inductive definitions in the logic. Importantly, to ensure strong normalization, we have to restrict the possible positions in the shape where "X marks the spot". Otherwise, least and greatest fixed points degenerate into general recursive types, which make the logic inconsistent. For example, the type $\mu X.X \to X$ admits terms with no normal form. Such problematic definitions can be ruled out by forbidding occurrences of X in negative positions of the type (cf. Wadler [Wad90d]).

Preserving strong normalization is not only useful for interactive theorem proving, but also for programming. Inductive and coinductive types are also known as data and codata, and have been proposed by Turner [Tur04] for typed functional programming languages that are guaranteed total/free of divergence, and yet can express recursive definitions and infinite structures, in a demand-driven programming style. Codata is also believed to be suitable for a principled reconciliation of object-oriented and functional programming (cf. Downen et al. [Dow+19]).

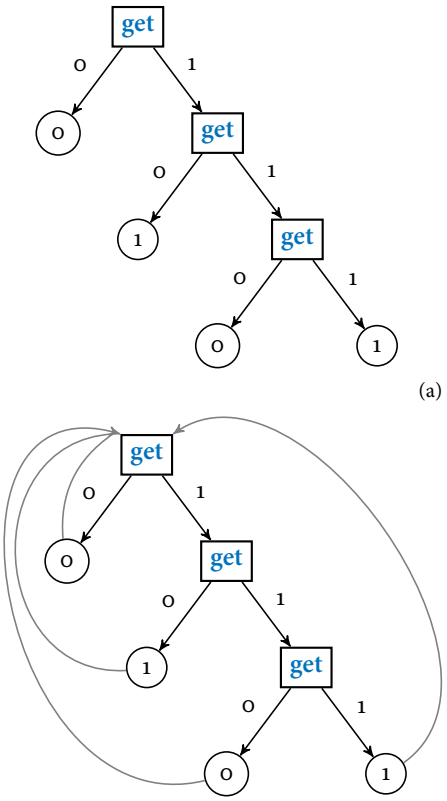
The list and pull stream definition can alternatively be written in data and codata notation, similar to Downen et al's paper. The notation is perhaps more amenable to programmers:

```
data List[A] =
  | nil: 1 → List[A]
  | cons: A, List[A] → List[A]

codata Stream[A] =
  | head: Stream[A] → Maybe[A]
  | tail: Stream[A] → Maybe[ Stream[A] ]

data Maybe[A] =
  | none: Maybe[A]
  | some: A → Maybe[A]
```

The sum types in the least fixed point notation correspond to named constructors, and the products to constructor parameters in the data definition. Dually, a codata definition defines a product of named destructors/observations on the codatum, each returning a sum of outcomes of the observation. In the case of the list shape applied to streams, the observations can be empty, which is why we enclose them in the Maybe type. The codata definition of streams more clearly conveys an interpretation of streams as objects with methods for accessing the head and tail.



$$f : \mathsf{Stream}_{\mathsf{pull}}[2] \to 2$$
$$f(0, \ldots) = 0$$
$$f(1, 0, \ldots) = 1$$
$$f(1, 1, 0, \ldots) = 0$$
$$f(1, 1, 1, \ldots) = 1$$

$$f^{\omega} : \mathsf{Stream}_{\mathsf{pull}}[2] \to \mathsf{Stream}_{\mathsf{pull}}[2]$$

Figure 2.1: Example (by Hancock, Pattinson, and Ghani [HPG09], with adaptations): (a) Representation of a continuous function f with discrete stream domain and discrete codomain, by well-founded trees (least fixpoints). Inner nodes represent effect invocations, tree branches possible outcomes and leaves return values. (b) Lifting of f to a continuous-stream function by nesting well-founded (least fixpoints) and non-well-founded trees (greatest fixpoints).

Stream Transformers as Interleaved Data and Codata

In 2009, Hancock, Pattinson, and Ghani [HPG09] would investigate the foundations of coroutines, streams, and Unix pipes (Section 2.1.3), providing an encoding of stream transformers, i.e., continuous functions from pull streams to pull streams

$$\mathsf{Stream}_{\mathsf{pull}}[A] \to \mathsf{Stream}_{\mathsf{pull}}[B],$$

in terms of interleaving/nesting of greatest and least fixed points from above. By continuous stream functions, they intuitively mean:

> *[…] to find out a finite amount of information about the [output], one need only provide a finite amount of information about the [input]*. (Hancock, Pattinson, and Ghani [HPG09])

This is a reasonable assumption for practical systems, because it rules out stream functions that need to read the contents of an entire infinite stream to produce output, an impossibility.

First, Hancock et al. define a representation of continuous functions having type

$$\mathsf{Stream}_{\mathsf{pull}}[A] \to B$$

in terms of well-founded trees/least fixed points:

$$\mathbf{T}_A[B] := \mu X.B + (A \to X),$$

which, for readability, we alternatively define as a data type definition in a programming language akin to ML/OCaml:

data T[A,B] =
| ret: $B \to$ T[A,B]
| **get**: $(A \to B) \to$ T[A,B]

That means, $\mathbf{T}_A[B]$ is the type of well-founded A-branching trees with leaves carrying B labels. Figure 2.1a graphically depicts a function

$$f : \mathsf{Stream}_{\mathsf{pull}}[A] \to B$$

in terms of the well-founded tree representation $\mathbf{T}_A[B]$. For the example, we set $A = B = 2$, which is the type of binary digits having values 0 and 1. Stream functions of the above type correspond to well-founded trees that have values from B in their leaves (constructor ret, circles in the graphic) and inner nodes marked with the constructor **get**, which have $|A|$-many branches, one for each value in A. We marked the constructor with blue font for emphasis. Intuitively, to compute the value of $f(s)$ for some input stream s, one traverses from the root of the tree to a leaf, as follows: If the current node is a **get**, then we demand the next value x from the stream s of type A, and follow the corresponding branch with label x. Once we reach a leaf ret y, then this is the result, i.e., $f(s) = y$. The inner **get** nodes designate points in the computation of f that read from the argument stream, and the outgoing branches designate the possible outcomes of the read.

Hancock et al. prove that this representation is complete, i.e., all functions $\mathsf{Stream}_{\mathsf{pull}}[A] \to B$ have a representation as a well-founded tree $\mathbf{T}_A[B]$. Conversely, each $\mathbf{T}_A[B]$ represents such a function. The well-foundedness of the trees, which is guaranteed by the least fixed point representation, certifies that the functions are total and continuous. For any input stream, the function result is determined by scanning a finite prefix. In this sense, least fixed points guarantee that *eventually*, a result is produced in a finite number of computation steps.

Furthermore, Hancock et al. define a representation of continuous functions with stream-valued codomains (i.e., Landin's stream transformers, Section 2.1.3)

$$\mathsf{Stream}_{\mathsf{pull}}[A] \to \mathsf{Stream}_{\mathsf{pull}}[B]$$

by nesting a least fixed point in a greatest fixed point:

$$\mathbf{P}_A[B] := \nu Y.\mathbf{T}_A[B \times Y] = \nu Y.\mu X.(B \times Y) + (A \to X).$$

Again, for readability, we may alternatively define this type in a codata definition, in a notation style similar to Downen et al. [Dow+19]:

codata P[A,B] =
| next: P[A,B] → T[A, B×P[A,B]]

That means, continuous stream functions are codata/greatest fixed points, representing non-well-founded trees, "stitched together" from well-founded trees. We can apply the observation next on a stream function representation, which yields a well-founded tree, returning a B *and* the continuation of the stream function. Figure 2.1b graphically depicts an example of that construction, where we lift Figure 2.1a to a stream-valued codomain. A continuous stream function scans finite segments of the input stream, and eventually computes the next value of the output stream, along with its continuation, which is again a $\mathbf{P}_A[B]$ value. Note that in the general case, unlike the example, the continuation in the leaves is history sensitive, i.e., the behavior of the returned continuation may depend on the path taken from the root to the leaf.

If we program stream computations with this presentation in a total functional language, then the interleaving of greatest and least fixed points guarantees productivity: assuming the input stream is productive, then we always ensure that eventually, the stream function produces its next value, when demanded. Recent works have investigated the Curry-Howard correspondence between linear temporal logic (LTL) and functional reactive programs, e.g., Jeffrey [Jef12] and Cave et al. [Cav+14]. Cave et al. exploit this property of interleaved greatest and least fixed points, using similar stream constructions as shown here, for live and fair reactive programs. However, for interpreting types as LTL propositions, they include a modality $\circ T$ in the type system, for distinguising between now and the next time step.

Finally, the constructions by Hancock et al. shown here are related to algebraic effects (Plotkin and Power [PP03]) and effect handlers (Plotkin and Pretnar [PP09]), where the latter work was published in the same year as Hancock et al.'s paper. General computations with effects are also modeled by well-founded trees as depicted in Figure 2.1a. Hancock et al. use a specialized instance, where there is one effect operation **get** for reading from the input stream. In the algebraic setting, other effect operations may occur in the inner nodes of the trees (cf. Lindley [Lin14]). While Hancock et al.'s and Plotkin's line of work are independent, they both rely on a older, well-known construction: the well-founded trees are free monads (cf. Awodey [Awo10]). We will introduce algebraic effects in Section 2.3.3.

Revisiting Push/Pull Streams

The push/pull stream construction from Section 2.1.6 is characterized by a greatest fixpoint and a modality $\Diamond[\cdot]$:

$$\mathsf{Stream}[T]_{\mathsf{push/pull}} := \nu Y.\Diamond[1 + T \times Y],$$

where the modality represents some notion of eventuality that delays the result, or has a side effect. For example, we could define $\Diamond$ to mean the well-founded tree construction $\mathbf{T}_A[B]$ by Hancock et al. [HPG09], a Future, a thunk, or an event source accepting a callback. The point is that the $\Diamond[\cdot]$ marks the position in the stream type where the consumer might suspend, and control flips from internal to external choice (Section 2.1.4).

The combination of sequencing by lists/streams and an eventuality in the push/pull streams shows why we should be careful to not conflate *event values* and *eventuality*, as we insisted in Section 2.1.1. The profound difference between these notions is revealed in the $\mathsf{Stream}[T]_{\mathsf{push/pull}}$ definition. This type intuitively states that the stream "potentially produces the next event value", i.e., type parameter T represents event values, but not an eventuality. If we conflated the two, we could define a stream type as a potentially infinite sequence of eventualities:

$$\mathsf{Stream}'[T]_{\mathsf{push/pull}} := \nu Y.\, 1 + \Diamond[T] \times Y,$$

which has different qualities. For instance, a stream of the former type ensures that the production order of event values matches the observation order by the consumer, whereas the latter does not give this guarantee. It might matter to an application to be able to discern the order. For instance, some event correlation systems have sequence patterns, e.g., SASE [Agr+08], and the second stream type seems inadequate for this purpose. For this reason, our Cartesius model in Chapter 3 relies on a variant of the order-preserving push/pull stream type.

Finally, it is worth noting that if we instantiate the modality in the abstract push/pull stream type (leaving out the 1 case for nil) to the well-founded trees $\Diamond[B] := \mathbf{T}_A[B]$ for some type A, then the push/pull stream type becomes a stream transformer in the sense of Hancock, Pattinson, and Ghani [HPG09]:

$$\nu Y.\Diamond[T \times Y] = \nu Y.\mathbf{T}_A[T \times Y].$$

That means, a stream representation becomes a representation of a transformation of one stream into another stream, by adding a side effect.

2.1.8 *Summary*

The history of events and streams in systems goes back to the 1950s to 1960s, and the quest for programming languages representations to handle their unboundedness is just as old. Care should be taken with the terminology "event", which might conflate event values as an evidence of a past happening and eventualities, i.e., the potential of a happening in the future. These notions result in quite different programming language representations.

Events and stream representations in programming languages are fundamentally instances of dual notions: algebra/coalgebra, least/greatest fixpoint, data/codata, and induction/coinduction each express the same ideas in different form. The classic distinction in the design space for stream types is (1) pull-based streams: represented as greatest fixpoints, unfolds, existential types. (2) push-based streams: represented as least fixpoints, folds, universal types, generalizing event-driven programming to multiple notifications. However, push/pull streams reconcile the two notions into one, by means of a greatest fixpoint construction, and an eventuality representing an effect.

Programming languages supporting extensible effect systems and representations of algebras and coalgebras, are suitable for expressing all of these different representations of events and streams.

2.2 EVENT CORRELATION FAMILIES

In this section, we provide examples of concrete systems and languages from the different event correlation families: Complex event processing (CEP), stream processing, reactive programming, and concurrent programming. The point is to show that they all express related ideas and exhibit common themes in design and functionality.

2.2.1 *Complex Event Processing*

Complex Event Processing (CEP) features sequence patterns, aggregations and timing constraints for events [DIG07; Dem+06]. For example, in the SASE+ system (taken from Agrawal et al. [Agr+08]), the following example pattern

```
1 PATTERN SEQ(Stock+ a[], Stock b)
2 WHERE skip_till_next_match(a[], b) {
3   [symbol]
4   and a[1].volume > 1000
5   and a[i].price > avg(a[..i-1].price)
6   and b.volume < 0.8*a[a.LEN].volume }
7 WITHIN 1 hour
```

reports each stock that rises monotonically (Line 5, for all indices $i > 0$), *aggregating* the monotonic event sequence in `a[]`, where the rise is ended by an abrupt decline (Line 6). The scope of the pattern is delimited by a *sliding window* of 1 hour duration (Line 6). The `SEQ` combination of the `a[]` event sequence and the terminating `b` event is called a *complex event* (Line 1). All `a` and `b` events should refer to the same stock (Line 3).

Event correlations in CEP systems are expressed in a system-specific pattern language, on top of monolithic non-programmable runtime systems, i.e., the semantics are fixed and cannot be adapted to specific application/domain needs. Furthermore, systems such as SASE+ above offer no proper programming language integration. This aspect of event correlation feels like a throwback to the 1980s, where integrations of database and programming languages took their first steps. For example, Copeland and Maier [CM84] were among the first to integrate database and object-oriented languages. We have more to say on programming integration of event correlation syntax in Chapter 4.

The semantics of CEP languages are ad-hoc variants of automata theory (cf. Hopcroft, Motwani, and Ullman [HMU06]), extended as needed to suit specific CEP systems. These circumstances make it difficult to compare systems. For instance, are the automata in SASE+ as expressive as the automata in the Cayuga system (Demers et al. [Dem+06]), or TESLA (Cugola and Margara [CM10])? What makes answering these questions difficult is the lack of complete formal systems specifications in published research.

Furthermore, automata/state machines are not a good way to think about and specify correlations for end users and programmers, because of operating at a low level of abstraction. It is easier to think in declarative pattern languages, which is why event correlation systems end up reinventing similar syntax designs. Yet, automata are good intermediate representations for compilation and

optimizations, but as mentioned above, the variations in features and automaton structure in systems makes it difficult to "glue" (in the spirit of Hughes [Hug89], cf. Section 2.1.5) features and traits of different automata structures from CEP systems together.

2.2.2 *Stream Processing*

Stream processing systems are descendants of database systems, and accordingly have foundations based on variants of relational algebra adapted to streaming and time. For instance, the CQL language (Arasu, Babu, and Widom [ABW06; ABW04] and Arasu and Widom [AW04]) has a notion of "streams", which is different from the stream notions we discussed in Section 2.1. In such "stream-relational algebra" systems, streams are possibly infinite multisets (bags) of tuples, which carry a timestamp, i.e., infinite multisets of event values in our terminology. Relations in this context are functions from timestamps to relations in the sense of relational algebra, i.e., time-indexed relations. The semantics defines operators to convert from streams to relations and back.

The following example from [ABW04] defines a CQL query:

```
1 Select Istream(Close.item_id)
2 From Close[Now], Open[Range 5 Hours]
3 Where Close.item_id = Open.item_id
```

This query correlates streams `Close` and `Open` representing end and start events for auctions, by a join. It reports all auctions that are closed within a time window of 5 hours after their opening.

The distinction between stream processing systems and CEP systems is not crisp and hybrid languages exist, i.e., embeddings of sequence patterns into CQL. However, such hybridization, or "glueing" efforts are still an open research problem (cf. Cugola and Margara [CM12]). One other feature they have in common is time data attached to event values and notions of time windows, which restrict the temporal scope of a query. Windows are not well-understood in these domains, and their behavior can vastly differ in concrete implementations, a problem which has been studied by Dindar et al. [Din+13] and Botan et al. [Bot+10].

2.2.3 *Reactive Programming*

This family of event correlation approaches constitutes languages and systems for abstracting over state and dynamic change by static, immutable descriptions. That is, time-changing behavior is represented in terms of the possible history of changes over time (called behaviors or signals). Such first-class behaviors are composed from primitive behaviors and combinators. Essentially, the stream notions we discussed in Section 2.1, both in the demand/pull and push style are used as programming language representations for these change histories.

> *[…] it is the user's temporal existence that imposes state on the system. If the user could step back from the interaction and think in terms of streams of [values] rather than individual transactions, the system would appear stateless.*
>
> — Abelson and Sussman [AS96]

Reactive programming stems from dataflow programming in Lucid (Wadge and Ashcroft [WA85]) from 1974. Lucid's notion of dataflow corresponds to demand-based streams which are declared by equations. Once again, an example is in order (from [WA85]), defining the ever-delightful Fibonacci sequence in Lucid:

```
fibs = 1 fby (fibs + (0 fby fibs))
```

The Fibonacci sequence is defined by the above recursive dataflow equation, where `fby` designates what the next value in the sequence should be: The sequence *fibs* starts with the constant 1, which is followed by the sum of the next value of *fibs* and the sequence starting with the constant 0 followed by the next value of *fibs*.

Importantly, the dataflow notation exhibits a common trait found in reactive languages, which is the two-fold nature of variables and expressions. We have that the variable *fibs* represents both (1) the entire Fibonacci sequence and (2) an individual, anonymous value inside the sequence, which is used inside an expression with arithmetic and number constants to define the values in the sequence. This two-fold nature of variables will become important for our PolyJoin embedding for extensible event correlation syntax embedding into programming languages (Chapter 4).

After Lucid, subsequent developments resulted in *synchronous dataflow*, such as in Esterel (Berry [Ber00], 1984) and Lustre (Caspi et al. [Cas+87], 1987), which are tailored to real-time processing and used for control software in avionics and signal processing in hardware. Synchronous dataflow adds clocks into the dataflow model, so that streams are regularly sampled by a clock. The dataflow equations would now denote functions from time to a domain of values, instead of plain sequences. These languages work under the synchrony assumption: reactions to input changes are instantaneous.

In 1997, functional reactive programming (FRP, Elliott and Hudak [EH97]) would marry synchronous dataflow and lazy, higher-order functional programming within the Haskell language, for interactive applications, GUIs, games and simulations. It features signals, which correspond to the functions from time to values in synchronous dataflow, and events for discrete changes and switching of signals. Perhaps confusingly, FRP conflates events with sequential streams of event values tagged with time as metadata, in our terminology from Section 2.1.1. FRP has a denotational semantics, where time is continuous, yielding a resolution-independent specification of time-changing behavior. This is not to be confused with the continuous stream function characterization from Section 2.1.7, though any reasonable implementation on a digital computer would certainly fullfill the latter continuity requirement as well.

The denotational FRP semantics leaves it open how to implement it on discrete computers. Its implementations may exhibit time and space leaks, i.e., sampling a signal might trigger a cascade of expensive computations and too much history of past values might be remembered, exhausting memory. This has led to other formulations of restricted FRP forms. For example, Yampa (Nilsson, Courtney, and Peterson [NCP02], 2002) re-forumlates FRP in Haskell using the concept of arrows (Hughes [Hug00]). FRP formulations that avoid leaks in implementations are still an active area of research (cf., e.g., Krishnaswami [Kri13]).

Perhaps confusingly, functional reactive programming is also used to name integrations of discrete dataflow into languages with first-class functions, which do not share the continuous semantics of FRP. These kinds of reactive programming approaches have semantics based around propagating changes along a dependency graph representation of the dataflow. One example of these approaches is FrTime (Cooper and Krishnamurthi [CK06], 2006), which brings dataflow to call-by-value languages (Scheme), and later into the browser, in the form of FlapJax (Meyerovich et al. [Mey+09], 2009).

The classic FRP approach is demand-driven, and thus has the drawbacks associated with that style, e.g., latency issues, which we discussed in Section 2.1.4. To address these shortcomings, Elliott [Ell09] proposed in 2009 a reformulation of FRP, combining data- and demand-driven processing (push/pull FRP), which we discussed in Sections 2.1.6 and 2.1.7.

Reactive extensions (Meijer [Mei12] and ReactiveX [Rea11], 2012), which we discussed in Section 2.1, proposes a discrete reactive programming model based on the observer pattern and push-based streams.

ELM (Czaplicki and Chong [CC13], 2013) reconciles discrete, asynchronous computation with the FRP signals model, by relaxing the synchronicity of computation and pipelining event computations. The signals dependency graph can be broken up into separate synchronous subgraphs by programmers.

REScala (Salvaneschi, Hintz, and Mezini [SHM14], 2014) is a variant of reactive programming in Scala with a graph-based semantics, and has been used to study the marriage of concurrency and distribution with reactive programming (Drechsler et al. [Dre+14; Dre+18]).

As mentioned in the beginning, notation-wise, reactive languages share the lucid-style specifications of dataflows in terms of variables representing both the entire time-changing value and its current value. For illustration, the following FrTime example

```
(lift-strict (λ (y z) (/ y z)) (posn-x mouse-pos) (width window))
```

joins the mouse's absolute `x` coordinate signal with the `width` signal of the GUI window into a signal computing the relative coordinate. Changes of the values of the input signals cause the joint signal's value to be automatically re-computed.

We argue that the way dataflows are joined/merged into compound dataflows in reactive programming is a specialized form of event correlation, in the sense that changes in signals are representable as discrete event notifications and their combination in compound dataflows is a correlation. Typically, only the most recent values of the input signals are correlated. It is not obvious how the richer join features of the CEP/stream domains translate into respective reactive programming notions. However, the evident similarity in notation compared to correlation patterns, resp. stream joins cannot be ignored.

2.2.4 *Concurrent Programming*

Concurrent programming languages feature declarative patterns for synchronizing and coordinating processes, and for selective communication. The following example in the JoCaml language (Mandel and Maranget [MM14] and Conchon and Le Fessant [CL99]) defines a *join pattern* for discerning two interesting situations:

```
1 def wait() & finished(r) = reply Some r to wait
2 or  wait() & timeout()   = reply None to wait
```

The pattern synchronizes on three communication channels `wait`, `finished` and `timeout`. Either a consumer's `wait` message coincides with a producer's `finished` message (top pattern) or the producer takes too long (bottom pattern), where a `timeout` occurs first. This is a variant of correlation on communication channels using pattern matching. Importantly, the pattern matching cases are concurrently active, and do not have a sequential, fall-through semantics unlike standard pattern matching or switch statements in general-purpose programming languages.

The foundations of concurrent programming languages stem from the π-calculus by Milner [Mil99], which models concurrent and mobile processes communicating via channels and rendezvous points. Concurrent ML (CML,Reppy [Rep88; Rep91; Rep93]) is a well-known implementation of the π-calculus, having first-class, synchronous communication events (eventuality in our terminology from Section 2.1.1) that can be composed by combinators. Synchronous means that a communication endpoint suspends until matched with another communication partner (rendezvous), before continuing. There are also designs that integrate asynchronous communication into CML, e.g., the work by Ziarek, Sivaramakrishnan, and Jagannathan [ZSJ11], as well as designs combining CML-style concurrency with implicit data parallelism, e.g., Manticore (Fluet et al. [Flu+08]).

The above example in JoCaml is an implementation of the Join Calculus (Fournet and Gonthier [FG96] and Fournet [Fou98]), which is a descendant of the π-calculus that is restricted to unidirectional, asynchronous message passing and lexically-scoped channels. The Join Calculus is a more practical to implement version of the π-calculus amenable to distribution. The full π-calculus would require an implementation of distributed, global consensus, which is impractical (cf. [Fou98]). Join patterns have been shown to be implementable in an efficient and scalable way (cf. Turon and Russo [TR11]). Besides JoCaml, an OCaml variant with join patterns, there are other programming languages featuring join patterns, e.g., Polyphonic C$\sharp$ (Benton, Cardelli, and Fournet [BCF04]) and CPL (Bračevac et al. [Bra+16]).

What is the difference between concurrent programming and event-driven programming? Paykin, Krishnaswami, and Zdancewic [PKZ16] study a Curry-Howard correspondence [Wad15] between event-driven programming and a combination of linear time temporal logic (LTL) and linear logic (Girard [Gir87]). They show that concurrent and event-driven programming are closely related. Firstly, Paykin et al.'s notion of events corresponds to an eventually modality $\Diamond[\cdot]$, and the effective difference between the two is that concurrent programming adds selective communication (case statements), which corresponds to the linear time axiom in LTL:

$$\Diamond A \wedge \Diamond B \Rightarrow \Diamond(A \wedge \Diamond B \vee \Diamond A \wedge B),$$

meaning that "eventually, A happens before B or B happens before A". We view their results as evidence that synchronization in concurrent programming is an instance of event correlation.

Furthermore, we view actor systems (Hewitt, Bishop, and Steiger [HBS73] and Agha [Agh90]) which feature message-passing concurrency as instances of event correlation. For instance, Erlang (Armstrong, Virding, and Williams [AVW93]), Elixir and Akka are well-known implementations of actors. Passing a messages is effectively the same as an event notification, carrying an event value. Thus, actors implement forms of event processing and event correlation on messages in their inbox.

There is a relation between concurrent programming languages based on channels and actor languages. Recently, Fowler, Lindley, and Wadler [FLW17] have argued that these notions are different, but are often confused by researchers and practitioners. They formally prove that they can simulate each other. However, channel-based communication more succinctly expresses complex communication patterns, whose definition becomes considerably more involved in actor systems.

What can be said about the relation between the various notions of streams (Section 2.1) and channels? We are not aware of a formal investigation relating these concepts. However, it is well-known that synchronous channels can encode asynchronous buffered channels that decouple producers and consumers [Rep88; Rep91; Rep93]. This construction has similar qualities to push/pull streams (Sections 2.1.6 and 2.1.7). For instance, perhaps not surprisingly, the asynchronous FRP language Elm [CC13] has a denotational semantics mapping reactive programs to concurrent ML, which makes use of the buffered channel construction.

Finally, JErlang (Plociniczak and Eisenbach [PE10]) combines join patterns and actors in an Erlang dialect, for a high-level specification of synchronization patterns within an actor. The following example [PE10]

```
receive {get, X} and {set, X} -> {found, X} end
```

defines an actor that pairs up a `get` and a `set` message, resulting in a `found` message, effectively implementing a memory cell by message passing. Without the join pattern, the example would require a low-level state machine implementation operating on the actor's inbox for correlating the message pairs.

2.2.5 *Conclusion*

In this section, we gave examples and historical perspectives on the four event correlation families. We find that they express very similar ideas. Essentially, they provide declarative pattern specifications for joining, respectively merging unbounded information. A common theme are pattern notations for joining/merging sources, having variables as abstract representatives for an unbounded information source/stream, expressing relations between elements from multiple sources, to describe an entire joined collection. The patterns are *intensional* specifications, i.e., specifications of a collection by the properties of its elements, e.g.,

$$X = \{n \in \mathbb{N} \mid 1 \leq n \leq 3\},$$

whereas an *extensional* specification describes the collection itself, e.g.,

$$X = \{1, 2, 3\}$$

(cf. Cook [Coo09]).

So far, it has been difficult to "glue together" (in the spirit of Hughes [Hug89], cf. Section 2.1.5) event correlation features from the different event correlation families and within families. We also find that programming language approaches provide clearer semantics compared to systems approaches, whereas the latter has more diverse features for event correlation, e.g., windows, sequence and timing patterns.

Event correlation is fundamentally about joining/merging unbounded sources, such as streams. Historically, it has been a long standing issue how to program joins in a concise, elegant, and purely functional manner. As Abelson and Sussman [AS96] note in their classic textbook, such attempts tend to degenerate into stateful, imperative programming:

> *Thus, in an attempt to support the functional style, the need to merge inputs from different [event sources] reintroduces the same problems that the functional style was meant to eliminate.* (Abelson and Sussman [AS96])

We conclude that correlations are stateful and effectful, as a "fact of life", and that one should consider principled ways to integrate and separate effects into a language for programming event correlations.

2.3 COMPUTATIONAL EFFECTS AND EXTENSIBLE LANGUAGES

The study of programming languages in mathematics has led to techniques for modular and extensible theories of programming languages, their syntax and their semantics. In this context, considerable effort has been made to account for side effects in the semantics of programming languages. Practically all useful and interesting programming language features have side effects. In the preceding discussions, we encountered many effects related to event correlation. Thus, studying techniques for modeling and structuring effects gives us important clues how to obtain "à la carte" event correlation systems, and their language implementations. In this section, we recapitulate important developments in this area, leading to algebraic effects and effect handlers.

2.3.1 *Monads and Monad Transformers*

In the transitional period between the 1980s and 1990s, Moggi [Mog89; Mog91] proposed using monads from category theory to uniformly structure the denotational semantics of languages with features inducing side effects, e.g., state, I/O, continuations and nondeterminism. Shortly thereafter, Wadler [Wad92] proposed and popularize using monads to structure functional programs with side effects in Haskell.

A monad is a type class in Haskell with associated operations `return` and `bind`, often written infix as `a >>= b := bind a b`.

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

A value is. A computation does.

— Levy [Lev04]

Intuitively, an expression of type `m a` designates a computation returning a value of type `a`, contained in a context `m`. We can put a pure value into the context, via `return`, and we can compose computations in the same context `m1 >>= \x -> m2`, essentially a sequential composition, where the computation on the right-hand side may depend on the returned value of the left-hand side computation. Importantly, this is a referentially transparent, immutable description of a computation with side effects, preserving equational reasoning and parametricity (Wadler [Wad89]). The actual effects are isolated in the context `m` and cannot leak into the rest of the pure language when executed.

In an earlier work, Wadler [Wad90a] generalizes list and set comprehension syntax to monads, which emulates the syntax of direct-style, sequential imperative programming languages in Haskell. For example,

```
effectful :: IO ()
effectful = do putStr "Say something, please"
               line <- getLine
               putStrLn ("You said: " ++ line)
```

defines a functional program performing input and output to the console, safely isolated in the type `IO ()`, which designates a computation in the monad for

input/output, eventually returning a result of the unit type. The **do** block provides a direct style, imperative looking specification of an effectful program, and is syntactic sugar for

```
putStr "Say something, please"
  >> getLine
       >>= \line ->
             putStrLn ("You said: " ++ line)
```

in terms of functions and function composition using the monad combinators.

Monads and comprehensions go beyond Haskell and functional languages. One particularly important example in the context of this thesis is LINQ for language-integrated queries, developed for the .NET platform by Meijer, Beckman, and Bierman [MBB06], and formalized by Cheney, Lindley, and Wadler [CLW13]. LINQ is a descendant of monad comprehensions and provides a uniform and seamless way to embed queries into programming languages, supporting many different representations, e.g., external and in-memory databases, stream queries, and XML, etc. The following LINQ example in C♯ selects the names and salaries from an employee database:

```
var results = from e in Employees
              where e.Age < 40
              select new {e.Name, e.Salary};
foreach (var result in results) {
  Console.WriteLine(result);
}
```

Behind the scenes, the query syntax is desugared into a monad-based representation, similar to how the **do** notation is desugared.

Composing Effects with Monad Transformers

There can be more than just one monad in a Haskell program. Users can add custom side effects by defining additional monads, such as reading and writing to an ambient shared state. Hence, monads are a design tool to model and include new effectful language features into Haskell, "à la carte". Typically, one would like to use multiple language features in one program, e.g., I/O and local state. This requires a mechanism to compose monads, and the predominant approach in the Haskell world are monad transformers, introduced by Liang, Hudak, and Jones [LHJ95] in 1995.

Essentially, monad transformers compose multiple monads into a new monad that stacks them together. One crucial point of composing effects is that expressions written against particular monad/effect, should be reusable in a context having more effects, e.g., the example above uses the function

```
putStr :: String -> IO ()
```

inducing the `IO` effect, and it should be reusable in a context combining `IO` and nondeterminism effects (assuming a monad `Nondet a`). This requires a mechanism for lifting the type `String -> IO ()` into `String -> Nondet (IO ())`, or `String -> IO (Nondet ())`, depending on the composition order.

Choosing a different nesting order may induce different semantics of the effects. For example, `Nondet (State Int)` would describe backtracking computations, where each alternative branch has a local state, whereas `State (Nondet Int)` describes computations with a global state which is shared by the alternatives.

Critique of Monads and Monad Transformers

Kammar, Lindley, and Oury [KLO13] criticize monads as having issues with *modular abstraction* and *modular instantiation*. The former means that functional programs in the monadic style are usually written against a specific monad (implementation) instead of an abstract interface, so that implementation details leak into clients. For example, a client program would be written against a specific monad implementation for state, instead of abstractly against an interface for any monad that supports abstract `set` and `get` operations. Modular instantiation means that each sub-interface of a larger interface can be independently instantiated with an implementation, oblivious of the whole composition.

Modular abstraction and instantiation can be in part recovered with the type class mechanism in Haskell and the monad transformers we just described. However, monad transformers have a reputation of being inconvenient to use and lacking robustness, because of the necessary lifting, which cannot always be automated. Manual lifting makes client programs sensitive to nesting order of monads and thus violates the modular instantiation property above. Making monad transformers easier to use is an active area of research (cf. Schrijvers and Oliveira [SO11]).

To paraphrase, monads "don't glue together as well as one would like them to". Considering our goal for modular and extensible event correlation systems with freely composable correlation features, we certainly would like to have better ways to define, compose and program with effects. Below, we discuss alternatives to monads and monad transformers.

2.3.2 *Extensible Denotational Language Specifications*

In 1994, about a year before monad transformers were published, Cartwright and Felleisen [CF94] would propose a way to define extensible denotational language specifications, but their approach would not become as popular. It is, however, a precursor of things to come. Their approach, called "extended direct semantics" is superior to the monadic way of defining and composing new effects/language features. The essence of the problem for composing effects/language features in a modular way is to make programs using a specific set of features work seamlessly in a context with more features. Monad transformers are not entirely "seamless", in this regard, as we have argued earlier.

Cartwright and Felleisen provide a cleaner solution to the composition problem. Their work is basically an early variation on algebraic effects and effect handlers, which we discuss in the next section. It is nevertheless useful to consider their point of view on obtaining modular and extensible languages.

According to Cartwright and Felleisen, the problem of developing sublanguages that seamlessly compose into bigger languages is a matter of "stable denotations", i.e., how to structure the denotational semantics of languages in a reusable way. A denotation is a function from the syntax of a language to a semantic domain

$$[\![\cdot]\!] : \mathcal{S} \to \mathcal{D},$$

basically a fold over the structure of the syntax. For instance, programming language interpreters and compilers are denotations. Intuitively, a denotation of a language is "stable" if we can define it once and for all, and reuse it without further changes in the definition of the denotation of an extended language.

For example, this is a denotation for a simple language with numbers and addition, which interprets the syntax in terms of the actual natural numbers:

$$\begin{aligned} [\![\cdot]\!]_{\mathsf{num}} &: \mathcal{S}_{\mathsf{num}} \to \mathbb{N} \\ [\![\hat{n}]\!]_{\mathsf{num}} &= n \in \mathbb{N} \\ [\![e_1 \hat{+} e_2]\!]_{\mathsf{num}} &= [\![e_1]\!]_{\mathsf{num}} + [\![e_2]\!]_{\mathsf{num}} \end{aligned}$$

In the example, we write $\hat{n}$ to tell the syntactic representation of numbers and addition apart from their semantic counterparts.

This is another language for a read-only state, having a **get** operation:

$$\begin{aligned} [\![\cdot]\!]_{\mathsf{rd}} &: \mathcal{S}_{\mathsf{rd}} \to \mathbb{N} \to \mathbb{N} \\ [\![\mathbf{get}]\!]_{\mathsf{rd}} &= \lambda s.s \end{aligned}$$

In this case, we give the syntax a denotation in terms of functions from natural numbers to natural numbers, i.e., read-only state is interpreted as a function (in the notation of the λ-calculus) accepting the state.

It is easy to conceive a combination of the two languages, leading to the language with natural numbers, addition and ambient, read-only state. This is an expression in the composed languages:

$$\hat{2} \mathbin{\hat{+}} \mathbf{get} \mathbin{\hat{+}} \mathbf{get}.$$

However, what is the denotational semantics for the compound language? Here is a straightforward, distinctly "unstable" solution:

$$\begin{aligned} [\![\cdot]\!] &: \mathcal{S}_{\mathsf{num,rd}} \to \mathbb{N} \to \mathbb{N} \\ [\![\hat{n}]\!] &= \lambda s.n \\ [\![e_1 \hat{+} e_2]\!] &= \lambda s.\, [\![e_1]\!]\ s + [\![e_2]\!]\ s \\ [\![\mathbf{get}]\!] &= \lambda s.s \end{aligned}$$

The problem is that the dependence on the state parameter in the denotation unnecessarily leaks into the denotation of the syntax for natural numbers and addition. We had to change and redefine the denotation for numbers, which has not really changed. But their previous denotation could not be reused as it is, because the state has to be passed along in the new denotation. While the example is just a toy, its problem becomes more profound in realistic programming languages having many more features, such as exceptions, continuations and I/O. E.g., the Cartwright and Felleisen paper studies the denotation of fragments of the Scheme language and their composability.

The insight provided by Cartwright and Felleisen is that programs can be understood as agents interacting with an external environment, an "administrator" that manages resources, such as state and the file system. A language feature represents specific client/server interactions with the administrator, who knows how to "handle" the feature/effect. Their insight provides a way to structure denotations for languages in a stable manner.

We briefly explain their approach in terms of a modern presentation, which is due to Kiselyov [Kis17]. One chooses the semantic domain of computations $\mathcal{C}$, which distinguishes between pure values and requests/effects resolved by the administrator:

$$\mathcal{C} = \mathsf{V} : \mathcal{V} \uplus \mathsf{Fx} : \mathcal{E} \times (\mathcal{V} \to \mathcal{C}),$$

where we write $\uplus$ for the disjoint union of sets and assign labels V and Fx to distinguish the elements in the union. A computation either returns a value from the domain $\mathcal{V}$ or is an invocation of an effect from $\mathcal{E}$ with a continuation for the answer provided by the administrator. The domain $\mathcal{C}$ is parametric in the domain of values $\mathcal{V}$ and the possible effects $\mathcal{E}$, which depend on the composed language features. Here are stable denotations for the two example languages, which can be reused as-is in the denotation of larger languages having an extension of $\mathcal{C}$ as domain:

$$\begin{aligned}
[\![\hat{n}]\!] &= \mathsf{V}\ n \qquad\qquad & [\![\mathbf{get}]\!]_{\mathrm{rd}} &= \mathsf{Fx}\ \langle \mathsf{Get}, \lambda s.\mathsf{V}\ s\rangle, \\
[\![e_1 \hat{+} e_2]\!] &= \mathit{lift}\ [\![e_1]\!] & &\quad \text{where } \mathsf{Get} \in \mathcal{E} \\
&\quad (\lambda n_1.\mathit{lift}\ [\![e_2]\!] \\
&\qquad (\lambda n_2.\mathsf{V}\ (n_1 + n_2)))
\end{aligned}$$

We can define denotations for a given language feature, which is either pure or induces an effect, and blissfully ignore other language features in the definition. In contrast to the naïve composition of languages before, no extension with another language feature will require a refactoring of existing denotations.

The heavylifting is done in the *lift* function, which sequentially composes $\mathcal{C}$ computations:

$$\begin{aligned}
\mathit{lift}\ (\mathsf{V}\ x)\ k &= k\ x \\
\mathit{lift}\ (\mathsf{Fx}\ \langle e, k_1\rangle)\ k_2 &= \mathsf{Fx}\ \langle e, \lambda x.\mathit{lift}\ (k_1\ x)\ k_2\rangle
\end{aligned}$$

The composition function propagates occurrences of effects outwards, for being handled by the administrator, as determined by the second equation.

Extensibility is achieved by varying the definition of $\mathcal{E}$, the available effects/language features. A new effect/language feature can be separately developed and included. For example, here is a write effect for an ambient state:

$$[\![\mathbf{put}\ e]\!] = \mathit{lift}\ [\![e]\!]\ \lambda x.\mathsf{Fx}\langle \mathsf{Put}\ x, \lambda y.\mathsf{V}\ y\rangle$$

assuming that $\mathsf{Put} \in \mathcal{E}$. We assume that the context

Composition of languages is now easy, because the effect invocation is automatically threaded through the syntax to the top level, outside of the program, where the authority that responds to the request resides, pretty much how an exception is propagated. Control is relinquished to the external administrator, which is reminiscent of control transfer in coroutines (Section 2.1) and the "flip" from internal to external choice (Section 2.1.4).

The administrator resolves effect invocations, giving them a semantics. For example, we can describe it as a recursive function that evaluates computations and manages the local state for reading and writing:

$$\begin{aligned} \mathit{admin} &: \mathcal{V} \to \mathcal{C} \to \mathcal{C} \\ \mathit{admin}\ v\ (\mathsf{V}\ v') &= v' \\ \mathit{admin}\ v\ (\mathsf{Fx}\ \langle \mathsf{Get}, k\rangle) &= \mathit{admin}\ v\ (k\ v) \\ \mathit{admin}\ v\ (\mathsf{Fx}\ \langle \mathsf{Put}\ v', k\rangle) &= \mathit{admin}\ v'\ (k\ \langle\rangle) \end{aligned}$$

Thus, the design recipe for composable and extensible languages is: we represent programs in terms of $\mathcal{C}$ and may add new effect constructors. Their semantics is determined by the administrator, which requires adding a clause for the new effect. While programs are modular and extensible, the administrator seems to be monolithic in the Cartwright and Felleisen approach. Effect handlers in the next section improve over this situation, adding the possibility to locally vary the denotation of an effect.

The domain of computations $\mathcal{C}$ which is parametric over values types $\mathcal{V}$ and effects $\mathcal{E}$ is a generalized form of Hancock et al.'s well-founded tree construction for stream transformers from Section 2.1.7. $\mathcal{C}$ forms a monad, the *free monad* over $\mathcal{E}$, and *lift* for implicitly propagating effects is its bind operation. The solution to the composition issues with monads and monad transformers turns out to be a (free) monad.

2.3.3 *Algebraic Effects and Handlers*

Algebraic effects and effect handlers (or simply, effects and handlers) have roots in category theory and universal algebra. Plotkin and Power [PP03] studied in 2003 the equational theory/algebraic theory of Moggi's monadic effects. The algebraic view approaches effects from the point of view of syntax/interface of operations triggering the effect. For instance, it investigates the syntactic equational laws of effect operations and the possible denotations they induce.

The algebraic theory of effects assigns effect operations a type signature, e.g., $\mathbf{get} : 1 \to A$ and $\mathbf{put} : A \to 1$ are operations for reading and writing a memory cell holding A values (we will write concrete effect operations in blue font from now on). Some of the equational laws we would expect from these operations are (cf. Bauer [Bau18])

$$\mathbf{get}(\langle\rangle, \lambda s.\mathbf{put}(s, \kappa)) = \kappa\ \langle\rangle,$$

and

$$\mathbf{put}(s, \lambda\_.\mathbf{get}(\langle\rangle, \kappa)) = \mathbf{put}(s, \lambda\_.\kappa\ s).$$

This algebraic representation of computations should not be surprising by now, since it corresponds to the well-founded (computation) trees we encountered in Sections 2.1.6 and 2.3.2, carrying the continuation in the nodes.

In the algebraic theory, the equations together with the syntax signature of effect operation induce the free model (cf. [Bau18]), which consist of the quotient structure of the equivalence relation induced by the equations over syntax trees generated from variable symbols and the effect operations. The free model gives a canonical representation of computations with effects, and any other model of the algebraic theory can be obtained from it in a unique way. While the equations

are important, practical language implementations do not check their fullfillment, which is an undecidable problem in general. If no equations are present, then the free model corresponds to exactly the well-founded computation trees, and thus the free monad construction.

We have encountered specialized instances of the free monad. Here is its general representation as a data type:

data Free$[F[\cdot], A]$ =
| ret: $A \to$ Free$[F,A]$
| op: $F[$ Free$[F,A]\,] \to$ Free$[F,A]$

Free monads are abstract syntax with variables of type A and inner nodes determined by the operations in the type constructor $F[\cdot]$ (a functor in the language of category theory [Awo10]).

For instance, with

data Get$[A, K]$ =
| **get**: $(A \to K) \to$ Get$[A, K]$

we obtain the trees representing stream transformers (Section 2.1.6) by partial application of the first type parameter:

T$[A,B] \cong$ Free[Get$[A, \cdot], B]$

Effect handlers proposed in 2009 by Plotkin and Pretnar [PP09] are deconstructors of the effect operations, and technically transformations from computation trees to computation trees. We already encountered a variant of handler in the previous section, in the form of the *admin* function, that specifies a semantics for effect operations. Its type is literally a function from computations to computations. Practically, handlers are folds over computation tree structure. We will give more concrete examples of handlers in the next section.

Free monads are one possible and well-known way to embed algebraic effects in terms of abstract syntax and interpreters in to a programming language [KI15], with the associated overhead. However, it is also possible to have them as language primitives that compile to efficient code. We consider such language implementations in the following.

2.3.4 *A Primer on Effects and Handlers in Multicore OCaml and Koka*

Here, we give an example-driven overview of effects and handlers in the multicore OCaml language (Dolan et al. [Dol+17]) and the Koka language (Leijen [Lei17b]).

NOTES ON OCAML SYNTAX (Multicore) OCaml supports polymorphism in a Hindley-Milner typing discipline. Type variables in polymorphic definitions are lowercase identifiers preceded by a prime (`'`) and type constructors are written postfix, e.g.,

```
map: 'a list -> ('a -> 'b) -> 'b list
```

is the usual signature of the `map` function on lists. Certain situations require an explicit binding of type parameters in polymorphic definitions, in which case type parameters are not prefixed with the prime symbol, e.g.,

```
map: type a b. a list -> (a -> b) -> b list
```

is equivalent to the above.

Intuitively, effects and handlers generalize `try/catch/throw` for managed exceptions by enabling evaluation to resume back to the point in the program, where the effect was "thrown", as a form of coroutine.

Recall that effects entail having named, typed operations, which are user-defined. For example, this is the declaration of a failure effect:

```
effect Fail: unit -> 'a
```

It consists of the command `Fail`, taking a unit-valued argument and returning any type. Its polymorphic return type indicates that it cannot resume to the calling site. Failures are not supposed to return.

Effect commands are invoked with **`perform`**, e.g.,

```
let be_positive () =
  let i = read_int () in (* Read integer from keyboard *)
  if i > 0 then i else perform (Fail ())
```

In multicore OCaml, effects can be handled by the existing language facilities for exception handling and pattern matching, in the cases of **`try`** and **`match`** blocks. By handling command invocations, we give them an implementation/semantics. For instance, a handler may interpret failure as proper exceptions:

```
try be_positive () with
| effect (Fail ()) k ->
    raise (Invalid_argument "work on your attitude!")
```

or alternatively, may provide a default positive number:

```
try be_positive () with
| effect (Fail ()) k -> 42
```

or enclose the outcome in the `option` type:

```
match be_positive () with
| i                  -> Some i (* return clause *)
| effect (Fail ()) k -> None.
```

In the above handler clauses, the variable `k` binds the continuation/resumption of the computation that triggered the `Fail` effect, similar to the clauses of the administrator function in Section 2.3.2. The important difference is that these handlers are local and ad-hoc. Just as exceptions, command invocations are dynamically dispatched to the innermost handler in the evaluation context having a matching clause. That is, effect handling is a form of dynamic overloading.

The ability to bind the resumption makes handlers forms of delimited continuations. The following examples make use of the resumption, which may be invoked via **`continue`**, e.g., given the command

```
effect Read: unit -> int
```

the expression

```
try (perform (Read ())) + 1 with
| effect (Read ()) k -> continue k 41
```

evaluates to 42, whereas

```
try
  try (perform (Read ())) + 1 with
```

```
  | effect (Read ()) k -> continue k (perform (Read ()))
with
| effect (Read ()) _ -> -1 (* does not resume *)
```

evaluates to `-1`. This example also shows that multiple, local interpretations of the same effect may coexist in the same program.

We may lift second-class **`try`** handlers into first-class effect handlers, by means of first-class functions, e.g.,

```
let handler: (unit -> 'a) -> 'a = fun action ->
  try action () with
  | effect (Fail ()) k -> 42
```

And define a binary composition `|+|` on first-class handlers:

```
let (|+|) h1 h2 action = h1 (fun () -> h2 action)
```

which nests handlers in right (innermost) to left (outermost) order.

Effect Types in Koka

While multicore OCaml piggybacks on the pattern matching and exception handling mechanisms of OCaml, Koka has a more dedicated syntax for effect handlers. Furthermore, in contrast to multicore OCaml, Koka has effect types in functions via effect rows, keeping track of the possible effects that may be invoked. Here, we show these additional two traits of Koka, since the formal semantics of Cartesius in Chapter 3 is based on a λ-calculus with a Koka-style effect system.

The handlers we considered so far resume their continuation at most once, which does not always have to be the case. For example, we can implement nondeterminism effects with handlers ([Lei17b]), by resuming more than once. This is a Koka effect declaration

```
effect amb {
  flip() : bool
}
```

representing a coin flip. Here is a computation that performs the nondeterminism effect:

```
fun lottery(): <amb> string {
  if (flip()) then "You've won"
  else "You've lost"
}
```

Its type signature reveals that the nondeterminism effect occurs, via the `<amb>` row annotation. Here is a handler that resumes more than once, implementing that effect:

```
val h_amb = handler {
  x -> [x]
  flip() -> resume(False) + resume(True)
}
```

we can apply the handler like a function to computations for handling effects, e.g.,

```
h_amb(lottery)
```

results in the list

```
["You've won", "You've lost"].
```

```
module type Symantics = sig
  type 'a repr
  val lit: int -> int repr
  val (+): int repr -> int repr -> int repr
end
```

(a)

$$\frac{n \in \mathbb{Z}}{\vdash_{\text{exp}} \texttt{lit}\ n : \mathsf{Int}}$$

$$\frac{\vdash_{\text{exp}} e_1 : \mathsf{Int} \qquad \vdash_{\text{exp}} e_2 : \mathsf{Int}}{\vdash_{\text{exp}} e_1 + e_2 : \mathsf{Int}}$$

(b)

```
module Num = struct
  type 'a repr = 'a
  let lit n = n
  let (+) x y = plus x y

end
```

(c)

```
module PP = struct
  type 'a repr = string
  let lit n = sprintf "<%d>" n
  let (+) x y =
    sprintf "(%s + %s)" x y
end
```

(d)

Figure 2.2: Basic Tagless Final Examples.

In terms of types, `h_amb` is a function

```
h_amb: (() -> <amb | e> a) -> e list<a>
```

which transforms computations having the nondetermism effect into a computation without the effect, wrapping the result in a list. The type is effect-polymorphic, indicated by the type variable `e` in the row `<amb| e>`. Relating back to the problem of composing denotations of languages and lift definitions for one language into a larger language, intuitively, this means that `h_amb` works with computations having possibly more effects. The type also shows how a handler is a value that gives locally a denotation to a specific effect operation, unlike the global administrator in the Cartwright and Felleisen work.

2.4 A PRIMER ON TAGLESS INTERPRETERS

In this section, we introduce the tagless final approach by Carette, Kiselyov, and Shan [CKS09], upon which we define PolyJoin, a practical embedding for declarative event correlation patterns into programming languages (Chapter 4).

NOTES ON OCAML SYNTAX The discussion contains examples in OCaml, particularly its module system. We use green font to highlight the names of all things related to the module system: modules, module signatures, functors, e.g.,

```
module type Foo = sig
  type t
  val a: t
  val b: t
  val f: t -> t -> t
end
```

defines a *module signature* `Foo` having an abstract type `t` and values `a`, `b`, `f`. We introduce the other concepts of the module system as the discussion unfolds.

Traditionally, interpreters for DSLs are defined by structurally recursive functions (initial algebras) translating abstract syntax terms into a semantic domain, i.e., the interpreter function folds the abstract syntax terms. The latter are modeled by data types, "tagging" nodes with data constructors, and deconstructed by the interpreter function via pattern matching. In contrast, tagless final (1) encodes DSL terms with (typed) functions instead of data, in the style of Reynolds [Rey78] and (2) decouples interface and implementation (the interpreter) of these functions, obtaining a term representation which is indexed by their denotation. "Tagless" means that the syntax trees of the DSL are not reified as data, where each node is tagged with a data constructor and analyzed by pattern matching.

For example, the OCaml module signature in Figure 2.2a defines a tagless DSL for arithmetic expressions. This language supports integer constants and addition, via the syntax functions `lit` and addition `(+)`, in infix notation. We abstract over the concrete semantic representation of arithmetic expressions with the type constructor `'a repr` (read: expression of DSL type `'a`). The type signatures of these syntax functions embed both the grammar and the typing rules of the DSL in the host language's type system, using phantom types [LM99]. These signatures straightforwardly correspond to a premises-to-conclusion reading of natural deduction rules, with return type being conclusions and arguments being premises (Figure 2.2b). That is, tagless final models *intrinsically-typed* DSLs, where it is impossible to define ill-typed terms, by construction. We write $\vdash_{\text{exp}} M : A$ for expression typing, assigning DSL expression M the DSL type A.

In OCaml, we represent (groups of) concrete DSL terms as functors, accepting a `Symantics` module:

OCaml functors construct modules from other modules. The syntax `open S` makes the definitions within `S` visible in the current lexical scope, which would otherwise have to be qualified, e.g., `S.lit`.

```
module Exp(S: Symantics) = struct
  open S
  let exp1 = (lit 1) + (lit 2)
  let exp2 x y = exp1 + (lit 3) + x + y
end
```

In this example, `exp1` is a closed DSL term having host language type `int S.repr` and `exp2` a DSL term with two free variables, having the type

```
int S.repr -> int S.repr -> int S.repr
```

The interpretation of DSL terms depends on modules conforming to the `Symantics` signature above, i.e., DSL terms are parametric in their denotation/interpreter `S`, which the type signature makes explicit. For brevity, we will not explicitly show the surrounding functor boilerplate in subsequent example terms. Figure 2.2c and Figure 2.2d show two example interpreters/denotations for our arithmetic expressions DSL. `Num` interprets arithmetic expressions as OCaml's built-in integers and functions, whereas `PP` interprets them as their string representation:

```
module En = Exp(Num);; (* Instantiate Num interpretation *)
module Es = Exp(PP);;  (* Instantiate PP interpretation *)
En.exp1;;              (* yields int Num.repr = 3 *)
Es.exp1;;              (* yields int PP.repr = "(<1> + <2>)" *)
```

Furthermore, the tagless final approach supports modular extensibility of DSLs. For example, the DSL for arithmetic expressions can be extended to further include boolean expressions and connectives:

```
module type NumBoolSym = sig
  include Symantics
  val b_lit: bool -> bool repr
  val (&): bool repr -> bool repr -> bool repr
end
```

The syntax **include** copies the definitions of the given module signature into the new signature.

In a similar fashion, these additional functions can be separately implemented in a module and combined with the existing interpreters, using module composition, to yield extensions of the interpreters, such as `Num` and `PP`. We leave their extension with booleans as an exercise to the reader.

2.5 CHAPTER SUMMARY

Event correlation has its roots in developments from the 1950s to 1970s. Events and streams are fundamental abstractions in this domain and come in qualitatively different variations, e.g., there are push, pull, and hybrid push/pull streams. They can be explained in terms of categorically dual notions of algebra and coalgebra, and forms of eventuality/effects. Stream transformers are representable in terms of free monads and thus directly relate to effects in functional programming.

Our analysis of the different event correlation families suggests that they are expressions of very similar ideas. Common themes are pattern notations for joining/merging sources, having variables as abstract representatives for an unbounded information source/stream, expressing relations between elements from multiple sources, to describe an entire joined collection. Yet, they lack composability and uniformity. These issues are related to structuring and defining extensible languages and denotations, tying event correlation in yet another way to effects.

Effects and handlers culminated from the pursuit to structure denotational semantics and integrate side effects into pure languages. They are a flexible abstraction, combining delimited control, modular abstraction and modular instantiation. In the next chapter, we will show that effects and handlers are a suitable mechanism for fulfilling our vision of à la carte event correlation systems.

Part II

A SYNTAX AND SEMANTICS FOR EXTENSIBLE EVENT CORRELATION

In which we develop a novel semantic model for extensible event correlation in terms of algebraic effects and effect handling (back-end) as well as extensible declarative event pattern syntax in terms of a typed embedding into programming languages (frontend).

VERSATILE EVENT CORRELATION WITH ALGEBRAIC EFFECTS

3

This chapter shares material with the ICFP'18 paper of the same title [Bra+18].

SYNOPSIS We present CARTESIUS, the first language design to uniformly express variants of n-way joins over asynchronous event streams from different domains, e.g., stream-relational algebra, event processing, reactive and concurrent programming. We model asynchronous reactive programs and joins in direct style, on top of algebraic effects and handlers. Effect handlers act as modular interpreters of event notifications, enabling fine-grained control abstractions and customizable event matching. Join variants can be considered as cartesian product computations with "degenerate" control flow, such that unnecessary tuples are not materialized a priori. Based on this computational interpretation, we decompose joins into a generic, naïve enumeration procedure of the cartesian product, plus variant-specific extensions, represented in terms of user-supplied effect handlers.

3.1 INTRODUCTION

Events notify a software system of incidents in its dynamic environment. Examples of event sources are sensors, input devices, or network hosts. *Event correlation* means to make deductions about the state of the environment, given observations of events from different sources over time. For example, by correlating batches or streams of events, computer systems drive cars, trade stocks, or give recommendations; by correlating input events smartphones and tablet devices interpret touch gestures. In the words of Luckham [Luc01]: "*Events are related in various ways, by cause, by timing, and by membership*". Since events are asynchronous, computing such event relations amounts to defining n-way *synchronizers* (or *joins*) over events. To avoid teasing out subtle distinctions, we use the terms *join* and *correlation* interchangeably.

Different communities have thus far invented various specialized abstractions for joins. On the systems side, there are *complex event processing (CEP)* and *stream processing* systems [CM12]. On the programming languages side, there are *reactive programming* [Bai+13] and *concurrent programming languages*, e.g., [Rep91; Flu+08; FG96; CL99; BCF04]. Common to all families is that they support some sort of high-level specifications of joins, e.g., as declarative patterns or queries.

However, a commonly agreed-upon semantics for joins (and by extension event correlation) remains elusive. Not only is the number of features for relating events staggering (cf. the surveys by Cugola and Margara [CM12] and Bainomugisha et al. [Bai+13]), but already a single feature can be interpreted quite differently between systems and even within one family (e.g., time windows [Din+13]). This lack of clarity is an obstacle for choosing the right language/system: It is hard to determine its adequacy for expressing a desired event correlation behavior. Moreover, since each system provides specialized abstractions that cannot be easily changed, some applications may not find any adequate language/system to meet their requirements.

Motivated by these observations, the goal of our work is to contribute a language design and structured programming abstractions for defining n-way event joins with customizable, extensible, and composable semantics, which we call *versatile joins*. More specifically, our aims are:

MIX-AND-MATCH STYLE FEATURE COMPOSITION We enable *cross-domain composition* of event correlation features, *overloading of features* to express subtle differences, and *extensibility with new features*.

CONTROLLABLE MATCHING BEHAVIOR To model versatile joins, one needs to consider all possible ways joins can pattern match and process n infinite push streams. There are plenty of ways of doing this: aligning, skipping, duplicating, timing, depending on event values, or any combination thereof. To manage this complexity, the language needs generic control abstractions for coordination and alignment of streams.

DIRECT STYLE SPECIFICATIONS Asynchrony of events encourages programming in some form of continuation-passing style (CPS), due to inversion of control. Continuation-based programming is error-prone and does not scale in the large, leading to "callback-hell" [Edw09]. Hence, the language design should enable users to express asynchronous computations in direct style. This is easier to understand and more natural than working with continuations directly. Even more so in the case of n-way *joins*, where n interdependent continuations need to be coordinated.

This work proposes CARTESIUS – a domain-specific language for programming versatile joins over infinite event sequences that satisfies the above goals. Its design is guided by an informal intuition of *what a join is*:

PROPOSITION 3.1 (INTUITIVE INTERPRETATION)
A join (correlation) is a restriction of the cartesian product over the inputs. ♦

This view of joins is deliberately open-ended, since we aim for extensibility and customizability. We consider the pure cartesian product as the most general correlation, confining the space of possible results. Any other variant of event correlation is a restriction of the cartesian product, where a "restriction" can take many forms, e.g., a simple attribute filter on values, or a nondeterministic selection of values, which is an *effect*. We support user-defined effects as a way to specify, customize, and add new restrictions.

Proposition 3.1 defines what joins "are" in a way that is not far from a denotational semantics, e.g., relational algebra. At the same time, this informal view is already useful to derive a way of *doing joins*, which is simple yet realistic for implementation in a programming language:

PROPOSITION 3.2 (COMPUTATIONAL INTERPRETATION)
A join computation is an enumeration of the cartesian product over the inputs, with (user-defined) side effects influencing how the computation proceeds. ♦

Following this computational interpretation, the core of CARTESIUS provides a generic cartesian product implementation. Yet, despite working with such a naïve and expensive generic component, effects enable us to obtain computationally efficient variants of correlations in CARTESIUS. That is, the cartesian product

retains all observed events forever and materializes all combinations of events. In practice, though, only a small fraction of event combinations is relevant to the correlation computation, e.g., when zipping streams. User-defined effects can manipulate the control flow so that irrelevant event combinations are *never* materialized.

For language-level support of effects, CARTESIUS employs algebraic effects and handlers [PP03; PP09]. They adequately address our requirements on the language design. First, the algebraic view of effectful operations is well-suited for supporting extensibility and customizability: They enable purely functional definitions of effects, compose freely, and enable language extensions/customizations as libraries, *without changes to compiler or runtime*. Second, *effect handlers* capture control flow akin to coroutines [MI09; Con63], which lets users express asynchronous computations in direct style [Lei17a], diminishing the pains of inversion of control and supporting custom control abstractions.

The remainder of this chapter contributes:

- a high-level example-driven overview of CARTESIUS in Section 3.2;
- a formal semantics of a core calculus for effect handlers with parametric and effect polymorphism, λ_{cart}, in Section 3.3;
- a library-based design of CARTESIUS, as an embedding into λ_{cart}. The design features novel uses of algebraic effects and handlers: we encode event notifications, reactive programs, implicit variables and control abstractions for fine-grained coordination of asynchronous computations. We implemented the design in both Koka and multicore OCaml (Section 3.4).

To the best of our knowledge, this is the first work to (a) enable programmable event correlation with clearly defined semantics – a long standing problem for event languages – and (b) to identify and adopt algebraic effects and handlers as a viable method for solving the problem.

3.2 AN OVERVIEW OF CARTESIUS

This section presents a high-level overview of CARTESIUS from the perspective of end users who need to specify and customize joins. We assume a host language akin to ML with algebraic effects. Furthermore, we outline how our computational interpretation of joins integrates event correlation features from different domains.

3.2.1 *Event Sources*

There are two kinds of asynchronous event sources in CARTESIUS computations: Input event sources that connect to the external world, e.g., sensors, input devices, etc., and output event sources that are defined by correlation patterns. Both are called *reactives*.

CARTESIUS embodies reactives as values of the parametric data type $\mathsf{R}[T]$, where T is the type of the *payload* carried by events. An event of type T in CARTESIUS is actually a product type $(T \times \mathsf{Time})$ for some time representation Time. That is, an event embodies the evidence of something that *happened* at a particular point in time. This is unlike works on reactive programming (e.g.,

[Cav+14]), which view events of type T as the *potential to yield* a value in terms of an eventually modality $\Diamond[T]$. These two views are not incompatible, though. In fact, both show up in the definition of R[T], which intuitively corresponds (informally written $\approx$) to an infinite sequence of events in the first sense:

$$\mathsf{R}[T] \approx \nu Y.\Diamond[(T \times \mathsf{Time}) \times Y].$$

We write $\nu Y.T[Y]$, respectively $\mu X.T[X]$ for the *greatest fixpoint/least fixpoint* of a type term $T[X]$, to express type-level recursion (cf. Chapter 2).

That is, an event source of Cartesius *potentially yields events*, one after another. We adopt this representation from Elliott [Ell09] and discuss its precise definition and benefits in Section 3.3.

Commonly, CQL-like query languages [ABW04; KS09] and CEP languages [DIG07; Dem+06] exhibit timing information on data streams and events, which we capture above. The design space for representing time in these languages ranges from singular time stamps to sets of time stamps per event, trading off accuracy and space requirements [Whi+07]. Cartesius sets Time to an interval representation, for constant space usage, while still enabling an approximation of an event's history (cf. Section 3.3).

3.2.2 *Expressing Event Relations in Direct Style*

In Cartesius, we express versatile event correlation through declarative *correlation patterns*, which correlate events from one to many reactives, thereby transforming event data, or filtering events. A basic example is the following pattern:

```
1 let mouse: R[Int×Int] = ... //mouse input
2 let even_mouse_product: R[Int] =
3   correlate
4   { p from mouse
5     where (fst p) mod 2 = 0
6     yield (fst p) * (snd p) }
```

where the syntactic form **correlate** delimits the pattern body between the curly braces. This correlation pattern defines a reactive *even_mouse_product* correlating mouse position events, which are integer pairs. We filter the positions with an even first component using **where** and multiply their components within **yield**. The overall result of the correlation is a reactive emitting all multiplications of mouse positions satisfying the condition.

Note that this example is an asynchronous computation, but its syntax maintains the intuitive direct style of comprehension syntax similar to stream query languages. That is, *even_mouse_product* looks demand-driven (or pull-based), but it is reacting to event observations from the external mouse input (push-based). Internally, inversion of control is present, but it remains hidden to end-users.

Cartesius expresses event correlations as n-way correlation patterns of the form:

$$\textbf{correlate}\ \{\ x_1\ \textbf{from}\ r_1; \ldots; x_n\ \textbf{from}\ r_n\ \textbf{where}\ p\ \textbf{yield}\ e\ \}$$

which intuitively represents a transformation[1] of "shape"

$$\mathsf{R}[T_1] \Rightarrow \ldots \Rightarrow \mathsf{R}[T_n] \Rightarrow \mathsf{R}[T_{n+1}]$$

1 We write $A \Rightarrow B$ to indicate transformations in a broad, informal sense, and $A \to B$ for proper function types.

that forms an output reactive from n input reactives. The elements of the pattern syntax reflect this accordingly: (1) we bind individual event variables from input reactives with x_i **from** r_i, in the scope p and e, (2) intensionally (cf. Section 2.2.5) specify relations among the events $(x_i)_{1 \leq i \leq n}$ in the optional predicate **where** p, and then (3) apply a transformation **yield** e: $T_1 \times \ldots \times T_n \Rightarrow T_{n+1}$ to event combinations that are in the relation.

3.2.3 *Computational Interpretation*

Here, we motivate the cartesian product with side effects view (Proposition 3.2). The most basic correlation pattern we can define in CARTESIUS is an n-way cartesian product, i.e., the *cartesian* definition in Figure 3.2, top left, which produces all combinations $\langle x,y \rangle$ from the *left* and *right* reactives (Figure 3.1a). Predicates as in the preceding example reduce the number of generated combinations, i.e., *restrict* the cartesian product. This is a simple and intuitive way to think about the semantics of event correlation and close to a relational algebra interpretation.

We also want to derive the operational behavior of event correlation as a realistic basis for concrete implementations in a programming language. The challenge is to keep the specification of the operational behavior simple and extensible, ideally, close to the intuitive relational view while achieving efficiency of the computation. A naïve implementation would generate all combinations and then test against the predicate, which is expensive and leads to space leaks. Moreover, due to the asynchrony of reactives, we are forced to observe event notifications one-by-one. Hence, applying lazy techniques from demand-driven computation would not avoid the issue.

"Space leak" is a term used by the reactive programming community [Mit13; LH07; Kri13] that refers to a kind of memory leak, i.e., when a reactive computation retains "*too much history*" (Krishnaswami [Kri13]) in memory.

Our solution *a priori* avoids generating superfluous combinations. Specifically, we propose exposing overloadable, *user-defined effects* in the cartesian product computation. Simply by reinterpreting its effects, we force the computation into a specific operational behavior, so that the search space of combinations is cut down. In this way, end users can work with a generic, naïve generate-then-test implementation and turn it into a specialized, efficient computation.

For the concrete implementation of the operational behavior sketched above, CARTESIUS uses an effect system based on Plotkin et. al's algebraic effects and handlers [PP03; PP09]. Some elements of the correlation pattern syntax desugar into effect operations. One such element is e.g., **yield**. By itself, **yield** has no semantics (i.e., implementation) – it requires a run time context (a handler) in which it is interpreted. For now, it is not necessary to understand this abstraction in detail. The only thing of note is that programmers can write and apply effect handlers that give implementations to effect operations as a form of dynamic overloading. Effect handlers are the main programming abstraction for customizing and specializing joins. We explain the basics of effects and handlers in Section 3.3 and how they are used for event correlation in Section 3.4.

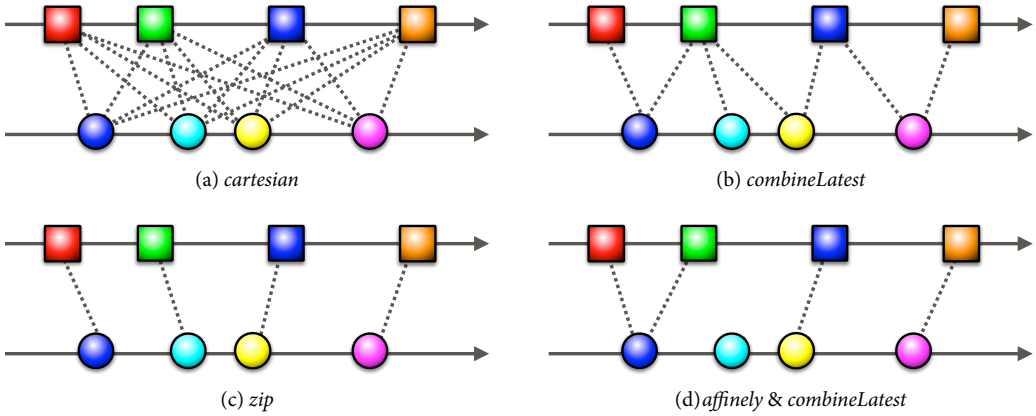


Figure 3.1: Example correlations over two reactives, represented by arrows. Coloured marbles represent events. Dashed lines indicate materialized pairings. We adopt the marble diagram notation from the Reactive Extensions documentation [Rea11]. Reactives are potentially infinite sequences that asynchronously emit event values (colored marbles) of a certain type (marble shapes).

3.2.4 *Customizing Matching Behavior with Algebraic Effects*

With the computational interpretation sketched above, modeling a specific correlation semantics becomes a matter of adding the right mixture of effect handlers. This aspect of the language design is crucial for defining extensions and mix-and-match style compositions of features. In the following, we exemplify this by specifying different correlation behaviors in Cartesius using the effect mechanism.

As a first example, we discuss the *combineLatest* combinator on asynchronous event sequences, e.g., as featured in the Reactive Extensions (Rx) library [Rea11]. As illustrated in Figure 3.1b, this combinator weakly aligns its inputs and always combines the *most recently* observed events. Figure 3.2, top right, shows its corresponding Cartesius definition. In Line 2-4, we apply the combinator *mostRecently* to both *left* and *right* reactives. This combinator creates an effect handler that acts on its argument reactive (*left* and *right* in the example). The ⊞ operator composes the two effect handlers, so that both apply to the correlation. We may read this line as "inject this (compound) effect handler into the cartesian product computation". The handlers are injected through an *implicit variable* *?restriction* into the computation. We use implicit variables starting with '*?*' for injecting dependencies (Section 3.3.4).

Correlation patterns always depend on the implicit variable *?restriction*, which by default is bound such that there is no restriction. During the computation of a correlation pattern, the effect injected by *?restriction* is applied. In the case of *combineLatest*, the injected effects force the cartesian product to provision only one element per input reactive, discarding events other than the latest one.

```
let cartesian: R[A] → R[B] → R[A × B] = λleft. λright.
  correlate
    { x from left
      y from right
      yield ⟨x,y⟩ }
```

```
let zip = λleft. λright.
  let implicit ?restriction =
    (mostRecently left)
    ⊞ (mostRecently right)
    ⊞ (aligning left right)
  in cartesian left right
```

```
let combineLatest = λleft.λright.
  let implicit ?restriction =
    (mostRecently left)
    ⊞ (mostRecently right)
  in cartesian left right
```

```
let affine_latest = λleft. λright.
  let implicit ?restriction =
    (affinely left)
    ⊞ (mostRecently left)
    ⊞ (mostRecently right)
  in cartesian left right
```

Figure 3.2: Corresponding CARTESIUS Code for Figure 3.1.

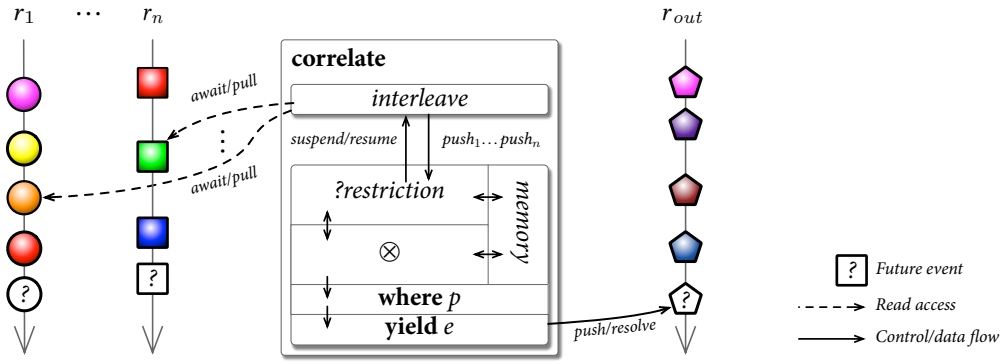


Figure 3.3: Overview: Sub-computations in Correlation Patterns.

PATTERN TRANSLATION Above, we specified a custom correlation behavior as a restriction of the cartesian product. To provide readers an overview of how this computation is structured underneath the surface, Figure 3.3 depicts the computational building blocks into which CARTESIUS translates n-way correlation patterns. Blocks designate computations, which interact by control or data transfer, indicated by solid arrows. A **correlate** pattern results in a number of sub-computations, which we describe top to bottom in the following.

The *interleave* sub-computation is a collection of n threads/strands, each independently iterating over one of the n input reactives. The dashed lines indicate that accessing the events of a reactive is potentially blocking. A reactive always has a finite prefix of materialized events and a tail that is yet-to-arrive in the future (indicated by question mark). For materialized input events, iteration proceeds in direct style, but suspends at the future tail with its continuation/-callback, until the environment asynchronously materializes the tail. The i-th interleaved thread exposes the events to the join by performing a distinct $push_i$ effect to the building blocks downwards.

The user-supplied *?restriction* computation handles (i.e., observes) these $push_i$ effects. It embodies a custom synchronization logic for aligning the reactives, in the role of a coordinator. For this purpose, *?restriction* has the power to suspend/resume individual strands. Another responsibility of *?restriction* is to interact with the fixed cartesian product computation (⊗) – control shifts between the two as a form of coroutine [MI09; Con63]. The computations *?restriction* and ⊗ share a local *memory* as a communication medium to control the event combinations that ⊗ will generate. The default *?restriction* (unrestricted cartesian product) will store all events it observes from *interleave* in *memory* for further processing by ⊗. The *mostRecently ?restriction* introduced above would only store the last observed event in *memory* and truncate older events. Once ⊗ generates event combinations, these pass into the filtering and transformation steps as specified by **where** and **yield**. Finally, **yield**ed output events are materialized into an output reactive.

COMPOSING CUSTOM CORRELATION BEHAVIORS. To illustrate how combinators can be composed, consider the following correlation pattern that specifies the well-known *zip* combinator (Figure 3.1c). It exhibits a stricter alignment than *combineLatest* by adding another effect handler (*aligning left right*) in Figure 3.2, bottom left. The (*aligning*) handler changes the *selection behavior* of the cartesian product so that events from *left* and *right* are processed in lockstep. That is, if the correlation computation receives the next event from *left*, it will not process further *left* events until the next *right* event, and vice versa. It does this by suspending/resuming the corresponding iteration strand (Figure 3.3). In conjunction with *mostRecently*, we ensure that paired up events are forgotten. The result is the familiar correlation behavior of *zip*. Supplied effect handlers execute in right-to-left order. For example, the restrictions imposed by (*aligning*) apply first to ensure lockstep processing.

We can flexibly change the behavior of correlations with additional effect handlers. For instance, *event consumption* is another aspect we may wish to control. By default, there is no bound on how often a correlation may combine an observed event with another one. For example, in *combineLatest*, if *left* emits just one event *e*, then *e* will forever be combined with all future events from *right*. We may wish to enforce *affine* use of events to avoid this. It suffices to supply the *affinely* effect handler, just as in the *affine_latest* combinator in Figure 3.2, bottom right. Its behavior is depicted in Figure 3.1d, i.e., each event of the top stream occurs at most once in a pairing with the bottom stream.

3.2.5 *Natural Specifications with Implicit Time Data*

Notice that variable bindings in correlation patterns directly project the *payload* of events. For instance, the definition of *even_mouse_product* directly accesses the first component of *p*. As mentioned, the events actually carry additional time information: a closed *occurrence time interval* $[\tau_1, \tau_2]$. Yet, so far, time neither shows up in the correlation pattern definitions nor in the types of the input reactives. This design choice enables programmers to write event relations and transformations naturally in terms of the payload, as long as they do not need explicit access to event times.

CARTESIUS provides access to event times through special variables. For each bound event variable x, CARTESIUS provides an implicit variable $?time_x$ that contains the time of x. The binding of $?time_x$ variables is managed internally by the join computation and exists only within the scope of the corresponding **correlate** pattern. For example, the pattern

```
1 correlate
2   { p from mouse; k from keys
3     where |(end ?time_p) − (end ?time_k)| ≤ 2 ms
4     yield ⟨a,b⟩ }
```

correlates all mouse movements and keyboard events where the occurrence time intervals end at most 2 milliseconds apart from each other (Line 3). The function *end* in the constraint refers to the end of the respective interval. Correspondingly, the function *start* refers to the beginning of the interval. When an event results from a correlation pattern, its time interval is the union of the intervals of the contributing events. With implicit variables, we avoid the notational overhead of wrapping and unwrapping the payload and time from n input events to form an output event.

3.2.6 *Time Windows as Contextual Abstractions*

Since effect handlers influence control flow, core features from the CQL-like query languages [ABW04; KS09] and CEP engines [DIG07; Dem+06] are expressible. One such core feature are *time windows*. The following example is adapted from EventJava [EJ09] to find new TV releases having five good reviews within a month:

```
1 correlate
2   { with (slidingWindow (1 Month) (1 Day))
3       release     from TVReleases
4       reviews(5) from TVReviews
5       where (distinct reviews)
6               (forall reviews (λx. (rating x) ≥ 3.5))
7               (forall reviews (λx. (model x) == (model release)))
8       yield release }
```

In the example, the pattern body is surrounded by the **with** syntactic form. As opposed to the implicit variable *?restriction*, which *injects* effect handlers into the middle of the pattern computation (Figure 3.3), the syntactic form **with** *encloses* the pattern computation with the given handler. This way, the entirety of the correlation pattern computation can be controlled. In our example, the *slidingWindow* handler bound by **with** manages multiple instances ("windows") of the correlation pattern: a new instance each day, processing events within the past month. This way, the *slidingWindow* handler emulates the design of stream query languages – windows impose a temporal scope in which the query executes.

EXPRESSIONS

	$x, y, z \ldots$		(Var)
	$\alpha^\kappa, \beta^\kappa, \gamma^\kappa \ldots$		(TVar)
	v	$::= \lambda x.e \mid \Lambda\alpha^\kappa.e \mid \mathsf{k}\ \overline{v} \mid com$	(Val)
	e	$::= x \mid v \mid e\ e \mid e\ [T^\kappa] \mid \mathsf{k}\ \overline{e} \mid \mathbf{fix}\ e \mid \mathbf{match}\ e\ \{\overline{p \mapsto e}\} \mid \mathbf{handle}\{h\}\ e$	(Exp)
	p	$::= x \mid \mathsf{k}\ \overline{p}$	(Pat)
	h	$::= x \mapsto e \mid h;\ com\ x\ x \mapsto e$	(HCls)
	com	$::= \cdots$	(Cmd)

TYPES

	T, U, V	$::= \alpha^* \mid T \to^\varepsilon T \mid D\ \overline{T} \mid \forall\alpha^\kappa.T$	(Typ)
	D	$::= \mathsf{Unit} \mid \mathsf{Nat} \mid \mathsf{Bool} \mid \mathsf{List}[T] \mid \cdots$	(Data)
	ε	$::= \alpha^\mathsf{e} \mid \langle\rangle \mid \langle com, \varepsilon\rangle$	(Row)
	κ	$::= * \mid \mathsf{e}$	(Kind)
	T^*	$::= T$	(TStar)
	T^e	$::= \varepsilon$	(TRow)
	Γ	$::= \varnothing \mid \Gamma, x : T \mid \Gamma, \alpha^\kappa$	(TCtx)

Figure 3.4: Syntax of λ_{cart}.

3.2.7 *Summary*

In summary, the injection mechanism via *?restriction* is the main extension point for changing the correlation behavior. Importantly, this mechanism avoids unclear and hard-coded correlation parameters that numerous CEP and stream query languages exhibit. We support scoped variations of the correlation semantics at different places in a program.

3.3 CORE LANGUAGE AND DATA TYPES

This section defines the formal syntax and semantics of a core language for effects and handlers, λ_{cart}. In Section 3.4, we will define the semantics of Cartesius' high level correlation patterns (Section 3.2) by a translation into λ_{cart}.

The core language λ_{cart} is based on the second-order call-by-value λ-calculus, extended with algebraic data types, pattern matching and recursion. In addition, it features native support for algebraic effects, effect handlers and row-based effect polymorphism. λ_{cart} is similar to Koka [Lei17b], based on a Hindley-Milner (HM) type-discipline, whereas our presentation borrows heavily from Biernacki et al. [Bie+18]. That is, we employ explicit type/effect abstraction and subtyping in a variant of the second-order λ-calculus (System F, Reynolds [Rey74] and Girard [Gir72]). However, we do not include the *shift* operator from Biernacki et al.'s work. We consider it important future work how this operator can help improve the effect safety of event correlation computations in Cartesius.

We assume that algebraic data type signatures are pre-defined and well-formed, e.g.,

$$\mathbf{type}\ \mathsf{List}[T] := \mathsf{nil} \mid \mathsf{cons}\ T\ \mathsf{List}[T]$$

EXPRESSION TYPING $\boxed{\Gamma \vdash e : T|\varepsilon}$

$$\Gamma, x{:}T, \Gamma' \vdash x : T|\langle\rangle \qquad \frac{\Gamma, x{:}T_1 \vdash e : T_2|\varepsilon}{\Gamma \vdash \lambda x.e : T_1 \to^{\varepsilon} T_2|\langle\rangle} \qquad \frac{\Gamma, \alpha^{\kappa} \vdash e : T|\langle\rangle}{\Gamma \vdash \Lambda\alpha^{\kappa}.e : \forall\alpha^{\kappa}.T|\langle\rangle}$$

$$\frac{\Gamma \vdash e_1 : T_1 \to^{\varepsilon} T_2|\varepsilon \qquad \Gamma \vdash e_2 : T_1|\varepsilon}{\Gamma \vdash e_1\ e_2 : T_2|\varepsilon} \qquad \frac{\Gamma \vdash e : \forall\alpha^{\kappa}.T_1|\varepsilon}{\Gamma \vdash e\ [T_2^{\kappa}] : T_1[T_2^{\kappa}/\alpha^{\kappa}]|\varepsilon} \qquad \frac{\Gamma \vdash e : T \to^{\varepsilon} T|\varepsilon}{\Gamma \vdash \mathbf{fix}\ e : T|\varepsilon}$$

$$\frac{\mathsf{k}\ (T_i)_{1\le i\le n} \in D\ \overline{S} \qquad (\Gamma \vdash e_i : T_i|\varepsilon)_{1\le i\le n}}{\Gamma \vdash \mathsf{k}\ (e_i)_{1\le i\le n} : D\ \overline{S}|\varepsilon} \qquad \frac{\begin{array}{c}\Gamma \vdash e : T_1|\varepsilon \qquad (\Gamma \vdash p_i : T_1, \Gamma_i)_{1\le i\le n} \\ (\Gamma, \Gamma_i \vdash e_i : T_2|\varepsilon)_{1\le i\le n} \qquad (p_i)_{1\le i\le n}\ \text{covers}\ T_1\end{array}}{\Gamma \vdash \mathbf{match}\ e\ \{(p_i \mapsto e_i)_{1\le i\le n}\} : T_2|\varepsilon}$$

$$\frac{com_{\overline{S}}{:}\ T_1 \to T_2}{\Gamma \vdash com_{\overline{S}} : T_1 \to^{\langle com_{\overline{S}}\rangle} T_2|\varepsilon} \qquad \frac{\Gamma \vdash e : T_1|\langle\overline{com}, \varepsilon\rangle \qquad \Gamma \vdash h : T_1\ ^{\langle\overline{com}\rangle}\!\Rightarrow^{\varepsilon} T_2}{\Gamma \vdash \mathbf{handle}\ \{h\}\ e : T_2|\varepsilon}$$

$$\frac{\Gamma \vdash e : T_1|\varepsilon_1 \qquad \Gamma \vdash T_1 \le T_2 \qquad \Gamma \vdash \varepsilon_1 \le \varepsilon_2}{\Gamma \vdash e : T_2|\varepsilon_2}$$

HANDLER TYPING $\boxed{\Gamma \vdash h : T_1\ ^{\varepsilon_1}\!\Rightarrow^{\varepsilon_2} T_2}$

$$\frac{\begin{array}{c}\Gamma, x_o{:}T_1 \vdash e_o : T_2|\varepsilon \qquad (com_i{:}\ U_i \to V_i)_{1\le i\le n} \\ (\Gamma, x_i{:}\ U_i, r_i{:}\ V_i \to^{\varepsilon} T_2 \vdash e_i : T_2|\varepsilon)\end{array}}{\Gamma \vdash \{x_o \mapsto e_o; (com_i\ x_i\ r_i \mapsto e_i)_{1\le i\le n}\} : T_1\ ^{\langle com_1,\ldots,com_n\rangle}\!\Rightarrow^{\varepsilon} T_2}$$

SUBTYPING $\boxed{\Gamma \vdash T_1^{\kappa} \le T_2^{\kappa}}$

$$\Gamma \vdash T^{\kappa} \le T^{\kappa} \qquad \Gamma \vdash \langle\rangle \le \varepsilon \qquad \Gamma \vdash \langle com_1, com_2, \varepsilon\rangle \le \langle com_2, com_1, \varepsilon\rangle \qquad \frac{\Gamma \vdash \varepsilon_1 \le \varepsilon_2}{\Gamma \vdash \langle com, \varepsilon_1\rangle \le \langle com, \varepsilon_2\rangle}$$

$$\frac{\Gamma \vdash T_1^{\kappa} \le T_2^{\kappa} \qquad \Gamma \vdash T_2^{\kappa} \le T_3^{\kappa}}{\Gamma \vdash T_1^{\kappa} \le T_3^{\kappa}} \qquad \frac{\Gamma, \alpha^{\kappa} \vdash T_1 \le T_2}{\Gamma \vdash \forall\alpha^{\kappa}.T_1 \le \forall\alpha^{\kappa}.T_2}$$

$$\frac{\Gamma \vdash T_1' \le T_1 \qquad \Gamma \vdash \varepsilon_1 \le \varepsilon_2 \qquad \Gamma \vdash T_2 \le T_2'}{\Gamma \vdash (T_1 \to^{\varepsilon_1} T_2) \le (T_1' \to^{\varepsilon_2} T_2')}$$

Figure 3.5: Type System of λ_{cart}.

EVALUATION CONTEXTS

$$\begin{array}{lll} \mathcal{E} & ::= & [\cdot] \mid \mathcal{E}\ e \mid v\ \mathcal{E} \mid \mathcal{E}\ [T^\kappa] \mid \mathsf{k}\ \overline{v}\ \mathcal{E}\ \overline{e} \mid \mathbf{fix}\ \mathcal{E} \mid \mathbf{match}\ \mathcal{E}\ \{\overline{p \mapsto e}\} \mid \mathbf{handle}\ \{h\}\ \mathcal{E} & \text{(ECtx)} \\ \mathcal{X}_{com} & ::= & [\cdot] \mid \mathcal{X}\ e \mid v\ \mathcal{X} \mid \mathcal{X}\ [T^\kappa] \mid \mathsf{k}\ \overline{v}\ \mathcal{X}\ \overline{e} \mid \mathbf{fix}\ \mathcal{X} \mid \mathbf{match}\ \mathcal{X}\ \{\overline{p \mapsto e}\} & \text{(XCtx)} \\ & & \mid \mathbf{handle}\ \{h\}\ \mathcal{X} \qquad \text{where } com \notin \mathcal{C}(h) \end{array}$$

HANDLER CAPABILITIES $\mathcal{C}(h)$: $\mathcal{C}(x \mapsto e) = \{\}$ $\mathcal{C}(h; com\ x\ y \mapsto e) = \{com\} \cup \mathcal{C}(h)$

DYNAMICS $\boxed{e_1 \longrightarrow e_2}$

$$\begin{array}{rcll} \mathcal{E}[(\lambda x.e)\ v] & \longrightarrow & \mathcal{E}[e\{v/x\}] & (\beta) \\ \mathcal{E}[(\Lambda\alpha^\kappa.e)\ [T^\kappa]] & \longrightarrow & \mathcal{E}[e[T^\kappa/\alpha^\kappa]] & \text{(TApp)} \\ \mathcal{E}[\mathbf{match}\ v\ \{(p_i \mapsto e_i)_{1\le i\le n}\}] & \longrightarrow & \mathcal{E}[e_j\sigma], \text{ where } v \Downarrow_{(p_i)_{1\le i\le n}} \langle p_j, \sigma\rangle & \text{(match)} \\ \mathcal{E}[\mathbf{fix}\ (\lambda x.e)] & \longrightarrow & \mathcal{E}[e\{\mathbf{fix}\ (\lambda x.e)/x\}] & \text{(fix)} \\ \mathcal{E}[\mathbf{handle}\ \{x \mapsto e; h\}\ v] & \longrightarrow & \mathcal{E}[e\{v/x\}] & \text{(ret)} \\ \mathcal{E}[\mathbf{handle}\ \{h\}\ \mathcal{X}_{com}[com\ v]] & \longrightarrow & \mathcal{E}[e\{v/x, (\lambda y.\mathbf{handle}\ \{h\}\ \mathcal{X}_{com}[y])/r\}], & \text{(handle)} \\ & & \text{where } (com\ x\ r \mapsto e) \in h,\ y \text{ is fresh} & \end{array}$$

PATTERN MATCHING $\boxed{v \Downarrow_p \sigma \qquad v \Downarrow_{\overline{p}} \langle p', \sigma\rangle}$

$$v \Downarrow_x \{v/x\} \qquad \frac{(v_i \Downarrow_{p_i} \sigma_i)_{i\in\{1\ldots n\}}}{(\mathsf{k}\ v_1 \ldots v_n) \Downarrow_{(\mathsf{k}\ p_1\ldots p_n)} \uplus_{i=1}^n \sigma_i} \qquad \frac{v \Downarrow_p \sigma}{v \Downarrow_{p\,\overline{p}'} \langle p, \sigma\rangle} \qquad \frac{\forall \sigma'.v \not\Downarrow_p \sigma' \qquad v \Downarrow_{\overline{p}'} \langle p'', \sigma\rangle}{v \Downarrow_{p\,\overline{p}'} \langle p'', \sigma\rangle}$$

Figure 3.6: Dynamic Semantics of λ_{cart}.

is the type of lists. Examples of data values are: $\langle\rangle$ is the unit value of type Unit, true and false are of type Bool, $(\text{cons}_T\ v\ \text{nil}_T)$ is of type List$[T]$ if v is of type T, (S (S o)) is of type Nat and $\langle v_1, v_2\rangle$ is of pair type $\langle T_1, T_2\rangle$ if v_1 (resp. v_2) is of type T_1 (resp. T_2). For readability, we write numeric literals for Nat in the obvious way, and write list values in the usual bracket notation, e.g., $[0, 1] = \text{cons}_{\text{Nat}}\ 0\ (\text{cons}_{\text{Nat}}\ (\text{S}\ 0)\ \text{nil}_{\text{Nat}})$.

3.3.1 *Formal Syntax*

Figure 3.4 shows the formal syntax of λ_{cart}, which we explain in the following.

EXPRESSIONS We assume a countable set of expression variables and type variables. Because we support polymorphism over both values and effect rows, we annotate type variables with the kind $*$ (type) or e (effect). In examples, we sometimes omit the kind if it is unambiguous and we omit explicit type abstraction and application.

Values v are either a λ-abstraction, or a polymorphic type or effect abstraction. Furthermore, values are applications of data constructors to values, written $\mathsf{k}\ \overline{v}$, and correspondingly $D\ \overline{T}$ for instantiations of data types, in a fashion similar to Lindley, McBride, and McLaughlin [LMM17]. Finally, effect operations *com* are values as well, since commands have function-type signatures (cf. Section 2.3.3).

An expression e is either an expression variable, a value, an application, a type or effect application, a data constructor application, the fixpoint of an expression for recursive definitions, a pattern matching expression with a sequence of pattern clauses or the application of a second-class handler h to a computation e. Patterns match against nested data constructor applications and may bind variables, where we assume the usual linear patterns, i.e., no variable symbol occurs more than once in a pattern p.

TYPES A type is either a type or effect variable, depending on the kind, a function type, where ε indicates the which effects the function may induce, an instantiation of a predefined data type $D\ \overline{T}$ or a universal type, which may quantify either over types or effects.

As already mentioned, we assume that data types D are predefined, well-formed and always receive the right number of type arguments.

Similar to Koka [Lei17b], effect rows consist of a sequence of effect names, terminated by either an empty row or an effect type variable.

To streamline typing and subtyping rules, we sometimes write T^{κ} to either mean types or effect rows, depending on the kind annotation κ.

As is usual, we have typing contexts Γ, which are finite sequences containing assumptions about the types of free variables, or the bound type and effect variables in scope.

3.3.2 *Static Semantics*

Figure 3.5 shows the typing and subtyping rules of λ_{cart}, which we explain in the following.

EXPRESSION TYPING The formal judgment $\Gamma \vdash e : T|\varepsilon$ intuitively means that under typing context Γ, the expression e has type T and may induce the effects ε.

The rule for variables is standard and requires that the variable is bound in the current typing context. We assume that this is the rightmost variable binding, i.e., x is not bound in Γ'.

The rule for λ-abstraction is standard. Note that the effects ε induced by the body of the abstraction become latent, i.e., they are annotated at the function arrow and induced once the function is applied to an argument.

Following Biernacki et al. [Bie+18], the rule for type and effect abstraction requires that the abstracted expression e is free of side effects.

The rule for expression application induces the side effects which are latent in the function type. Applying a type or effect to a polymorphic value instantiates the polymorphic type accordingly.

The rule for fixpoints is standard and includes the effects of the definition.

A data constructor application is well-typed, if the number and types of the argument expressions match the requirements in the data type specification.

The rule for pattern matching on data types ensures that the pattern clauses exhaustively cover the data type definition.

As explained before, pre-defined commands have a function-type signature and induce themselves in the effect row when applied.

The rule for applying a handler to an expression specifies that a handler eliminates part of the induced effects in the effect row of the expression. We will give more detailed explanations on effect handling in the subsequent discussions.

Finally, the subsumption rule is standard. If an expression can be typed with T_1 and effects ε_1, then it can be typed with any other type T_2 and effects ε_2, which are at most as precise as the former, according to the subtyping relation, explained below. This rule can be used to unify the effects in sub-expressions of compound expressions.

HANDLER TYPING The formal judgment $\Gamma \vdash h : T_1 \, ^{\varepsilon_1}\!\Rightarrow^{\varepsilon_2} T_2$ intuitively states that the effect handler h transforms a computation of type T_1 inducing effects ε_1 into a computation of type T_2 inducing effects ε_2. Handlers have a return clause, specifying how to transform the result of the handled computation, as well as clauses for commands. For regularity, we assume that commands take exactly one argument type (multiple arguments can be encoded with data types). A command clause additionally binds a second variable, which is the underlying resumption of the computation, accepting the codomain type of the commands' type signature. We give more detailed explanations on the dynamic and static semantics of handlers below.

SUBTYPING The subtyping rules govern how effectful computations can be lifted and composed. Essentially, they model that the effect rows behave like multisets of command names, and thus effects can be permuted so that expressions fit into different usage contexts. Note that the usual contravariance rule applies to the domain of function types, whereas both the codomain and effect annotation are covariant.

3.3.3 *Dynamic Semantics*

Figure 3.6 shows the operational semantics of λ_{cart} in terms of the reduction relation $e_1 \longrightarrow e_2$ on expressions, using evaluation contexts (Felleisen and Hieb [FH92]). The rules (β), (TAPP), (MATCH) and (FIX) are standard, governing function application, type application, pattern matching and recursion, respectively.

3.3.4 *Algebraic Effects and Handlers*

Algebraic effects and handlers [PP03; PP09] enable structured programming with user-defined effects in pure functional languages. Compared to more established language abstractions for effects, i.e., monads [Mog89; Mog91; Mog89; Wad92] and monad transformers [LHJ95], algebraic effects and handlers compose more freely and conveniently, because they support modular instantiation and modular abstraction via effect interfaces [KLO13]. Semantically, algebraic effects can be modeled in terms of free monads [Swi08; KI15]. However, in this work, we treat effects and handlers as language primitives. Intuitively, handlers and effect commands are generalizations of **try/catch/throw** for managed exceptions. The difference is that handlers can resume evaluation at the point where the effect (e.g., thrown exception) occurred.

Correspondingly, λ_{cart} specifies the syntactic sort *com* for a set of *commands* (Figure 3.4), designating effectful operations. We assume that each command *com* has a predefined type $com\colon T_1 \to T_2$, i.e., commands are functions from the client's perspective. We write concrete commands in bold blue font, e.g., **yield**: $\alpha \to \mathsf{Unit}$.

An effect handler h is a finite sequence of clauses, with one mandatory return clause $x \mapsto e$ and optional command clauses $com\ x\ r \mapsto e$, specifying how to handle/interpret a specific selection of commands. We always assume that for each command there is at most one corresponding clause in each handler. Handlers are second-class and applied to computations invoking effects using the **handle** $\{h\}\ e$ form. Intuitively, a handler h is a computation transformer $T_1\ {}^{\langle\overline{com}\rangle}\!\Rightarrow^{\varepsilon}\ T_2$, turning a computation of result type T_1 and effects $\langle\overline{com}\rangle$ (handled by h's clauses) to a computation with result T_2 and new effects ε (cf. handler typing in Figure 3.5).

Reduction rules (RET) and (HANDLE) (Figure 3.6) govern the behavior of handler application. The former rule applies the return clause[2] of the handler for transforming the final result of the computation. The latter rule specifies how to handle effects invoked by the computation. Similarly to managed exceptions, a command invocation *com v* shifts the control flow to the currently innermost handler with the capability to handle *com*. Just as Leijen [Lei17b] does, we express this by restricting the evaluation context $\mathcal{X}_{com}$, i.e., evaluation may focus under a handler application only if h does not handle *com*.

Once control flow shifts into a handler clause $com\ x\ r \mapsto e$, the argument to the command is bound to the first variable x and the *resumption* of the computation is bound to the second variable r. That is, e has the capability to resume the command invocation with an answer value. Hence, effects and handlers implement a more structured form of delimited continuation [KLO13; BP15; For+17]. Note that λ_{cart} employs *deep handlers*, i.e., the resumption re-applies the current handler to the rest of the computation, reflecting the intuition that handlers are folds over computation trees [Lin14].

Effect Typing

We assume a row-based type and effect system as in Koka, which assigns effect rows to arrow types. For example,

$$map\colon \forall\,\alpha\,\beta\,\mu.(\alpha \to^{\langle\mu\rangle} \beta) \to^{\langle\rangle} \mathsf{List}[\alpha] \to^{\langle\mu\rangle} \mathsf{List}[\beta]$$

is effect polymorphic, indicated by the universally quantified effect variable μ. Because *map* applies the supplied function elementwise to its second argument, it overall induces the same effects μ. The order of effects in rows does not matter, e.g., $\langle$**yield**, **fail**, $\mu\rangle$ is equivalent to $\langle$**fail**, **yield**, $\mu\rangle$, which is ensured by subtyping and subsumption (Figure 3.5).

For simplicity, in contrast to Koka, we do not group commands into effect interfaces. Instead, each command induces itself as effect, e.g., if $com\colon T_1 \to T_2$ is the predefined signature of *com*, then $T_1 \to^{\langle com\rangle} T_2$ is the type assigned to *com* at the level of expressions. However, in examples we sometimes abbreviate sets of commands in effect rows by a single effect name. We further allow polymorphic commands, writing type parameters and instantiation as subscripts, e.g., $\mathbf{return}_{\alpha}\colon \alpha \to \mathsf{Unit}$.

2 In examples, we omit the return clause, if it is an identity $x \mapsto x$ and omit curly braces in favor of indentation.

DERIVED SYNTAX

$$\begin{array}{rcll}
\textbf{let } x = e_1 \textbf{ in } e_2 & \rightsquigarrow & (\lambda x.e_2)\ e_1 & \text{(Let)} \\
\textbf{let rec } x = e_1 \textbf{ in } e_2 & \rightsquigarrow & (\lambda x.e_2)\ (\textbf{fix}\ \lambda x.e_1) & \text{(Letrec)} \\
e_1; e_2 & \rightsquigarrow & (\lambda x.e_2)\ e_1, \text{ where } x \notin \mathrm{fv}(e_2) & \text{(Seq)} \\
\{T\}^{\varepsilon} & \rightsquigarrow & (\mathsf{Unit} \to^{\varepsilon} T) & \text{(Thunk Typ)} \\
\{e\} & \rightsquigarrow & \lambda x.e, \text{ where } x \notin \mathrm{fv}(e) & \text{(Thunk)} \\
\textbf{handler}\{h\} & \rightsquigarrow & \lambda thnk.\textbf{handle}\,\{h\}\,(thnk\ \langle\rangle) & \text{(Handler Val)} \\
\textbf{with } e_1\ e_2 & \rightsquigarrow & e_1\,\{e_2\} & \text{(With)} \\
?var & \rightsquigarrow & \textbf{var}\ \langle\rangle & \text{(Implicit Use)} \\
\textbf{let implicit } ?var\colon T = e_1 \textbf{ in } e_2 & \rightsquigarrow & \textbf{let } x = e_1 \textbf{ in } (\textbf{handle}\,\{\textbf{var}\ y\ k \mapsto k\ x\}\ e_2), & \text{(Implicit Def)} \\
 & & \text{where } x, y, k \text{ are fresh and } \textbf{var}\colon \mathsf{Unit} \to T &
\end{array}$$

HANDLER COMBINATORS

$$\_ \boxplus \_ := \lambda hl_1.\lambda hl_2.\lambda thnk.\textbf{with } hl_1 \textbf{ with } hl_2\ (thnk\ \langle\rangle) \quad \text{(Handler Composition)}$$

Figure 3.7: Derived Syntax and Combinators.

$$\textbf{handler}(y = e)\{x_0 \mapsto e_0;\ (com_i\ x_i, k_i \mapsto e_i)_{1\le i\le n}\} \quad \rightsquigarrow \quad \lambda thnk.\,(\textbf{handle}\,\{h'\}\ (thnk\ \langle\rangle))\ e$$
$$\text{where } h' = (x_0 \mapsto \lambda y.e_0;\ (com_i\ x_i, k_i \mapsto \lambda y.e_i)_{1\le i\le n})$$

Figure 3.8: Derived Syntax for Parameterized Effect Handlers.

First Class Handlers and Combinators

Recall from Section 3.2 that we frequently use (1) combinators to construct restriction handlers as *values*, (2) handler composition $\boxplus$ and (3) implicit variables for injecting handlers. Accordingly, we define convenience syntax for first class handlers and thunks in terms of functions and second class handlers (Figure 3.7). Encoding first class handlers in this way keeps the core language simple, i.e., handler values become thunk-accepting functions and handler application becomes function application. Correspondingly, if a second class handler h has type $T_1\ ^{\langle\overline{com}\rangle}\Rightarrow^{\varepsilon}\ T_2$, then its first class encoding $\textbf{handler}\{h\}$ has type $\{T_1\}^{\langle\overline{com},\varepsilon\rangle} \to^{\varepsilon} T_2$ (cf. Figure 3.5).

To clearly convey the intent (and for aesthetic reasons), we use Eff-style **with** syntax [BP15] for applying handler values. Composing handlers ($\boxplus$) simply becomes nested handler application.

We further stipulate the usual **let** and **let rec** binding forms, and sequencing $e_1; e_2$.

We also make frequent use of *parameterized handlers* [PP09; KLO13] in a notation similar to Koka, i.e., $\textbf{handler}(x = e)\{h\}$, binding the result value of e to a variable x, which is accessible from all clauses of handler h, and can be modified in the resumption. Figure 3.8 shows the syntax translation for first-

```
1 let accum = λn. handler(s = n)
2   x ↦ s
3   yield val resume ↦
4     resume ⟨⟩ (s + val)
```

```
1 let interactive = handler
2   yield val resume ↦
3     println val;                 //print to console
4     match readchar ⟨⟩            //read keyboard
5       \cr ↦ resume ⟨⟩            //proceed on carriage return
6       _   ↦ ⟨⟩                   //otherwise, terminate
```

Figure 3.9: Different Interpretations of yield.

class parameterized handlers, as a variant of the first-class handler encoding from Figure 3.7. The translation binds the annotated handler parameter y as a λ-abstraction in each individual clause of the original handler. We show a usage example of parameterized handlers immediately below.

Example

In the following example, we consider the command yield : Nat → Unit and the computation

$$c := \mathit{map}\ (\mathbf{yield})\ [1, 2, 3, 4],$$

where *map* is the standard mapping function on lists. We may turn this mapping computation into an accumulating computation, *without changing* c, by enclosing the computation with an appropriate handler for yield, e.g., *accum* in Figure 3.9. This handler sums up the yielded values in its handler parameter and would take the following high-level evaluation steps:[3]

$$\begin{aligned}
&\mathbf{with}\ (\mathit{accum}\ 0)\ (\mathit{map}\ (\mathbf{yield})\ [1, 2, 3, 4]) \\
\longrightarrow^*\ &\mathbf{with}\ (\mathit{accum}\ 0)\ (\mathsf{cons}\ (\mathbf{yield}\ 1)\ (\mathit{map}\ (\mathbf{yield})\ [2, 3, 4])) \\
\longrightarrow^*\ &\mathbf{with}\ (\mathit{accum}\ 1)\ (\mathsf{cons}\ \langle\rangle\ (\mathit{map}\ (\mathbf{yield})\ [2, 3, 4])) \\
\longrightarrow^*\ &\mathbf{with}\ (\mathit{accum}\ 3)\ (\mathsf{cons}\ \langle\rangle\ (\mathsf{cons}\ \langle\rangle\ (\mathit{map}\ (\mathbf{yield})\ [3, 4]))) \\
\longrightarrow^*\ &\mathbf{with}\ (\mathit{accum}\ 10)\ [\langle\rangle, \langle\rangle, \langle\rangle, \langle\rangle] \\
\longrightarrow\ &10.
\end{aligned}$$

Note that the resumption of a parameterized handler now takes an extra argument, which is the new value of the handler parameter (Line 4 of *accum*).

As the previous example shows, handlers give meaning to commands, such as yield. We can easily give it a quite different meaning by applying another handler, e.g., *interactive* in Figure 3.9. This handler makes c print its elements to the console, where its progression is controlled by the user's keyboard interactions. The choice to progress has "flipped" from *internal* to *external* choice, thanks to handler clauses exposing the resumption. That is, effects and handlers support coroutining [MI09; Con63].

3 We leave a proper reduction trace as an exercise to the reader.

SYNTAX

$$\tau \in \mathcal{T} \qquad \text{(TStamp)}$$
$$b ::= \tau \mid \bot \mid \top \qquad \text{(TBound)}$$
$$v ::= \ldots \mid b \qquad \text{(Val)}$$
$$e ::= \ldots \mid e \sqcup e \mid e \leq e \qquad \text{(Exp)}$$

DYNAMICS

$$\frac{b_1 \leq b_2}{b_1 \leq b_2 \longrightarrow \mathsf{true}}\ (\leq\text{T}) \qquad \frac{b_1 \not\leq b_2}{b_1 \leq b_2 \longrightarrow \mathsf{false}}\ (\leq\text{F}) \qquad \frac{\tau_1'' = \tau_1 \sqcap \tau_1' \qquad \tau_2'' = \tau_2 \sqcup \tau_2'}{\langle \tau_1, \tau_2 \rangle \sqcup \langle \tau_1', \tau_2' \rangle \longrightarrow \langle \tau_1'', \tau_2'' \rangle}\ (\sqcup)$$

Figure 3.10: Syntax and Semantics of Time Stamps and Intervals.

Implicit Variables as Effects.

We encode implicit variables and bindings from Section 3.2 in terms of effects (Figure 3.7, rules *(Implicit Def)* and *(Implicit Use)*). For each implicit variable declaration *?var* we associate a command **var**: Unit → *T* (without question mark). Clients (i.e., e_2 in *(Implicit Def)*) invoke this command to retrieve the bound value, where an occurrence *?var* desugars into the command invocation **var** ⟨⟩ by rule *(Implicit Use)*. Note that effect typing naturally enables static tracking of implicit dependencies, e.g., the effect row assigned to e_2 would have the shape ⟨**var**, ε⟩, mentioning the occurrence of *?var*. This form of implicit variables resembles the *dynamic scoping with static types* work by Lewis et al. [Lew+00], since command dispatch and hence accessing the implicit value is dynamic. For static implicits, as in Haskell or Scala, we would require coeffects [POM14].

The Role of Handlers in Cartesius

We can think of commands as effect constructors in computations. Dually, handlers deconstruct/observe effects as the computation unravels, i.e., they are *co-algebraic*. Later, in Section 3.4, we exploit this intuition to encode event notifications as effects and event observers as handlers. Yet, this is not the only use of handlers: Similarly to the *interactive* example above, handlers enable us to "flip" from direct style to callback style, transparently. Last, but not least, handlers realize restrictions (Section 3.2.4) of the cartesian product in the sense of Proposition 3.2. That is, effects occurring in the cartesian product computation can be locally reinterpreted by handlers, changing how the computation behaves.

3.3.5 *Time and Event Values*

We assume a discrete time model to track the arrival times of event notifications (Figure 3.10). The set $\mathcal{T}$ is an infinite set of discrete, totally ordered *time stamps* τ. In timing predicates, we lift time stamps to *time bounds* b, extending $\mathcal{T}$ to a complete lattice with a least element $\bot$ ("minus infinity"), a greatest element $\top$ ("plus infinity"), and greatest lower bound ($\sqcap$) as well as least upper bound ($\sqcup$) operations. We overload $\leq$ to denote the lifted order on time bounds.

Recall that an *event value* is evidence of a past situation of interest (Section 3.2.1). It is either supplied externally or the result of joining one to many other event values. Event values have type **type** Ev[T] := ev T ⟨Time, Time⟩, where the first component is the *content/payload* and the second component is the *occurrence time interval* $\langle \tau_1, \tau_2 \rangle$ in pair notation. For each external event, we assume that the runtime assigns a discrete time interval containing only the event's arrival time. Joining n event values $(\text{ev } v_1\ i_1), \ldots, (\text{ev } v_n\ i_n)$ results in an event value $(\text{ev } f(v_1, \ldots, v_n)\ (i_1 \sqcup \ldots \sqcup i_n))$, which merges the payloads via a function f and the time intervals with the $\sqcup$ operator, yielding the smallest interval containing all given intervals. In Cartesius, the merge function f corresponds to the **yield** expression in patterns (Section 3.2.2). We discuss it further in Section 3.4.

3.3.6 *Asynchrony and Push/Pull Reactives*

Since event sources are asynchronous, we must commit to a push-based processing model. Inadvertently, this leads to a programming style where control is inverted, requiring difficult to manage callbacks ("callback hell" [Edw09]). We already demonstrated by the *interactive* example above how effect handlers reconcile inversion of control with direct style, iterative programs. What remains to be shown is how reactives from Section 3.2.1 integrate with this programming pattern.

In λ_{cart}, reactives R[T] correspond to a co-inductive list type, which is inspired by Elliott [Ell09]:

$$\textbf{type}\ \mathsf{R}[T] := \mathsf{Future}[\mathsf{R}'[T]] \qquad \textbf{type}\ \mathsf{R}'[T] := \mathsf{rnil} \mid \mathsf{rcons}\ \mathsf{Ev}[T]\ \mathsf{R}[T].$$

The data type Future[T] represents async/await style futures [Bie+12; SPL11; HM19], which implement the potential $\Diamond[T]$ of an event source to yield its next event value. Futures can be implemented on top of algebraic effects. We elide the definition and treat Future[T] as abstract (cf. Leijen [Lei17a] and Dolan et al. [Dol+17] for possible implementations). The only way to introduce and eliminate future values is via effects. Command

$$\textbf{async}_{\alpha,\mu} : \{\alpha\}^{\langle\mu\rangle} \to \mathsf{Future}[\alpha]$$

immediately returns a future to the caller, asynchronously executing the given thunk. The elimination command

$$\textbf{await}_{\alpha} : \mathsf{Future}[\alpha] \to \alpha$$

returns the result value of a completed future, if it is completed and otherwise suspends the caller. In effect rows, we simply abbreviate all the effects associated with asynchrony as **async**. We assume that at the top-level, there is always a handler present for handling the **async** effects.

Since R[T] resembles a (possibly infinite) list, we can now write direct style iterative programs over the events originating from an event source, in a fashion similar to the *interactive* example. For this purpose, we define the polymorphic iteration combinator *eat* (Figure 3.11), which applies a function elementwise to all events of a given reactive.

The positions of R[T] wrapped in Future[$\cdot$] precisely mark where internal choice (pull) may "flip" to external choice (push). That happens if the next event is not yet available (Line 2 of *eat*). Thus, we have raised the level of abstraction and do not need to worry about low level callback functions.

```
1 let rec eat: ∀α T.R[T] →⟨⟩ (Ev[T] →⟨α⟩ Unit) →⟨async,α⟩ Unit =
2   Λα.ΛT.λr.λf. match (await_T r)
3     rcons hd tl ↦ (f hd); eat_{α,T} tl f
4     rnil        ↦ ⟨⟩
```

Figure 3.11: Direct Style, Asynchronous Iteration.

3.3.7 *Interleaving*

We conclude this section with one more abstraction for asynchrony. Recall from Figure 3.3, that correlation patterns require concurrent executions of threads/strands. One can define a combinator for that, called *interleave*, on top of the asynchrony system above. Here is its simplified signature:

$$\mathit{interleave}\colon \forall \mu.\mathsf{List}[\{\mathsf{Unit}\}^{\langle \mathbf{async},\mu\rangle}] \to^{\langle \mathbf{async},\mu\rangle} \mathsf{Unit}.$$

The combinator concurrently executes a list of independent, asynchronous computation strands, passed as thunks. We elide its implementation and refer for the details to Leijen [Lei17a]. Moreover, we assume that the scheduling of the concurrent execution is fair and allow the syntax $e_1 \parallel \ldots \parallel e_n \rightsquigarrow \mathit{interleave}\,[e_1, \ldots, e_n]$ for readability.

The above type is noteworthy: The effect polymorphism in *interleave* states that all the effects occurring in the given asynchronous strands *are observable by the caller*. For example,

$$\{\mathbf{await}_T\ x\} \parallel \{\mathbf{println}\ \mathit{"foo"}\} \parallel \{\mathbf{yield}\ 1\}$$

induces the effects ⟨**async**, **println**, **yield**⟩ to the context in which it is invoked.

This is an important property of the interleaving combinator, which we exploit for the implementation of Cartesius in the next section.

3.4 EVENT CORRELATION WITH ALGEBRAIC EFFECTS

In Section 3.3, we defined required preliminaries for Cartesius in terms of λ_{cart}, from effects and handlers to event values, reactives and interleaving. In this section, we present the implementation of Cartesius from Section 3.2 as a shallow embedding in λ_{cart}. Our design follows the computational interpretation of joins (Proposition 3.2 and Figure 3.3). In particular, while Section 3.2 motivated Cartesius from the perspective of end users, this section explains the overall framework that enables library writers to define new, custom join behaviors.

We prototyped Cartesius in two languages with native support for algebraic effects, Koka [Lei17b] and multicore OCaml [Dol+17]. The implementations are available at:

`http://bracevac.org/correlation`

3.4.1 *The Marriage of Effects and Joins: Correlate by Handling*

We start with motivating the underlying structure of correlation computations in Cartesius (depicted in Figure 3.3), which yields a useful, modular organization principle in terms of effects and handlers.

One can straightforwardly encode event correlation behavior over $n \geq 1$ reactives by *nested iteration*. Recall that in Section 3.3, we introduced abstractions for direct style iteration over asynchronous reactives. In particular, we defined the higher order iteration function *eat* (Figure 3.11), which "feeds" observed events one-by-one to its function argument f. Thus, nested iteration takes the form

$$eat\ r_1\ (\lambda ev_1.eat\ r_2\ (\lambda ev_2.\cdots eat\ r_n\ (\lambda ev_n.\ process\ \langle ev_1,\ldots,ev_n\rangle)\cdots))$$

for reactives r_1:R$[T_1]$,…, r_n:R$[T_n]$ and a function *process*: (Ev$[T_1]$, …, Ev$[T_n]$) → Unit, which is a callback on completed n-tuples, encapsulating a concrete event correlation behavior. The nesting enforces a sequential selection of events, i.e., before the computation binds event ev_i, it must bind (and **await**) all events ev_j, $j < i$.

This programming pattern is not uncommon and can be found in similar contexts, e.g., language-embedded query APIs and comprehension notations based on monads [Wad90a; MBB06; CLW13]. In these APIs, multiple selections translate into nested applications of the well-known combinator

$$bind\text{:}^4\ \mathsf{M}[A] \to (A \to \mathsf{M}[B]) \to \mathsf{M}[B]$$

on a monad M$[\cdot]$. For illustration, we show an example correlation in Rx.NET, using the monad-based LINQ in Appendix A.1.

Unfortunately, such nesting can *not* faithfully model all kinds of joins over asynchronously arriving events, because of the induced sequential selection order. In event correlation, reactives produce events in an arbitrary order. For example, the *combineLatest* combinator from Section 3.2 must continue to process events from one reactive, even if the other has stopped producing events.

Interleaved Binding

The problem is that the (static) notation order of event bindings determines the dynamic selection order during event correlation, while, in general, the two should be independent. One solution is to extend monads with new combinators and laws, e.g., Joinads [PS11; PMS11]. However, the same drawbacks to using monads apply, which we mention in Section 3.3.4. In this work, we use algebraic effects and handlers for decoupling the (static) notation order of event bindings from the dynamic selection order during correlation. Instead of nesting the n iterations, we *juxtapose* them, by using the *interleave* combinator (Section 3.3.7):

$$\{eat\ r_1\ \lambda ev_1.e_1\}\ \|\ \cdots\ \|\ \{eat\ r_n\ \lambda ev_n.e_n\}.$$

This composition puts each iteration in a separate computation strand. No event binding takes precedence over the other, all iterations proceed independently and concurrently. Our solution seamlessly fits in our setting, where algebraic effects and handlers are a key element of the design. As a side-effect, the solution also represents a novel use of algebraic effects and handlers.

4 *bind* is also known as *SelectMany* and *flatMap*.

Correlate by Handling

How can the iterations be correlated if the interleaving separates them? They can be correlated by effect handling! We simply follow the example in Section 3.3.4, applying it to reactives instead of lists. That is, we elementwise invoke effect commands:

$$\{eat\ r_1\ \lambda ev_1.\mathbf{push}_1\ ev_1\} \parallel \cdots \parallel \{eat\ r_n\ \lambda ev_n.\mathbf{push}_n\ ev_n\},$$

which after η-conversion ($\lambda x.fx \equiv f$) becomes:

$$\{eat\ r_1\ (\mathbf{push}_1)\} \parallel \cdots \parallel \{eat\ r_n\ (\mathbf{push}_n)\}. \tag{3.1}$$

We let each iteration in the interleaving invoke a distinct, *fresh* effect

$$\mathbf{push}_i\colon \mathsf{Ev}[T_i] \to \mathsf{Unit}.$$

Its purpose is to *expose each produced event as an effect*. The effect name and signature are position dependent, so that at the type and term level, one can discern which input a **push** belongs to.

Importantly, the effect type of the interleaving (3.1) is revealing the nature of event correlation. Recall from Section 3.3.7 that the effects occurring in the interleaved strands propagate to the calling evaluation context. Thus, the effect type of expression (3.1) is

$$\langle \mathbf{push}_1, \ldots, \mathbf{push}_n, \mathbf{async} \rangle,$$

which intuitively states that at any time, in any order and arbitrarily often, events from the n reactives may "pop into existence" as effect invocations. That concisely characterizes the problem of asynchronous event correlation!

Moreover, the effect type hints at how to correlate events from n reactives. The interleaving is namely the ideal place to implement event correlation, since it is where all events are exposed as effects to the calling context. That is, we enclose the interleaving (3.1) with effect handlers and *correlate by handling*:

DEFINITION 3.3 (HANDLER-BASED EVENT CORRELATION)
An asynchronous n-way join computation over reactives r_1 to r_n is a composition of effect handlers $h_1 \boxplus \cdots \boxplus h_k$ enclosing the interleaved iteration

$$\mathbf{with}\ (h_1 \boxplus \cdots \boxplus h_k)\ (\{eat\ r_1\ (\mathbf{push}_1)\} \parallel \cdots \parallel \{eat\ r_n\ (\mathbf{push}_n)\})\ \langle\rangle),$$

so that all the $\mathbf{push}_i$ are *discharged*. That is, its type has the form

$$(h_1 \boxplus \cdots \boxplus h_k)\colon \{\mathsf{Unit}\}^{\langle \mathbf{push}_1,\ldots,\mathbf{push}_n,\mathbf{async},\varepsilon\rangle} \to^{\langle \mathbf{async},\varepsilon\rangle} \mathsf{Unit}.$$

♦

The above definition yields a useful, modular organization principle for event correlation computations. Interleaved iteration can be defined once and for all, as a generic component (Figure 3.3) and there is a clearly defined structure to computations. All event occurrences "join" in the context of the interleaving, where they can be correlated by handling. Effect handlers are modular units of composition. They interpret underlying event notifications, embodied by the family of **push** effects, which form an interface. The implementation effort for event correlation reduces to programming handlers against that interface.

CARTESIUS is all about interpreting interleavings of **push** effects with a suitable handler context $(h_1 \boxplus \cdots \boxplus h_k)$. In the remainder of this section, we will define an encoding of correlation patterns (Figure 3.3) and the constrained cartesian product (Proposition 3.2) in terms of such a handler context.

```
let join_shape_n = {
  with (trigger_n(?pattern)   // match pattern
    ⊞ memory_n                // memory state
    ⊞ reify_n                 // cartesian product
    ⊞ ?restriction_n          // inject restrictions
    ⊞ forAll_n)               // store events
      ?streams_n ⟨⟩ }         // inject interleaving
```

```
let reify_n = handler
  {push_i ev resume ↦
    let entry = find ev (get_i ⟨⟩) in
      // trigger cartesian product
      forEach (trigger_n) (cartesian_i entry);
      GC ⟨⟩; // delete used up events
      resume ⟨⟩}_{1≤i≤n}
```

```
let forAll_n = handler
  {push_i ev resume ↦
    set_i ((ev, inf) :: (get_i ⟨⟩));
    resume (push_i ev)}_{1≤i≤n}
```

```
let memory_n =
  ⊞_{i=1}^{n} handler(s: List[Ev[T_i] × Count] = nil)
        get_i ⟨⟩ resume ↦ resume s s
        set_i  s' resume ↦ resume ⟨⟩ s'
```

```
type Pat_n = {Ev[T_1] × ⋯ × Ev[T_n]}^⟨yield,fail⟩
let trigger_n = handler(pattern: Pat_n)
  trigger_n tuple resume ↦
    pattern tuple; resume ⟨⟩ pattern
```

Figure 3.12: Shape of n-way Join Computations.

3.4.2 *Core Framework of Cartesius*

Now we make use of Definition 3.3 and encode a generic, extensible cartesian product computation (Section 3.2.4 and Figure 3.3) in terms of handlers.

Interface

Our encoding requires the following signature of effect commands

$$\begin{aligned}
\mathbf{push}_i &: \mathsf{Ev}[T_i] \to \mathsf{Unit},\ 1 \le i \le n \\
\mathbf{trigger}_n &: (\mathsf{Ev}[T_1] \times \cdots \times \mathsf{Ev}[T_n]) \to \mathsf{Unit} \\
\mathbf{get}_i &: \mathsf{Unit} \to \mathsf{List}[\mathsf{Ev}[T_i] \times \mathsf{Count}],\ 1 \le i \le n \\
\mathbf{set}_i &: \mathsf{List}[\mathsf{Ev}[T_i] \times \mathsf{Count}] \to \mathsf{Unit},\ 1 \le i \le n
\end{aligned}$$

which is indexed by the number n of input reactives and their event types $T_1,\ldots,T_n$.

Next to the $\mathbf{push}_i$ commands for event notifications from Section 3.4.1, there are $\mathbf{get}_i$/$\mathbf{set}_i$ commands for reading/writing a local memory ("mailbox"), one for each input. This is the effect interface to the local, shared memory component in Figure 3.3, which retains event notifications for processing. Each memory is an association list, storing currently relevant event values along with their *lifespan* of **type** Count := fin Nat | inf. This type specifies how often (finitely or arbitrarily often) an in-memory event value may be reused in combination with other event values. By default, in-memory event values have infinite lifetime (inf). This policy may be changed by restriction handlers, e.g., affine events (Section 3.2.4) would have lifespan (fin 1). Once a correlation computation materializes a candidate n tuple, it is passed to the command $\mathbf{trigger}_n$, for testing against the **where** predicate (Figure 3.3).

Generative Effects

Above, we specified an *indexed family of effects* to express that each instantiation of a join computation uses different, instance-specific version of the effects **push**, **get**, **set**, and **trigger**. Such definitions are known as *generative effects*. These kinds of effects are still an open research problem and not readily supported in current language implementations. For simplicity, we assume a sufficient supply of predefined, differently named commands. We further discuss in Section 3.5 how we approximated generative effects in our implementations.

Implementation

Here, we define the generic, extensible n-way cartesian product implementation, in terms of handlers for the $\mathbf{join}_n$ signature above. The implementation is represented by the computation thunk $join\_shape_n$ in Figure 3.12, which composes (i.e., layers) a number of sub-handlers. These sub-handlers are in close correspondence to the schematic boxes inside **correlate** in Figure 3.3, in upside down order. In the following, we explain the individual handlers comprising this shape, bottom to top.

At the bottom, there is the implicit variable (cf. Section 3.3.4)

$$?streams_n\colon \{\mathsf{Unit}\}^{\langle \mathbf{push}_1,\ldots,\mathbf{push}_n,\mathbf{async}\rangle},$$

which accepts any computation invoking the **push** effects for correlation, as an external dependency. Its is by default bound to the interleaved iterations from Definition 3.3. The binding can be overridden externally, e.g., our encoding of time windows requires binding a different computation (cf. Section 3.4.5).

The $forAll_n$ handler in $join\_shape_n$ reacts to each $\mathbf{push}_i$ command by adding the supplied event to the i-th memory with an initially unbounded lifespan. Note that the *resume* invocation at the end implements a coroutine behavior (Section 3.3.4) for the i-th iteration and forwards the current $\mathbf{push}_i$ command further up the context, to give other handlers the chance to process the command invocation. Iteration is continued as soon as one layer in the stack resumes with the unit value, or it is stopped if a handler decides to not invoke the resumption.

The $?restriction_n$ implicit variable in $join\_shape_n$ is our main extension point for changing the behavior of the cartesian product by external injection of restriction handlers. We showed its usage in Section 3.2 and postpone how to program concrete restriction handlers to Section 3.4.4.

The $reify_n$ handler in $join\_shape_n$ materializes the cartesian product over all in-memory event values having non-zero life time, each time a new $\mathbf{push}_i$ event notification reaches this layer. It then invokes the $\mathbf{trigger}_n$ command for each resulting n-tuple (see below). The function

$$cartesian_i\colon (\mathsf{Ev}[T_i] \times \mathsf{Count}) \to \mathsf{List}[\mathsf{Ev}[T_1] \times \ldots \times \mathsf{Ev}[T_n]]$$

computes the candidate tuples from the just observed event ev and the memory contents for all inputs $j \neq i$. After triggering, used-up event values are garbage collected from the memory. This is the terminal coroutine layer for all $\mathbf{push}_i$ commands, because it does not forward $\mathbf{push}_i$ effect before resuming (Line 7 of $reify_n$).

The $memory_n$ handler in $join\_shape_n$ maintains and threads the current state of the memory through the join computation. That is, it keeps the n-tuple of all n memory states in handler parameters and answers the $\mathbf{get}_i/\mathbf{set}_i$ commands (Section 3.4.2) by retrieving/updating the i-th memory.

Finally, the $trigger_n$ handler is responsible for testing each materialized candidate tuple that is propagated via the $\mathbf{trigger}_n$ effect against the constraints in the correlation pattern, which is bound to the implicit variable ?*pattern* as part of the correlation pattern translation (see next Section). Once a candidate tuple satisfies the constraints, it is marked as consumed. That is, the life time counters of its n components are decreased, and it is inserted into the output reactive of the correlation computation.

3.4.3 *Correlation Pattern Translation*

Language-integrated comprehension syntax is well-understood for monadic APIs [Wad90a], which translates into monad combinators. However, in this work, we investigate how comprehension syntax alternatively could be translated in terms of algebraic effect handlers.

Now, we translate correlation patterns (Section 3.2) into λ_{cart} (Section 3.3), in two steps: (1) desugar the body of patterns (the part enclosed by **correlate**$\{\ldots\}$) into λ_{cart} expressions and (2) instantiate a $join\_shape_n$ computation (Section 3.4.2) with the translated pattern. Figure 3.13 formalizes the two translation steps in rule (CORRELATE). We explain the details in the following.

Generative Effects for Pattern Matching

Our correlation pattern translation requires generative effects that link the pattern syntax with the correlation computation. The signatures of these effects are dependent on the number and types of variable bindings (x_i **from** e_i) and the output type in the pattern.

Effect $\mathbf{bind}_n$ declares that the correlation pattern extracts candidate n-tuples from n input reactives. Its purpose is linking the variable bindings (x_i **from** e_i) to ordinary λ_{cart} variable bindings.

Effect $\mathbf{yield}_n$ signals the run time that the resulting event should be appended to the output reactive of the correlation (cf. Figure 3.3), whereas $\mathbf{fail}_n$ signals a failed pattern match attempt.

Effects $\mathbf{time}_{x_i}$ are the effects associated with implicit variables (cf. Section 3.3.4) for binding the time values of input events in the pattern.

Step 1: Pattern Body Translation

The translation $[\![cb]\!]^n_{\text{pat}}$ maps the body of a pattern cb to a computation thunk performing pattern matching on data in λ_{cart}. All the right-hand sides of **from** are grouped into the parameter list of $\mathbf{bind}_n$ (Line 2), which is invoked to extract an n-tuple from the inputs. The left-hand side variables of **from** are grouped into binders of the pattern matching clause (Line 3), which decomposes a supplied n-tuple. The **where** and **yield** clause of the pattern are transformed into an **if** expression. If all constraints are satisfied, then the **yield** effect is invoked with the result. Otherwise, **fail** is invoked.

PATTERN SYNTAX

c ::= **correlate** cb (CPat)

cb ::= $\{\overline{x \textbf{ from } e}\ \textbf{where}\ \overline{e}\ \textbf{yield}\ e\}$ (CBody)

CORRELATION PATTERN TRANSLATION

correlate $cb \rightsquigarrow correlate_n\ [\![cb]\!]^n_{\text{pat}}$, where n is the arity of cb (Correlate)

GENERATIVE EFFECT DECLARATIONS

From $cb = \{\ (x_i \textbf{ from } e_i)_{1\le i\le n}\ \textbf{where}\ (e_{p_i})_{1\le i\le k}\ \textbf{yield}\ e_r\ \}$ generate

$$\begin{aligned}\textbf{bind}_n&: (\mathsf{R}[T_1] \times \ldots \times \mathsf{R}[T_n]) \to (\mathsf{Ev}[T_1] \times \ldots \times \mathsf{Ev}[T_n])\\ \textbf{yield}_n&: \mathsf{Ev}[T_{n+1}] \to \mathsf{Unit}\\ \textbf{fail}_n&: \mathsf{Unit} \to \mathsf{Unit}\\ (\textbf{time}_{x_i}&: \mathsf{Unit} \to \mathsf{Time})_{1\le i\le n}\end{aligned}$$

where $\Gamma \vdash e_i : T_i|\varepsilon$ and $\Gamma \vdash e_r : T_{n+1}|\varepsilon$ for some Γ, ε.

PATTERN BODY TRANSLATION $\boxed{[\![\cdot]\!]^n_{\text{pat}} : cb \to e}$

$[\![\ \{\ (x_i \textbf{ from } e_i)_{1\le i\le n}\ \textbf{where}\ (e_{p_i})_{1\le i\le k}\ \textbf{yield}\ e_r\ \}\ ]\!]^n_{\text{pat}} :=$

```
{ match (bind_n ⟨e_1, ..., e_n⟩)
    ⟨ev x_1 t_1, ..., ev x_n t_n⟩ ↦
      (let implicit ?time_{x_i} : Time = t_i)_{1≤i≤n} in
        if (e_{p_1} ∧ ··· ∧ e_{p_k})
        then yield_n (ev e_r (t_1 ⊔ ··· ⊔ t_n))
        else fail_n ⟨⟩ }
```

CORRELATE DELIMITER TRANSLATION

```
let correlate_n = λbody. with (gen ⊞ setup_n) (body ⟨⟩)
let setup_n = handler
  bind_n ⟨r_1, ..., r_n⟩ body ↦
    let implicit ?pattern = body in
    let implicit ?streams = {{eat r_1 (push_1)} ‖ ··· ‖ {eat r_n (push_n)}} in
      join_shape_n ⟨⟩
```

Figure 3.13: Correlation Pattern Translation.

```
1 let mostRecently_{n,i} = handler
2   push_i ev resume ↦
3     set_i (⟨ev, inf⟩ :: nil);
4     resume (push_i ev)
```

```
1 let affinely_{n,i} = handler
2   push_i ev resume ↦
3     //replaces ev's lifetime in memory:
4     set_i (update (get_i ⟨⟩) ev (fin 1));
5     resume (push_i ev)
```

Figure 3.14: Simple Restriction Handlers.

Following Section 3.2.5, this desugaring separates the payloads of events from their intervals, so that programmers can write natural constraints and transformations. Accordingly, time intervals can be accessed via the implicit variable $?time_{x_i}$, which is another important use case of generative effects in CARTESIUS (cf. Section 3.4.2).

Finally, the argument expression to **yield** re-wraps the result, implementing the event joins from Section 3.3.5.

Step 2: Correlate Delimiter

We encode the **correlate** delimiter simply as a λ-abstraction $correlate_n$ and apply it to the translated pattern body from the previous step. This abstraction applies the handler $gen \boxplus setup_n$ to its argument.

The $setup_n$ handler stages an instance of the join shape computation (Figure 3.12), linking it with the previous translation step, using implicit variables. The translated pattern body invokes $\mathbf{bind}_n$, which in turn is handled by $setup_n$. In this way, its resumption *body* captures Lines 2-7 of the translated pattern as a function into the implicit variable ?*pattern*. Recall that ?*pattern* is invoked by the $trigger_n$ handler (Section 3.4.2) with each tuple materialized during event correlation. Another responsibility of $setup_n$ is constructing and binding the interleaved iterations (Section 3.4.1) from the given reactives r_1 to r_n to $?streams_n$, so that the $join\_shape_n$ instance correlates its event notifications.

The responsibility of the effect handler *gen* is transforming **yield**ed values from effects back to ordinary data values:

$$gen\colon \forall \mu.\{\mathsf{Unit}\}^{\langle \mathbf{yield},\mathbf{fail},\mathbf{async},\mu\rangle} \to^{\langle \mathbf{async},\mu\rangle} \mathsf{R}[T_{n+1}],$$

so that other correlations may query the results.

3.4.4 *Implementing Restriction Handlers*

Without restrictions, the $join\_shape_n$ (Figure 3.12) computes the n-way cartesian product, where time and space requirements grow arbitrarily large. One or more restriction handlers must be assigned to the implicit variable ?*restriction* (cf. Section 3.2.4), in order to obtain a different correlation computation. By default, ?*restriction* is bound at the top level to an identity $\mathbf{handler}\{x \mapsto x\}$, which does not influence the computation. Now, we define concrete implementations of the restriction handlers from Section 3.2.4.

```
let suspendable_i = handler
 push_i ev resume ↦
  push_i ev;
  match peek_i ⟨⟩
   true ↦ resume ⟨⟩
   false ↦
    stash_i (async_thread (resume));
    ⟨⟩
```

```
type Strand_i = Maybe[{Unit}^⟨push_i,stash_i,peek_i,async⟩]
let play_pause_i =
 handler(run: Bool = true, strand: Strand_i = none)
  pause_i ⟨⟩ resume ↦ resume ⟨⟩ false strand
  play_i ⟨⟩ resume ↦ match ⟨run, strand⟩
      ⟨false, some k⟩ ↦ k ⟨⟩; resume ⟨⟩ true none
      ⟨false, none⟩ ↦ error ⟨⟩ //illegal state
      _ ↦ resume ⟨⟩ run strand
  stash_i k resume ↦ resume ⟨⟩ run (some k)
  peek_i ⟨⟩ resume ↦ resume run run strand
```

Figure 3.15: Handlers for Suspending/Resuming Interleaved Iterations.

```
type S_i = Maybe[Ev[T_i]]
let aligning_⟨i,j⟩ =
 handler(s_i: S_i = none, s_j: S_j = none)
  push_i ev resume ↦
   pause_i ⟨⟩;
   tryRelease_⟨i,j⟩ (some ev) s_j (resume)
  push_j ev resume ↦
   pause_j ⟨⟩;
   tryRelease_⟨i,j⟩ s_i (some ev) (resume)
```

```
let tryRelease_⟨i,j⟩ = λs_i.λs_j.λresume.
 match ⟨s_i, s_j⟩
  //synchronisation complete:
  ⟨some ev_1, some ev_2⟩ ↦
   //stash current strand:
   resume ⟨⟩ none none;
   //sequentially process events:
   push_i ev_1; push_j ev_2;
   //resume iterations:
   play_1 ⟨⟩; play_2 ⟨⟩
  //synchronisation incomplete:
  _ ↦ resume ⟨⟩ s_i s_j
```

Figure 3.16: Restriction Handler for Aligning Interleaved Iterations.

Simple Restrictions

The *mostRecently*$_i$ handler (Figure 3.14, left-hand side) simply truncates all but the last observed event from the ith memory state. That is, it effectively restricts the memory for the ith input reactive to one cell.

The *affinely*$_i$ handler (Figure 3.14, right-hand side) sets the lifespan counter of events to 1, i.e., each event value from the i-th input reactive can occur in at most one pattern match.

Advanced Restriction Handlers: Play/Pause Iterations

One of the design challenges we identified in Section 3.1 is controllable matching behavior. We enable it with generic abstractions that coordinate the interleaved iteration over reactives (Definition 3.3), e.g., for expressing the join behavior of *zip* (Section 3.2.4).

We implemented the capability to suspend/resume individual *eat* r_i (**push**$_i$) iterations for join computations. For example, we can make the i-th iteration suspendable as follows:

$$\{\textbf{with}\ \mathit{suspendable}_i\ (\mathit{eat}\ r_i\ (\textbf{push}_i))\}.$$

The *suspendable*$_i$ handler (Figure 3.15, left-hand side) implements a simple interception of the **push**$_i$ effect (Line 3), which originates from the underlying iteration. That way, we can capture the continuation of the iteration in *resume* and store it for later, if the surrounding correlation computation decides to suspend the iteration. Line 4 invokes the effect **peek**$_i$: Unit → Bool, which signals whether the iteration continues or not. In the latter case (Line 7), *suspendable*$_i$ stores *resume* via the ambient state effect **stash**$_i$: $\{\text{Unit}\}^{\langle \textbf{push}_i, \textbf{stash}_i, \textbf{peek}_i, \textbf{async} \rangle} \to \text{Unit}$.[5] The function *async_thread* is part of the asynchrony implementation and ensures that the asynchronous thread context of *resume* is properly captured by **stash**$_i$.

The effects **peek**$_i$ and **stash**$_i$ are handled by the *play_pause*$_i$ handler (Figure 3.15, right-hand side), which carries the **stash**ed suspension state and the ith iteration's continuation in its parameters. Additionally, *play_pause*$_i$ handles the two commands **pause**$_i$: Unit → Unit and **play**$_i$: Unit → Unit. They provide an interface to the correlation computation for imperatively signalling the suspension/resumption of the ith iteration strand. **play**$_i$ resumes the stashed iteration, if previously **pause**$_i$ was invoked (Line 6). Importantly, other ongoing iterations in the interleaving remain unaffected and continue processing.

We can now implement a wide range of coordination strategies in the join computation, such as *aligning* from Section 3.2.4. The *aligning*$_{n,S}$ handler (Figure 3.16, left-hand side) causes an input subset $S \subseteq \{1, \ldots, n\}$, $|S| > 1$, to be iterated in lockstep. We exemplify it for two inputs i, j, which have the *suspendable* capability, but the code extends to more inputs in an obvious way. The handler implements a synchronization barrier on the interleaved iterations for inputs i and j. To this end, it implements a simple state machine: On each **push** notification from either input, we pause the underlying iteration and store the event in the handler parameter. Then, we invoke *tryRelease* (Figure 3.16, right-hand side) to check if the iterations are in lockstep. In this case, we forward the buffered event notifications further up the handler stack towards the cartesian implementation (Line 8). Once the surrounding handler context finished processing these two event values, Line 10 resumes the interleaved iteration.

To support suspension/resumption of all strands, minor modifications to the framework are necessary. One is replacing the implicit binding in ?*streams*$_n$ with iterations wrapped in *suspendable*. The other is applying the handlers $\boxplus_{i=1}^{n}$*play_pause*$_i$ immediately after *trigger*$_n$ in *join_shape*$_n$ (Figure 3.12).

Linear/Affine Effects.

Handlers in λ_{cart} support by default multi-shot resumptions, which can have problematic interactions with resources. For instance, the **push**$_i$ effects need to be one-shot. Otherwise, future event notifications might be incorrectly duplicated. For these kinds of effects, a linear typing discipline would be appropriate [Wad90c]. However, if a correlation incorporates **pause**$_i$/**play**$_i$, then **push**$_i$ requires an affine type. Integrating linear/affine types with algebraic effect handlers is an active area of research. [Lei17a]

5 Since *resume* accepts a unit value, it is a thunk of type $\{\text{Unit}\}^{\langle \textbf{push}_i, \textbf{stash}_i, \textbf{peek}_i, \textbf{async} \rangle}$.

```
1 let slidingWindow_n = λlength.λperiod.
2   handler
3     bind_n ⟨r_1, ..., r_n⟩ body ↦
4       let implicit ?pattern = body in
5         with (winArbiter_n length period ⟨r_1, ..., r_n⟩)
6           ({eat r_1 (push_1)} ‖ ··· ‖ {eat r_n (push_n)}) ⟨⟩
```

```
1 let winArbiter_n = λlen.λp.λrs.
2   handler(win_state = nil)
3     {push_i ev resume ↦
4       let win_state′ =
5         updateState len p win_state rs ev
6       in dispatch_i win_state′ ev;
7         resume ⟨⟩ win_state′ }_{1≤i≤n}
```

Figure 3.17: Handlers for Sliding Windows.

3.4.5 *Windows*

With effect handlers, windows (cf. Section 3.2.6) become a purely contextual restriction, which is orthogonal to join definitions, i.e., existing restriction handlers are reusable as-is. We give a brief sketch of sliding windows. Many other kinds of windows can be defined similarly.

We model a windowed correlation as a stateful computation that manages zero or more running copies of a $join\_shape_n$ instance (Figure 3.12), one per window, where windows may overlap. Figure 3.17, left-hand side defines the $slidingWindow_n$ delimiter. A declaration of a window overrides the default setup behavior of the $correlate_n$ form (Figure 3.13). The difference is that now handler $winArbiter_n$ directly interprets the $\mathbf{push}_i$ commands of the interleaved iteration.

The $winArbiter_n$ handler (Figure 3.17, right-hand side) manages a set of active join shapes in its handler parameter *win_state*. Each time a new event value is pushed by the interleaved iteration, the *updateState* logic determines if new windows need to be allocated or expired ones need to be discarded, e.g., due to the passage of time.[6] Next, $dispatch_i$ multicasts the event to all active windows for which it is relevant. In the sliding window case, it is relevant if its occurrence time interval (Section 3.3.5) is entirely contained within a sliding window's start and end times. Iteration resumes once all relevant windows finished processing the event.

For each window instance,$winArbiter_n$ binds the interleaved iteration to a "facade", which is maintained by the *updateState* function. The facade consists of an interleaving of n streams, which are dynamically allocated per window.

3.5 APPROXIMATING GENERATIVE EFFECTS

We have shown in Section 3.4 how to embed CARTESIUS into a λ-calculus with algebraic effects and handlers. There is a semantic gap to cross when deriving executable implementations from our design, in particular concerning generativity and typing (Section 3.4.2). In this section, we discuss how such conceptual challenges impacted the embedding of CARTESIUS into Koka [Lei17b] and multicore OCaml [Dol+17], two state of the art implementations of algebraic effects.

6 In this design, we take the time data from the event values to calculate time passage. Other designs are possible, such as active timer interrupts.

The definitions in Section 3.4 are generative in the sense that they should be bound to a join definition, which is a value. Two different join definitions should have their own, non-overlapping **push** effects, because they can have different arities and input types.

Both languages have rather complimentary strengths: Koka has strong effect typing, but has no support for generative effects. On the other hand, multicore OCaml has no effect typing, but generativity can be encoded by using OCaml's powerful module system.

Our multicore OCaml implementation, encodes generative effects with the built-in module system.[7] For example, a **push** effect is part of a module interface

```
module type SLOT = sig
  type t
  effect Push: t -> unit
end
```

and this is how we put it to use, by the example of a very simplified binary join definition, which is a functor:

```
module Join2(T: sig type t0 type t1 type result end) =
struct
  module I0 = Slot(struct type t = T.t0 evt end)
  module I1 = Slot(struct type t = T.t1 evt end)
  effect Trigger: I0.t * I1.t -> unit
  let trigger p = perform (Trigger p)
  let reify action =
     try action () with
     | effect (I0.Push v) k ->
        forEach trigger (cartesian0 v); gc (); continue k ()
     | effect (I1.Push v) k ->
        forEach trigger (cartesian1 v); gc (); continue k ()
   (* ... *)
end
```

Note that two different slot instances `I0` and `I1` are allocated, each describing its own **push** effect. The **trigger** effect's type of the join definition is dependent on the two slot instances' type members. We can discern the two different push effects, because they are defined in two separate module instances.

In Koka, the push effect would need to be defined using a polymorphic effect interface:

```
effect cart<a> { push(index: int, v: a): () }
```

where the command carries an extra index parameter to discern which reactive emitted an event. Now, a handler clause has to compare the index argument at runtime to determine whether it is responsible for handling a push effect:

```
handler {
  push(i, v) ->
    if (equal(i,1)) then
      forEach trigger (cartesian1 v); gc (); resume ()
    else resume (push(i,v)) }
```

7 We thank Matija Pretnar for pointing out the encoding in a personal correspondence. The idea is originally due to Robert Atkey, realized at Dagstuhl Seminar 16112 [Bau+16], and inspired by an alternative encoding of generative effects due to Oleg Kiselyov using his delim/cc library for OCaml.

Another drawback of the Koka implementation is that unlike the multicore OCaml version, it does not support multiple instantiations of the `cart<a>` interface with different types, e.g., to join reactives of type Int and String. Due to parametricity, handlers for `cart<a>` must treat the pushed event values uniformly. Given that Koka has no notion of type classes, this is quite limiting, e.g., when printing event values for debugging purposes. We were forced to give up polymorphism for the `cart<a>` effect interface, fixing the type of reactives to Int.

We conclude that generativity is useful, if not elementary to our modeling approach for event correlation, but not yet well supported in current languages with built-in algebraic effects. Furthermore, our prototypes support only a handful of hard-coded *join arities*. That means, the respective implementations allow joining only up to a limited number m of inputs at the same time, for a fixed constant $m \in \mathbb{N}$. From an end user's (and moral) perspective, it should be possible to support *all* arities $n \in \mathbb{N}$, though.

In Chapter 5, we will study safe programming abstractions for generative effects that overcome this limitation and significantly improve over the implementation presented here.

3.6 RELATED WORK

To the best of our knowledge, this is the first work to propose algebraic effects and handlers as a common substrate to model semantic variants of reactive computations. In this section, we compare our work against other approaches that are similar in scope, i.e., unifying models.

Brooklet by Soulé et al. [Sou+10] is a calculus for stream processing languages that models stream computations as static, asynchronous and acyclic data flow graphs. For example, stream-relational algebra languages, such as CQL [ABW06] and batch processing languages can be compiled into the calculus. The calculus emphasizes the topology of stream computations, which makes it well-suited as an intermediate representation for domain-agnostic optimizations at the operator graph level [Hir+13]. The internal behavior of a graph node remains a "black box" in the form of a deterministic pure state transition function, acting on a (possibly shared) state and a number of input queues. Cartesius is similar in that there is a strict separation between impurity and purity, mediated by algebraic effects. However, Brooklet itself has no means of specifying the internal combination behavior of nodes and is not language embedded. Cartesius is complementary, enabling the declaration of the behavior inside the black boxes. Coordination and state sharing among nodes can be achieved by lexically enclosing with suitable handlers.

Streams *à la carte*, by Biboudis et al. [Bib+15], abstract over the semantics of stream pipelines using Oliveira et al.'s object algebra approach [OC12; Oli+13], which models algebras in terms of OO interfaces and generics. Algebra instances are similar to handlers in the sense that both are modular folds over a given signature of operations. However, object algebras do not capture delimited control and must be passed as an explicit parameter to programs. The design by Biboudis et al. facilitates switching backend to execute a stream program, e.g., between push and pull streams. In contrast, Cartesius fixes the choice to Elliott's push/pull streams [Ell09] to support direct style correlation patterns in consumers, even though event production is push-based. This works well in combination with effect handlers, because of their ability to capture the continuation.

Stream fusion, to completeness by Kiselyov et al. [Kis+17], uses staging to remove abstraction overhead of a range of pull-based stream combinators. The generated primitive, loop-based code rivals hand-written implementations. Pull-based streams can be thought of as generators or unfolds. A key finding of Kiselyov et al. is staging of zip – which is complex, because the two streams need to be advanced in lock step, yet a (non-linear) stream may need to be advanced a statically unknown number of times to produce one element. Their solution involves ensuring that one of the streams is linear through reification; they argue that there is no way around this inefficiency, even with hand-written code. In our case, we already reify the memory states, so aligning them for zipping is not particularly difficult.

The SECRET model [Bot+10; Din+13] examines windows in stream processing engines and provides a unifying description of vastly different windowing behaviors among varied systems. In CARTESIUS, we model windows as contextual restrictions (see Section 3.4.5), and leave the exact behavior of how events are dispatched to windows (e.g., sliding) open to implementors.

Ziarek, Sivaramakrishnan, and Jagannathan [ZSJ11] contribute a composable and extensible design for asynchronous events, based on the Concurrent ML API. Their work targets general purpose, asynchronous programming, while the focus of CARTESIUS is on joining asynchronous event streams. Nevertheless, some notable similarities exist to handler-based event correlation (Definition 3.3). Handling the interleaved iterations is similar to applying CML's `choose` combinator to CML events. The effect handlers for **push** correspond to the case analysis of `choose`. Among the differences are that `choose` is one shot, defining a single event, whereas CARTESIUS continuously applies the case analysis to event sequences, producing multiple events. Moreover, in our context, events are effect invocations and not the events arriving on the streams being composed. Finally, their work supports parallel executions, whereas CARTESIUS so far only supports single step, interleaving concurrency.

3.7 CHAPTER SUMMARY

By virtue of algebraic effects and handlers, we have conceived with CARTESIUS an "à la carte" event correlation system, taming the high complexity and variability in the field. Our design has the following traits:

UNIFORM Definition 3.3 gives a *uniform interface* for all join variants of joins. That is, effect handlers that discharge/interpret push notifications embody a specific join variant. Operationally, event correlation can be uniformly understood in terms of specific selection and consumption patterns on n local mailboxes of event observations, which arise from a coroutine-like interaction of a cartesian product computation with user-defined restriction handlers. This yields a *uniform organization* of the underlying computation, as depicted in Figure 3.3.

EXTENSIBLE New restriction handlers can be freely programmed, requiring knowledge of the effect interface for event notifications and access to the local mailboxes of observations. Definition 3.3 permits that restriction handlers induce *new* effects/features (abstracted over by the ε meta variable in the effect row) as needed to implement additional functionality.

MODULAR Programmers can understand extensions/restriction handlers simply by reading their definitions and the (effect) type signatures of subexpressions/dependencies. It is clear which effects are handled and what other effects might be induced.

COMPOSABLE Restriction handlers are freely composable by stacking them. Especially, this enables compositions of features from across event correlation families.

SAFE Effect typing ensures that compositions of restriction handlers "do not go wrong" and that all event notifications in Definition 3.3 are handled.

EXPRESSIVE A join computation can be written once, and receive different local interpretations in different usage contexts, due to the decoupling of effect interfaces and implementation by effect handlers. Programmers can define libraries of interpretations. In this way, different event correlation variants can coexist in an application. In Chapter 6, we conduct a survey validating that CARTESIUS can express a significant set of event correlation features from across all families.

EFFICIENT The coroutine-like interaction between the cartesian product and restriction handlers makes it possible to avoid needless materialization of event tuples, despite having modular decomposition into these components. In Chapter 7 we perform empirical measurements that validate this claim.

The overall architecture of CARTESIUS combines traits of (1) push- and pull-based reactive programming to model asynchronous data flow and (2) actors [HBS73; Agh90] with n mailboxes that communicate by message passing, where messages are effect invocations. The internal behavior of these actors is determined by cartesian products interacting with restriction handlers.

Importantly, some computational join variants cannot be understood as mere passive observers of incoming messages. The prime example is the zip combinator, which requires the *aligning* restriction handler that interacts with and coordinates the asynchronous stream iterations (Section 3.4.4). This is another coroutine-like interaction, this time not with the part of the join computation that produces output (cartesian product), but the part that produces input, i.e., the interleaving of the n asynchronous stream iterations. Our handler-based abstractions for suspending/resuming enable this interaction, which is a form of n-way, simultaneous coroutine for coordination logic. We are not aware of any work on coroutines that contributes a similar mechanism.

One issue with the design of CARTESIUS so far is its heavy use of generative (or first-class) effects. For example, we assign each join instance a fresh set of **push** effect declarations and all our effect handler definitions have to be understood relative to a context which has the right number and the right types of **push** effects. We marked this dependence by indexing the definitions with numbers. On pen and paper, we can assume that respective effect instances are pre-allocated. However, in a real programming language implementation, we need a way to linguistically express and program with generative effects. Our prototype implementations in Section 3.5 do not have this ability. The

consequence is that they support only a hard-coded number of join arities, i.e., the prototypes can only correlate up to a specific number n of reactives, but not more. Our prototypes also lack the declarative correlation pattern syntax we have been using in examples (Section 3.4.3).

It turns out that these two seemingly separate issues are connected and we will address them in the next two chapters, obtaining a complete realization of CARTESIUS in a practical programming language, supporting the event correlation pattern syntax and the ability to correlate any finite number of reactives.

POLYJOIN: POLYVARIADIC, EXTENSIBLE EVENT CORRELATION PATTERNS

4

This chapter shares material with [Bra+19].

SYNOPSIS Event correlation systems adopted the well-known and intuitive join notation from database query languages, along with the techniques for typed programming language embedding (e.g., LINQ [MBB06; CLW13]). However, these approaches are essentially descendants of the work on monad comprehensions, which are not well-suited for event correlation patterns: The monadic semantics of variable bindings in joins is unsuitable for the concurrency/data flow semantics in some event correlation systems. That is, mainstream embeddings cannot accommodate the entire design space for event correlation systems.

We propose PolyJoin, a novel approach to type-safe embedding of event patterns that captures the essence of established notations in the field of event correlation. Importantly, it permits more general denotations than mainstream embeddings and it is by design extensible with new pattern syntax. To achieve this, PolyJoin combines *tagless interpreters* [CKS09] with *typed polyvariadic functions* [Dan98; Rhi09; FI00], i.e., functions accepting a variable number $n \in \mathbb{N}$ of arguments, each heterogeneously typed. Event patterns are syntax forms with uncurried, n-ary higher-order abstract syntax (HOAS) variable bindings [HL78; PE88]. In contrast to previous works, our approach achieves the separation between polyvariadic interfaces and polyvariadic implementations.

Our message is that taglessness and polyvariadicity are sufficient to obtain practical and extensible embeddings for common join and dataflow notations. Importantly, PolyJoin is independently deployable from Cartesius, and suitable as a declarative frontend for other event correlation systems. Patterns in PolyJoin neither require dependent types, nor code generation nor dedicated compiler support. Our approach is practical, type-safe, lightweight and portable. An implementation in OCaml is readily usable. The motto is: More power to system programmers, and less burden for compiler implementers!

4.1 INTRODUCTION

The previous Chapter 3 on Cartesius first and foremost addresses a uniform and extensible *semantics* for event correlation computations. There, we already proposed a pen-and-paper syntax design for declarative event correlation patterns that integrate control flow restrictions. Here, we consider how to practically achieve embeddings of declarative patterns in a programming language, i.e., the main matter of this chapter is extensible *syntax*.

The integration of event patterns into programming languages has been a rather neglected question so far in the field of event correlation. Such integration can be realized by dedicated syntax and compiler extensions or by *embedding*, i.e., by expressing the domain-specific language (DSL) of event patterns in terms of a host language's linguistic concepts (Hudak [Hud96]). Our work focuses on embeddings that are *statically type-safe* and *polymorphic* in the sense that they allow re-targeting event pattern syntax into different semantic representa-

tions (cf. Hofer et al. [Hof+08]). Polymorphic embeddings are crucial to support the semantic diversity of event correlation. Moreover, for practical reasons, we focus on embeddings that are implementable in mainstream programming languages.

Existing event correlation systems either have no language embeddings or rely on language-integrated query techniques originally developed for database systems. An example in the first category is Esper [Esp06], which only supports formulating event patterns/queries as strings, which are parsed and compiled at runtime, possibly yielding runtime errors. Examples in the second category are Trill [Cha+14] and Rx.Net [Rea11], which employ LINQ [MBB06; CLW13] and express event patterns in database join notation.

The reliance on database query notation for correlation patterns has historical and pragmatic reasons: (1) some event correlation approaches have their roots in the database community [CM12], featuring similar declarative query syntax, e.g., the CQL language [ABW06]. (2) intuitively, one may indeed think of event patterns and join queries as instances of a more abstract class of operations: that which associate values originating from different sources, e.g., databases, in-memory collections, streams, channels. (3) language-integrated query techniques for databases are mature and readily usable for implementations of event correlation systems.

Yet, language-integrated query techniques for databases are not adequate for event patterns. Integration techniques for database languages, such as LINQ, are descendants of monad comprehensions (Wadler [Wad90a]) and translate queries to instances of the monad interface (Moggi [Mog89; Mog91] and Wadler [Wad92]). These techniques are statically type-checked and they are polymorphic in the sense that they support arbitrary monad instances.

However, join queries over n sources translate to n variable bindings, which *sequentially* induce data dependencies, as this is the only interpretation of variable bindings that the monad interface permits. We argue that these sequential bindings cannot express important cases of event correlation patterns, which require a *parallel* binding semantics, to faithfully address the concurrency of external event sources.

Hence, we re-think how to embed event patterns into programming languages and consider alternatives to the monad interface, in order to properly support general models for event correlation. Specifically, adequate embeddings should support polymorphism in the semantics of event pattern variable bindings. For example, CARTESIUS in Chapter 3 requires a parallel binding semantics.

To address these requirements, we propose POLYJOIN, a novel approach for type-safe, polymorphic embeddings of event patterns. It embeds event patterns as a typed DSL in OCaml and retains much of the familiar notation for database joins.

POLYJOIN fruitfully combines two lines of research: (a) the *tagless final* approach by Carette, Kiselyov, and Shan [CKS09], which yields type-safe, extensible and polymorphic embeddings of DSLs, and (b) *typed polyvariadic functions*, which are *arity generic*, accepting a list of n parameters for all $n \in \mathbb{N}$ and each of the n parameters is *heterogeneously-typed* [Kis15].[1] Event patterns and joins naturally are instances of polyvariadic functions: users can join n heterogeneously-typed event sources, for arbitrary $n \in \mathbb{N}$. The two lines of research complement

1 Polyvariadic functions were first studied by Danvy [Dan98] to encode a statically type-safe version of the well-known `printf` function in terms of combinators.

each other well in PolyJoin: (1) with polyvariadic function definitions, we can generalize Carette et al.'s tagless embeddings from single to n-ary binders in the higher-order abstract syntax (HOAS) [HL78; PE88] encoding, and (2) with tagless embedding, we can separate polyvariadic interfaces and polyvariadic implementations. These traits are crucial for event pattern embeddings, because they enable "polymorphism in the semantics of variable bindings". Event patterns become DSL terms with n-ary HOAS variable bindings (interface) and concrete tagless interpreters determine what the bindings mean (implementation).

However, polyvariadic function definitions are notoriously difficult to express in typed programming languages that are not dependently typed, like most mainstream languages. We achieve compatibility with mainstream languages by explicit tracking of the shape and type of a pattern's variable context within the type system of the host language, requiring only type constructor polymorphism, bounded polymorphism and binary product types.

PolyJoin is portable: variants of the tagless final approach coincide with object algebras [OC12; Oli+13] in OO languages. Thus, while this work chooses OCaml as the host language, PolyJoin works in principle in other languages, both functional and OO, e.g., Haskell [CKS09], Scala [Hof+08] or Java [Bib+15].

Overall, our work shows that it is possible to have a purely *library-level* embedding of event patterns, in the familiar join query notation, which is not restricted to monads. PolyJoin is type-safe and extensible. It is readily usable as an embedding for general event correlation systems, supporting semantically diverse event correlation.

In summary, the contributions of this chapter are as follows:

- A systematic analysis of why existing monadic style embeddings for database-style queries are inadequate for embedding event patterns (Section 4.2).
- A novel tagless final embedding for event patterns into mainstream programming languages, which is polyvariadic, statically type-safe, polymorphic, extensible, and modular (Section 4.3). This embedding constitutes the core of PolyJoin - we provide a formalization and an encoding of it in pure OCaml.
- The metatheory of PolyJoin (Section 4.5), providing a type-preserving translation into OCaml, a type soundness result and a parametricity result. We guarantee that (1) end users cannot define ill-formulated or ill-typed event patterns, (2) patterns with arbitrary arity are expressible and (3) tagless interpreters for event patterns are free to choose how pattern variables are bound.
- A discussion on how to implement concrete tagless interpreters for PolyJoin in OCaml, which have to provide a polyvariadic implementation of a polyvariadic signature/interface (Section 4.6 and 4.7).

This chapter is the first of two parts, discussing the basics of tagless interpreters and PolyJoin. Chapter 5 is the second part, which contributes the embedding of Cartesius into multicore OCaml with PolyJoin, a qualitative evaluation, discussions and related work.

4.2 PROBLEM STATEMENT

In this section, we argue that the monad interface is too limiting for the integration of event correlation systems into programming languages.

4.2.1 *Event Patterns versus Join Queries*

Throughout this chapter, we make use of the example:, *"if within 5 minutes a temperature sensor reports ≥ 50°C, and a smoke sensor is set, trigger a fire alarm"* (Cugola and Margara [CM12]), which is an event pattern that relates sensor events by their attributes and timing.

For illustration, the pseudo-code in Figure 4.1a reflects how programmers could write the fire alarm event pattern in terms of a LINQ-like, embedded join query over event sources. The first two lines select/bind all events originating `from` the temperature sensor and smoke sensor to the variables `t` and `s`, respectively. The `where` clause specifies which pairs of temperature and smoke events are relevant, i.e., those that occur at points in time at most 5 minutes apart, the smoke event's value is `true` and the temperature event's value is above 50° Celsius. The `yield` clause generates a new event from each relevant pair, in this example a string-valued event containing a warning message along with the temperature value. Overall, the join query correlates the `temp_sensor` and `smoke_sensor` event sources and forms a new event source yielding fire alert messages.

One may think of event sources as potentially infinite sequences of discrete event notifications, which are pairs of a value and the event's occurrence time. We write concrete event sequences within angle brackets (`< >`). For instance,

```
temp_sensor: float react =
  < (20.0, 2), (53.5, 4), (35.0, 5), (60.2, 8) >
smoke_sensor: bool react =
  < (true, 9), (false, 10), (true, 12) >
```

are concrete event sequences. That is, `temp_sensor` produces `float`-valued events (the temperature in degrees Celsius) and `smoke_sensor` produces `bool`-valued events (sensor detects smoke or not). We name the type of event sources `'a react`, using OCaml notation.

With the concrete event sequences above as input and assuming that the second components of the events specify occurrence times in minutes, our example join query yields this event sequence:[2]

```
< ("Fire: 53.5", [4,9]), ("Fire: 60.2", [8,9]), ("Fire: 60.2",
    [8,12]) >
```

Even though event patterns in real systems are notationally similar to join queries in databases, there are significant semantic differences between the two:

INVERSION OF CONTROL Event correlation computations are *asynchronous and concurrent*. In particular, they have no control when event notifications occur. Event sources run independently and produce event notifications at their own pace, i.e., *control is inverted* as opposed to traditional collection or database queries, which are demand-driven. That is, data is enumerated only if the join

2 To determine the occurrence time of the events produced by a join, we follow the strategy from Chapter 3, i.e., we merge the two occurrence times into the smallest interval containing both. In the notation, we replace singleton intervals $[t, t]$ with just t.

```
from (t <- temp_sensor)
  from (s <- smoke_sensor)
    where within(s,t,5 minutes) && s && t >= 50.0
      yield (format "Fire: %f" t)
```

(a) Monadic Database Query Notation.

```
join ((from temp_sensor) @. (from smoke_sensor) @. cnil)
  (fun ((temp,t1), ((smoke,t2), ())) ->
    where (within t1 t2 (minutes 5.0)) %& smoke %& (temp %>= 50.0)
          (yield (format "Fire %f" temp)))
```

(b) PolyJoin/OCaml Version.

Figure 4.1: Event Correlation Example: Fire Alarm.

computation decides to access it. Dually, an event correlation computation passively observes and reacts to the enumeration of data, i.e., it is subject to *external choice*, because its environment controls when and which event notifications are sent.

SEMANTIC DIVERSITY Complex-event and stream processing systems exhibit great semantic diversity in event correlation behavior because events in conjunction with time can be correlated in different ways. For example, the event pattern "*a followed by b*" applied to the event sequence $\langle a\ b\ b \rangle$ may match once or twice, depending on whether the correlation computation consumes the *a* event the first time or not. In general, it is application-specific which correlation behavior is best suited. Especially in heterogeneous computing environments, no single correlation behavior satisfies all requirements and hence different semantic variants should be expressible/composable.

4.2.2 *Monadic Embeddings are Inadequate for Event Patterns*

We already gave semantic explanation in Chapter 3 why monad computations the way they are used in collection libraries cannot express general event correlation computations. Here, we analyze the problem from the perspective of syntax and polymorphic embedding, i.e., how the common join query syntax is closely tied to monads and why this is limiting the polymorphism of embeddings.

Monadic query embeddings have important traits, which are as important for event patterns:

(STATIC) TYPE SAFETY They are type-safe and integrate seamlessly into the (higher-order) host programming language. The compiler statically rejects all ill-defined queries.

POLYMORPHIC EMBEDDING They are polymorphic embeddings [Hof+08], because all instances of the monad interface are admissible representations.[3] Since monads encompass a large class of computations, queries in the monadic style may denote *diverse behavior*. In particular this includes *asynchronous and concurrent* computations, such as event correlation, via the well-known continuation monad [Mog89; Mog91; Gon12].

A closer inspection, though, reveals that the monadic translation of join queries cannot capture all event correlation behaviors.

Semantics of Joins in Monadic Embeddings

Monadic embeddings, such as LINQ, map queries to monads, i.e., type constructors $C[\cdot]$ with combinators

$$\begin{aligned} \mathit{return} &: \forall\alpha.\alpha \to C[\alpha] \\ \mathit{bind} &: \forall\alpha.\forall\beta.C[\alpha] \to (\alpha \to C[\beta]) \to C[\beta] \end{aligned}$$

satisfying the following laws

$$\mathit{bind}\ c\ (\mathit{return}) = c \tag{4.1}$$

$$\mathit{bind}\ (\mathit{return}\ x)\ f = f\ x \tag{4.2}$$

$$\mathit{bind}\ (\mathit{bind}\ c\ f)\ g = \mathit{bind}\ c\ (\lambda y.\mathit{bind}\ (f\ x)\ g) \tag{4.3}$$

which describe that $C[\cdot]$ models a notion of sequential computation with effects, respectively a collection type. Accordingly, LINQ's metatheory (cf. Cheney, Lindley, and Wadler [CLW13]) is specifically tailored to this interface and its laws. In particular, *bind* models a variable binder in continuation-passing style (CPS), extracting and then binding an element of type α out of the given $C[\alpha]$ shape and then continuing with the next computation step, resulting in $C[\beta]$. Like all monadic embeddings, LINQ encodes joins by nesting invocations of *bind*, so that an n-way join query

```
from (x1 <- r1) ⋯ from (xn <- rn) yield (x1,...,xn)
```

denotes a nested monad computation

$$\mathit{bind}\ r_1\ (\lambda x_1.\mathit{bind}\ r_2\ (\lambda x_2.\cdots\mathit{bind}\ r_n\ (\lambda x_n.\ \mathit{return}\ (x_1,\ldots,x_n))\cdots))$$

where `from`-bindings correspond to nested *bind* invocations and `yield` to *return*. The monad laws determine a rigid sequential selection of elements from the input sources r_1 to r_n, in the order of notation. That means, for all $i \in \{1,\ldots,n\}$, the monadic join computation binds an element from r_i to variable x_i *after* it has bound all variables x_j, $j < i$.

Limitations of Monadic Embeddings

Sequential binding is an unfaithful model of the asynchrony and concurrency of event sources

and thus inadequate for event patterns. For example, it has the following limitations:

3 Strictly speaking, database languages require the MonadPlus type class [Haso8], i.e., monads with additional zero and plus operations for empty bag and bag union.

UNBOUNDED INCREASE IN LATENCY Suppose we embedded the fire alarm example (Section 4.2.1) using LINQ. The computation is able to bind a `smoke_sensor` event to `s` only after it has bound a `temp_sensor` to `t`. That is, it must first unnecessarily wait on `temp_sensor` to continue, even if a new event is already available from `smoke_sensor`. This would increase the latency of the computation, e.g., if past `temp_sensor` events paired with the new `smoke_sensor` event could immediately yield a new fire alarm event. The next `temp_sensor` event could come arbitrary late and thus delay already available alarm events arbitrarily long. This unbounded increase in latency is incompatible with online data elaboration in CEP systems.

LIMITED EXPRESSIVITY The rigid binding order *cannot* express important event correlation behaviors which require interleaving or parallelism of bindings. Consider correlating the sources in the fire alarm example to define a stream that *always reflects the two most up to date values* of smoke and temperature.[4] If one of the sources stops producing events and the other continues, then a monadic version of this computation becomes stuck, since it forever blocks on the non-productive source. This example requires a "parallel" variable binding semantics, where the notation order of binders bears no influence on the computation's selection order at runtime.

Key Issue: What Should "from" Mean?

In summary, monadic query embeddings are polymorphic over the monad instance $C[\cdot]$, where the monad interface confines to join computations with a sequential variable binding semantics. Hence, monadic embeddings are too weak to express event patterns in join notation, because the latter may require other semantics for variable bindings in patterns, e.g., as defined by Applicatives [MP08], Arrows [Hugoo] or in the Join Calculus [FG96]. That is to say:

The semantics of the `from`-binding constitutes an additional dimension of polymorphism.

Monadic embeddings fix this dimension to a single point. However, the domain of event correlation requires embedding techniques that are parametric in this dimension as well: So that (1) systems programmers can correctly integrate an existing event correlation engine into a programming language, while keeping the familiar/traditional join notation. And (2), one uniform embedding technique can accommodate diverse event correlation semantics in one application.

4.2.3 *Unifying Event Patterns and Join Queries*

The previous analysis suggests that we need to re-think the embedding of the join syntax into the host programming language. We propose that the join syntax translates to a representation which is more general than nested monadic binds. And from now on, we let "join" refer to both database joins and event correlation computations, since they both "associate values originating from different sources". As a first step, we model this intuition by an informal type signature:

4 This is sometimes referred to as *Combine Latest* event correlation behavior, which captures the semantics of reactive programming languages. We already encountered this semantic variant in Chapter 3.

DEFINITION 4.1 (n-WAY JOIN TYPE SIGNATURE)
At the type-level, joins are computation-transforming functions with a signature of the form

$$S[\alpha_1] \times \cdots \times S[\alpha_n] \to S[\alpha_1 \times \cdots \times \alpha_n]$$

for some type constructor $S[\cdot]$ and for all element types $\alpha_1, \ldots, \alpha_n$ and all $n \geq 0$.♦

That is, a join is a function merging heterogeneous n-tuples of computations having a shape $S[\cdot]$ (e.g., database, event source or effect) into a computation of n-tuples. For $n \in \{0, 1\}$, we obtain a constant function, respectively an identity. The case $n > 1$ is more interesting, e.g., for $n = 2$ we obtain

$$S[\alpha_1] \times S[\alpha_2] \to S[\alpha_1 \times \alpha_2],$$

which in the terminology of Mycroft, Orchard, and Petricek [MOP16] is an *effect control-flow operator*, i.e., the (effect) $S[\cdot]$ appears left of the function arrow. This signature accommodates merge implementations that are suitable for event correlation: They are allowed to perform the effects of the arguments in any order, even in an interleaved or parallel fashion. For comparison, if we instantiate monadic bind (Section 4.2.2) for merging, we obtain a different control-flow operator

$$S[\alpha_1] \to (\alpha_1 \to S[\alpha_1 \times \alpha_2]) \to S[\alpha_1 \times \alpha_2].$$

By parametricity [Wad89], *bind* continues with the next step after the input effect $S[\alpha_1]$ has been performed, with a resulting "naked" α_1 which has been yielded by the shape $S[\cdot]$. Parallel composition with other shapes is impossible.

Therefore, Definition 4.1 gives a unifying interface for both worlds: the type signature permits both the nested monadic bind construction for database joins (Section 4.2.2), and concurrent merges, as required by event correlation. Function values of this signature are a good target denotation for the join syntax. Such functions are called *polyvariadic* [Kis15], i.e., functions polymorphic in both the number and (heterogeneous) type of input shapes/events sources, reflecting that users can formulate join queries over an arbitrary, but finite number of differently-typed sources.

We face a two-fold challenge: (1) representing polyvariadic functions as the denotation for join syntax and (2) modeling a polymorphic embedding of join syntax that makes use of the polyvariadic representation. Both should be expressible purely in terms of the linguistic concepts of a typed *mainstream* programming language (e.g., dependent types are forbidden). However, such an implementation is rewarding, because we can do it purely as a library, without being dependent on compiler implementers to adjust their comprehension support of the language, which may never even happen. More power to systems programmers and less burden for compiler writers!

4.3 TYPE-SAFE POLYVARIADIC EVENT PATTERNS WITH POLYJOIN

In this section, we present PolyJoin, an embedding of join syntax into OCaml, which is both polymorphic and polyvariadic. It permits more general interpretations of variable bindings in comparison to the predominant monadic comprehensions found in modern programming languages. For example, PolyJoin admits parallel bindings that are needed for event correlation. PolyJoin is

lightweight: it requires neither compiler extensions, nor complicated metaprogramming, nor code generation techniques. And it is statically type-safe: the OCaml compiler checks that joins/event patterns cannot go wrong. Programmers can readily use PolyJoin for language integration of event correlation systems, offering high-level, declarative event patterns syntax to end users.

4.3.1 *Fire Alarm, Revisited*

As a first taste of how clients specify event patterns in PolyJoin, Figure 4.1b shows the PolyJoin version of the fire alarm example (Figure 4.1a), using the `join` form. Note that this is *pure OCaml code* and notationally close to the original example. To avoid clashes with standard OCaml operators, we prepend connectives in the event pattern syntax by `%`. A more significant difference in the notation is the separation of `from`-bindings (Line 1) from the actual body of the pattern (Lines 2-4), to avoid nested bindings. The `@.` symbol is a right-associative concatenation of `from`-bindings (cf. Section 4.3.3) into a list of bindings, with `cnil` being the empty list of bindings.

Line 2 defines the pattern's variables and body in terms of an OCaml function literal and OCaml variables, in higher-order abstract syntax (HOAS) [PE88; HL78]. This way, we avoid the delicate and error-prone task of modeling variable binding, free variables and substitution for the DSL by ourselves, instead delegating it to the host language OCaml. We deconstruct bound events into their value and their occurrence time, using OCaml's pattern matching, e.g., `(temp,t1)`.

Our approach extends the work by Carette, Kiselyov, and Shan [CKS09] from single variable HOAS bindings to uncurried n variable bindings (using nested binary pairs), for arbitrary $n \in \mathbb{N}$. We statically enforce that the number n of pattern variables and their types is consistent with the number and types of supplied `from`-bindings: If `temp_sensor` (resp. `smoke_sensor`) is an event source of type `float react` (resp. `bool react`), then pattern variable `time` (resp. `smoke`) is bound to `float` (resp. `bool`) events originating from that source in the body of the pattern.

It is impossible to define ill-typed patterns in PolyJoin: The type system of the host language (OCaml) checks and enforces the correct typing of the event pattern DSL. For example, if we added another `from`-binding to the pattern in Figure 4.1b, then OCaml's type checker would reject it, because bound pattern variables do not match (underlined in the snippet below):

```
join ((from temp_sensor) @. (from smoke_sensor) @. (from p_sensor
    ) @. cnil)
      (fun ((temp,t1),((smoke,t2),())) -> ...)
(* Error: This pattern matches values of type unit but a pattern
    was expected which matches values of type float repr * unit
    *)
```

4.3.2 *Tagless Final Embeddings*

PolyJoin is based on the tagless final approach, which we introduced in Section 2.4. We recommend reading that section before proceeding.

VARIABLE CONTEXTS

$$\Gamma ::= \varnothing \mid x : A, \Gamma$$

EXPRESSIONS AND PATTERNS $\boxed{\Gamma \vdash_{\text{exp}} M : A} \quad \boxed{\Gamma \vdash_{\text{pat}} P : A}$

$$\frac{\Gamma(x) = A}{\Gamma \vdash_{\text{exp}} x : A}\ (\textsc{var}) \qquad \frac{\Gamma \vdash_{\text{exp}} M : \mathsf{Bool} \qquad \Gamma \vdash_{\text{pat}} P : A}{\Gamma \vdash_{\text{pat}} \texttt{where}\ M\ P : A}\ (\textsc{where}) \qquad \frac{\Gamma \vdash_{\text{exp}} M : A}{\Gamma \vdash_{\text{pat}} \texttt{yield}\ M : A}\ (\textsc{yield})$$

$$\frac{\Gamma \vdash_{\text{ctx}} \Pi : \vec{A} \qquad \vec{A} \triangleright \vec{B} \qquad \Gamma, \overrightarrow{x : B} \vdash_{\text{pat}} P : C}{\Gamma \vdash_{\text{exp}} \texttt{join}\ \Pi\ (\vec{x}.P) : \mathsf{Shape}[C]}\ (\textsc{join})$$

CONTEXT FORMATION $\boxed{\Gamma \vdash_{\text{var}} V : A} \quad \boxed{\Gamma \vdash_{\text{ctx}} \Pi : \vec{A}}$

$$\frac{\Gamma \vdash_{\text{exp}} M : \mathsf{Shape}[A]}{\Gamma \vdash_{\text{var}} \texttt{from}\ M : A}\ (\textsc{from}) \qquad \Gamma \vdash_{\text{ctx}} \texttt{cnil} : \varnothing\ (\textsc{cnil}) \qquad \frac{\Gamma \vdash_{\text{var}} V : B \qquad \Gamma \vdash_{\text{ctx}} \Pi : \vec{A}}{\Gamma \vdash_{\text{ctx}} V\ @.\ \Pi : B, \vec{A}}\ (\textsc{cat})$$

CONTEXT SHAPE TRANSLATION $\boxed{\vec{A} \triangleright \vec{B}}$

$$\vec{A} \triangleright \vec{A}\ (\textsc{id})$$

Figure 4.2: Core Syntax and Typing Rules of PolyJoin.

While the focus of this work is polyvariadic and polymorphic event pattern embeddings, basing PolyJoin on the tagless final approach enables promising future applications for our line of research: in conjunction with multi-stage programming (e.g., [RO10; Kis14]), the approach in principle supports modular, library-level compilation pipelines for DSLs, e.g., as exemplified in [CKS09] and [SKK16].[5]

4.3.3 *Core PolyJoin*

Here, we develop PolyJoin as a tagless final DSL. We make a simplification to focus on the core ideas: event patterns do not expose timing (e.g., `within` in Figure 4.1b) or other implicit metadata on events. We address these features in Chapter 5.

Figure 4.2 defines the core syntax and typing rules of PolyJoin in natural deduction style and Figure 4.3 the corresponding tagless encoding as a OCaml module signature.

5 We consider staging and optimizing code that contains algebraic effects and handlers important future work, but no current implementation of staging offers this capability.

```
module type Symantics = sig
  type 'a shape (* Shape[·] Constructor *)
  (* Judgments (cf. Figure 4.2): *)
  type 'a repr     (* ⊢exp · : A *)
  type 'a pat      (* ⊢pat · : A *)
  type ('a,'b) ctx (* combination of ⊢ctx · : A and A ▷ B *)
  type 'a var      (* ⊢var · : A *)
  (* Context Formation and Shape Translation: *)
  val from: 'a shape repr -> 'a var
  val cnil: (unit,unit) ctx
  val (@.): 'a var -> ('c, 'd) ctx -> ('a * 'c, 'a repr * 'd) ctx
  (* Expressions and Patterns: *)
  val yield: 'a repr -> 'a pat
  val where: bool repr -> 'a pat -> 'a pat
  val join: ('a, 'b) ctx -> ('b -> 'c pat) -> 'c shape repr
end
```

Figure 4.3: Tagless Final Representation of Core PolyJoin.

```
module type Hl = sig
  type 'a el
  type _ hlist =
    | Z : unit hlist
    | S : 'a el * 'b hlist
              -> ('a * 'b) hlist
end
```

(a)

```
module HList(E: sig type 'a t end) =
struct
  type 'a el = 'a E.t
  type _ hlist =
    | Z : unit hlist
    | S : 'a el * 'b hlist
              -> ('a * 'b) hlist
  let nil = Z
  let cons h t = S (h,t)
end
```

(b)

Figure 4.4: Heterogeneous Lists Definition.

Expressions and Patterns

As before, we define syntax/typing rules $\Gamma \vdash_{\text{exp}} M : A$ for the syntactic sort of expressions, but this time with a context Γ of free expression variables, because we now model join patterns having variable bindings. Additionally, we introduce a new sort for *patterns*, $\Gamma \vdash_{\text{pat}} P : A$, meaning that pattern P yields events of type A under context Γ. In this language, patterns consist only of `where` (constraints/filter) and `yield` (lift expression of type A to an event of type A). However, we can always extend the language with more pattern forms, as needed. We designate meta variables representing finite ordered sequences with arrows, e.g., $\vec{A}$ is a sequence of types.

```
module StdContext(T: sig type 'a repr type 'a shape end) = struct
  open T
  type _ var = Bind: 'a shape repr -> 'a var
  module Ctx = HList(struct type 'a t = 'a var end)
  type (_,_) shape = (* Shape translation judgment *)
    | Base: (unit, unit) shape
    | Step: ('s, 'a) jsig  -> ('t * 's, 't repr * 'a) shape
  (* Implementation of Context Formation (Figures 4.2 and 4.3) *)
  type ('a,'b) ctx = ('a,'b) shape * 'a Ctx.hlist
  let from = fun s -> Bind s
  let cnil = (Base, Ctx.nil)
  let (@.): type a c d. a var -> (c, d) ctx -> (a * c, a repr * d) ctx
      = fun v (shape, ctx) -> (Step shape, Ctx.cons v ctx)
end
```

Figure 4.5: Default Variable Context Representation with Heterogeneous Lists.

```
module type MonadPlus = sig
  type 'a m
  val return: 'a   -> 'a m
  val bind:   'a m -> ('a -> 'b m) -> 'b m
  val zero:   unit -> 'a m
  val plus:   'a m -> 'a m -> 'a m
end
module MonadJoin(M: MonadPlus) = struct
  type 'a repr  = 'a
  type 'a shape = 'a M.m
  type 'a pat   = 'a M.m
  (* Contexts: cf. Figure 4.5: *)
  include StdContext(struct type 'a repr = 'a type 'a shape = 'a M.m)
  let where b p = if b then p else (M.zero ())
  let yield v = M.return v
  let pair: 'a repr -> 'b repr -> ('a * 'b) repr = fun a b -> (a,b)
  (* Nested monadic bind by induction over the context derivation: *)
  let rec join: type a b c. (a,b) ctx -> (b -> c pat) -> c repr = fun ctx k ->
    match ctx with
    | (Base, Ctx.Z) -> k ()
    | (Step n, Ctx.S (Bind ls, hs)) ->
       M.bind (fun x -> join (n,hs) (fun xs -> k (x, xs)))
end
module ListM = struct (* Example instantiation with lists *)
  type 'a m = 'a list
  let return x = [x]
  let bind l k = flatten (map k l)
  let zero () = []
  let plus l1 l2 = concat l1 l2
end
module MonadL = MonadJoin(ListM)
```

Figure 4.6: Sequential Monadic Joins in PolyJoin.

The Essence of Polyvariadic Joins

Rule (JOIN) formalizes the essence of polyvariadic n-join expressions, which we exemplified in Section 4.3.1. A join expression requires a valid pattern context Π consisting of `from` bindings. In the premise, context formation $\Gamma \vdash_{\text{ctx}} \Pi : \vec{A}$ certifies that Π is well-formed, exposing at the type-level that Π has the *context shape* $\vec{A}$, which is an ordered, *heterogeneous* sequence of types describing the number and element types of the bindings in Π.

Terminology: The context shape $\vec{A}$ is not to be confused with the abstract Shape$[\cdot]$ type constructor for input sources. We use the term "shape" for both concepts if there is no ambiguity.

Importantly, contexts Π and Γ are part of different domains of discourse: the former is a representation of the pattern variable context *in the object language/DSL*, whereas the latter is a context of the *meta language*, i.e., the language in which we reason about the object language, its expressions and free variables (mathematics or the OCaml programming language). Tracking the shape $\vec{A}$ as witnessed by Π in the type system will be crucial for programming polyvariadic definitions and performing type-level computations later on.

As we motivated in Section 4.2.3, there is a type-level functional dependency between the context of `from` bindings and the number and type of pattern variables in joins. Formally, context shape $\vec{A}$ translates to a context shape $\vec{B}$, written $\vec{A} \triangleright \vec{B}$. The latter shape describes the number and types of available pattern variables, which are used in the rightmost premise of rule (JOIN). This premise defines the body of the join pattern. Its variables are assumptions of a derivation ending in a pattern form P. The derivation generalizes the usual implication introduction rule of natural deduction to n assumptions and is represented in OCaml as an uncurried, n-ary function value/HOAS binding. Intuitively, the body P of the join pattern defines how to construct a single output element of type C, from elements $\vec{x}$ extracted from the join's input sources. As a result, the type of the whole join expression lifts this specification for single C elements into a whole collection Shape$[C]$. Here, the abstract type constructor Shape$[\cdot]$ corresponds to the type constructor $S[\cdot]$ in Definition 4.1. Overall, the `join` form defines an n-ary join followed by a map

$$\overrightarrow{\text{Shape}[A]} \to \text{Shape}[\vec{A}] \to \text{Shape}[C]$$

in the sense of our earlier definition. We make this correspondence more precise in Section 4.5.

Context Formation

In contrast to standard treatments of variable bindings and context (e.g., the typing context Γ), the pattern context representation Π in the DSL is nameless, i.e., they are *not* sequences of variable/type pairs $\overrightarrow{x : A}$, because we cannot directly model variable names in OCaml's type language. Instead, pattern contexts Π just witness the shape $\vec{A}$, which is enough to compute the appropriate signature of a join pattern.

Variable introduction $\vdash_{\text{var}} V : A$ witnesses that a binding term V introduces an anonymous variable of type A. In core PolyJoin, only `from`-bindings can introduce a variable via rule (FROM). The latter establishes that an anonymous variable of type A must come from an input source term M of Shape$[A]$. The rules for context formation $\vdash_{\text{ctx}} \Pi : \vec{A}$ specify that contexts are inductively formed by the terms `cnil` (empty context) and `@.` (prepending of variable to context).

Context Shape Translation

The purpose of the judgment $\vec{A} \triangleright \vec{B}$ is controlling how the types of pattern variables are computed from the shape $\vec{A}$ of the pattern context Π. The core version of PolyJoin trivially establishes $\vec{B} = \vec{A}$, i.e., the ith pattern variable binds a DSL term representing an element of the type A_i. The point is to enable customizations, e.g., in Chapter 5, we will customize the formal rules for this judgment, to model event pattern languages with timing.

4.4 OCAML REPRESENTATION OF POLYJOIN

Following the principles of Section 4.3.2, we define the intrinsically-typed syntax of PolyJoin in an OCaml module signature (Figure 4.3). Each of the judgment forms presented here corresponds to an abstract type constructor (which we marked in the comments) and each rule to a function. Meta variables in judgments become type variables in the functions. Note that the OCaml version bundles context formation and shape translation into a single type constructor/judgment. This way, we avoid requiring the user to manually supply the derivation of the context shape translation judgment, since it can be inductively defined from the structure of the context formation. Accordingly, the context formation rules `cnil` and `@.` also compute the context shape translation in the second type parameter of `ctx`.

Indeed, given this module signature, the unification-based Hindley-Milner (HM) type system of OCaml is sufficient for computing the correct type of the polyvariadic `join` form, for any expressible pattern context. For example, a partial application with three bindings

```
module Exp(S: Symantics) = struct
  open S  (* ... *)
  let exp =
    join ((from src1) @. (from src2) @. (from src3) @. cnil)
end
```

yields a three-ary join pattern

```
val exp: ((a1 S.repr * (a2 S.repr * (a3 S.repr * unit)) -> 'b S.
    pat)  -> 'b S.shape S.repr
```

as intended, given that `src`i, has type

```
a$i$ S.shape S.repr
```

for $i \in \{1, 2, 3\}$. We model ordered, heterogeneous sequences of types via nested binary product types in OCaml.

4.5 METATHEORY

Thus far, for simplicity, we did not strictly distinguish between PolyJoin, the formal language in Figure 4.2 and its encoding in OCaml (`Symantics` signature in Figure 4.3). We can justify this lack of distinction, by showing that there is a homomorphism (i.e., structure-preserving embedding) from the formal language into OCaml. This homomorphism asserts that the encoding of the formal system in the `Symantics` signature is reasonable. We can thus switch the meta language

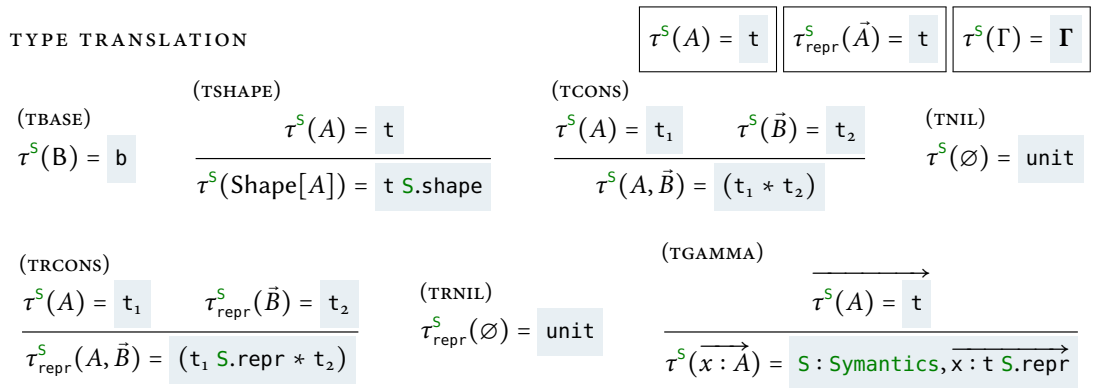


Figure 4.7: Translation of Core PolyJoin into `OCaml` : Types.

in which we reason about the event pattern DSL from mathematics to OCaml. We may then also transfer knowledge about the pen and paper formalization to the OCaml version and thus derive properties about the tagless DSL and its possible interpreters. This works in principle for any other language into which we wish to embed PolyJoin, as long as we are able to define a homomorphism.

To derive the homomorphism, we define translation rules for the types (Figure 4.7) and terms (Figure 4.8) in the formal language into OCaml types and terms. The translations are context-dependent on a `Symantics` instance `S`, due to the tagless final encoding. For readability, we highlight resulting OCaml entities in blue. We describe the translations below.

TYPE TRANSLATION The translation $\tau^{\mathsf{S}}(A)$ maps PolyJoin types to OCaml types and its homomorphic extension to variable contexts $\tau^{\mathsf{S}}(\Gamma)$, mapping variable contexts to variable contexts in OCaml. The translation rules for types are straightforward. For example, we map sequences of expression types $\vec{A}$ into nested binary pair types, associating to the right and terminating with the `unit` type by rules (TCONS) and (TNIL). In some cases, we require that the translated vector's element types are enclosed in the `S.repr` type constructor, which we ensure by the $\tau^{\mathsf{S}}_{\mathsf{repr}}(\vec{A})$ translation. Importantly, we prepend the resulting OCaml context with the given interpreter instance `S` in (TGAMMA) and all free variables range over expressions `S.repr`.

TERM TRANSLATION The term translation witnesses the homomorphism from PolyJoin into OCaml, i.e., for each typing derivation in the formal language, there is a corresponding typing derivation in OCaml, obtainable by the translation rules. Note that the translation $[\![\cdot]\!]^{\mathsf{S}}_{\mathsf{ctx}}$ simultaneously translates context formation and context shape translation, because the structure of context shape translation is determined by the context formation. Context shape translation exists solely at the type level and does not require a term representation.

TERM TRANSLATION

$$\boxed{[\![\Gamma \vdash_{\text{exp}} M : A]\!]^{\mathtt{S}}_{\text{exp}} = \tau^{\mathtt{S}}(\Gamma) \vdash \mathtt{m} : \tau^{\mathtt{S}}(A)\ \mathtt{S.repr}}$$

$$\boxed{[\![\Gamma \vdash_{\text{pat}} P : A]\!]^{\mathtt{S}}_{\text{pat}} = \tau^{\mathtt{S}}(\Gamma) \vdash \mathtt{p} : \tau^{\mathtt{S}}(A)\ \mathtt{S.pat}}$$

$$\boxed{[\![\Gamma \vdash_{\text{ctx}} \Pi : \vec{A},\ \vec{A} \triangleright \vec{B}]\!]^{\mathtt{S}}_{\text{ctx}} = \tau^{\mathtt{S}}(\Gamma) \vdash \mathtt{pi} : \mathtt{(}\ \tau^{\mathtt{S}}(\vec{A})\ \mathtt{,}\ \tau^{\mathtt{S}}_{\text{repr}}(\vec{B})\ \mathtt{)}\ \mathtt{S.ctx}}$$

$$\boxed{[\![\Gamma \vdash_{\text{var}} V : A]\!]^{\mathtt{S}}_{\text{var}} = \tau^{\mathtt{S}}(\Gamma) \vdash \mathtt{v} : \tau^{\mathtt{S}}(A)\ \mathtt{S.var}}$$

(TVAR)

$$\left[\!\!\left[\frac{\Gamma(x) = A}{\Gamma \vdash_{\text{exp}} x : A}\right]\!\!\right]^{\mathtt{S}}_{\text{exp}} = \frac{\tau^{\mathtt{S}}(\Gamma)(\ \mathtt{x}\ ) = \tau^{\mathtt{S}}(A)\ \mathtt{S.repr}}{\tau^{\mathtt{S}}(\Gamma) \vdash \mathtt{x} : \tau^{\mathtt{S}}(A)\ \mathtt{S.repr}}$$

(THWERE)

$$\frac{[\![\Gamma \vdash_{\text{exp}} M : \text{Bool}]\!]^{\mathtt{S}}_{\text{exp}} = \tau^{\mathtt{S}}(\Gamma) \vdash \mathtt{m} : \mathtt{bool}\ \mathtt{S.repr} \qquad [\![\Gamma \vdash_{\text{pat}} P : A]\!]^{\mathtt{S}}_{\text{pat}} = \tau^{\mathtt{S}}(\Gamma) \vdash \mathtt{p} : \tau^{\mathtt{S}}(A)\ \mathtt{S.pat}}{[\![\Gamma \vdash_{\text{pat}} \mathtt{where}\ M\ P : A]\!]^{\mathtt{S}}_{\text{pat}} = \tau^{\mathtt{S}}(\Gamma) \vdash \mathtt{where\ m\ p} : \tau^{\mathtt{S}}(A)\ \mathtt{S.pat}}$$

(TYIELD)

$$\frac{[\![\Gamma \vdash_{\text{exp}} M : A]\!]^{\mathtt{S}}_{\text{exp}} = \tau^{\mathtt{S}}(\Gamma) \vdash \mathtt{m} : \tau^{\mathtt{S}}(A)\ \mathtt{S.repr}}{[\![\Gamma \vdash_{\text{pat}} \mathtt{yield}\ M : A]\!]^{\mathtt{S}}_{\text{pat}} = \tau^{\mathtt{S}}(\Gamma) \vdash \mathtt{yield\ m} : \tau^{\mathtt{S}}(A)\ \mathtt{S.pat}}$$

(TJOIN)

$$\frac{\begin{array}{c}[\![\Gamma \vdash_{\text{ctx}} \Pi : \vec{A},\ \vec{A} \triangleright \vec{B}]\!]^{\mathtt{S}}_{\text{ctx}} = \tau^{\mathtt{S}}(\Gamma) \vdash \mathtt{pi} : \mathtt{(}\ \tau^{\mathtt{S}}(\vec{A})\ \mathtt{,}\ \tau^{\mathtt{S}}_{\text{repr}}(\vec{B})\ \mathtt{)}\ \mathtt{S.ctx} \\ \left[\!\!\left[\Gamma, \overrightarrow{x : B} \vdash_{\text{pat}} P : C\right]\!\!\right]^{\mathtt{S}}_{\text{pat}} = \tau^{\mathtt{S}}(\Gamma), \overrightarrow{\mathtt{x} : \tau^{\mathtt{S}}(B)\ \mathtt{S.repr}} \vdash \mathtt{p} : \tau^{\mathtt{S}}(C)\ \mathtt{S.pat}\end{array}}{[\![\Gamma \vdash_{\text{exp}} \mathtt{join}\ \Pi\ (\vec{x}.P) : \text{Shape}[C]]\!]^{\mathtt{S}}_{\text{exp}} = \tau^{\mathtt{S}}(\Gamma) \vdash \mathtt{join\ pi\ (fun}\ [\![\vec{x}]\!]_{\text{v}}\ \mathtt{->\ p)} : \tau^{\mathtt{S}}(C)\ \mathtt{S.shape\ S.repr}}$$

(TFROM)

$$\frac{[\![\Gamma \vdash_{\text{exp}} M : \text{Shape}[A]]\!]^{\mathtt{S}}_{\text{exp}} = \tau^{\mathtt{S}}(\Gamma) \vdash \mathtt{m} : \tau^{\mathtt{S}}(A)\ \mathtt{S.repr}}{[\![\Gamma \vdash_{\text{var}} \mathtt{from}\ M : A]\!]^{\mathtt{S}}_{\text{var}} = \tau^{\mathtt{S}}(\Gamma) \vdash \mathtt{(from\ m)} : \tau^{\mathtt{S}}(A)\ \mathtt{S.var}}$$

(TCNIL)

$$[\![\Gamma \vdash_{\text{ctx}} \mathtt{cnil} : \varnothing,\ \varnothing \triangleright \varnothing]\!]^{\mathtt{S}}_{\text{ctx}} = \tau^{\mathtt{S}}(\Gamma) \vdash \mathtt{cnil} : \mathtt{(unit, unit)}\ \mathtt{S.ctx}$$

(TCAT)

$$\frac{\begin{array}{c}[\![\Gamma \vdash_{\text{var}} V : B]\!]^{\mathtt{S}}_{\text{var}} = \tau^{\mathtt{S}}(\Gamma) \vdash \mathtt{v} : \tau^{\mathtt{S}}(B)\ \mathtt{S.var} \\ [\![\Gamma \vdash_{\text{ctx}} \Pi : \vec{A},\ \vec{A} \triangleright \vec{D}]\!]^{\mathtt{S}}_{\text{ctx}} = \tau^{\mathtt{S}}(\Gamma) \vdash \mathtt{pi} : \mathtt{(}\ \tau^{\mathtt{S}}(\vec{A})\ \mathtt{,}\ \tau^{\mathtt{S}}_{\text{repr}}(\vec{D})\ \mathtt{)}\ \mathtt{S.ctx}\end{array}}{\begin{array}{c}[\![\Gamma \vdash_{\text{ctx}} V\ @.\ \Pi : (B, \vec{A}),\ (B, \vec{A}) \triangleright (C, \vec{D})]\!]^{\mathtt{S}}_{\text{ctx}} = \\ \tau^{\mathtt{S}}(\Gamma) \vdash \mathtt{(v\ @.\ pi)} : \mathtt{(}\ \tau^{\mathtt{S}}(B)\ \mathtt{*}\ \tau^{\mathtt{S}}(\vec{A})\ \mathtt{,}\ \tau^{\mathtt{S}}(C)\ \mathtt{S.repr\ *}\ \tau^{\mathtt{S}}_{\text{repr}}(\vec{D})\ \mathtt{)}\ \mathtt{S.ctx}\end{array}}$$

PATTERN VARIABLE BINDING TRANSLATION

$$[\![\langle\rangle]\!]_{\text{v}} = \mathtt{()} \qquad [\![x, \vec{y}]\!]_{\text{v}} = \mathtt{(x,}\ [\![\vec{y}]\!]_{\text{v}}\ \mathtt{)}$$

Figure 4.8: Translation of Core PolyJoin into `OCaml` : Terms.

THEOREM 4.2 (TYPE PRESERVATION)
Let `S` *be a module implementing the* `Symantics` *signature in Figure 4.3. Then all of the following hold:*

- *If* $\Gamma \vdash_{\text{exp}} M : A$, *then* $\tau^{\mathtt{S}}(\Gamma) \vdash \mathtt{m} : \tau^{\mathtt{S}}(A)$ `S.repr` *in OCaml.*
- *If* $\Gamma \vdash_{\text{pat}} P : A$, *then* $\tau^{\mathtt{S}}(\Gamma) \vdash \mathtt{p} : \tau^{\mathtt{S}}(A)$ `S.pat` *in OCaml.*
- *If* $\Gamma \vdash_{\text{ctx}} \Pi : \vec{A},\ \vec{A} \triangleright \vec{B}$, *then* $\tau^{\mathtt{S}}(\Gamma) \vdash \mathtt{pi} : (\tau^{\mathtt{S}}(\vec{A}), \tau^{\mathtt{S}}_{\text{repr}}(\vec{B}))$ `S.ctx` *in OCaml.*
- *If* $\Gamma \vdash_{\text{var}} V : A$, *then* $\tau^{\mathtt{S}}(\Gamma) \vdash \mathtt{v} : \tau^{\mathtt{S}}(A)$ `S.var` *in OCaml.*

Where the OCaml terms `m`, `p`, `m`, `v` *are determined by the translations in Figure 4.8.* ♦

PROOF The term translation rules in Figure 4.8 are a constructive proof of this theorem. The rules form the cases of a proof by mutual induction over the derivations for expressions, patterns, context formation and pattern variable bindings. Rule premises are applications of the induction hypotheses and rule conclusions are the proof of the respective case. The resulting typing derivations are valid in OCaml, because the type checker accepts the `Symantics` signature, `S` is an instance and the derivation steps are admissible by the type signature of the syntax encoding. □

Term translation gives us a mechanical way of embedding intrinsically-typed PolyJoin terms into OCaml and it will always succeed:

COROLLARY 4.3 (OCAML EMBEDDING OF POLYJOIN)
If $\Gamma \vdash_{\text{exp}} M : A$, *then the OCaml type checker accepts the functor definition:*

```
module Exp(S:Symantics) = struct
  open S
  let exp = fun x : t S.repr -> m
end
```

where

$$[\![\Gamma \vdash_{\text{exp}} M : A]\!]^{\mathtt{S}}_{\text{exp}} = \tau^{\mathtt{S}}(\Gamma) \vdash \mathtt{m} : \tau^{\mathtt{S}}(A)\ \mathtt{S.repr}$$

and

$$\tau^{\mathtt{S}}(\Gamma) = \mathtt{S} : \mathtt{Symantics}, \overrightarrow{\mathtt{x} : \mathtt{t}\ \mathtt{S.repr}}.$$

Moreover, the type checker assigns the functor signature

```
functor (S:Symantics) -> sig
  val exp: t S.repr -> τ^S(A) S.repr
end.
```

♦

PROOF Consequence of the Type Preservation Theorem 4.2. □

Analogous corollaries hold for patterns P, pattern contexts Π and pattern variables V, which we leave as an exercise to the reader.

Corollary 4.3 refers to one embedding direction, but what about the other? Not every OCaml expression of type `t S.repr` has a corresponding expression in PolyJoin, because OCaml supports general recursion and thus permits well-typed PolyJoin expressions that do not have normal forms.

SOUNDNESS The OCaml type checker enforces the syntax and typing rules of POLYJOIN and thus rejects any ill-typed or ill-formulated pattern. It ensures that evaluation of closed patterns and expressions *never gets stuck*. These guarantees continue to hold when extending POLYJOIN with new expression or pattern syntax forms. In the tagless final approach, we have that "*The soundness of the object language's type system with respect to the dynamic semantics specified by a [tagless] interpreter follows from the soundness of the metalanguage's type system.*" (Carette, Kiselyov, and Shan [CKS09]). OCaml contains some language features that are unsound, e.g., type casts. However, if we restrict ourselves to a sound fragment of OCaml without these problematic features, it is guaranteed that any event pattern expression we define in OCaml cannot "go wrong" with respect to any tagless interpreter that implements `Symantics`. Below, we repeat the soundness argument by Carette, Kiselyov, and Shan [CKS09] in more formal terms:

THEOREM 4.4 (TYPE SOUNDNESS)
Consider a sound fragment of OCaml that includes the module system and let `I` *be a module (tagless interpreter) that implements the* `Symantics` *signature in Figure 4.3.*
If

$$\varnothing \vdash_{\text{exp}} M : A$$

and

$$\left[\!\left[\varnothing \vdash_{\text{exp}} M : A\right]\!\right]^{\mathtt{S}}_{\text{exp}} = \tau^{\mathtt{S}}(\varnothing) \vdash \mathtt{m} : \tau^{\mathtt{S}}(A)\ \mathtt{S.repr},$$

let

```
module Exp =
  (functor (S:Symantics) -> struct open S let exp () = m end)
  (I),
```

be the instantiation of expression M in terms of the interpreter `I`*, then the term* `Exp.exp ()` *has type* $\tau^{\mathtt{I}}(A)$ `I.repr` *and its evaluation* is not stuck. ♦

PROOF By the Type Preservation Theorem 4.2, the translation of M to `m` succeeds having the type $\tau^{\mathtt{S}}(A)$ `S.repr` in the context `S : Symantics`. By construction, `Exp.exp ()` receives the type $\tau^{\mathtt{I}}(A)$ `I.repr` from the functor application to `I`. If the evaluation of `Exp.exp ()` were stuck, then this expression would be a counterexample to the soundness of the assumed OCaml fragment. □

Well-typed programs cannot "go wrong".

— Milner [Mil78]

The notion of "not stuck" intuitively means that there are no type errors occurring during the execution of the tagless interpreter `I` on the given expression. Type errors are ill-defined or undesired situations during evaluation (cf. Pierce [Pie02]). Importantly, depending on the chosen OCaml fragment, the expression may still diverge or throw an exception, though.

4.5.1 *Polyvariadicity*

Here, we establish that `join` is a syntactic representation of a class of polyvariadic functions over the abstract Shape[·] type, relating back to Definition 4.1 for n-way joins. We require a few simple lemmas first:

LEMMA 4.5 (CONTEXT INVERSION)
If $\Gamma \vdash_{\text{ctx}} \Pi : A_1, \ldots, A_n$, *then*

$$\Pi = \texttt{from}\ M_1\ \texttt{@}.\ \cdots\ \texttt{@}.\ \texttt{from}\ M_n\ \texttt{@}.\ \texttt{cnil}$$

for some expressions $M_1, \ldots, M_n$, *such that*

$$(\Gamma \vdash_{\text{exp}} M_i : \mathsf{Shape}[A_i])_{1 \le i \le n}.$$ ♦

PROOF By a straightforward induction over the derivation of $\Gamma \vdash_{\text{ctx}} \Pi : A_1, \ldots, A_n$. □

The Context Inversion Lemma asserts that the shape $\vec{A}$ of the pattern variable context is a *type-level reflection* of the number of input sources and their element types. Π is a *witness* of this type-level fact at the term level.

LEMMA 4.6 (JOIN INVERSION)
If $\Gamma \vdash_{\text{exp}} \texttt{join}\ \Pi\ (\vec{x}.P) : A$, *then*

1. $\Gamma \vdash_{\text{ctx}} \Pi : A_1, \ldots, A_n$.
2. $A_1, \ldots, A_n \triangleright A_1, \ldots, A_n$.
3. $\Gamma, x_1 : A_1, \ldots, x_n : A_n \vdash_{\text{pat}} P : B$, *for some type* B.
4. $A = \mathsf{Shape}[B]$, *for the same* B *above.* ♦

PROOF Any derivation of $\Gamma \vdash_{\text{exp}} \texttt{join}\ \Pi\ (\vec{x}.P) : A$ must end with rule (JOIN) (Figure 4.2), since no other rule matches. The lemma follows immediately from the premises and inversion on the context shape translation. □

LEMMA 4.7 (POLYVARIADICITY OF THE JOIN SYNTAX)
For all $n \in \mathbb{N}$, *all types* $A_1, \ldots, A_n$, *all variables* $x_1, \ldots, x_n$, $y_1, \ldots, y_n$, *all patterns* P, *and all types* B *such that* $y_1 : A_1, \ldots, y_n : A_n \vdash_{\text{pat}} P : B$, *let*

$$\Gamma = x_1 : \mathsf{Shape}[A_1], \ldots, x_n : \mathsf{Shape}[A_n]$$

and

$$\Pi = (\texttt{from}\ x_1\ \texttt{@}.\ \cdots\ \texttt{@}.\ \texttt{from}\ x_n\ \texttt{@}.\ \texttt{cnil}),$$

then

$$\Gamma \vdash_{\text{exp}} \texttt{join}\ \Pi\ (y_1, \ldots, y_n.P) : \mathsf{Shape}[B].$$ ♦

PROOF It is straightforward to construct a derivation of $\Gamma \vdash_{\text{ctx}} \Pi : \vec{A}$ and $\vec{A} \triangleright \vec{A}$. Together with the assumption $y_1 : A_1, \ldots, y_n : A_n \vdash_{\text{pat}} P : B$, we obtain a derivation of the goal by application of rule (JOIN). □

The lemma above essentially states that `join` describes families of functions indexed by the context shape $\vec{A}$:

$$\left(\overrightarrow{\mathsf{Shape}[A]} \to \mathsf{Shape}[B]\right)_{\vec{A}}$$

and that pattern variable contexts Π are term representations of this index within the object language/DSL. At a given index $\vec{A}$, we obtain such a function by binding the free variables in Γ by an n-ary function abstraction at the meta level, e.g.,

$$\boldsymbol{\lambda}\vec{x}.\texttt{join}\ \overrightarrow{\texttt{from}\ x}\ (\vec{y}.P),$$

where we write $\boldsymbol{\lambda}$ to indicate that the abstraction is in the meta language and not in the object language PolyJoin. This function family is a variant of the n-way join signature (Definition 4.1) which we motivated earlier.

Importantly, the polyvariadicity of the `join` syntax carries over to OCaml in a specific way, which gives useful guarantees. The following theorem describes how polyvariadicity manifests in the tagless final approach:

THEOREM 4.8 (POLYVARIADICITY OF THE OCAML JOIN SYNTAX)
In the `Symantics` *module signature (Figure 4.3)*

```
join: ('a, 'b) ctx -> ('b -> 'c pat) -> 'c shape repr
```

corresponds to a polyvariadic function signature*, i.e., a family* `f` *of signatures*

```
f: (t₁ * ⋯ * tₙ, t₁ repr * ⋯ * tₙ repr) ctx
     -> (t₁ repr * ⋯ * tₙ repr -> u pat) -> u shape repr
```

indexed by all $n \in \mathbb{N}$, all n-vectors of types $\texttt{t}_1 \cdots \texttt{t}_n$ *and all types* `u`*. That is, in all OCaml contexts* `S:Symantics, open S`*:*

1. *Every* `join` *expression accepted by the OCaml type checker receives type parameter instantiations of the form*
   - $\texttt{'a} = \texttt{t}_1 * \cdots * \texttt{t}_n$
   - $\texttt{'b} = \texttt{t}_1\ \texttt{repr} * \cdots * \texttt{t}_n\ \texttt{repr}$
   - $\texttt{'c} = \texttt{u}$

   for some $n \in \mathbb{N}$ and types $\texttt{t}_1 \cdots \texttt{t}_n$ *and* `u`*.*
2. *For any given $n \in \mathbb{N}$, and any n-vector of types* $\texttt{t}_1 \cdots \texttt{t}_n$*, the OCaml type checker accepts the partial application*

   ```
   let exp = fun (x₁ : t₁ shape repr)⋯(xₙ : tₙ shape repr) ->
     join (from x₁ @.⋯@. from x₁ @. cnil)

   (* exp: t₁ shape repr -> ⋯ -> tₙ shape repr ->
           (t₁ repr * ⋯ * tₙ shape repr -> 'c pat) ->
           'c shape repr *)
   ```

   with the type parameter instantiation
   - $\texttt{'a} = \texttt{t}_1 * \cdots * \texttt{t}_n$
   - $\texttt{'b} = \texttt{t}_1\ \texttt{repr} * \cdots * \texttt{t}_n\ \texttt{repr}$
   - $\texttt{'c} = \texttt{'c}$

   ⧫

PROOF

1. By application of the Type Preservation Theorem 4.2 to the Join Inversion Lemma 4.6.
2. By application of the Type Preservation Theorem 4.2 to the Polyvariadicity Lemma 4.7. □

GUARANTEES Theorem 4.8 certifies that we faithfully capture the syntax and type signature of declarative n-way join patterns on individual elements out of the input shapes, using nothing but OCaml:

- *Clients/end users* cannot specify ill-formulated or ill-typed patterns. They can formulate patterns of any arity, and the type-level relation between inputs and pattern variables is automatically computed, just by supplying the context Π, which specifies the inputs. No additional syntactic burden is imposed on end users.
- *Implementations of tagless interpreters* are guaranteed that the abstract type variables in the type of `join` are constrainted to type-level sequences having a specific shape. Below, we discuss what we can predict about the behavior of concrete tagless interpreters, just from the polyvariadic signature of `join`.

4.5.2 *Polymorphic Variable Binding*

In Theorem 4.8, we established that the OCaml signature of `join` models a family of signatures. Recall the inferred type from part 2 of the theorem,

```
t₁ S.shape S.repr -> ··· -> tₙ S.shape S.repr ->
(t₁ S.repr * ··· * tₙ S.shape S.repr -> 'c S.pat) ->
'c S.shape S.repr
```

which is indexed by the shape `t₁ * ··· * tₙ` of the pattern variable context (we prefixed the types with the context `S`). Importantly, for any $n \in \mathbb{N}$, we may generalize the signature to a polymorphic signature, since we quantify over all n-vectors of type indexes:

```
'a₁ S.shape S.repr -> ··· -> 'aₙ S.shape S.repr ->
('a₁ S.repr * ··· * 'aₙ S.shape S.repr -> 'c S.pat) ->
'c S.shape S.repr
```

By parametricity [Wad89], we can predict how any concrete tagless interpreter `S` functions at any index n: It may emit output of type `'c` into the result

```
'c S.shape S.repr
```

where the `'c` elements can only come from the given n-ary HOAS pattern abstraction. Obtaining the pattern representation `'c S.pat` requires binding n expressions, which by their type must come from the n input shape expressions. That is to say: a tagless interpreter `S` extracts combination of n-tuples from the input shapes to compute the output. It also must have a way to lift its pattern representation `'c S.pat` into a shape.

Importantly, the type signature does not prescribe *how* or in *what order* the interpreter selects n-tuples. More varied ways of variable bindings are permitted, compared to monadic approaches. In this sense, PolyJoin achieves polymorphism in the semantics of variable bindings.

4.6 INTERMEZZO: HETEROGENEOUS SEQUENCES IN OCAML

Our OCaml encoding of PolyJoin models n-vectors, respectively heterogeneous type sequences by right-associative, nested binary pair types (cf. type translation Figure 4.7). Due to the term representation of pattern contexts Π, it is useful to have a data type for heterogeneous sequences and combinators for manipulating them, in order to program concrete tagless interpreters for PolyJoin.

We employ generalized algebraic data types (GADTs) [JG08; CH03] for an encoding of heterogeneous lists, similar to [KLS04], that is:

```
type _ hlist = Z : unit hlist
             | S : 'a * 'b hlist -> ('a * 'b) hlist
```

GADTs implement a form of bounded polymorphism by allowing non-uniform type-parameter instantiations in constructor declarations, in contrast to standard polymorphic data type definitions. For instance, one can tell that `hlist` is a GADT from the underscore in the type parameter position. The constructors `Z` (empty list) and `S` (list cons) constrain the shape of the possible types that can be filled into the type parameter. In the case of `hlist`, the type parameter describes the exact shape of a heterogeneous list value. For example, the value

```
S (1, S ("two", S (3.0, Z)))
```

has type `(int * (string * (float * unit))) hlist`. GADTs enable the compiler to prove more precise properties of programs. A classic example is the safe head function on `hlist`

```
let safe_head: type a b. (a * b) hlist -> a =
  function S (x,_) -> x
```

which statically guarantees that it can be invoked only with non-empty list arguments. This works because we constrain the shape of the argument's type parameter to `(a * b) hlist`. Hence, by the definition of `hlist` above, the argument can only be of the form `S (x,_)`.

Uniform Heterogeneity

We sometimes require stronger invariants on the heterogeneous context/list shape certifying that elements are uniformly enclosed in a given type constructor. For example, having a heterogeneous list of homogeneous lists: `int list * (string list * (float list * unit))`. It is hard to enforce such invariants using only the bare `hlist` type above, because the type parameters of elements `'a` in the `S` constructor quantifies over any type. We overcome this issue by defining a *family* of constrained `hlist` types, which is parameteric over a type constructor `'a el`, applied element-wise (Figure 4.4a). We may create concrete instances of constrained `hlists` in the form of OCaml modules, with the functor `HList` (Figure 4.4b). For example, the modules

```
(* Standard hlist *)
module HL = HList(struct type 'a t = 'a end)
(* hlist of lists *)
module Lists = HList(struct type 'a t = 'a list end)
```

define unconstrained `hlists` and `hlists` of homogeneous lists, respectively. To distinguish between concrete `hlist` variations in programs, we qualify the types and constructors by the defining module's name, e.g., the function

```
let rec enclose: type a. a HL.hlist -> a Lists.hlist =
  function
  | HL.Z -> Lists.nil
  | HL.S (hd,tl) -> Lists.(cons [hd] (enclose tl))
```

is a polyvariadic function that element-wise converts unconstrained `hlists` into `hlists` of homogeneous lists.

Shape Preservation

The definition of `enclose` above features a subtle, but important design principle. Its signature

```
type a. a HL.hlist -> a Lists.hlist
```

is polyvariadic, because it quantifies *over all possible shapes* `a` of `hlists`. Notice that the same shape `a` also appears in the codomain of the function type. That means, `enclose` is *shape preserving*, i.e., it does not change the number of elements or their basic types. But it changes the enclosing type constructor.

The invariance of the type parameter `a` is a way of encoding the relation between heterogeneous input and output in OCaml's type system, without the need of advanced type-level computations. From the definition of the `HL` and `Lists` modules and the invariance in shape `a`, the above signature certifies that for all $n \geq 0$, `enclose` takes n-tuples

$$(a_1, \ldots, a_n)$$

to n-tuples

$$(a_1 \texttt{ list}, \ldots, a_n \texttt{ list}).$$

Shape preservation is useful to compose independently developed polyvariadic components in a type-safe manner. The idea is that if two components are indexed by the same context shape, then they are composable. For instance, consider:

```
module ListRefs = HList(struct type 'a t = 'a list ref end)
let rec refs: type a. a Lists.hlist -> a ListRefs.hlist =
  function
  | Lists.Z -> ListRefs.Z
  | Lists.S (hd, tl) -> ListRefs.S (ref hd, (refs tl)).
```

The shape-preserving function `refs` stores a heterogeneous list of lists into heterogeneously-typed reference cells. We can compose the two previous example functions to obtain a new shape-preserving function:

```
let composed: type a. a HL.hlist -> a ListRefs.hlist =
  fun hl -> refs (enclose hl).
```

Generic Operations on Heterogeneous Lists

In examples, we make use of higher-order functions over `hlists`, which are definable in terms of OCaml functors. We show their definitions in Appendix B.1. For example, the `HMAP` functor defines a *shape-preserving*, element-wise conversion of `hlists`. For example, the following code

```
1 let enclose: type a. a HL.hlist -> a Lists.hlist =
2   (* declare a local module instance for mapping between standard
       hlists and hlists of lists *)
3   let module M = HMAP(HL)(Lists)in
4   M.map {M.ftor = fun x -> [x]}
```

re-implements the `enclose` function above in terms of mapping over heterogeneous lists. Due to the second-class status of the module system, programming with these operations requires appropriate module instantiations beforehand (e.g., Line 3). The last line shows a polymorphic record in OCaml, which we employ to define first-class polymorphic functions for mapping.

4.7 PROGRAMMING POLYVARIADIC TAGLESS INTERPRETERS

Here, we address how to program concrete tagless interpreters for PolyJoin as modules implementing `Symantics` (Figure 4.3). Tagless interpreters essentially define a denotational semantics for the PolyJoin syntax, i.e., bottom-up folds over the syntax tree into a semantic domain. How does it work with polyvariadic syntax forms, such as `join`?

Recall that our tagless final encoding of the `join` signature

```
join: ('a, 'b) ctx -> ('b -> 'c pat) -> 'c shape repr
```

models polyvariadicity by quantifying over all possible pattern variable contexts `('a,'b) ctx`. The type parameter `'a` is a type-level index witnessed by the pattern variable context, determining the signature `'b` of the join pattern variables.

Since denotations/tagless interpreters are bottom-up folds, the implementation of `join` can only assume and work with the denotations of the context and the pattern abstraction. We need a suitable denotation for the `('a, 'b) ctx` type of contexts. The Context Inversion Lemma 4.5 tells us that the derivations of the context formation contains the input shapes over which to join. The inputs are certainly necessary information to get hold of for reasonable implementations of `join`.

The Context Inversion Lemma points to a simple standard denotation for contexts: The *derivations trees* of the context formation judgment (Figure 4.2) over closed contexts, $\vdash_{\text{ctx}} \Pi : \vec{A}$. We encode the derivation trees of this formal system in terms of a GADT in OCaml:

We instantiate the abstract context signature (Figure 4.3) in terms of constrained heterogeneous lists and other GADTs, as shown in Figure 4.5. The variable representation `'a var` becomes a GADT, whose constructor `Bind` encloses an input source representation and links it to an element variable. The context representation `'a ctx` is instantiated to the constrained heterogeneous list type `Ctx.hlist`, which stores `Bind` values. Thus, contexts are concrete data values that the implementation of the interpreter is free to inspect and manipulate. We also need to model the context shape translation judgment from Figure 4.2, which is codified by the `('a, 'b) shape` GADT (Figure 4.5, Lines 5-7). As explained in Section 4.4, context formation and shape translation are combined into a single judgment, which translates to their product in Figure 4.5, Line 9.

Hence, interpreters of `join` really are proofs by induction over the context derivation, which we can express in OCaml using a recursive function over a GADT representation of the derivation. GADTs enable case analyses over derivations.

4.7.1 *Example: Sequential Monadic Bind*

We conclude with an example PolyJoin interpreter, implementing an old friend: the nested monadic translation from LINQ/comprehensions (Section 4.2). Figure 4.6 shows the full implementation.[6] The interpreter denotes expressions as OCaml values and works with all instances of the `MonadPlus` class, similar to LINQ. In this interpreter, we also set the pattern type constructor to the monad type constructor and use the `zero` element to indicate failure in the `where` clause of the pattern.

Given a `MonadPlus` instance, the tagless interpreter `MonadJoin` implements the n-nary nested monadic bind, by a straightforward structural induction over the standard context representation (Figure 4.5) in the function `bind_all` (Figure 4.6, Line 18-22). This function encloses a given continuation function `k` of arity n with n layers of monadic list bindings. The n-ary body of the join pattern is the innermost layer of this nesting (Line 18).

Here is an example instantiation with standard lists as the monad:

```
1 open MonadL
2 join ((from [1]) @. (from ["2";"3"]) @. (from [4.0;5.0]) @. cnil)
3      (fun (x, (y, (z,())))) -> (yield (pair x (pair y z))))
4 (* result: (int * (string * float)) list =
5    [(1, ("2", 4.)); (1, ("2", 5.)); (1, ("3", 4.)); (1, ("3", 5.)
      )] *)
```

The point is to show that the PolyJoin embedding is at least as expressive as LINQ or comprehensions. In the next chapter, we show that PolyJoin is strictly more expressive, supporting not only sequential variable bindings

4.8 CHAPTER SUMMARY

PolyJoin captures the *essence* of join notations found in database and event correlation systems. It enables a type-safe programming language integration of event patterns in terms of the tagless final approach and polyvariadic syntax forms. We enable the latter by an explicit type-level representation that describes the number and types of the variables that an event pattern contains. Our embedding is polymorphic in the sense that the pattern syntax supports different denotations, i.e., we effectively support different backends.

The indexing technique ensures that typed join patterns of arbitrary, but finite arity can be formulated (frontend) and that tagless interpreters provide a backend implementation that has the ability to correctly and uniformly handle all arities and all types of inputs. Embedding the object language of event patterns into a typed host/meta language has the benefit of exploiting the latter's type system to check and enforce the types of the former.

In general, event patterns require a non-monadic variable binding semantics and are thus not well-supported by state of the art language-integrated query approaches. The combination of polyvariadic syntax forms having n-ary HOAS variable bindings with the tagless final approach eliminates this deficiency. We support patterns with an arbitrary number of event sources and heterogeneous event types, while not requiring a host language with dependent types.

6 For the example, we extended the DSL with support for constructing binary pairs (Figure 4.6, Line 16).

5 COMING FULL CIRCLE: EMBEDDING CARTESIUS WITH POLYJOIN

This chapter shares material with [Bra+19].

SYNOPSIS With the aid of PolyJoin, we implement the first fully polyvariadic version of Cartesius with a proper embedded pattern syntax, improving the initial prototype we discussed in Chapter 3.[1] Recall that the effect-based model of Cartesius makes heavy use of generative algebraic effects. We contribute programming abstractions for *context polymorphism* and *position polymorphism*, which give rise to safe, reusable, modular and convenient handlers of polyvariadic generative effects. They can be defined once and are safely usable in the context of any n-way join, as long as certain compatibility constraints are satisfied, which we check and enforce statically.

One particular problem that we solve is safe callback logic for joining n communication partners in the presence of external choice, which we call the "focusing continuations problem". We believe that this is an important contribution beyond event correlation, because it applies to implementations of asynchrony/concurrency libraries as well as typed embeddings of process calculi.

Importantly, Cartesius is an interesting larger example and representative of systems having event patterns that expose implicit metadata (e.g., event timing). With a simple design variation of PolyJoin, we show how to safely embed extensible syntax designs for this class of systems. Another important point of this chapter is showcasing how to program a larger polyvariadic backend against PolyJoin's polyvariadic interfaces.

5.1 CONTRIBUTIONS

The contributions of this chapter are as follows:

- An embedding of Cartesius, yielding its first type-safe and fully polyvariadic implementation in multicore OCaml.
- We introduce extensions to core PolyJoin to properly support event correlation pattern designs based on implicit metadata associated to events, e.g., time or geolocation. (Section 5.3).
- We address how to program safe and reusable effect handlers for join computations in the context of arbitrary arities, via the notions of *context polymorphism* and a refinement called *position polymorphism*, for safely accessing and linking with sub-parts of a context.
- An evaluation of the PolyJoin version of Cartesius, comparing it against our initial prototype (Section 5.7). The PolyJoin version adds declarative pattern syntax, has exponential savings in code size, supports any arity and significantly reduces programmer effort when defining extensions.

1 All implementation variants of Cartesius are available at http://bracevac.org/correlation.

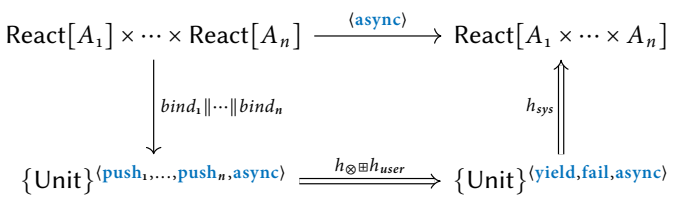


Figure 5.1: Join Computations in CARTESIUS.

```
join ((from temp_sensor) @. (from smoke_sensor) @. cnil)
      ((most_recently p0) |++| (most_recently p1))
      (fun ((temp,t1), ((smoke,t2), ())) ->
        yield (pair temp smoke))
```

Figure 5.2: Example: Join Pattern with CARTESIUS-style Restrictions.

This chapter constitutes the second of two parts. We recommend reviewing the first part introducing the POLYJOIN embedding (Chapter 4). Detailed knowledge of Chapter 3 on CARTESIUS is not required. We give a high-level summary of the system. We also recommend reviewing multicore OCaml in Section 2.3.4.

5.2 BRINGING CARTESIUS FROM THEORY INTO PRACTICE

In this section, we briefly recapitulate the CARTESIUS model based on algebraic effects and handlers from Chapter 3, and discuss interesting and noteworthy implementation challenges we need to solve, in order to successfully marry CARTESIUS with POLYJOIN.

The diagram in Figure 5.1 illustrates how polyvariadic join computations in the sense of Definition 4.1 (top row) relate to CARTESIUS' effect-based representation (bottom row). In this setting, function types (→) are annotated with Koka-style effect rows [Lei17b], indicating the side effect a function call may induce. We write concrete effect types in blue font. CARTESIUS joins perform global asynchrony effects ⟨**async**⟩ (cf. [Dol+17; Lei17a]), accounting for the inversion of control present in event correlation (Section 4.2.1). The other kind of arrow (⇒) indicates *effect handlers*, which transform effectful computations into other effectful computations. The overall idea is that *event notifications* are effects, *event sources* are computations inducing the effects and effect handlers are *event observers*. Sections 3.3 and 3.4 explain the full details.

CARTESIUS combines n input sources into a unit-valued effectful computation of type

$$\{\mathsf{Unit}\}^{\langle \mathbf{push}_1,\ldots,\mathbf{push}_n,\mathbf{async}\rangle}$$

$$\kappa_i : A_i \to M[A_1 \times \cdots \times A_n]$$
$$\kappa_i = \lambda x : A_i . m_1 \otimes \cdots \otimes m_{i-1} \otimes \langle x \rangle \otimes m_{i+1} \otimes \cdots \otimes m_n \quad 1 \leq i \leq n$$
$$(\_ \otimes \_) : \forall \alpha\, \beta . M[\alpha] \to M[\beta] \to M[\alpha \times \beta]$$

Figure 5.3: The Focusing Continuations Problem.

by a *parallel composition of bindings* (left column). We write computation types in curly braces (i.e., they designate thunks) annotated with a row specifying the possible side effects. Intuitively, this computation is an asynchronous process that in parallel subscribes to n input event sources, interleaving and forwarding their event notifications. We model these notifications in terms of a family of n heterogeneous effect operations $(\mathbf{push}_i)_{1 \leq i \leq n}$, which correspond to a sequence of effect declarations in multicore OCaml, informally written:

$$(\texttt{effect Push}_i\texttt{: 't}_i\texttt{ -> unit})_{1 \leq i \leq n}$$

That is, effects are named operations in the algebraic effect setting, having the signature of functions. Computations invoke them to induce the effect. Here, $\texttt{Push}_i$ carries in its parameter a typed event value from the i-th event source.

The effect row statically certifies that the binding is indeed parallel: in contrast to sequential monadic binding (cf. Section 4.2.2) there is no control dependency between the event notification effects, i.e.,

$$\langle \mathbf{push}_1, \ldots, \mathbf{push}_n, \mathbf{async} \rangle \equiv \langle \mathbf{push}_{\pi(1)}, \ldots, \mathbf{push}_{\pi(n)}, \mathbf{async} \rangle$$

are equivalent in the effect type system, for any permutation $\pi : \{1 \ldots n\} \to \{1 \ldots n\}$, certifying that the event notifications may occur in any order and arbitrarily often.

Invocations of effect operations are discharged by effect handlers, intuitively generalizations of exception handlers, which can resume back (similarly to coroutines [MI09]). Hence, effect handlers define a custom semantics/implementation of effect operations and are transformations ($\Rightarrow$) between effectful computations.

The bottom row of Figure 5.1 indicates that an effect handler $h_\otimes \boxplus h_{user}$ transforms the above event-observing process into a process that generates and tests correlated n-tuples, either yielding or discarding them through effects:

```
effect Yield: t₁ × ⋯ × tₙ -> unit (* Output tuple *)
effect Fail:  unit -> 'a          (* Discard tuple *)
```

The handler $h_\otimes \boxplus h_{user}$ is a composition ($\boxplus$) of a system default handler $h_\otimes$ and a user-defined handler h_{user}, which implements the actual event correlation logic. Intuitively, the correlation logic is an n-way coroutine that coordinates the n event subscriptions over the event sources. Finally, the handler h_{sys} is part of the system runtime and sends generated n-tuples into the output event source by handling the **yield**/**fail** effects.

5.2.1 *Callback Logic*

As part of the join computation, the system handler $h_{\otimes}$ in Figure 5.1 stores observed event notifications locally in n heterogeneous mailboxes. Each new event notification triggers a generic cartesian product computation over these mailboxes, for generating and testing n-tuples that satisfy the join's constraints.

Due to the asynchrony of the n event sources, the above strategy requires registering n heterogeneous callbacks/continuation functions $(\kappa_i)_{1 \leq i \leq n}$, which together implement the cartesian product. In more formal terms, the callbacks have the shape depicted in Figure 5.3, where $M[\cdot]$ is a collection type for mailboxes and $\otimes$ a binary cartesian product operator on mailboxes. Each callback κ_i represents the behavior when the i-th event source fires its next event, which is bound to the x variable. i.e., it replaces the i-th mailbox with the singleton mailbox $\langle x \rangle$ in the cartesian product over all mailboxes $m_i : M[A_i]$.

Writing callbacks for joining asynchronous computations leads to another manifestation of the binding problem for event correlation (cf. Section 4.2.2), this time in terms of the well-known continuation monad.[2] To avoid these problems and enable low latency reactions, the n callbacks perform a "focusing in the middle" of the expression $m_1 \otimes \cdots \otimes m_n$ for each possible position. The possible focus positions reflect the choices that the external environment can make to supply events (inversion of control).

5.2.2 *Restriction Handlers*

CARTESIUS supports a way to compose and inject user-defined effect handlers (the h_{user} component in Figure 5.1), called *restriction handlers*. They model composable sub-computations that change the behavior of the whole computation. Without custom restrictions, the variable bindings in CARTESIUS join patterns have a universal quantification semantics, executing the cartesian product above.

Figure 5.2 shows a preview of our POLYJOIN embedding for CARTESIUS.[3] The pattern specifies that the `most_recently` restriction handler shall apply to the first (`p0`) and second (`p1`) event source of the join (Line 2). Or equivalently: the first and second pattern variable should have the most recently binding semantics. For now, we leave `p0`, `p1` underspecified, they are essentially nameless indices [Bru72] into the context of the join. Intuitively, the operator `|++|` composes restriction handlers ($\boxplus$).

These restriction handlers change the behavior of the default cartesian product to achieve the *combine latest* correlation semantics, which is not expressible with nested joins (cf. Section 4.2.2). For example, from the input

```
temp_sensor  = < (120,  1), (50,    3), (20,   5) >
smoke_sensor = < (true, 1), (false, 2), (true, 4) >
```

we get

```
< ((120, true),  [1,1]),
  ((120, false), [1,2]),
```

2 Mainstream libraries for asynchronous programming (e.g., C♯ [Bie+12] Tasks or Scala Futures [EPF13]) are essentially instances of the continuation monad and join multiple computations by nesting callback subscriptions, which introduces unnecessary sequential dependencies.

3 We do not achieve the exact *lexical* syntax of the event correlation patterns from Chapter 3, but come pretty close with pure OCaml. The relation between the POLYJOIN version of the syntax and CARTESIUS is: **`let correlate = join`**. Readers who grew to like the **correlate** delimiter can mentally (and while programming) substitute one for the other for some relief.

```
((50,  false), [2,3]),
((50,  true),  [3,4]),
((20,  true),  [4,5]) >.
```

That is, the correlation always reflects the most up to date input events.

The restriction handler mechanism enables changing the variable binding semantics of the correlation at the level of individual variables, in contrast to Section 4.3, where we had a polymorphic, but *uniform* variable binding semantics for all n pattern variables. For example, if we leave out the `most_recently p1` restriction in Line 2, then the second variable binding would retain the default cartesian product semantics: "join the most up to date value of the first event source with *all* events from the second".

Moreover, CARTESIUS features restriction handlers constraining more than one variable binding simultaneously. For example, if we replaced Line 2 with the restriction handler `aligning p0 p1`, then we would obtain a correlation that *zips* its inputs (cf. Section 3.4.4).

```
< ((120,true), [1,1]), ((50,false), [2,3]), ((20,true), [4,5]) >.
```

The general form of this restriction is `aligning` K, i.e., a given *subset* $K \subseteq \{1 \dots n\}$ of inputs should be aligned.

5.2.3 *Conclusion and Challenges*

The CARTESIUS design introduces additional (polyvariadic) syntax elements and sub-components, which we need to address for a statically type-safe embedding and implementation of the language. We will show that these are well within reach of POLYJOIN's conceptual tools.

POLYVARIADIC EFFECT DECLARATIONS AND HANDLERS The effect declarations of CARTESIUS have type signatures which are functionally dependent on the types of event sources to be joined. Yet, no linguistic concept in the language of *declarations* allows calculating a sequence of heterogeneous effect declarations from the types of inputs. This is a new instance of polyvariadicity, because so far, we addressed it in the type and expression languages only. Another issue is how to idiomatically write polyvariadic handlers for these effects.

EVENT METADATA CARTESIUS patterns expose time data of events and accordingly constraints on time data, e.g., `within` in Figure 4.1b.

THE FOCUSING CONTINUATIONS PROBLEM The family $(\kappa_i)_{1 \le i \le n}$ of callback functions is a non-trivial polyvariadic definition. The issue is calculating a polyvariadic representation of the different focusing positions and replacements in the heterogeneously typed sequence

$$(m_i)_{1 \le i \le n} : M[A_1] \times \cdots \times M[A_n]$$

over a mailbox collection type $M[\cdot]$. Seemingly, we require a non-trivial type-level computation to represent the focusing. I.e., given the mailbox sequence type, calculate ($\rightsquigarrow$) the type of all possible ways to "punch holes" ($[\,\cdot\,]$) into the mailbox sequence:

$$M[A_1] \times \cdots \times M[A_n] \rightsquigarrow (M[A_1] \times \cdots \times M[A_{i-1}] \times [\,\cdot\,] \times M[A_{i+1}] \times \cdots \times M[A_n])_{1 \le i \le n},$$

A zipper is a data structure for efficient traversal and updates of functional data structures, e.g., lists, trees, etc. It decomposes the data into a focus point and its neighborhood. For uniformly parametric data types, zippers are easy to define and work with. The heterogeneity of the mailbox types makes zippers unwieldy, because the zipper structure has to be mirrored at the type level. For n mailboxes the heterogeneous zipper has a type description of size $O(n^2)$.

where the choice of "hole punching" and eventual "plugging" is at the discretion of the external environment. This type is effectively the zipper [Hue97] over the mailbox sequence type. Nevertheless, this problem is worthwhile solving, because it is of general interest for asynchrony and concurrency implementations in statically-typed sequential languages. For instance, Paykin, Krishnaswami, and Zdancewic [PKZ16] investigate in their work on event-driven programming how to statically calculate zippers of data types by differentiation. In this work, we propose a practical technique based on effect handling to achieve this goal.

SAFE, REUSABLE, MODULAR RESTRICTION HANDLERS Is important that users should not be able to specify ill-defined restrictions in a join pattern. E.g., providing a De Bruijn index that refers to a non-existent position in the context or supplying a K which is not a subset of $\{1 \ldots n\}$ in the case of `aligning` K. It is also important that the implementer of restrictions can avoid repetition and provide statically safe, yet maximally reusable definitions. E.g., if the `aligning` K value is compatible in the context of an n-way join, then it should be compatible for an $(n+1)$-way join, too. Finally, to foster extensibility and modularity, new kinds of restriction handlers should be programmable separately and independently from concrete join computations.

5.3 EXTENDING POLYJOIN

We extend core PolyJoin from Section 4.3 with metadata and contextual extensions to provide a type-safe embedding of the Cartesius event correlation patterns. Figure 5.4 defines the extended PolyJoin rules, highlighting the changes and additions relative to the version in Figure 4.2.

5.3.1 *Metadata*

The abstract type Meta, represents metadata carried by event values from event sources. E.g., the occurrence times of events (as in Cartesius) or geolocation coordinates as in EventJava [EJ09]. We expose metadata for each variable in the join pattern body, by changing the previous context shape translation judgment accordingly (rules (TZ) and (TS)). Metadata representations can be merged using the binary operator $\sqcup$ (rule (MMERGE)). In the HOAS variable representation, end users can reuse OCaml's deep pattern matching on function parameters to decompose the bound event variables into value and metadatum, e.g., Figure 5.2, Line 3.

A tagless interpreter requires a policy for computing the metadata of joined events from the metadata of input events. However, we do not expose merging explicitly in the pattern syntax. E.g., the pattern body in Figure 5.2 does not mention the output event's metadatum in the `yield` form. This design reduces syntactic noise in the pattern and prevents users from incorrectly merging the metadata for the resulting event, potentially violating invariants of the underlying system.

While merging is implicit in the end user pattern syntax, it should be explicitly stated in the syntax type signature, to obligate the tagless interpreter to provide an implementation for merging. We modify the pattern formation rules accordingly: The pattern formation judgment $\vdash^{\vec{C}}_{\text{pat}} P : A$ now states that P is a pattern yielding A events and requires a metadata context derived from the shape $\vec{C}$. By unification, the requirement $\vec{C}$ will be matched to the shape $\vec{A}$ of the pattern context in the (JOIN) rule.

5.3.2 *Contextual Extensions*

In order to support restriction handlers in the pattern syntax, we introduce an abstract syntax for contextual extensions $\vdash^{\vec{A}}_{\text{ext}} X$, meaning that X is an injectable extension, which depends on/can only be used in contexts of shape $\vec{A}$. This is to ensure that restriction handlers as in Figure 5.2 cannot refer to non-existent positions in the context of bindings. Furthermore, we assume contextual extensions are monoids, having an empty extension (rule (EEMPTY)) and a merge operation (rule (EMERGE)), just as the restriction handlers in CARTESIUS.

5.3.3 *OCaml Representation*

The extensions to POLYJOIN translate straightforwardly to an OCaml module signature, in the same manner as before (Section 4.4). Figure 5.6 shows the new `Symantics` signature. As before, judgment forms correspond to abstract types and type constructors and their type parameters to the metavariables. The function `mmerge` corresponds to the metadata merge operator $\sqcup$ in Figure 5.4. In example code, we will continue to use the $\sqcup$ symbol for readability.

5.3.4 *Predicates on Metadata*

The extended version of POLYJOIN enables new predicates (i.e., boolean expressions in patterns) over metadata, for which it provisions the abstract type Meta. We can easily add new syntax forms for time-based predicates in the tagless final encoding, without changes to existing tagless interpreters for other language features. We show an extension of the language in Figure 5.7, in the `SymanticsPlus` signature. Recall from Section 3.3.5 that in CARTESIUS, events carry time intervals (pairs of time stamps) as metadata. We encode this in OCaml with

```
type meta = time * time
```

where the type `time` is a numeric type for time stamps, having a total order. Accordingly, `SymanticsPlus` defines the syntax signatures for operations on the metadata and time at the level of expressions (`repr`). We marked their purpose in the comments.

These expressions enable a definition of the `within` constraint in our running fire alarm example (Figure 4.1b). The constraint is definable as "derived syntax" (i.e., function abstraction) in the tagless DSL. The functor `DerivedSyntax`, which depends on `SymanticsPlus` instances, shows the definition: We compare the merged intervals (their least upper bound) against the given time span.

EXPRESSIONS AND PATTERNS $\boxed{\Gamma \vdash_{\text{exp}} M : A}$ $\boxed{\Gamma \vdash^{\vec{C}}_{\text{pat}} P : A}$

$$\frac{\Gamma(x) = A}{\Gamma \vdash_{\text{exp}} x : A}\ (\textsc{var}) \qquad \frac{\Gamma \vdash_{\text{exp}} M : \mathsf{Bool} \qquad \Gamma \vdash^{\vec{C}}_{\text{pat}} P : A}{\Gamma \vdash^{\vec{C}}_{\text{pat}} \texttt{where}\ M\ P : A}\ (\textsc{where}) \qquad \frac{\Gamma \vdash_{\text{exp}} M : A}{\Gamma \vdash^{\vec{C}}_{\text{pat}} \texttt{yield}\ M : A}\ (\textsc{yield})$$

$$\frac{\Gamma \vdash_{\text{exp}} M : \mathsf{Meta} \qquad \Gamma \vdash_{\text{exp}} N : \mathsf{Meta}}{\Gamma \vdash_{\text{exp}} M \sqcup N : \mathsf{Meta}}\ (\textsc{mmerge})$$

$$\frac{\vdash_{\text{ctx}} \Pi : \vec{A} \qquad \vec{A} \triangleright \vec{B} \qquad \Gamma \vdash^{\vec{A}}_{\text{ext}} X \qquad \Gamma, \overrightarrow{x : B} \vdash^{\vec{A}}_{\text{pat}} P : C}{\Gamma \vdash_{\text{exp}} \texttt{join}\ \Pi\ X\ (\vec{x}.P) : \mathsf{Shape}[C]}\ (\textsc{join})$$

CONTEXTUAL EXTENSIONS $\boxed{\Gamma \vdash^{\vec{A}}_{\text{ext}} X}$

$$\Gamma \vdash^{\vec{A}}_{\text{ext}} \texttt{enil}\ (\textsc{eempty}) \qquad \frac{\Gamma \vdash^{\vec{A}}_{\text{ext}} X \qquad \Gamma \vdash^{\vec{A}}_{\text{ext}} Y}{\Gamma \vdash^{\vec{A}}_{\text{ext}} X\ \texttt{|++|}\ Y}\ (\textsc{emerge})$$

CONTEXT FORMATION $\boxed{\Gamma \vdash_{\text{var}} V : A}$ $\boxed{\Gamma \vdash_{\text{ctx}} \Pi : \vec{A}}$

$$\frac{\Gamma \vdash_{\text{exp}} M : \mathsf{Shape}[A]}{\Gamma \vdash_{\text{var}} \texttt{from}\ M : A}\ (\textsc{from}) \qquad \Gamma \vdash_{\text{ctx}} \texttt{cnil} : \varnothing\ (\textsc{cnil}) \qquad \frac{\Gamma \vdash_{\text{var}} V : B \qquad \Gamma \vdash_{\text{ctx}} \Pi : \vec{A}}{\Gamma \vdash_{\text{ctx}} V\ @.\ \Pi : B, \vec{A}}\ (\textsc{cat})$$

CONTEXT SHAPE TRANSLATION $\boxed{\vec{A} \triangleright \vec{B}}$

$$\varnothing \triangleright \varnothing\ (\textsc{vz}) \qquad \frac{\vec{A} \triangleright \vec{B}}{C, \vec{A} \triangleright (C \times \mathsf{Meta}), \vec{B}}\ (\textsc{ts})$$

Figure 5.4: PolyJoin with Metadata and Contextual Extensions (Changes Highlighted).

CONTEXT SHAPE (MULTI)PROJECTION $\boxed{\vdash n : A \in \vec{B}}$ $\boxed{\vdash \vec{n} : \vec{A} \sqsubseteq \vec{B}}$

$$\vdash \texttt{pz} : A \in A, \vec{B}\ (\textsc{pz}) \qquad \frac{\vdash n : A \in \vec{B}}{\vdash \texttt{ps}\ n : A \in C, \vec{B}}\ (\textsc{ps}) \qquad \vdash \langle\rangle : \varnothing \sqsubseteq \vec{A}\ (\textsc{mz}) \qquad \frac{\vdash n : C \in \vec{B} \qquad \vdash \vec{m} : \vec{A} \sqsubseteq \vec{B}}{\vdash n, \vec{m} : C, \vec{A} \sqsubseteq \vec{B}}\ (\textsc{ms})$$

Figure 5.5: Projection for DeBruijn Indices and Sets of Indices.

5.4 METATHEORY

Our extensions to core PolyJoin with contextual extension and implicit metadata merging (Figure 5.4) require changes to the syntax signature of the existing `join` form and the pattern formation judgment. The latter is now indexed by the shape $\vec{A}$ of the pattern variable context, to ascertain that (1) the pattern is compatible with the usage context $\vec{A}$ and (2) requires compatible *implicit bindings* of the event metadata. Of course, to accommodate our syntax design from Chapter 3, `join` now accepts contextual restrictions X for specifying restrictions.

How do these changes affect the metatheory we established for core PolyJoin in the previous chapter? The answer is: not much. The changes are benign, so that the metatheory from the previous chapter continues to hold, with appropriate adaptations of the type and term translations and the theorems. The additional shape index in the pattern judgment is a pure type-level marker and has no footprint on the syntax (i.e., it is a phantom type à la Leijen and Meijer [LM99]). We elide the specification of the straightforward, but tedious revision of type and term translation.

THEOREM 5.1 (TYPE PRESERVATION OF EXTENDED POLYJOIN)
Let `S` *be a module implementing the* `Symantics` *signature in Figure 5.6. Then all of the following hold:*

- *If* $\Gamma \vdash_{\text{exp}} M : A$, *then* $\tau^{\mathtt{S}}(\Gamma) \vdash \mathtt{m} : \tau^{\mathtt{S}}(A)$ `S.repr` *in OCaml.*
- *If* $\Gamma \vdash^{\vec{C}}_{\text{pat}} P : A$, *then* $\tau^{\mathtt{S}}(\Gamma) \vdash \mathtt{p} : (\ \tau^{\mathtt{S}}(\vec{C})\ , \tau^{\mathtt{S}}(A))$ `S.pat` *in OCaml.*
- *If* $\Gamma \vdash_{\text{ctx}} \Pi : \vec{A},\ \vec{A} \triangleright \vec{B}$, *then* $\tau^{\mathtt{S}}(\Gamma) \vdash \mathtt{pi} : (\tau^{\mathtt{S}}(\vec{A}), \tau^{\mathtt{S}}_{\mathtt{repr}}(\vec{B}))$ `S.ctx` *in OCaml.*
- *If* $\Gamma \vdash_{\text{var}} V : A$, *then* $\tau^{\mathtt{S}}(\Gamma) \vdash \mathtt{v} : \tau^{\mathtt{S}}(A)$ `S.var` *in OCaml.*
- *If* $\Gamma \vdash^{\vec{C}}_{\text{ext}} X$, *then* $\tau^{\mathtt{S}}(\Gamma) \vdash \mathtt{xt} : \tau^{\mathtt{S}}(\vec{C})$ `S.ext` *in OCaml.*

Where the OCaml terms `m`, `p`, `m`, `v`, `xt` *. are determined by the type and term translation rules.* ♦

PROOF In the a similar fashion to the previous chapter, a revised version of the type and term translation rules witnesses the homomorphism. □

LEMMA 5.2 (JOIN INVERSION IN EXTENDED POLYJOIN)
If $\Gamma \vdash_{\text{exp}} \mathtt{join}\ \Pi\ X\ (\vec{x}.P) : A$, *then*

1. $\Gamma \vdash_{\text{ctx}} \Pi : A_1, \ldots, A_n$.
2. $(A_1, \ldots, A_n) \triangleright ((A_1, \mathsf{Meta}) \ldots, (A_n, \mathsf{Meta}))$.
3. $\Gamma \vdash^{A_1,\ldots,A_n}_{\text{ext}} X$.
4. $\Gamma, x_1 : (A_1, \mathsf{Meta}), \ldots, x_n : (A_n, \mathsf{Meta}) \vdash^{A_1,\ldots,A_n}_{\text{pat}} P : B$, *for some type* B.
5. $A = \mathsf{Shape}[B]$, *for the same* B *above.* ♦

PROOF Any derivation of $\Gamma \vdash_{\text{exp}} \texttt{join}\ \Pi\ X\ (\vec{x}.P) : A$ must end with rule (JOIN) (Figure 5.4), since no other rule matches. The lemma follows immediately from the premises and inversion on the context shape translation. □

LEMMA 5.3 (POLYVARIADICITY OF THE EXTENDED JOIN SYNTAX) *For all $n \in \mathbb{N}$, all types $A_1, \ldots, A_n$, all variables $x_1, \ldots, x_n$, $y_1, \ldots, y_n$, all extensions X, all patterns P, and all types B such that $\Gamma \vdash^{A_1,\ldots,A_n}_{\text{ext}} X$, and*

$$y_1 : (A_1, \texttt{Meta}), \ldots, y_n : (A_n, \texttt{Meta}) \vdash^{A_1,\ldots,A_n}_{\text{pat}} P : B,$$

let

$$\Gamma = x_1 : \text{Shape}[A_1], \ldots, x_n : \text{Shape}[A_n]$$

and

$$\Pi = (\texttt{from}\ x_1\ @.\ \cdots\ @.\ \texttt{from}\ x_n\ @.\ \texttt{cnil}),$$

then

$$\Gamma \vdash_{\text{exp}} \texttt{join}\ \Pi\ X\ (y_1, \ldots, y_n.P) : \text{Shape}[B].$$ ⧫

PROOF It is straightforward to construct a derivation of $\Gamma \vdash_{\text{ctx}} \Pi : \vec{A}$ and $\vec{A} \triangleright (A, \vec{\text{Meta}})$. Together with the assumed derivations, we obtain a derivation of the goal by application of rule (JOIN). □

We obtain an analogue of the Polyvariadicity Theorem 4.8 for OCaml, this time with metadata and extensions:

THEOREM 5.4 (POLYVARIADICITY OF THE EXTENDED OCAML JOIN SYNTAX) *In the* `Symantics` *module signature (Figure 5.6)*

```
join: ('a,'b) ctx -> 'a ext -> ('b -> ('a,'c) pat) -> 'c shape repr
```

corresponds to a polyvariadic function signature*, i.e., a family* `f` *of signatures*

```
f: (t1 * ··· * tn, (t1 repr * meta repr) * ··· * (tn repr * meta repr)) ctx
     -> (t1 * ··· * tn) ext
     -> ((t1 repr * meta repr) * ··· * (tn repr * meta repr)
           -> (t1 * ··· * tn, u) pat)
     -> u shape repr
```

indexed by all $n \in \mathbb{N}$, all n-vectors of types $\texttt{t}_1 \cdots \texttt{t}_n$ and all types u. That is, in all OCaml contexts `S:Symantics, open S`*:*

1. *Every* `join` *expression accepted by the OCaml type checker receives type parameter instantiations of the form*
   - `'a = t1 * ··· * tn`
   - `'b = (t1 repr * meta repr) * ··· * (tn repr * meta repr)`
   - `'c = u`

   for some $n \in \mathbb{N}$ and types $\texttt{t}_1 \cdots \texttt{t}_n$ and `u`*.*
2. *For any given $n \in \mathbb{N}$, and any n-vector of types $\texttt{t}_1 \cdots \texttt{t}_n$, the OCaml type checker accepts the partial application*

```
let exp = fun (x1 : t1 shape repr)⋯(xn : tn shape repr) ->
  join (from x1 @.⋯@. from x1 @. cnil)

(* exp: t1 shape repr -> ⋯ -> tn shape repr ->
        (t1 * ⋯ * tn) ext ->
        ((t1 repr * meta repr) * ⋯ * (tn repr * meta repr)
             -> (t1 * ⋯ * tn, 'c) pat)
         -> 'c shape repr *)
```

with the type parameter instantiation

- `'a` = $\texttt{t}_1 * \cdots * \texttt{t}_n$
- `'b` = $(\texttt{t}_1\ \texttt{repr} * \texttt{meta repr}) * \cdots * (\texttt{t}_n\ \texttt{repr} * \texttt{meta repr})$
- `'c` = `'c`

♦

PROOF

1. By application of the Type Preservation Theorem 5.1 to the Join Inversion Lemma 5.2.
2. By application of the Type Preservation Theorem 5.1 to the Polyvariadicity Lemma 5.3. □

By virtue of Theorem 5.4, the parametricity reading of the `join` signature from Section 4.5.2 continues to hold in the extended version of PolyJoin. That is, tagless interpreters join inputs from selections of n-tuples and may freely choose the variable binding strategy in the pattern.

TYPE SOUNDNESS An identical version to the Type Soundness Theorem 4.4 from the previous chapter continues to hold in the extended version of PolyJoin and is proven in exactly the same manner. Note that because we have chosen multicore OCaml as the host language, it may happen that well-typed expressions terminate with unhandled effect invocations, just like in the exceptions case.

5.5 EMBEDDING THE CORE OF CARTESIUS WITH POLYJOIN

We provide a high-level outline of our PolyJoin tagless interpreter for Cartesius, in the multicore OCaml dialect. The interpreter interprets join patterns as multicore OCaml code.

5.5.1 *Polyvariadic Effect Declarations*

We obtain polyvariadic effect declarations by representing heterogeneous sequences of effect *declarations* in the language of *types and expressions*.

First, we represent single effects as expressions/values. We use the same folklore encoding of *generative* or *first-class effects*, which we already utilized in our non-polyvariadic prototype (Section 3.5). To recap, Figure 5.8 shows the first-class effect encoding in OCaml. A first-class effect is represented by a first-class module instance of the signature `SLOT`, carrying effect declarations. Their signature depends on the abstract **`type`** `t`, which represents the element type of

```
module type Symantics = (* ... *)
  type 'a shape    (* Shape[·] Constructor *)
  type meta        (* Meta Type *)
  (* Judgments (cf. Figure 5.4): *)
  type 'a repr     (* ⊢_exp · : A *)
  type ('c,'a) pat (* ⊢^C_pat · : A *)
  type ('a,'b) ctx (* combination of ⊢_ctx · : A and A ▷ B *)
  type 'a var      (* ⊢_var · : A *)
  type 'a ext      (* ⊢^A_ext · *)
  (* Context Formation and Shape Translation: *)
  val from: 'a shape repr -> 'a var
  val cnil: (unit,unit) ctx
  val (@.): 'a var -> ('c, 'd) ctx -> ('a * 'c, ('a repr * meta repr) * 'd) ctx
  (* Contextual Extensions: *)
  val enil: unit -> 'a ext
  val (|++|): 'a ext -> 'a ext -> 'a ext
  (* Expressions and Patterns: *)
  val mmerge: meta repr -> meta repr -> meta repr (* Metadata merge (⊔) *)
  val yield: 'a repr -> ('c,'a) pat
  val where: bool repr -> ('c,'a) pat -> ('c,'a) pat
  val join: ('a,'b) ctx -> 'a ext -> ('b -> ('a,'c) pat) -> 'c shape repr
end
```

Figure 5.6: Tagless Final Representation of Extended PolyJoin.

```
module type SymanticsPlus = sig
  include Symantics
  (* greatest element, positive infinity *)
  val infty:   time repr
  (* least element, negative infinity *)
  val ninfty:  time repr
  (* comparison *)
  val (%<=):   time repr -> time repr -> bool repr
  (* time span *)
  val span:    meta repr -> time repr
  (* representation of minutes *)
  val minutes: float -> time repr
end
module DerivedSyntax(S:SymanticsPlus) = struct
  open S
  let within: meta repr -> meta repr -> time repr -> bool repr =
    fun m1 m2 mb -> (span (m1 ⊔ m2)) %<= mb
end
```

Figure 5.7: Syntax Extensions for Time Predicates in OCaml.

```
module type SLOT = sig
  type t
  effect Push: t -> unit
  effect Get:  unit -> t mailbox
  effect Set:  t mailbox -> unit
end
type 'a slot =
  (module SLOT with type t = 'a)
```

```
module type YF = sig
  type t
  effect Yield: t -> unit
  effect Fail:  unit -> 'a
end
type 'a yieldfail =
  (module YF with type t = 'a)
```

Figure 5.8: Generative Effects from Modules in Multicore OCaml.

```
(* type a b. a Slots.hlist -> b handler *)
let memory slots = poly_handler slots (fun (s: (module SLOT)) ->
  let module S = (val s) in
  let mbox: S.t mailbox ref = ref (mailbox ()) in
    fun action -> try action () with
      | effect (S.Get ()) k -> continue k !mbox
      | effect (S.Set m)  k -> mbox := m; continue k ())
```

Figure 5.9: Example Polyvariadic State Handler.

```
let where: bool repr -> ('c,'a) pat -> ('c,'a) pat =
  fun cond body meta yf ->
    if cond then (body meta yf) else fail_with yf
```

```
let yield: 'a repr -> ('c,'a) pat =
  fun result meta yf -> (result, (merge_all meta))
```

Figure 5.10: CARTESIUS Pattern Implementation in Context/Capability-Passing Style.

```
let join: type s a b. (s, a) ctx -> s ext -> (a -> (s, b) pat) -> b shape repr
  = fun ctx extension pattern_body ->
      (* (effect Push_i: s_i -> unit)_{1≤i≤n}: *)
      let slots: s Slots.hlist = gen_slots_from ctx in
      (*  effect Yield: b -> unit and effect Fail: unit -> 'c: *)
      let yf: b yieldfail = gen_yieldfail () in
      (* instantiate and run bottom-right corner of Figure 5.1 *)
      let pstreams = parallel_bind ctx slots in
      let h⊗       = gen_default_handler slots yf pattern_body in
      let h_user   = extension ctx in
      let h_sys    = gen_sys_handler yf in
      Async.spawn (h_sys |+| h⊗ |+| h_user) pstreams
```

Figure 5.11: Implementation of CARTESIUS Join Patterns.

an input event source. A slot module carries an instance-specific `Push` effect declaration, as well as `Set` and `Get` effects for retrieving/updating a mailbox of local observations (Section 5.2.1). Similarly, we encapsulate the `Yield` and `Fail` effects (cf. Figure 5.1) in the `YF` module.

A sequence of effect declarations $(\texttt{Push: t}_i\texttt{ -> unit})_{1 \le i \le n}$ turns into a heterogeneous list $(\texttt{t}_1 * \cdots * \texttt{t}_n)$ `Slots.hlist` of first-class modules.

First-class effect instances are capability values that programmers can pass around and manipulate. E.g., Lines 7 and 8 of the left definition define the

```
'a slot
```

capability type. We use the **`with type`** syntax to equate the type variable `'a` with the abstract type `t` in the respective module. In this way, the abstract effect type becomes visible in the language of types.

Finally, we represent heterogeneous sequences of effect declarations by heterogeneous lists of first-class effects values:

```
module Slots = HList(struct type 'a t = 'a slot end)
```

Shape preservation (Section 4.6) ensures that the effect types are consistent with the input and variable types of the join computation. That is, if an n-way join computation has shape `'a`, then a value of type `'a Slots.hlist` carries n capabilities to push events, each matching the corresponding input source's type.

5.5.2 *Heterogeneous Effect Handling*

The first-class effect encoding enables heterogeneous handlers. We represent them by ordinary handlers plus context, accepting the first-class `slot` effects:

```
type 'a handler  = (unit -> 'a) -> 'a
type 'ctx ext    = 'ctx Slots.hlist -> unit handler
let (|++|) h1 h2 = fun ctx -> (h1 ctx) |+| (h2 ctx)
```

Values of type `'ctx ext` may calculate a handler depending on these effects. Since we model asynchronous processes that do not return, we let the handlers have the `unit` return type. For example, the function

```
1  let print_pushes: type a. (int * (string * a)) ext =
2   fun slots ->
3     module IntS = (val (head slots)) in
4     module StrS = (val (head (tail slots))) in
5     fun action -> try action () with
6     | effect (StrS.Push s) k ->
7         println(s);
8         continue k ()
9     | effect (IntS.Push i) k ->
10        println(int_to_str(i));
11        continue k ()
```

calculates a handler that prints integer- and string-valued push-notifications to the console, by accessing the respective first-class effect instances and handling their `Push` effects. The point is that we can simultaneously handle heterogeneous instantiations of these effects in a type-correct way within one handler. I.e., the `print_pushes` handler handles both a string- and integer-valued instance of `Push`. The variants can be discerned by assigning the first-class effects to a local module declaration (Lines 2-3), and then qualify via the module's name the intended effect declaration (Lines 5-6).

```
1 let poly_handler: type a b. a Slots.hlist -> ((module SLOT) -> b handler) -> b
      handler =
2     fun slots f ->
3       (* Turn slots into list of existentials *)
4       let e_slots: (module SLOT) list = abstract slots in
5       let handlers = List.map f e_slots in
6       List.fold_right (|+|) handlers
```

Figure 5.12: Context-polymorphic Effect Handler Calculation from Generating Functions.

5.5.3 *Context Polymorphism*

Moreover, the `print_pushes` handler above is safely applicable in infinitely many contexts, because it is parametric over all context shapes

```
(int * (string * a))
```

for all `a`. We call this trait *context polymorphism*. Note how the function's type signature

```
(int * (string * a)) ext =
  (int * (string * a)) Slots.hlist -> unit handler
```

guarantees that it is usable in all context shapes having *at least* two inputs sources with element type `int` and `string`.

More generally, we can define polyvariadic handlers, by calculations over heterogeneous lists. E.g., Figure 5.9 shows the polyvariadic handler `memory`. This handler is part of $h_{\otimes}$ in Figure 5.1 and handles the mailbox effects. That is, it maintains n distinct mailbox states in reference cells (Line 4) and handles n distinct `Set` and `Get` effects, by reading/writing the corresponding mailbox.

We calculate polyvariadic handlers via the combinator `poly_handler` (Figure 5.12). It is essentially mapping the supplied function over the n first-class effects and then composing the resulting handlers into a single handler.

The general takeaway from the example is that tracking the context *shape* $\vec{A}$ (respectively `'a` in OCaml) induced by the pattern context formation (Figure 5.4) at the type level enables context polymorphic, polyvariadic components and their composition. These components can then be used in a type-safe manner for an entire polyvariadic backend implementation/tagless interpreter having PolyJoin's polyvariadic frontend syntax.

5.5.4 *Implicit Time Data in Patterns*

We let the syntactic sort of patterns denote functions types as one possible representation of context dependence, i.e.,

```
type ('c,'a) pat = 'c Meta.hlist -> 'a yieldfail -> 'a repr *
    meta repr
```

accepting a heterogeneous list of metadata, i.e.,

```
module Meta = HList(struct type 'a t = meta repr end)
```

and a capability to yield and fail (cf. Figure 5.8, right). The end result is a pair of output event together with its metadatum, which is derived from the metadata of the input events.

Metadata and capabilities must be explicitly threaded by the tagless interpreter of the pattern syntax. Their essential uses occur in the implementations of `where` and `yield` (Figure 5.10). In the case of `where`, if all the constraints are satisfied in `cond`, we continue with the body, passing down the context. Otherwise we cancel the current pattern match attempt by invoking the `Fail` effect from the given capability `yf`. The failure invocation is abstracted in the function `fail_with yf`. In the case of `yield`, we return the given result and merge the implicit metadata of the input with `merge_all: 'a Meta.hlist -> meta`.

The point is that the function-based pattern type interpretation forces a tagless interpreter to supply a value for implicitly merging metadata (e.g., extracted from the n tuple of input events) before a join pattern can produce an event.

5.5.5 *Join Pattern Implementation*

Figure 5.11 shows the implementation of the join pattern form. The code is a direct translation of the diagram in Figure 5.1. First, it allocates first-class effect instances and then continues setting up the concurrent join computation. The last line corresponds to executing the bottom-right corner of Figure 5.1 in an asynchronous thread. The point is that the join signature in PolyJoin indeed supports non-sequential event binding, since this is a concurrent join computation reacting to events as they come.

5.5.6 *Solving the Focusing Continuations Problem*

Defining a polyvariadic version of Cartesius' callback logic (Sections 5.2.1, 5.2.3 and Figure 5.3) was the most challenging part of the implementation. Fortunately, the tools we have so far enable a simple solution (Figure 5.13), which does not require type-level zippers. We exploit that effect handling is a form of dynamic overloading.

The combinator `callback` represents *all* the callbacks $(\kappa_i)_{1\leq i\leq n}$ from Figure 5.3. It is a polyvariadic handler for the $\texttt{Push}_i$ effects, so that the effect handling clause of the i-th effect implements callback κ_i (Lines 5-9). For an event notification `x`, we first retrieve the mailboxes of the join computation, replacing the i-th position by `x`, via the `focus` combinator. Then, we compute the cartesian product over these mailboxes, passing all the n-tuples to a `consumer` function. This parameter represents the part of the system that tests tuples against the join pattern, yielding or failing.

The `focus` combinator is at the core of the solution, exploiting a synergy with effect handlers, to obtain a type-safe focus and replacement into the heterogeneous list of mailboxes (modeled by the type `a Mail.hlist`). Lines 6-7 of `focus` codify the approach. We retrieve all mailboxes by invoking all of the n `Get` effects polyvariadically (function `get_all`). To focus into the given position, we *locally* handle the i-th `Get` effect and *override* its meaning, i.e., we answer the effect invocation for the focused position by passing the supplied event `x` in a mailbox. For the non-focused positions, `focus` does not handle the `Get` invocations, instead delegating the handling to the calling context. We assume that the `memory` handler (Figure 5.9) is in the context, to define the default semantics.

```
(* Callback logic *)
let callback slots consumer = poly_handler slots (fun (si: (module SLOT)) ->
  let module Si = (val si) in
  fun action -> try action () with
  | effect (Si.Push x) k ->
      (* m_1, ..., m_{i-1}, ⟨x⟩, m_{i+1}, ..., m_n *)
      let mboxes = focus slots s x in
      (* m_1 ⊗ ··· ⊗ m_{i-1} ⊗ ⟨x⟩ ⊗ m_{i+1} ⊗ ··· ⊗ m_n *)
      crossproduct mboxes consumer; continue k ()

(* Focusing into m_1, ..., m_n *)
let focus: type a b. a Slots.hlist -> b slot -> b ev -> a Mail.hlist =
  fun slots si x ->
    let module Si = (val si) in
    (* Replace i-th position by handling *)
    try get_all slots () with
    | effect (Si.Get ()) k -> continue k (mailbox x)

(* Bulk retrieval of all mailboxes m_1, ..., m_n *)
let get_all: type a. a Slots.hlist -> unit -> a Mail.hlist = fun slots () ->
  let module M = HMap(Slots)(Mail) in
    M.map { M.f = fun (s': (module SLOT)) ->
      let module S' = (val s') in perform (S'.Get ()) } slots
```

Figure 5.13: Multicore OCaml Solution to the Focusing Continuations Problem.

Finally, Figure 5.14 summarizes our focusing solution graphically. Each box represents a stack frame of the dynamic execution context, with the bottom-most being currently executed. We effectively implement *dynamic binding*, using the deep binding strategy [Mor98], in terms of effects (dynamic variables) and stacking handlers (dynamic variable bindings).

5.6 SAFE, REUSABLE AND MODULAR RESTRICTION HANDLERS

Here, we solve the challenge of writing restriction handlers generically (Section 5.2.3). By the design of CARTESIUS it should be enough to know about the `Push`, `Get` and `Set` effect interfaces in order to write restrictions. These interfaces are a natural abstraction barrier, to not burden programmers with the details of the underlying join implementation. We show a simple solution for programming restriction handlers that is both *context-polymorphic* and *position-polymorphic*, so that one definition is compatible with join instances of different arity and we have fine-grained control where to apply restrictions.

5.6.1 *Type-Safe Pointers/De Bruijn Indices*

Recall that we apply restriction handlers to specific positions via De Bruijn indices (e.g., Figure 5.2, Line 2). Formally, a De Bruijn index corresponds to a derivation of the shape projection judgment $\vdash n : A \in \vec{B}$ having rules (PZ) and (PZ) (Figure 5.5). The judgment asserts: the index number n proves that type

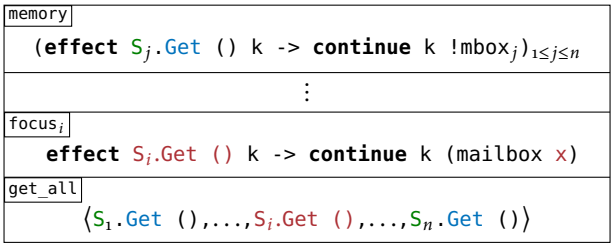


Figure 5.14: Illustration of Dynamic Execution Context during Focusing.

```
(* ⊢ n : A ∈ B⃗ (Figure 5.4) *)
type (_,_) _ptr =
  | Pz: ('a, 'a * 'b) _ptr
  | Ps: ('a, 'b) _ptr
          -> ('a, 'c * 'b) _ptr
type ('a,'b) ptr =
  unit -> ('a,'b) _ptr
```

```
(* ⊢ n⃗ : A⃗ ⊑ B⃗ (Figure 5.4) *)
type (_,'a) _mptr =
  | Mz: (unit, 'a) _mptr
  | Ms: (('c, 'b) _ptr * ('a, 'b) _mptr)
           -> ('c * 'a, 'b) _mptr
type ('a,'b) mptr =
  unit -> ('a,'b) _mptr
let mz ()      = Mz
let ms p ps () = Mn (p (), ps ())
```

Figure 5.15: GADTs for Type-safe Pointers and Sets of Pointers.

```
let p0 () = Pz       (* ('a, 'a * 'b) ptr *)
let p1 () = Ps Pz    (* ('a, 'b * ('a * 'c)) ptr *)
let p2 () = Ps Ps Pz (* ('a, 'b * ('c * ('a * 'd))) ptr *)
let ps () =          (* ('a * ('b * unit), 'b * ('c * ('a * 'd))) mptr *)
  (ms p2 @@ ms p0 @@ mz) ()
```

Figure 5.16: Example: Type-safe Pointers and Sets of Pointers in OCaml.

```
module HPtr(H: Hl) = struct
  let rec proj: type a ctx. (a, ctx) _ptr -> ctx H.hlist -> a H.el =
    fun n hlist -> match n, hlist with
      | Pz,   H.S (hd, _) -> hd
      | Ps n, H.S (_, tl) -> proj n tl

  let rec mproj: type xs ctx. (xs, ctx) _mptr -> ctx H.hlist -> xs H.hlist =
    fun mptr hlist -> match mptr with
      | Mz         -> H.nil
      | Ms (i, ps) -> H.cons (proj i hlist) (mproj ps hlist)
end
```

Figure 5.17: Type-safe Element Access.

A is contained in the heterogeneous sequence $\vec{B}$. The rules of shape projection straightforwardly translate to a binary GADT `_ptr` (Figure 5.15, left), which is well-known by functional programmers. This leads to the type `ptr` of *type-safe pointers* into heterogeneous contexts, which communicate *context requirements* at the type-level. We show examples of type-safe pointers along with their types in Figure 5.16. The pointer type relates to the formal system in Figure 5.4 in that we let the type of a pointer value n represent a polymorphic *judgment scheme*, from which all derivable ground judgments $\vdash n : A \in \vec{B}$ can be instantiated. We can interpret the scheme as a *context requirement*. E.g., the polymorphic type[4] of the pointer `p1` certifies that it points at the second element `'a` (highlighted in red) of contexts *having at least two elements*. This is due to the context shape `'b * ('a * 'c)` being polymorphic in the tail `'c`.

By induction over the shape projection derivation, we can leverage these type-level assertions to define a type-safe projection function `proj` on heterogeneous lists, where projecting the i-th element always succeeds (Figure 5.17). I.e., if `(i,a) _ptr` holds, then any `hlist` of shape `a` contains an `i`-typed element, which can be retrieved. The definition we show is contained in a functor `HPtr`, to make the projection function parametric over all the "uniformly heterogeneous" lists (Section 4.6).

5.6.2 *Sets of Type-Safe Pointers*

On top of shape projection, we define the shape multiprojection judgment $\vdash \vec{n} : \vec{A} \sqsubseteq \vec{B}$ (Figure 5.4), which is a straightforward extension. It reads: the sequence $\vec{n}$ of De Bruijn indices points at multiple positions having types $\vec{A}$ within a heterogeneous sequence $\vec{B}$. The corresponding OCaml GADT `_mptr` is defined in Figure 5.15 and enables sets of type-safe pointers (type `mptr`). Similarly to the type-safe pointers, the sets describe polymorphic judgment schemes for context requirements. For example, the set of pointers `ps` in Figure 5.16 points into the third and first positions, and requires a context of at least three elements. Finally, we define a type-safe multiprojection function `mproj` (Figure 5.17), which guarantees that projecting the pointed-at positions in the pointer set yields the heterogeneous list of projected values.

5.6.3 *Position Polymorphism for Restriction Handlers*

Being parametric over type-safe pointers/sets enables library writers to impose more refined constraints on context-polymorphic functions (cf. Section 5.5.3). We call this refinement *position polymorphism*, and it enables generic restriction handlers for CARTESIUS. For instance, we define the `most_recently` restriction in Figure 5.18. This restriction changes the semantics of the i-th pattern variable, by truncating past event observations. I.e., it handles the i-th `Push` effect of the join and overwrites the contents of the i-th mailbox with just the current event notification (Lines 6-9). Accessing the required i-th first-class effect instance is just a matter of projecting it from the available instances, using the given pointer (Lines 3-5). The type-safe pointers guarantee that access always succeeds. Analogously, restriction handlers on sets, e.g., `aligning` (cf. Section 5.2.2), are functions parametric over type-safe pointer sets

4 The distinction between `_ptr` and `ptr` is necessary, due to the value restriction in ML/OCaml [Wri95]. The pointers must be functions in order to fully generalize the type parameters.

```
let most_recently: type ctx i. (i, ctx) ptr -> ctx ext = fun ptr slots ->
  (* Type-safe pointers and projection on first-class effects *)
  let module SPtr = HPtr(Slots) in
  (* Project the given first-class effect declaration *)
  let module Si = (val (SPtr.proj (ptr ()) slots)) in
  fun action -> try action () with
    | effect (Si.Push x) k ->
        (* Truncate the mailbox to the most recent value *)
        perform Si.Set (mailbox x); continue k (perform Si.Push x)
```

Figure 5.18: Example: Position-polymorphic Restriction Handler (`most_recently`).

```
aligning: type xs ctx. (xs,ctx) mptr -> ctx ext
```

which use the multiprojection function to focus on the given positions `xs` in `ctx`. We show the implementation code in the appendix, Appendix C.1.

5.6.4 *Summary*

Our context- and position-polymorphic function definitions give important static guarantees for both system programmers and end users, checked and enforced by the OCaml compiler. First, recall that we keep track of the join shape $\vec{A}$ in the context formation judgment (Figure 4.2). The type variable `'a` in the abstract type `('a,'b) ctx` is the OCaml equivalent of the shape. It serves as a type-level index and "glue" that binds polyvariadic backend components and restriction handlers together. Components that share the same shape `'a` in their signature are composable.

Context polymorphism (Section 5.5.3) ensures that we can write polyvariadic components that compose with the backend implementation of any join instance of arbitrary shape. The type-level shape `'a` is usually accompanied by value-level content, which we think of as a polyvariadic interface for embedding a component into the backend implementation. In the case of Cartesius, the accompanying contents are the first-class effect declarations of type `'a Slots.hlist` (Section 5.5.1), from which we calculate effect handlers (e.g., Figure 5.9).

Position polymorphism ensures that we can write polyvariadic components that only need access to a specific part of the join shape `'a`. By construction, the access "cannot go wrong", because we track usage context requirements in type-safe pointers and sets. We simply demand in the signature of position-polymorphic definitions that the requirement equals the abstract join shape against which we write the implementation. E.g., `('i,'a) ptr -> 'a ext` for single positions (Figure 5.18), respectively `('xs,'a) mptr -> 'a ext` for sets of positions.

Furthermore, type-safe pointers and sets prevent end users from applying restrictions to wrong positions. I.e., referenced positions always are within bounds and have types that are compatible with the requirements of a restriction. Again, we ensure this by matching the join shapes. I.e., rule (Join) (Figure 5.4) matches the join shape demanded by the given extensions to the join shape $\vec{A}$ supplied by the join pattern syntax.

Table 5.1: Comparison between PolyJoin Version and Prototype of Cartesius.

	PolyJoin	Prototype
Features		
Heterogeneity	✓	✓
Arity-Genericity	✓	
Context Polymorphism	✓	
Position Polymorphism	✓	
Declarative Pattern Syntax	✓	
Implementation Code Size[a]		
Core (Fig. 5.1)	$O(1)$	$O(n^2)$
Restriction Handlers (Over c Positions)	$O(1)$	$O(n^{c+1})$
Restriction Handlers (Over Position Sets)	$O(1)$	$O(2^n)$
Join Instance Size[b]	$O(n)$	$O(n)$
Extensibility Dimensions		
Syntax	✓	
Restriction Handlers	✓	~[c]
Computational Effects	✓	✓

a n = maximum arity in use.
b n = number of inputs.
c Requires separate copy per supported arity.

5.7 EVALUATION AND DISCUSSION

Table 5.1 compares our PolyJoin-based implementation of Cartesius against the initial prototype we discussed in Section 3.5 Our version has significant advantages over the prototype. We discuss them below.

5.7.1 *Features*

In terms of features, the PolyJoin version of Cartesius is fully polyvariadic, i.e., it supports heterogeneous sources and any finite join arity. The original prototype satisfies "one half" of polyvariadicity, i.e., its backend does not support arity abstraction and offers only a handful of hard-coded arities. Each supported arity entails a separate copy of the backend code that has to be separately maintained, with little to no code sharing. Similarly, absence of context and position polymorphism (Section 5.6.4) greatly increases programming effort for both the backend and restriction handlers (we elaborate the issue below). Moreover, the prototype has no declarative frontend syntax for join patterns, so that end users require detailed knowledge of backend components and their composition to specify joins.

5.7.2 *Asymptotic Code Size*

Hard-coded arities, lack of arity abstraction and lack of context/position polymorphism lead to significant code duplication and severely undermine the composability and extensibility of the system. This is at odds with the goal to create an extensible and general event correlation system as originally envisioned. The issues become apparent when considering the *code size* of the respective implementations, reflecting programming effort.

In the prototype, the maximum supported arity n must be statically provisioned and accordingly, at most n copies of the backend, one for each $i \in \{1, \ldots, n\}$ have to be either manually programmed or pre-generated. If a client intends to specify a join pattern that exceeds the current maximum arity, then a new copy of the backend must be generated beforehand. In the worst case, the prototype backend has $O(\Sigma_{i=1}^{n} i) = O(n^2)$ size. This is because each arity i leads to the dependence on i first-class effect declarations and contains effect handlers with i cases for these effects (e.g., the `memory` handler in Figure 5.9).

Similar reasons apply in the case of restriction handlers, where we distinguish between restrictions parametric in a constant number c of positions (e.g., `most_recently` in Figure 5.18 has $c = 1$ position parameters) and restrictions parametric in position sets (e.g., `aligning` restriction Section 5.2.2). In the prototype, a separate handler definition must be written for *each arity* and *each combination* of positions, respectively *each subset* of $\{1, \ldots, n\}$. The code sizes become $O(n^{c+1})$ and $O(2^n)$, accordingly. Context and position polymorphism in PolyJoin reduce the programming effort to a single definition and thus have constant code size. Clients can specify join patterns of arbitrary arity and automatically, a suitable instance of the backend is calculated on demand.

In both versions, the size of a join computation at runtime is linear in the number n of joined input sources, because a join computation allocates n first-class effect instances. However, this is not a predictor of the overall runtime behavior, e.g., computational cost and memory requirements during a join's execution. These measures are highly dependent on the semantic variant specified by users. We refer to the measurements we conduct in Chapter 7 for the runtime behavior of example join variants.

5.7.3 *Extensibility Dimensions*

The PolyJoin version of Cartesius features an extensible pattern language, because of the reliance on the tagless final approach (Section 4.3). New forms of syntax can be easily added to the frontend language.

Restriction handlers are an orthogonal way to extend the system. They are subcomponents that can be separately developed, because only knowledge about an effect interface is required, represented by the first-class `SLOT` effects (Figure 5.8). However, the table entry for the prototype is only half-checked (~). While new restriction handlers are programmable, they are hardcoded against a specific join arity, due to the lack of context polymorphism.

Finally, since both versions are designed around algebraic effects and effect handlers, it is possible to induce new kinds of effects via restriction handlers and in the PolyJoin case additionally in the implementations of new syntax forms.

5.7.4 *Discussion*

LESSONS LEARNED The CARTESIUS case study heavily focuses on practical polyvariadic programming with first-class effects and handlers. While this is already a significant contribution for the algebraic effects community, it is also of value for the design and implementation of asynchrony and concurrency languages as well as typed embeddings of process calculi.

(1) the techniques demonstrated are of general utility for implementing polyvariadic backends against polyvariadic frontends: abstract type-level context shapes, context and position polymorphism as well as heterogeneous lists for safely programming and composing polyvariadic sub-components.

(2) POLYJOIN is a flexible and extensible design methodology that captures the essence of join and synchronization pattern frontend syntaxes.

(3) polyvariadicity "naturally" occurs in concurrency and asynchrony systems, since communication endpoints partaking in exchanges and synchronization are usually of heterogeneous type and arbitrary in number. The interface of Definition 4.1 on which we base POLYJOIN is adequate to set up their synchronization logic, enabling low-latency reactions and eliminating unnecessary blocking.

(4) the focusing problem we successfully solve in Section 5.5.6 is a manifestation of external choice, which is an important aspect of concurrency. While we implemented our solution in terms of effects and handlers, it could be achieved with other dynamic binding approaches, e.g., [Kis14]. Importantly, the dynamic-binding solution does *not* require a heterogeneous zipper structure and its quadratically-growing type-level description, while still being statically type-safe.

SUPPORTED SYSTEMS In general, any system is embeddable with POLYJOIN for which a suitable tagless interpreter can be written for the module signature in Figure 4.3, respectively Figure 5.6, if the system supports metadata in patterns. These already permit a wide range of backends. (1) any system that already has a LINQ-based frontend or is based on a monadic embedding can be plugged into POLYJOIN using the construction we show in Figure 4.6. (2) general, metadata-based event correlation backends can be plugged into POLYJOIN, which we have exemplified with CARTESIUS in this chapter. Importantly, we support more than only library-level implementations in the same host language and cover external systems, e.g., embedding the JVM-based Esper [Esp06] in OCaml. Embedding such systems additionally requires language bindings, e.g., Ocaml-Java [Cle14]. However, this is an orthogonal issue and we assume that a suitable binding exists. Once such a binding is in place, then POLYJOIN can act as its type-safe frontend to end users.

REPRESENTING CONTEXT DEPENDENCE In the embedding of CARTESIUS (Section 5.5), we represented implicit context dependence by function abstraction. In Chapter 3, we proposed an encoding of context dependence by means of ambient, implicit parameters in the style of [Lew+00]. These can be straightforwardly encoded in a language with algebraic effects and handlers. However, these would require generating *effect names* from the variable names in the pattern, which is impossible without proper compiler support. This makes the idea based

on implicit parameters less portable. In languages with compile-time ad-hoc polymorphism (e.g., Wadler and Blott [WB89] and Odersky et al. [Ode+18]) we could avoid the syntactic overhead of threading context through the join computation.

ON PORTABILITY PolyJoin makes use of advanced OCaml features: GADTs, phantom types, (first-class) modules, second-class higher-kinded polymorphism and Hindley-Milner type inference. However, it is portable to languages outside the ML family, as long as the target host language can express some form of bounded polymorphism and type constructor polymorphism. GADTs and phantom types can be encoded with interface/class inheritance, generics and subtyping. For illustration, Appendix B.2 shows a Scala version of PolyJoin. We anticipate that programmers can more conveniently and directly implement our ideas in languages with ad-hoc polymorphism, although our Scala example does *not* require it and works with local type inference [PT98]. The more "functional" the host language, the easier it is to port and the more "natural" the DSL syntax appears. We expect that PolyJoin should be expressible reasonably well even in modern Java versions with lambdas. However, as other uses of tagless final/object algebras in Java show [Bib+15], higher-kinded polymorphism has to be encoded indirectly, using the technique by Yallop and White [YW14], which increases the amount of required boilerplate.

ALTERNATIVE BINDER REPRESENTATIONS We investigated variants of PolyJoin that support an n-ary curried pattern notation, which has less syntactic noise than destructuring nested tuples:

```
join ((from a) @. (from b) @. (from c) @. cnil)
     (fun x y z () -> (yield y))
```

We show the full tagless final specification for currying in Appendix B.3. However, we found that the programming style is less convenient than the uncurried version, because in the encoding that we considered, join has the following signature

```
val join: ('shape, 'ps * 'res) ctx -> 'ps -> 'res shape repr
```

where the type parameter `'ps` for the pattern body hides the fact that it is a curried n-way function. To recover this information, structural recursion over the context formation rules is required (as in Figure 4.6). In contrast, we found that the uncurried style enabled a more natural, sequential programming style, which works well with heterogeneous lists, as exemplified by the Cartesius case study (Chapter 5).

5.7.5 *Future Work*

DISJUNCTIONS PolyJoin supports join patterns that are conjunctive, i.e., working on n-way products. However, we did not address *disjunctive* joins, yet, which are general notions of patterns with cases (coproducts). For example, the CML choose combinator [PKZ16]

$$\Diamond A \otimes \Diamond B \Rightarrow \Diamond(A \otimes \Diamond B \oplus \Diamond A \otimes B)$$

is a linear logic version of pattern matching events with cases.

Another example for disjunction is the Join Calculus [FG96]. In future work, we would like to investigate a syntax design that reasonably integrates disjunction with the event patterns of PolyJoin. Rhiger [Rhi09] shows that pattern cases are in principle expressible in terms of typed combinators in higher-order functional languages.

SHAPE HETEROGENEITY The joins we cover in PolyJoin assume a fixed type constructor/shape $S[\cdot]$, but it seems useful to have a declarative join syntax that supports joining over heterogeneous shapes $S_1[\cdot], \ldots, S_n[\cdot]$ into an output shape $S_{n+1}[\cdot]$. E.g., $S_i \in \{\mathsf{Channel}, \mathsf{DB}, \mathsf{Future}, \mathsf{List}, \ldots\}$. It seems promising to reduce the shape-heterogeneous case to the shape-homogeneous case, by defining a common `'a adapter` shape that encapsulates how to extract values from heterogeneous shapes.

5.8 RELATED WORK

LANGUAGE-INTEGRATED QUERIES AND COMPREHENSIONS Similarly to this work, Facebook's Haxl system [Mar+14; Mar+16] recognizes that monad comprehensions inhibit concurrency. Their primary concern is reducing latency, exploiting opportunities for data parallelism and caching in monadic queries that retrieve remote data dependencies, without any observable change in the result. Thus, they analyze monad comprehensions and automatically rewrite occurrences of monadic bind to composition on applicatives [MP08], whenever data parallelism is possible. In contrast, our work is concerned with discerning and correlating the arrival order of concurrent event notifications, which may lead to different results.

In their T-LINQ/P-LINQ line of work, Cheney, Lindley, and Wadler [CLW13] argue for having a quotation-based query term representation having higher-order features, such as functional abstraction. Their system guarantees that well-typed query terms always successfully translate to a SQL query and normalization can avoid query avalanche. Suzuki, Kiselyov, and Kameyama [SKK16] show that the T-LINQ approach by Cheney et al. is definable in tagless final style, improving upon T-LINQ by supporting extensible and modular definitions. Najd et al. [Naj+16] generalize the LINQ work by Cheney et al. to arbitrary domain specific languages, using quotation, abstraction and normalization for reusing the host language's type system for DSL types. They rely on Gentzen's subformula property to give guarantees on the properties of the normalized terms by construction, e.g., absence of higher-order constructs or loop fusion. However, none of these works address general polyvariadic syntax forms as in PolyJoin, outside of nested monadic bind. We consider it important future work to integrate these lines of research with ours.

The join interface we propose in Definition 4.1 is an n-ary version of Joinads, which underlie F# computation expressions [PS14; SPL11; PS11]. The latter are a generalization of monad comprehensions allowing for parallel binders, similar to PolyJoin. However in contrast to our work, F# computation expressions are not portable, requiring deep integration with the compiler and are not as extensible and customizable. E.g., we support tight control of the pattern variable signature, via shape translation and support extensibility with new syntax forms. Furthermore, we support pattern syntax designs for systems with implicit metadata.

POLYVARIADICITY Danvy [Dan98] was the first to study polyvariadic functions to define a type-safe printf function in pure ML and inspired many follow-up works, e.g., [FI00; Asa09]. Rhiger [Rhi09] would later define type-safe pattern matching combinators in Haskell. Similarly to our work, Rhiger tracks the shape of pattern variable contexts at the type-level and supports different binding semantics.

However, the above works rely on curried polyvariadic functions, while we choose an uncurried variant, in order get hold of the entire variable context in the tagless final join syntax. This makes it easier to synthesize concurrent event correlation computations, because usually, all the communicating components must be known in advance (cf. our discussion on alternative binder representations in Section 5.7.4). We postulate that usually some variant of the focusing problem (Section 5.5.6) manifests in the concurrency case, where each communication endpoint contributes one piece of information to a compound structure (e.g., the cartesian product of mailboxes in Cartesius), but the code representing the contributions (e.g., callback on the i-th event source) is position-dependent and heterogeneously-typed.

To the best of our knowledge, this work's combination of (1) tagless final, (2) abstract syntax with polyvariadic signature, (3) GADT-based context formation and (4) type-level tracking of the variable context shape is novel. Importantly, the combination achieves modular polyvariadic definitions, separating polyvariadic interfaces and polyvariadic implementations. Interactions with modules and signatures as in PolyJoin have not been studied in earlier works on polyvariadicity. Some works even point out the lack of modularity in their polyvariadic constructions, e.g., the numerals encoding by Fridlender and Indrika [FI00]. GADTs, which none of the previous works on polyvariadicity consider, are the missing piece to close this gap, as they enable inversion/inspection of the context formation.

Lindley [Lin08] encodes heterogeneously-typed many-hole contexts and context plugging for type-safe embeddings of XML code in ML. His solution can express constraints on what kind of values may be plugged into specific positions in a heterogeneous context. In our work, such constraints would be useful for controlling that only certain combinations of restriction handlers (Section 5.6) can be composed and applied. However, his representation of heterogeneous sequences differs from ours and it remains open how to reconcile the two designs with each other, which we consider interesting future work.

Weirich and Casinghino [WC10b; WC10a] study forms of polyvariadicity in the dependently-typed language Agda [Nor07] in terms of typed functions dependent on peano number values. Dependent types grant more design flexibility for polyvariadic definitions, at the expense of not being supported by mainstream languages. To the best of our knowledge, works on dependently-typed polyvariadicity have not investigated modularity concerns so far. McBride [McB02] shows how to emulate dependently-typed definitions, including polyvariadic definitions, in Haskell with multi-parameter type classes and functional dependencies. While more the above approaches are more powerful, they are less portable and not as lightweight as PolyJoin.

HIGHER-ORDER ABSTRACT SYNTAX The idea of higher-order abstract syntax (HOAS) was first introduced by Huet and Lang [HL78]. Later, the paper by Pfenning and Elliott [PE88] popularized HOAS, which they implemented for the Ergo project, a program design environment. HOAS eliminates the complications of explicit modeling of binders and substitution in abstract syntax. One of their motivations was simplifying the formulation of syntactic rewriting rules in formal language development, and ensuring the correctness of variable bindings. Similarly to this work, they too recognize limitations of a purely curried specification style for n-ary bindings and propose products for uncurried HOAS bindings with nested binary pairs, which required changes to their implementation and unification algorithm. Our approach works "out of the box" with modern higher-order languages, both with HM and local type inference. However, while their motivation for uncurried binders were matters of formal syntax representation and manipulation, we are motivated by denotation of syntax, i.e., uncurried n-ary binding forms satisfactorily enable concurrency implementations.

5.9 CHAPTER SUMMARY

We defined a fully polyvariadic multicore OCaml implementation of CARTESIUS with declarative event pattern syntax. CARTESIUS includes additional syntax elements for event correlation patterns, i.e., specifications of restriction handlers and timing attributes of events. We cover both by a straightforward variation of core POLYJOIN from the previous chapter. Independently of these variations, the implementation demonstrates that POLYJOIN's uncurried HOAS representation of pattern variables accommodates concurrent and interleaved variable bindings.

Type-level indexing over the pattern variable context shape $\vdash_{\text{ctx}} \Pi : \vec{A}$ is the essential design ingredient that allows programming polyvariadic tagless interpreters and easily scales to larger code bases. The index asserts modularity and composability of software components, and acts as a "glue" at the type level. Interestingly, the same type-level index $\vec{A}$ has *different* term-level representations in different places. For example, the index is witnessed by the pattern context Π and the heterogeneous lists of generative `SLOT` effects. In this sense, the index exists "everywhere and yet nowhere".

Furthermore, type-level shape indexing enables context- and position-polymorphism, yielding effect handlers of generative effects that are safe and convenient to use, similar to ordinary effect handlers. The handlers for the CARTESIUS core backend as well as the restriction handlers we can write this way in multicore OCaml are notationally very close to our formalization in Chapter 3. The formalization indicated dependence on generative effect instances by number indices, which translates in our implementation to type-safe pointer access into a context of generative effect instances. We can ensure that restriction handlers are always compatible with the context in which they are used, via the type-level index and type-safe pointers. This generalizes without issues to sets of positions. Finally, we substantially improve composability, reusability and compatibility over implementations with hard-coded arities.

For asynchronous computations, such as in event correlation, we found that computations follow a common pattern. That is, evaluation must focus "in the middle" of a heterogeneous shape and then collapse this shape during the correlation. Equivalently, for each variable in the (heterogeneous!) context, we

must synthesize a callback that relates the event notifications for the current variable binding with the variable binding of the rest of the context. We fruitfully exploited a synergy with algebraic effects and effect handling to polyvariadically implement this callback logic.

Part III

EVALUATION

In which we empirically evaluate the expressiveness and performance of CARTESIUS.

SURVEY: EXPRESSIVITY OF CARTESIUS

6

This chapter shares material with the ICFP'18 paper "Versatile Event Correlation with Algebraic Effects" [Bra+18].

SYNOPSIS In this chapter, we empirically investigate the expressivity of CARTESIUS (Chapter 3). We validate that CARTESIUS enables modeling and composing a range of join semantics across the families of CEP/streaming engines, reactive programming languages/frameworks, as well as concurrent programming languages. For this purpose, we survey representative works across these families, comparing their features with those of CARTESIUS. Furthermore, we discuss classes of complementary features that our work does net yet cover.

6.1 SURVEY

The focus of the survey is on features for event joins. Table 6.1 and Table 6.2 summarize the surveyed works (rows) and feature categories (columns) related to joins. A checkmark (✓) indicates that a feature is readily available in the respective system. Half-checked (~) indicates that the feature is supported to a limited degree or can be implemented on top of the available abstractions. An empty ( ) cell indicates that a feature is not supported.
Below, we discuss the surveyed works and the considered feature categories.

6.1.1 *Surveyed Works*

We subdivide the surveyed works by the four event correlation families we consider in this thesis. For simplicity, we put complex event and stream processing systems into one category. While they have different roots, there is sometimes no crisp distinction whether a given system belongs either in one or the other familiy (cf. Cugola and Margara [CM12]). Here, we briefly introduce each work in each family, in their chronological order of appearance.

COMPLEX EVENT AND STREAM PROCESSING We consider the following works:

- Snoop (Chakravarthy and Mishra [CM94] and Chakravarthy et al. [Cha+94], 1994), which is a CEP system coming from the active database community, i.e., event-driven database architectures that detect and react to changes.
- Rapide (Luckham et al. [Luc+95] and [Rap97], 1995), which is a hybrid object-oriented and event-based language for prototyping distributed systems architectures.
- Esper (EsperTech Inc. [Esp06], 2006), which is a CEP engine that integrates into the Java Virtual Machine (JVM). It features a query language, which is derived from SQL, adding event patterns and time windows.
- Cayuga (Demers et al. [Dem+06], 2006), which is a general-purpose publish/subscribe system that correlates notifications by nondeterministic finite automata (NFA) [HMU06].

- SASE+ (Diao, Immerman, and Gyllstrom [DIG07], 2007 and Agrawal et al. [Agr+08], 2008), which is a streaming system tailored to detecting sequences and aggregations (Kleene plus [HMU06]) with a query language that combines traits of SQL and regular expressions.
- EventJava (Eugster and Jayaram [EJ09], 2009), which is variant of Java with language integration of complex event patterns into methods of classes.
- TESLA (Cugola and Margara [CM10], 2010), which is an event pattern language based on a first-order temporal logic and compiles to a custom variant of nondeterministic finit automata (NFA).
- Google Data Flow (Akidau et al. [Aki+15], 2015), which is the underlying processing model of Google Cloud Dataflow and deployed for statistics collection, billing and anomaly detection in Google's services.

REACTIVE PROGRAMMING We consider the following works:

- FrTime (Cooper and Krishnamurthi [CK06] and Cooper [Coo08], 2006) and Flapjax (Meyerovich et al. [Mey+09], 2009). The former is an implementation of push-based functional reactive programming that integrates in call-by-value languages (e.g., Scheme) for interactive applications, whereas the latter is a continuation of the work in the context of reactive programming in web applications with JavaScript.
- Push/Pull FRP (Elliott [Ell09], 2009), which contributes a reactive programming library in Haskell that combines traits from both pull- and push-based processing. We adopted the push/pull stream representation from this work for our reactive data type in Chapter 3.
- Reactive Extensions (ReactiveX, Rx) (Meijer [Mei12] and [Rea11], 2011), which is a widely-used, general-purpose asynchronous reactive programming library based on the observer pattern [Gam+94], respectively the continuation monad [Gon12]. It was first implemented on the .NET platform, but has since been ported to many other languages, e.g., Java, Scala, JavaScript, Clojure and Swift.
- Asynchronous C♯ (Bierman et al. [Bie+12], 2012), which provides asynchrony abstractions in the C♯ standard library, enabling direct-style programming in the presence of inversion of control. This library is representative of other languages/libraries that use similar abstractions, e.g., futures and promises. In Chapter 3, we encoded these abstractions on top of algebraic effects and handlers.
- Elm (Czaplicki and Chong [CC13], 2013), which is an asynchronous functional reactive language that has a design similar to the Haskell language and compiles to JavaScript for web-based reactive applications.

CONCURRENT PROGRAMMING We consider the following works:

- Concurrent ML (CML) (Reppy [Rep91], 1991), which is an extension of the Standard ML language with abstractions for higher-order, concurrent, functional programming, based on communication channels and synchronizable events. Their event notion corresponds to "potential to yield a value", respectively an eventually modality in linear temporal logic (cf. [PKZ16]), which is not the same as our notion of event for notifications of something that happened (cf. our discussion in Section 3.2.1).
- JoCaml (Conchon and Le Fessant [CL99], 1999), which is an implementation of the Join Calculus (Fournet and Gonthier [FG96]) in a dialect of OCaml. It features declarative synchronization (join) patterns, which are concurrently active case statements that compete against messages.
- Polyphonic C♯ (Benton, Cardelli, and Fournet [BCF04], 2004), which is another implementation of the Join Calculus which integrates into the object-oriented C♯ language and its classes and methods, enabling inheritance of join patterns.
- Manticore (Fluet et al. [Flu+08], 2008), which combines concurrency primitives of CML (explicit concurrency) with data parallelism (implicit concurrency). The latter enables parallelization of sequential syntax forms, such as array comprehensions. Furthermore, it features parallel bindings and case statements.

6.1.2 *Surveyed Feature Categories*

Category: Event Relations

This category is concerned with features pertaining to defining relations between event notifications, which is one of the hallmarks of event correlation. We distinguish between three basic subcategories of event relations:

SEQUENCE Pertains to linguistic concepts for correlating by a notion of observation order between different events of a join, e.g., "event a from s_1 is followed by event b from s_2". Some languages specialize in detecting contiguous sequences, e.g., SASE+ [Agr+08], which we exemplified in Chapter 2. Other languages, e.g., TESLA [CM10], also allow partial orders in a join, i.e., a subset of the input events must occur in a sequence, while others may occur in an arbitrary order.

Notably, concurrent programming languages characteristically lack correlation by sequence patterns. Concurrent ML (CML) and Manticore are half-checked, because they have case statements that allow synchronization on two events and discernment of the occurrence order in the clauses. JoCaml and Polyphonic C♯, which are implementations of the Join Calculus [FG96] cannot discern order, due to a nondeterministic selection and consumption of messages.

Table 6.1: Overview of Supported Join Features in Cartesius and other Works.

	Event Relations			Time Model		Windows				Event Selection				Event Consumption	
	Sequence	*Attribute*	*Timing*	*T. Stamp*	*Interval*	*Fixed*	*Sliding*	*Session*	*Custom*	*FIFO*	*LIFO*	*Position*	*Nondet.*	*Custom*	*Linear*	*Multiple*
Complex Event/Stream Processing																
Snoop[CM94; Cha+94]	✓	✓		✓			✓			✓	✓		✓			✓
Rapide [Luc+95; Rap97]	✓	✓	✓	✓		✓	✓			✓						✓
Esper [Espo6]	✓	✓	✓	✓		✓	✓			✓	✓	✓				✓
Cayuga [Dem+06]	✓	✓	✓		✓					✓			✓			✓
SASE+[DIG07; Agr+08]	✓	✓		✓			✓			✓			✓			✓
EventJava [EJo9]	✓	✓	✓	✓		✓		✓		✓				≈	✓	
TESLA [CM10]	✓	✓	✓	✓						✓	✓	✓	✓		✓	✓
Google Data Flow [Aki+15]	✓	✓	✓	✓		✓	✓	✓					✓		✓	✓
Reactive Programming																
FrTime[CKo6; Cooo8] / Flapjax[Mey+09]	✓	✓								✓					✓	✓
Push/Pull FRP [Ello9]	✓	✓	✓	✓						✓					✓	✓
ReactiveX [Rea11; Mei12]	✓	✓				≈	≈	≈	≈	✓	✓	✓	✓	✓	✓	✓
Asynchronous C♯ [Bie+12]	✓	✓					≈			✓	✓	✓	✓	✓	✓	✓
Elm [CC13]	✓	✓								✓					✓	✓
Concurrent Programming																
CML [Repg1]	≈	✓	≈										✓		✓	
JoCaml [CL99]		≈											✓		✓	
Polyphonic C♯ [BCF04]													✓		✓	
Manticore [Flu+08]	≈	✓	≈										✓		✓	
Cartesius	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 6.2: Overview of Supported Join Features in CARTESIUS and other Works (cont).

	Ordering		Non-occurrence	Merging	Extensions
	Temporal	*Custom*			
CEP/Stream					
Snoop	✓			✓	
Rapide	✓	✓		✓	
Esper	✓	✓	✓	✓	
Cayuga	✓		✓	✓	
SASE+	✓		✓	✓	
EventJava	✓				
TESLA	✓		✓		
Google Data Flow	~	~		✓	
Reactive Prog.					
FrTime / Flapjax	✓				
Push/Pull FRP	✓				
ReactiveX		~	✓	✓	✓
Asynchronous C♯			✓		
Elm	✓				
Concurrent Prog.					
CML			✓	✓	
JoCaml			✓	✓	
Polyphonic C♯			✓	✓	
Manticore			✓	✓	
CARTESIUS	✓	✓		~	✓

ATTRIBUTE Pertains to constraints on the event contents, e.g., for filtering or associating by a common id. Most of the considered works readily support this feature. Exceptions are languages based on the Join Calculus [FG96], i.e., JoCaml [MM14] and Polyphonic C♯ [BCF04]. The former allows deep pattern matching on constructors in join patterns, but relating the contents of two or more events can only happen *after* the join triggers. This does not fit well together with the linear event consumption semantics.

TIMING Pertains to expressing the relative time distance between event pairs, e.g., "match if events *a* and *b* occur at most 10ms apart from each other". Note that this is not the same as a time window, which is a scoping/grouping construct. We consider in a separate category below.

Timing relations are most prominently available in CEP/streaming systems. Exceptions are Snoop and SASE+, which support time windows, though. In reactive and asynchronous programming languages, timing relations are not supported as primitive notions. However, as part of event merging, push/pull FRP supports some form of comparing timing of events. In concurrent programming language, synchronization by timing relations is not directly supported. However, CML and Manticore in principle allow restricted forms, i.e., racing against timeout events generated by a local time source, to discern whether some other event happens before the timeout occurs.

Category: Time Model

This category is concerned with specifying any time information attached to events. We distinguish between one dimensional time stamps and two dimensional intervals. The latter enables finer-grained distinctions in combination with sequence relations [Whi+07]. In the CEP and stream processing families, time models and timing are much more common than in asynchronous reactive or concurrent programming. The time model of CEP and stream systems is tied to the architecture of the system for ordering guarantees. Additionally, event times are not always explicitly exposed in the specification language, e.g., in SASE+ and Snoop. Even though they often provide time-related signals, most reactive programming languages do not attach timing information to events. This is why their time-related columns are empty ( ). Our work supports a time model based on intervals, but it can be in principle be customized for other time models.

Category: Windows

This category is concerned with limiting the extent of correlations with some additional context specification:

FIXED Pertains to absolute windows, e.g., "consider all events from May 1 to May 30".

SLIDING Pertains to possibly overlapping windows of a given size and period, e.g., "a window of 1 month each week".

SESSION Pertains to windows having event-driven indicators of their start/end and are thus not statically predictable, e.g., "a window from when user logs in until the user logs out".

CUSTOM Pertains to the support for customization and extension with new kinds of windows in a given system.

Notions of windows are prominently found in the CEP and stream processing families. This coincides in most cases with the support for timing in event relations. Esper and Google Data Flow model both have the most exhaustive support for windows. However, no surveyed approach in the CEP and stream processing families supports extension and customization with new windows. We argue that this is a problem, since windows are a delicate issue in these families and no good foundations exist. This is witnessed by the work on the SECRET model [Din+13; Bot+10], which shows that window implementations may behave vastly different between systems. If such semantic differences matter to an application, then an extensible system would in principle allow implementing differently-behaving kinds of windows.

Windows can be encoded in terms of higher-order streams from the available combinators in Reactive Extensions. Due to the lack of a time model, the related columns are only half checked (~): Events can be tagged with timestamps from some time source, but the underlying system enforces no ordering or monotonicity for these windows. For other programming language approaches in both the reactive and concurrent families, we are not aware of any viable encodings of windows.

Our work in principle supports the dicussed kinds of windows and extensions, in a similar fashion to the window implementation in Section 3.4.5.

Category: Event Selection

This category is concerned with supported strategies/policies for how events are selected for correlation from event sources as well as linguistic concepts for the specification of such strategies:

FIFO Pertains to selecting the events in received order.

LIFO Pertains to prioritizing the most-recently seen events, e.g., if hardware sensors report events in rapid succession, then usually the most recent one is relevant and older ones can be discarded.

POSITIONAL Pertains to selecting events by a positional specification, e.g., "the first n events of stream s_1 with every other event of stream s_2".

NONDETERMINISTIC Pertains to selection of events that is ambiguous or random.

CUSTOM Pertains to a customization and extension with new selection policies.

The most common selection policy is FIFO, which is shared by CEP and stream processing as well as reactive languages. Of course, the concurrent programming languages by their nature select concurrent events nondeterministically.

Google Data Flow uses the out-of-order processing model by Li et al. [Li+08], where order in join operators pertaining to time is not enforced to reduce latency. Consequently, we mark their event selection as nondeterministic. Furthermore, we mark systems based on NFAs with nondeterministic selection.

None of the surveyed works in the CEP and processing family support customization/extension of the event selection beyond the ones that they come equipped with. One notable exception is EventJava, but only to a limited degree: their underlying engine allows configuration changes to adapt the matching/selection policy, but their surface language, a Java dialect, does not expose a linguistic concept to specify the policy.

In the case of reactive languages, we find that not all of them support arbitrary selection policies. The reason is that these languages maintain a synchronous dataflow runtime, in order to avoid *glitches* [Edw09; CK06], i.e., temporary inconsistencies in computed dataflow values due to propagation delays in updates. Both Reactive Extensions and Asynchrounous C♯, however, support arbitrary selection policies, since they have no glitch-free propagation system.

Our work enables very flexible implementations of event selection by defining suitable restriction handlers. They have the capability to manipulate and select from the memory/mailboxes of event observations in user-defined ways.

Category: Event Consumption

This category is concerned with supported strategies/policies for how events are consumed during correlations as well as linguistic concepts for the specification of such strategies:

LINEAR Pertains to consuming events exactly when they are part of a matching combination of events, so that they are not available for further matching. For example. the `zip` combinator is linear.

MULTIPLE Pertains to consuming events multiple times or up to a given number of times. For example, in a cartesian product, an event in one stream is consumed as many times as the size of the other stream.

Most of the upper half (CEP, stream processing and reactive languages) allows that events are consumed multiple times, whereas the bottom half supports linear consumption. In concurrent programming languages, events are only linear, since they are used for synchronizing concurrent processes with side effects. One would not want to uncontrollably change the world multiple times.

Our work supports control over the event consumption by life time counters of entries in the memory/mailboxes of a join instance.

Category: Ordering

This category is concerned with linguistic concepts to specify ordering on correlated results, respectively maintenance of order invariants:

TEMPORAL Pertains to the underlying system maintaining order of correlated events by time.

CUSTOM Pertains to ordering correlated events in user-defined ways.

Maintaining an order of event data by time is a common trait of all surveyed CEP and stream processing systems. Notably, Rapide and Esper support different notions/dimensions of time in events, from different clocks. For example, time data could be supplied externally from event sources, or it could be the local machine time, or composed from multiple other criteria. Google Data Flow is half-checked, due to the reliance on the out-of-order processing model.

We classify some of the reactive programming language as maintaining a temporal order, in the sense that their underlying dataflow model maintains consistency of the computation, avoiding glitches we discussed earlier. This requires care on how updates in the dataflow model are propagated. For example, the push/pull FRP model maintains events as potentially infinite lists of notifications tagged with time, such that time monotonically increases. Similarly, FrTime and Elm maintain consistency of signals.

In contrast, Reactive Extensions does not have a propagation system to maintain time and consistency. However, streams can be tagged with custom clock sources, but programmers have to manually take care of ensuring monotonicity invariants, which is why we put a half checkmark. Not surprisingly, due to the lack of timing, concurrent programming languages do not have notion of ordering.

CARTESIUS supports an ordering on intervals that maintains monotonicity on the end of timestamp of intervals. It also can be customized to support other time models.

Category: Non-Occurrence

This category is concerned with features that enable expressing that some event(s) should *not* occur. To guarantee that non-occurrence results in productive specifications, one can add timing constraints or time windows that induce a finite waiting time or rely on anchoring events, e.g., "match *a* followed by *b*, where no *c* occurred in between" or "no *a* event should occur within 5 seconds".

Esper supports such forms of non-occurrence in queries, as well as the NFA-based Cayuga, SASE+ and TESLA.

Some of the reactive programming languages we surveyed cannot linguistically express non-occurrence checks, since it does not make sense in their consistency-maintaining/glitch-free dataflow system. However, nodes in their underlying propagation system might still signal "no change" messages as part of their model, such as in Elm. Reactive Extensions and Asynchronous C♯ in principle allow programmatic checking by using mutable shared state if an asynchronous source fired. Concurrent programming languages allow non-occurrence checks by racing against timeout events.

We have not yet investigated how to reasonably integrate non-occurrence into CARTESIUS and leave it for future work. An open question is how non-occurrence should be incorporated into the syntax design of correlation patterns. Furthermore, asynchrony necessitates that non-occurrences need some form of time guard or anchor, such as above, for guaranteeing that specifications do not induce infinite waiting times. Otherwise, a correlation can never be sure

that a non-occurring source does not fire at some arbitrary point in the future. Especially, due to our decomposition into cartesian product and restrictions, unlimited waiting might result in unlimited buffering of event notifications, which is to be avoided.

Category: Merging/Union

This category is concerned with merging events of multiple sources into one source, e.g., by a sum type $\mathrm{R}[A] \to \mathrm{R}[B] \to \mathrm{R}[A + B]$. While this is a dual operation to joining, which produces products/tuples, we include it here, because it is a useful complementary operation for correlations, and it shows what our work does not yet address.

Most CEP and stream processing systems support merging in some form. However, we could not find corresponding operators in EventJava and TESLA. For similar reasons as stated for the previous category, a notion of merge does not make sense in reactive languages. However, Reactive Extensions supports merges, due to its lack of a propagation system. Concurrent programming languages can compute sums of events by constructions with combinators and disjunctions/cases on events.

While Cartesius focuses on joins as products/conjunctions, it is in principle possible to support merging/disjunction, which is why we mark our work half-checked. The handler-based event correlation Definition 3.3 allows such an interpretation, because (1) the **push** effects are already interleaved and thus mergeable and (2) it is left open how these effects are handled/discharged. While the rest of the framework turns the effects into products, we could handle them differently in principle. We consider it important future work to add merging/disjunction.

Category: Extensions

This category is concerned with the question: Is the work open to extension with new correlation variants? Almost all surveyed systems are closed to extension, except for Reactive Extensions and our work. While Reactive Extensions can support arbitrary implementations of stream joins, due to its integration into general purpose OO languages, it is far less structured than our work and does not support first-class control flow and coroutining for interactions with generic components as we do.

6.2 RELATED WORK

The survey by Cugola and Margara [CM12] focuses on CEP and stream processing systems and the survey by Bainomugisha et al. [Bai+13] focuses on reactive programming languages. Our primary concern is investigating the supported features for joining of the covered systems and how they compare to our work, whereas their surveys are much broader in scope and not as focused. Furthermore, neither of the two surveys considers concurrent programming languages, and we consider in part systems that are newer.

6.3 CHAPTER SUMMARY

Caveat emptor: At first glance, Table 6.1 and Table 6.2 may suggest that CARTESIUS is superior in comparison. However, this is because in this chapter, we focus on event joins only, while other works have a broader or different scope. Other systems certainly exhibit more features not related to joining, that are nevertheless useful for correlations. For example, one category of features we have not considered in this work are aggregations, such as averages over windows or top k items by some specific ordering. We consider these important future work. Our message is that we can cover relevant features around event joins from *across all event correlation families*, delivering a unifying model, based on a small foundation with clear semantics and an extensible design. We thus conclude that we have captured the essence of event correlation. Due to the free composability of features and the locality of effect handlers, we obtain a highly adaptable and customizable event correlation system.

7

SYNOPSIS In this chapter, we assess the performance of our CARTESIUS implementations in multicore OCaml in terms of synthetic microbenchmarks.[1] Our motivations are: (1) How effective is the computational interpretation we proposed in Chapter 3? (2) What is the abstraction overhead of the control flow restrictions imposed by effect handlers in CARTESIUS? (3) What is the performance gap between CARTESIUS and high-performance stream libraries?

7.1 ASSESSMENT OF THE COMPUTATIONAL INTERPRETATION

This section is from our ICFP 2018 paper [Bra+18], and gives quantitative arguments that the computational interpretation from Chapter 3 is effective, in the sense that the way we structure and compose join computations from a cartesian product and restriction handlers enables join implementations which are more efficient than a naïve generate-then-test approach, by a priori avoiding the materialization of unneeded tuples. To validate this claim, we conducted a microbenchmark on our initial multicore OCaml prototype from Chapter 3. Nevertheless, it is sufficient to answer the question.

Table 7.1: Impact of restriction handlers on a three-way cartesian product. n is the total number of random input events, distributed evenly among three input reactives.

	memory ($\frac{\#\text{events}}{100\ \text{iterations}}$)	**throughput** ($\frac{\#\text{events}}{\text{sec}}$)	**#tuples**
3-way cartesian product			
with $n = 3 \cdot 370$ events[a]	610.5	23.06	50653000
mostRecently restriction			
with $n = 3 \cdot 3700000$ events	3	279632.76	11099998
affinely restriction			
with $n = 3 \cdot 3700000$ events	≈ 0	301311.34	3700000
zip restriction			
with $n = 3 \cdot 3700000$ events	≈ 0	372513.73	3700000

a Amount reduced due to high computational cost.

SETUP AND METRICS We performed microbenchmarks using join definitions in our multicore OCaml implementation. We generated streams of random integer-valued events, 10, 000, 000 in total, evenly distributed among the inputs of a 3-way join. We tested four variants of a 3-way join in our initial multicore OCaml prototype, processing: (1) an unrestricted cartesian product, (2) one with *mostRecently* restriction on all inputs, (3) one with the *affinely* restriction on all inputs, and (4) a 3-way zip in terms of both the *aligning* and *mostRecently*

1 The implementations of the benchmarks are available at `http://bracevac.org/correlation`.

restrictions (Section 3.4.4). We executed the benchmarks on a Mac Pro, 3 GHz Intel Xeon E5-1680 CPU, and 32 GB system memory. For each join definition, we measured (1) the average number of events retained in the join's memory, sampled after every 100 **push** notifications, (2) the average throughput in events per second, and (3) the total number of tuples generated in response to the input.

RESULTS Table 7.1 summarizes the results. Note that for the pure cartesian product, we used $n = 3x370$ as opposed to $n = 3x3700000$ for the other join variants. This reduction of the input size by several orders of magnitude was necessary due to the high computational cost of the pure cartesian product. While they process 10000 times more events per each input, the variants of the 3-way joins with effect handlers perform significantly faster and with negligible memory overhead:
(1) The memory consumption reflects the specified join semantics, e.g., *mostRecently* retains exactly one event for each input and *affinely* on average retains no event. In contrast, the cartesian product join grows linearly with the input over time.
(2) Likewise, joins restricted by effect handlers have a significantly higher throughput (between 12000 and 16000 times).
(3) The significant reduction of candidate tuples generated by the *mostRecently*, *affinely*, and *zip* joins demonstrates that effect handlers effectively avoid irrelevant computations, which leads to the significant memory reductions and throughput improvement. To put the numbers in perspective, the 3-way cartesian product would produce an amount of tuples on the order of 10^{19}, if it processed the full amount of input events under an even distribution among the input streams.

The n-way cartesian product's time complexity is sensitive to the distribution of the input data in the mailboxes. For instance, in Section 7.3 we measure a binary cartesian product with non-uniform input distribution, and manage to process 100, 000, 000 events with it in a reasonable time frame, because the input is split up into $10 \cdot 10,000,000$, resulting in as much pairings. An even distribution would lead to $50,000,000^2$ pairings for the binary cartesian product, an amount on the order of 10^{15}.

7.2 PERFORMANCE OF THE POLYVARIADIC IMPLEMENTATION

We performed more elaborate benchmarks, using the polyvariadic multicore OCaml implementation of CARTESIUS from Chapter 5. The polyvariadic implementation enables us to easily generate joins of any finite arity. We study if and how performance changes with increasing arity.

EXPERIMENTS We again tested the *affinely*, *mostRecently*, and *zip* join variants, each instantiated to the arity $n \in \{8, 12, 16, 20\}$. We altogether excluded the unrestricted cartesian product, due its high resource consumption and execution time when the input is evenly distributed. For each join variant and arity, we measured (1) the production rate of events over time, (2) the latency over time and (3) the overall throughput.

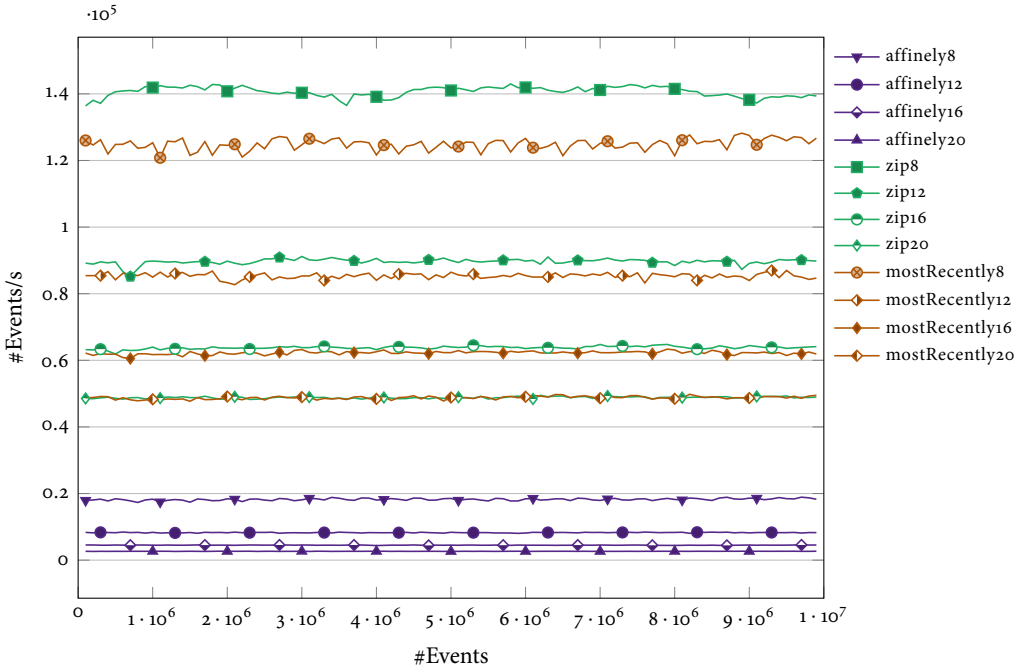


Figure 7.1: Microbenchmark: Mean event production rate over time in number of events per second (y-axis), over a time period processing $N = 10,000,000$ input events (x-axis). We measured each n-way join variant against arity $n \in \{8, 12, 16, 20\}$, as indicated by the legend.

INPUT As in the previous section, we pre-generated $N = 10,000,000$ events on the heap, evenly distributed among n streams, in accordance with the arity of the join variant.

SETUP We executed the microbenchmarks on a 2018 15-inch Macbook Pro model, with 32GB DDR4 memory and a 2.6Ghz Intel Core i7 CPU. We created the benchmarks with a custom code generator and compiled them to native code executables with multicore OCaml 4.06. For each join variant, we repeated the measurements 30 times and computed the mean. To avoid influences from previous runs, each iteration would first invoke the OCaml garbage collector. We determined progression of the computation by counting the number of input events pushed into the system, and took samples for production rate and latency every 100,000 events, for a total of 100 samples per run. A sample for production rate would involve measuring the wall-clock time for producing 1000 events. We use monotonic counters of the operating system with nanosecond resolution to measure wall-clock time. Similarly, a latency sample would measure how long it takes the join to produce an output after pushing the next event. Overall throughput is determined by dividing the total number N of processed events by the duration of the run, as reported by the wall-clock time.

RESULTS Figure 7.1 shows the measured mean production rate over time, in events per second. Figure 7.2 shows the measured mean latency over time in nanoseconds, and Figure 7.3 the overall throughput in events per secons, for each measured join variant and arity.

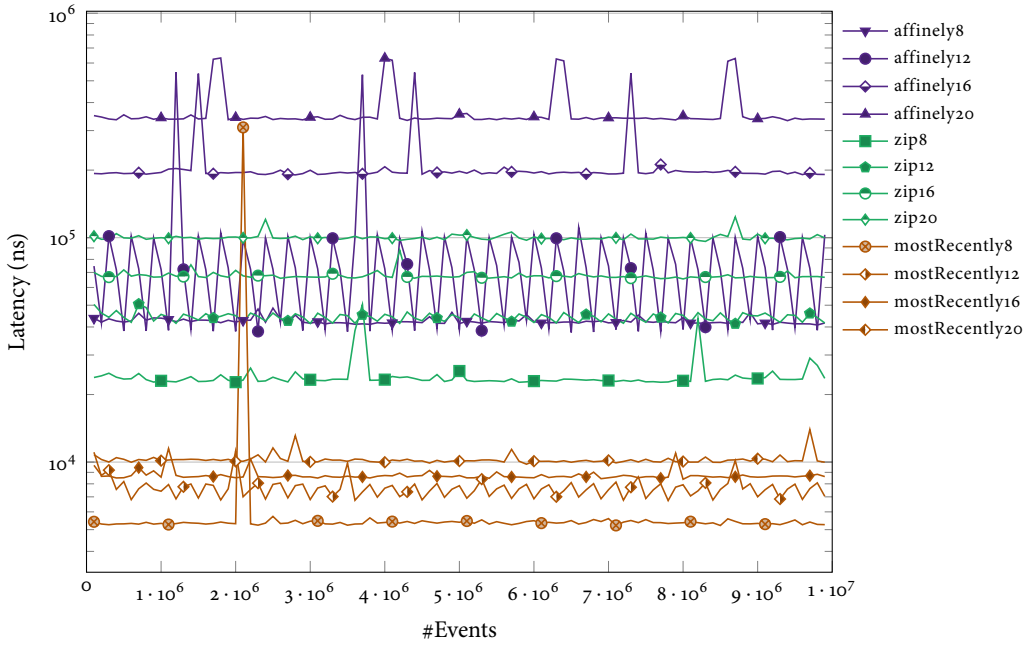


Figure 7.2: Microbenchmark: Mean latency over time in nanoseconds (y-axis, logarithmic scale, lower is better), over a time period processing $N = 10,000,000$ input events (x-axis). We measured each n-way join variant against arity $n \in \{8, 12, 16, 20\}$, as indicated by the legend.

DISCUSSION The results reveal that arity has an impact on performance, and also show characteristic behavior of the measured join variants.

Production rate (Figure 7.1) reveals how eagerly a join variant can produce output tuples from a pushed input event. Most of the variants maintain consistent production rate, but they differ in the magnitude. One notices that the production rates of *mostRecently* and *zip* are always close together for all considered arities. This is because *zip* incorporates *mostRecently* in its definition (cf. Section 3.4.4). The biggest gap between the two is at arity 8, and these two have the highest production rates and oscillation. We do not have an explanation why *mostRecently* oscillates so much, and why *zip* sometimes exhibits higher rates, even though it has a more restrictive association behavior (it implements a synchronization barrier). We observe for these two variants that the lower the arity, the higher the production rate. For *affinely*, the exact opposite is the case: the higher the arity, the lower the rate. This is because this restriction assigns a life time counter of 1 to incoming events. As soon as an event is output in a tuple, it is gone. Thus, at a higher arity, the join has to wait longer until more events enter its mailboxes, and it then exhibits a production burst.

The production bursts for *affinely* are visible in the measured latency as spikes (Figure 7.2). We observe the exact opposite to production rate, i.e., the higher the production rate, the lower the latency and vice versa. In the case of *zip*, we see now a bigger difference to *mostRecently*, which is due to its additional

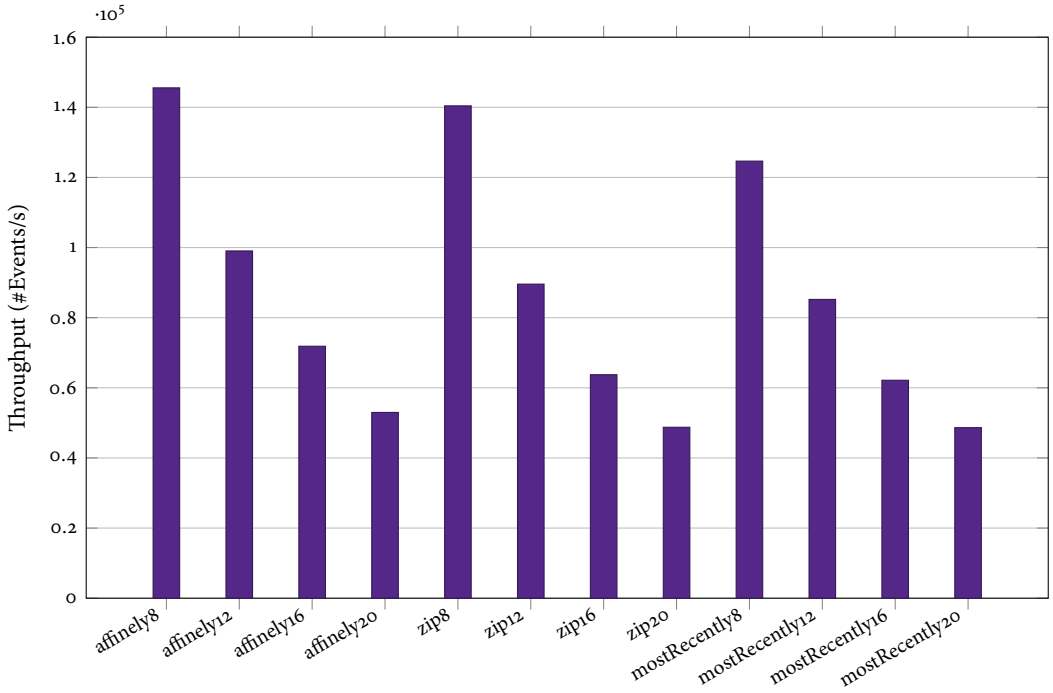
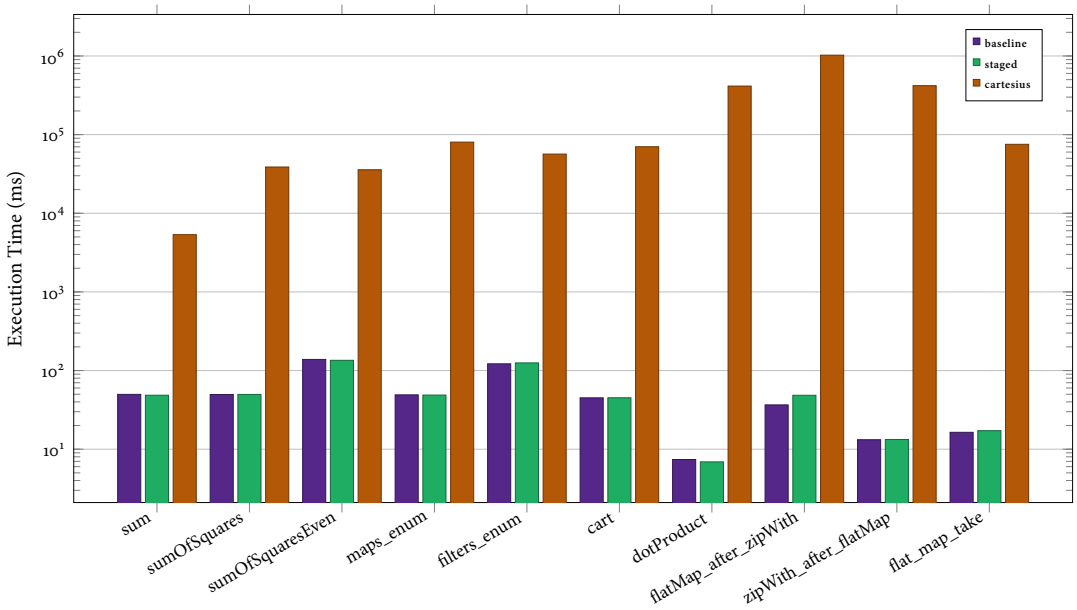


Figure 7.3: Microbenchmark: Mean throughput in number of events per second, for processing $N = 10,000,000$ input events (higher is better). We measured each n-way join variant against arity $n \in \{8, 12, 16, 20\}$, as indicated by the legend.

control flow restriction for enforcing a lock-step consumption of events, by suspending/resuming asynchronous producer strands. Higher *zip* arities tend to increase latency, though not by as much as in the *affinely* case. We do not have an explanation for the huge latency spike of *mostRecently* at arity 8.

While production rate and latency reveal varied behavior, they do not seem to be good predictors for the overall throughput (Figure 7.3). For instance, *affinely8* has the lowest production rate and highest latency, but the highest throughput. The dominant factor for throughput is the arity of the join. The higher the arity, the lower the throughput for all join variants.

The overall throughput measurements indicate that arity increase induces computational overhead. The reason for our belief is as follows: We have chosen a uniform distribution of events among the input streams and a round-robin scheduler for the underlying async/await implementation. Under these conditions, at least the *mostRecently* variant should be able to always be productive and react immediately upon the next event notification. Yet, even this variant exhibits reduced throughput under increasing arity. The likely culprit are effect handlers and the dispatch overhead of effects. Recall from Chapter 5 that we calculate n-deep stacks of effect handlers from context-polymorphic definitions. To the best of our knowledge, the multicore OCaml implementation of effects and handlers performs a linear search through the call stack to dispatch an effect



baseline	49.8	49.6	138.5	49.1	121.8	45	7.4	36.6	13.2	16.4
staged	48.6	49.7	135	48.8	125.2	45.1	6.9	48.5	13.3	17.2
cartesius	5,348.7	38,766.6	35,780.4	80,319.6	56,573.1	70,164.9	$4.14 \cdot 10^5$	$1.02 \cdot 10^6$	$4.19 \cdot 10^5$	75,335.4

Figure 7.4: Performance of Cartesius and the Strymonas library in the microbenchmark suite by Kiselyov et al. [Kis+17]. Mean total execution time in milliseconds for processing $N = 100,000,000$ events (y-axis, logarithmic scale, lower is better).

invocation to the right handler. Increasing the arity thus increases depth and dispatch time. We conclude that we need optimizations for improving effect dispatch in the polyvariadic Cartesius implementation, which we consider for future work.

7.3 COMPARISON AGAINST THE STRYMONAS LIBRARY

In this section, we investigate the performance gap between Cartesius and the state of the art, highly optimized Strymonas library ("Stream Fusion, to Perfection", by Kiselyov et al. [Kis+17]). Strymonas uses multi-stage programming (MetaOCaml, Kiselyov [Kis14]) to optimize the code of pull-stream pipelines defined with combinators and higher-order functions, so that it is on a par with optimal, hand-written versions using loops. Since optimizations are not the focus of this thesis, it is to be anticipated that results will not be in our favor. Nevertheless, it is useful to know how far we are way from perfection and what is the price of our modular and composable system design based on handlers.

EXPERIMENTS We recreate the Strymonas microbenchmark suite by Kiselyov et al. [Kis+17], it contains a number of integer-stream programs with different arrangements of stream combinators, e.g., compositions of map, filter, fold, and flatmap/bind.

- *sum*: sums up a stream by folding it.

- *sumOfSquares*: a stream map operation followed by fold.
- *sumOfSquaresEven*: a stream filter, followed by map, then fold.
- *cart*: the sum of the cartesian product on streams.
- *maps*: consecutive map operators on streams.
- *filters*: consecutive filter operators on streams.
- *dotProduct*: dot product, by folding after zipWith.
- *flatMap_after_zipWith*: applies zipWith + to a stream with itself, then computes the cartesian product with another.
- *zipWith_after_flatMap*: flips the operator order of the previous.
- *flat_map_take*: flatmap followed by take.

For each program, we measure the mean execution time.

INPUT We follow the input specification from the Strymonas paper, translated to push streams:

> All tests were run with the same input set. For the *sum*, *sumOfSquares*, *sumOfSquaresEven*, *maps*, *filters* we used an array of $N = 100,000,000$ small integers: $x_i = i \mod 10$. The *cart* test iterates over two arrays. An outer one of 10,000,000 integers and an inner one of 10. For the *dotProduct* we used 10,000,000 integers, for the *flatMap_after_zipWith* 10,000, for the *zipWith_after_flatMap* 10,000,000 and for the *flat_map_take* N numbers sub-sized by 20% of N. (Kiselyov et al. [Kis+17])

SETUP We executed the microbenchmarks on a 2018 15-inch Macbook Pro model, with 32GB DDR4 memory and a 2.6Ghz Intel Core i7 CPU. For obtaining results on Strymonas, we reused their OCaml version. Compiling and running their code required a different OCaml version: 4.02 with MetaOCaml. We were not able to compile their code with multicore OCaml.

We recreated the Strymonas benchmarks in CARTESIUS (multicore OCaml 4.06) for push streams. For implementing fold, we defined a custom handler of push events, that maintains a local state with the aggregated fold result. Just as in the Strymonas benchmark suite, for each experiment, we repeated the experiment 30 times and measured the wall-clock execution time, computing the mean in the end. To avoid influences from previous runs, each iteration would first invoke the OCaml garbage collector.

RESULTS Figure 7.4 reports the mean measured execution times in milliseconds on a logarithmic scale. Strymonas reports two versions: baseline and staged. The former is an optimized, hand-written version of each experiment, and the latter the result of the staging-based optimizations from their paper. Confirming the findings of their paper, hand-written and staged versions have nearly identical execution times, rarely exceeding 100ms time on our machine.

As anticipated, we are several orders of magnitude slower, from about 100 times slower in *sum*, up to about 60000 times in the *dot product*. We have abstraction overhead from the asynchrony system, e.g. the process scheduler of the async/await implementation, and the dispatch overhead of event notifications in terms of effect invocations, which is linear in the stacking-depth of handlers. Furthermore, we have a push/pull stream processing model, where the join computation maintains a growing memory of pushed event observations. In contrast, Strymonas works on arrays (pull) which have better cache locality, and thus more quickly are iterated over.

7.4 CHAPTER SUMMARY

In summary, our results demonstrate that restriction handlers work as intended. They cause changes in the behavior of the cartesian product that a priori do not materialize irrelevant event tuples. The abstraction overhead of our design and effect handlers is still very high, though. The polyvariadic CARTESIUS implementation induces overhead with increasing arity, which could be avoided with smarter effect dispatch strategies and better ways to generate handlers, e.g., fuse handlers into one big handler versus stacking multiple handlers. The comparison with Strymonas shows what "abstraction without guilt" might look like. For the future, we need multi-stage programming support for algebraic effects and handlers. No state of the art language implementation of effects and handlers supports staging, currently. In general, more research needs to be done in terms of data structures and evaluation strategies for getting better performance.

Part IV

EPILOGUE

CONCLUSION AND FUTURE WORK

8

8.1 SUMMARY AND CONCLUSIONS

The main objective of this thesis was to show that algebraic effects and handlers are a good abstraction for principled and versatile event correlation. They enabled us to overcome the "tower of babel syndrome" in the field, bringing a uniform, clear and effective model for correlating events. There is high variability in the design space for event correlation and our work embraces this variability. We successfully paved the way to the next generation of "à la carte" reactive systems, which are highly adaptable and highly customizable. Our work has additional utility as a model for studying fine-grained feature compositions and interactions between the different event correlation families. Last, but not least, not only did we approach event correlation from the angle of extensible semantics, but also from the angle of accompanying extensible syntax, resulting in an adequate technique for language integration of next generation reactive systems.

First, we investigated how to viably achieve a uniform and extensible semantics for event correlation computations with the aid of effects and handlers, resulting in Cartesius (Chapter 3), which we encoded in a λ-calculus with effects and handlers. We shifted the perspective of traditional views on data flow graphs from their topology to the internal behavior of nodes, identifying n-way joins as the fundamental class of operations underlying event correlation. Consequently, the focus of our efforts was on modeling the essence of joins.

We defined the computational interpretation of n-way joins as restricted variants of the cartesian product over n asynchronous streams of event notifications. Restrictions are effect handlers interacting with the cartesian product in the fashion of coroutines. This interaction with restrictions may induce changes in the behavior of the generic cartesian product. The control flow manipulations via effect handlers achieve that unnecessary n-tuples of event notifications are not materialized a priori. For example, the "combine latest" correlation exhibited by reactive programming languages materializes considerably fewer n-tuples than a naïve cartesian product followed by a filter. Operationally, variants of correlations have a specific "computational fingerprint", i.e., access and consumption patterns on n local mailboxes of event notifications.

Handlers are modular units that give effect operations an interpretation. At the same time, they are first-class notions of delimited control in the programming language. Since programming asynchronous computations involves control flow and continuations, handlers are an ideal abstraction for event correlation. The effect type signature in Definition 3.3 beautifully captures that event correlation is about handling of n distinct **push** effects (event notifications), which occur in any order and arbitrarily often. Programming effort for variants of event correlation focuses on defining and composing handlers for this effect signature/interface. This is already a useful interface for programming event

correlation, independently of our cartesian product interpretation. In the latter case, programmers additionally need to know about the **set** and **get** effects for accessing the mailboxes of local event observations. Overall, our model is simple, intuitive and practical for programming.

A stack of event-correlating handlers may introduce additional effects as needed, which can either be handled *internally*, by a handler upwards in the stack or *externally*, by the context outside of the correlation. An example of the former would be the ability to selectively suspend/resume one of the interleaved event producers (Section 3.4.4). The point is that our design is open to extension with new features and features can be freely composed. From the perspective of static typing, effect polymorphism in effect rows captures this extensibility. Together with effect subsumption, effect rows ensure that a given piece of code is compatible and composable in usage contexts with more effects. The locality of effect handlers ensures that different variants of event correlation can coexist in the same application.

Furthermore, the suspension/resumption capability introduces a second interaction point of the correlation, besides the cartesian product, this time with the asynchronous producers. This is useful to express some variants of correlation, e.g., *zip*, which require coordination with the producers to obtain their characteristic computational fingerprint.

Solving the issue of extensible event correlation semantics was just the first step. The next step was to cross the semantic gap from theory into practice, i.e., transitioning from a formal λ-calculus with effects to a real programming language implementation. In order to obtain a complete implementation of Cartesius, three issues needed to be resolved:

1. The formalization of Cartesius features a declarative correlation pattern syntax. The issue to solve was integrating this syntax into a real programming language implementation of algebraic effects and handlers, while keeping effort reasonable (e.g., avoid changing compiler implementations for the syntax support).

2. Cartesius requires notions of generative effects, i.e., the effect signatures for **push**, **get** and **set** are dependent on the number n and the types of input sources, which are heterogenously-typed. In the formalization, we assumed that each join definition is defined in a context with appropriately pre-defined effect declarations. However, in an actual implementation, we need linguistic concepts to allocate generative effects and define handlers for them. At the time of writing, not all state of the art programming language implementations of effects and handlers had support for generativity, e.g., Koka [Lei17b]. We implemented an early prototype in this language that supported only integer-valued events, because generativity is lacking. We were more successful with a multicore OCaml [Dol+17] version of the protoype, which expresses generative effects by means of first-class modules and thus joins over heterogeneously-typed event sources. One limitation remained, which is the next issue.

3. Besides heterogeneity, there is a second dimension to the generativity of the involved effect declarations: the dependence on the number n of input sources (arity) to join, i.e., an n-way join entails generating n distinct **push** effects declarations. While our multicore OCaml prototype supports heterogeneity, it does not support genericity over arity, and it is only able to join up to a hard-coded number of event sources.

We used the term *polyvariadicity* for the trait of simultaneous type-heterogeneity and arity-genericity induced by the last two issues. There is also a connection to the first issue, in the sense that the syntax too supports arbitrary arities and heterogeneity.

We first investigated the issue of bringing syntax support for declarative correlation patterns into programming languages (Chapter 4). Our initial hope was to utilize established state of the art techniques for language-integrated queries (e.g., LINQ [MBB06; CLW13]), or comprehensions. Unfortunately, they map queries to monads. We argued in detail why and how a translation of correlation patterns to monads is "bound" to fail: the way variable bindings are translated is problematic (Section 3.4.1 and 4.2). It limits the denotation of the syntax to event correlation computations with a priori fixed event selection order, prohibiting computations that react to event notifications "as they come". Hence, the monad-based translation does not accommodate the entire design space for event correlation systems. For this reason, we opted for handling/observing *interleavings* of event producers in Chapter 3, which is the right approach for event correlation. Thus, we developed the PolyJoin embedding for event correlation patterns, to accommodate our design.

In comparison to the state of the art monad-based approaches, PolyJoin permits more liberal interpretations of the variable bindings in patterns, including the interleaving resp. parallelism for dataflow and event correlation. This is due to the insight that correlation patterns and join syntax should be viewed as simultaneous binding forms of multiple variables, in contrast to single, nested bindings in monad-based approaches.

We implemented PolyJoin as a library-level embedding with the tagless final approach [CKS09] in (multicore) OCaml, which gives us an extensible abstract syntax for event correlation patterns that incorporates the type system of the patterns into the host language, thus preventing ill-typed and ill-formulated patterns. Especially, we did not have to change the compiler implementation to integrate the language of correlation patterns into the host language OCaml.

The tagless final approach models expression syntax as applications of functions from a signature/interface, leaving the denotation resp. interpretation completely abstract. For supporting correlation patterns, we extended the tagless final approach with support for simultaneous binding forms. Since the approach relies on higher-order abstract syntax (HOAS) [HL78; PE88], which reuses the host language's facilities for variable binding for the pattern variable bindings, we encoded simultaneous binding forms by first-class functions with polyvariadic type signatures.

The encoding of polyvariadic functions for correlation patterns requires type-level calculations that relate the types and number of input event sources to be joined with the type and number of variables in the pattern. We successfully embedded this calculation into OCaml's Hindley-Milner (HM) type system. It fully automatically and transparently computes the types, once the number and

types of event sources are known. This knowledge becomes available at the use site of a correlation pattern, once an end user specifies the event sources to be used in a correlation. No additional annotation burden is required from end users.

We proved that PolyJoin's correlation patterns are sound and that our encoding of polyvariadic functions adequately models the signature of the correlation pattern syntax, which is essentially a family of function types indexed by vectors of types (Section 4.5). Such a type vector serves as a description of the shape of the correlation pattern's variable context in the type system of the host language. Furthermore, keeping track of this index at the type level enables separation of polyvariadic signatures/interfaces from polyvariadic implementations. We can develop independent polyvariadic components and ensure that they safely compose by demanding they are instantiated at the same index. We used heterogeneous lists as term-level representations of polyvariadic components matching the type-level index.

While we had the practical need to support (multicore) OCaml for a complete Cartesius implementation, PolyJoin is in principle portable to other "mainstream" languages, because it has comparatively modest requirements on the host language's type system, e.g., we do not need dependent types for polyvariadic syntax forms, compared to some other solutions. We discussed matters of portability in Section 5.7.4.

Having solved the issue of portable syntax also gave us a solution for programming general polyvariadic definitions through indexing by the variable context shape. We finally had all the pieces to completely and satisfactorily lift Cartesius from theory into practice (Chapter 5), solving all of the three issues from above.

We combined the first-class module encoding of heterogeneously-typed effect declarations with heterogeneous lists to obtain polyvariadic generative effect declarations. Furthermore, we introduced the notion of *context polymorphism*, i.e., function definitions which are polymorphic over all possible context shapes of correlation patterns. These enable *polyvariadic effect handler definitions*. That means, we can write a single generic handler against a single abstract effect (e.g., **push**), which is representative for all concrete effect declarations in all concrete contexts of polyvariadic generative effect declarations. The polyvariadic handler can be instantiated to all concrete declaration contexts and simultaneously handle all the contained effects. Polyvariadic handlers are notationally very close to their respective formalizations in Chapter 3, where we used number indices to mark the context dependency. Importantly, our polyvariadic solution is a precise, concise, and type-safe programming language encoding of the formalization.

While context polymorphism is for handling all effects in a declaration context, we introduced *position polymorphism* as a refined notion, for projecting parts of a context and only handle parts of the effect declarations. This notion enabled concise and type-safe definitions for the restriction handlers that are composed and integrated with the generic cartesian product in correlations. We used an encoding of type-safe pointers and sets of pointers for projecting parts of contexts and ensuring that the composition and integration of restriction handlers into a correlation is safe and cannot refer to wrong or non-existent effect declarations.

A challenging issue was defining the generic cartesian product as part of the core framework of Cartesius polyvariadically (the *reify* handler, Section 3.4.2). The problem was that it requires a context-polymorphic handler that (1) abstracts over any particular context position and (2) accesses the rest of the (heterogeneous) context without that position and (3) performs a computation dependent on the current position and the rest of the context. This reflects that the correlation is subject to external choice by its environment, which may send event notifications (i.e., invoke a **push** effect) for any position in the context, in arbitrary order. We were able to solve this *focusing continuations problem* in a type-safe, polyvariadic manner, by exploiting that effect operations are dispatched dynamically (Section 5.5.6).

We conclude that polyvariadicity is a necessary accompanying phenomenon when reasoning about notions of joining and by extension event correlation. The (informal) mathematical practice of position-dependent definitions indexed by numbers must be made formally precise and linguistically expressible in a target programming language when transitioning from theory to practice. This holds especially true if the target language is strongly-typed, as in our case. General event correlation systems, such as Cartesius, need principled programming abstractions for safely and concisely programming polyvariadic, context-dependent definitions. We evaluated in Section 5.7.2 our encoding of polyvariadicity, context polymorphism and position polymorphism, by comparing against the initial prototype of Cartesius with hard-coded arities. The achieved exponential reduction in code size and the superior composability and extensibility of the polyvariadic version is evidence for the effectiveness of our programming abstractions.

Furthermore, we conclude that tagless interpreters and polyvariadic functions give rise to practical, portable, type-safe and extensible embedded syntax *frontends* for event correlation systems. PolyJoin provides evidence and a "design recipe" for accomplishing this. However, the portability of our *backend* based on algebraic effects and handlers depends on whether the target language is capable of expressing generative effect declarations by a secondary mechanism. In our case, we made use of OCaml's first-class modules to encapsulate generative effects declarations as a form of capability values. However, such a mechanism is not always given in a language, e.g., currently it does not seem possible to encode generative effect declarations in Frank [LMM17]. Integrating generative effects into the theory and language implementations of algebraic effects and handlers is nontrivial and is being actively researched. One of the first languages with effects and handlers, Eff by Bauer and Pretnar [BP15] originally featured generative effects (called instances), but the language has no effect types. Instances have been abandoned in newer Eff versions, due to design issues,[1] but its creators are investigating a new design for generative effects based on comodels [Bau18]. The more recent Helium language by Biernacki et al. [Bie+19] is an ML dialect featuring typed generative effects. In future work, we would like to investigate how Cartesius and PolyJoin integrate into their design.

We surveyed features related to joining across representative systems of all the event correlation families (Chapter 6). Our message is that we can cover relevant features around event joins from *across all event correlation families*, by a unifying model, based on a small foundation with clear semantics and

1 From a personal correspondence with Matija Pretnar, one of Eff's creators.

an extensible design. We thus conclude that we have captured the essence of event correlation. Due to the free composability of features and the locality of effect handlers, we obtain a highly adaptable and customizable event correlation system.

Finally, having a complete implementation of Cartesius, we measured and compared its performance (Chapter 7). Our results demonstrate that the handler-based realization of our computational interpretation imposes adequate control flow restrictions to avoid needless materialization of tuples, despite a seemingly inefficient modular decomposition into a generic cartesian product and extensions. However, we are orders of magnitude slower than state of the art, optimizing streaming libraries. This does not come as a surprise, because state of the art staging libraries are not yet available for language implementations of algebraic effects and handlers, and implementations are not yet mature enough. For example, our microbenchmarks show that nesting-depth of handlers impacts performance in multicore OCaml. We believe that our system provides a compelling and promising ground for research into optimizing compilers for effects and handlers.

8.2 FUTURE WORK

Our work opens up exciting opportunities for future research directions:

STRONG NORMALIZATION: GUARANTEEING LIVENESS, PRODUCTIVITY AND DEFORESTATION It seems useful and intriguing to eliminate data types and recursion and replace them as much as possible with computations and effect handlers. This way, we obtain a total functional language with extensible effects, exploiting the algebraic structure of the computation to encode data types in terms folds and replace recursion by induction, which comes "out of the box" with effects and handlers. There are good reasons to do this:

- Existing language implementations of effects and handlers implicitly assume an ambient divergence effect (e.g., Frank [LMM17]) or have ad-hoc measures to track divergence as effects (e.g., Koka [Lei17b]). In contrast, an encoding of (co)inductive types in a recursion-free effect language would enable guaranteed terminating code and enable a disciplined separation between total and diverging code. That is, if the effect system does not flag a term as divergent, then it truly should not be.
- Encoding data as effects opens pathways to a deforested (cf. Wadler [Wad90b] event correlation system, where needless construction of intermediary data structures is eliminated. This could help improve the performance of Cartesius.
- Well-typed, total effectful programs have a proposition-as-types/proofs-as-programs interpretation [Wad15]. Especially, (co)inductive types have been successfully utilized to define reactive programs that exhibit liveness (e.g., Cave et al. [Cav+14]). We could obtain a proofs-as-programs interpretation for event correlation systems.

While induction and folding come for free, it is unclear how coinduction, respectively unfolds (the foundation for demand-based streams, cf. Chapter 2) integrate with effects and handlers, without reintroducing problematic recursive definitions.

STAGING EFFECT HANDLERS – ABSTRACTION WITHOUT REGRET There are plenty of opportunities in the design and polyvariadic implementation of CARTESIUS to apply multi-stage programming in order to eliminate abstraction overhead. However, none of the state of the art multi-stage programming libraries, e.g., MetaOCaml (Kiselyov [Kis14]) and Lightweight modular staging (LMS, Rompf and Odersky [RO10]) support effects and handlers and have yet to catch up. One obvious application of staging is the calculation of polyvariadic restriction handlers, which we currently achieve by nesting first-class functions. Ideally, it should be possible to fuse all the stacks of calculated handlers into one handler.

In combination with the tagless final approach we utilize [CKS09], it should be possible to obtain a library-level optimization pipeline for POLYJOIN. In the case of LINQ, its feasibility has been demonstrated by Suzuki, Kiselyov, and Kameyama [SKK16], who show how the optimization rules for LINQ Cheney, Lindley, and Wadler [CLW13] can be expressed with tagless final and MetaOCaml as type-preserving transformations.

MORE PRINCIPLED CONTEXT ABSTRACTION WITH COMONADS AND COEFFECTS Some features of this work are related to context and computing in a context. For example, time windows define a context that limits the extent of a correlation. Furthermore, our notions of context and position polymorphism, as well as type-level indexing by an abstract context shape are expressions of context dependency and programming abstractions for ensuring context compatibility. The dual of effects, coeffects (Petricek, Orchard, and Mycroft [POM14]) as well as comonads (Orchard and Mycroft [OM12]) offer principled approaches for programming with context dependency. A practical integration of effects and coeffects is an interesting research direction.

Part V

APPENDIX

VERSATILE EVENT CORRELATION WITH ALGEBRAIC EFFECTS

A

A.1 EXAMPLE: EVENT CORRELATION IN RX.NET

Figure A.1 shows an excerpt of a larger example from the official repository of the .NET version of Reactive Extensions [NET13]. This examples shows how event correlations are programmed in this library, by using a mixture of LINQ comprehensions and ad-hoc applications of combinators when non-monadic variable binding semantics is required. I.e., the `CombineLatest` application in Lines 13-17 implements a parallel binding. The point is that the lack of customizable comprehension syntax requires ad-hoc applications of combinators of the underlying stream representation, leading to hard to understand correlation pattern specifications. Syntax and semantics become mixed, so that correlation specifications lack representation independence. I.e., a correlation specification could not be re-targeted to another semantic representation. It also becomes harder to apply algebraic optimizations as in the T-LINQ/P-LINQ work by Cheney, Lindley, and Wadler [CLW13].

```
var info =
  from beginAct in beginActivities
     let activityId = beginAct.Value.Id
     let endAct = endActivities.FirstAsync(e => e.Value.Id == activityId)
     let taskAs =
          from beginA in beginAs.TakeUntil(endAct).Where(a => a.Value.ActivityId ==
    activityId)
          from endA in endAs.FirstAsync(e => e.Value.Id == beginA.Value.Id)
          select new { Value = (int?)beginA.Value.Value, Start = beginA.Timestamp,
    End = endA.Timestamp }
     let taskBs =
          from beginB in beginBs.TakeUntil(endAct).Where(b => b.Value.ActivityId ==
    activityId)
          from endB in endBs.FirstAsync(e => e.Value.Id == beginB.Value.Id)
          select new { Value = beginB.Value.Value, Start = beginB.Timestamp, End =
    endB.Timestamp }
     from res in Observable.CombineLatest(
          endAct,
          taskAs.StartWith(/* ... */),
          taskBs.StartWith(/* ... */),
          (e, a, b) => new { e, a, b }).LastAsync()
     select new {
         Activity = activityId, StartTime = beginAct.Timestamp, EndTime = res.e.
    Timestamp,
          PayloadA = res.a.Value = null ? res.a.Value.ToString() : "(none)", DurationA
     = res.a.End - res.a.Start,
         PayloadB = res.b.Value ?? "(none)", DurationB = res.b.End - res.b.Start
     };
```

Figure A.1: Abridged Rx.NET Event Correlation Example from [NET13].

POLYJOIN

B

B.1 GENERIC OPERATIONS ON HETEROGENEOUS LISTS

We show the OCaml code for generic operations on heterogeneous lists in Figures B.1-B.4. Each operation is embodied by a second-class functor and parametric over one or more "uniformly heterogeneous" lists (cf. Section 4.6).

Some operations, i.e., mapping, folding and iteration, require first-class polymorphic functions. We use OCaml's polymorphic record types to encode first-class polymorphic values. For example, `HMap` accepts a polymorphic record `ftor` (Figure B.1), which uniformly maps elements that are in enclosed in the source hlist's element type to their counterpart enclosed in the target hlists's element.

B.2 SCALA VERSION OF POLYJOIN

To demonstrate the portability of PolyJoin, we show a Scala version in this section (Figures B.5-B.8). One notable aspect of this version is that it works well with local type inference [PT98], compared to the full type inference in OCaml. That is, the notation of pattern binders is at a similar level of conciseness and convenience as the OCaml version and does not require explicit type annotations by users.

B.3 CURRIED n-WAY JOINS

In this section, we show a version of PolyJoin with *curried* polyvariadic join patterns (Figure B.9). While currying in conjunction with the host language's pattern matching facilities leads to less syntactic noise in the HOAS variable bindings of patterns, it complicates programming of tagless interpreters, because the type signature of `join` hides information, i.e., the signature conceals that `'ps` in Figure B.9, Line 14 is a function.

```
module HMAP(S:Hl)(T:Hl) = struct
  type ftor = {f: 'a. 'a S.el -> 'a T.el}
  let rec map : type a. ftor -> a S.hlist -> a T.hlist =
    fun {f} -> function
      | S.Z -> T.Z
      | S.S (h,t) -> T.S (f h, map {f} t)
end
```

Figure B.1: OCaml Functor for Mapping over Heterogeneous Lists.

```
module HFOLD(H:Hl) = struct
  type 'a fold = {zero: 'a;
                  succ: 'b. 'b H.el -> (unit -> 'a) -> 'a }
  let rec fold: type a b. a fold -> b H.hlist -> a =
    fun {zero;succ} ->
      function
      | H.Z -> zero
      | H.S (x,t) -> succ x (fun () -> fold {zero;succ} t)
end
```

Figure B.2: OCaml Functor for Folding over Heterogeneous Lists.

```
module HFOREACH(H:Hl) = struct
  type foreach = {f: 'a. 'a H.el -> unit}
  let foreach: type a. foreach -> a H.hlist -> unit =
    fun {f} hs ->
      let module Units = HList(struct type 'a t = unit end) in
      let module M = HMAP(H)(Units) in
      ignore @@ M.map {M.f = f} hs
end
```

Figure B.3: OCaml Functor for Iterating over Heterogeneous Lists.

```
module HZIP(H1:Hl)(H2:Hl) = struct
  include HList(struct type 'a t = 'a H1.el * 'a H2.el end)
  let rec zip: type a. a H1.hlist -> a H2.hlist -> a hlist =
    fun h1 h2 ->  match h1, h2 with
      | H1.Z, H2.Z -> Z
      | H1.(S (x1,h1)), H2.(S (x2,h2)) -> S ((x1,x2), zip h1 h2)
end
```

Figure B.4: OCaml Functor for Zipping two Heterogeneous Lists.

```
import scala.language.higherKinds
//Module signatures become traits/interfaces
trait Symantics {
    type Ctx[A]
    type Var[A]
    type Repr[A]
    type Shape[A]
    def from[A](shape: Repr[Shape[A]]): Var[Repr[A]]
    val cnil: Ctx[Unit]
    def ccons[A,B](v: Var[A], ctx: Ctx[B]): Ctx[(A,B)]
    def join[A,B](ctx: Ctx[A])(pattern: A => Repr[Shape[B]]): Repr[Shape[B]]
    def yld[A](x: Repr[A]): Repr[Shape[A]]
    def pair[A,B](fst: Repr[A], snd: Repr[B]): Repr[(A,B)]
  }
```

Figure B.5: Scala Version of PolyJoin: Symantics Signature.

```
//For testing, extend the language with a syntax to lift lists of values to shape
     representations
trait SymanticsPlus extends Symantics {
   //note: A* means "variable number of A arguments"
   def lift[A](xs: A*): Repr[Shape[A]]
}

//Expression functors become functions/methods with path-dependent types
def test(s: SymanticsPlus) //infers path-dependent type s.Repr[s.Shape[(Double, (Int,
      String))]]
= {
  import s._
  //Note: with more effort, the syntax for constructing contexts can be made prettier
     , in infix notation
  val ctx = ccons(from(lift(1,2,3)), ccons(from(lift("one", "two")), ccons(from(lift
    (3.0,2.0,1.0)), cnil)))
  //Notational convenience is similar to the OCaml version
  join (ctx) { case (x,(y,(z,()))) =>
     yld(pair(z,pair(x,y)))
  }
  /* this wouldn't type check:
   join (ctx) { case (x,(y,())) =>
     yld(pair(z,pair(x,y)))
   }
   */
}
```

Figure B.6: Scala Version of PolyJoin: Example Expression.

```
//Example: sequential cartesian product over lists
object ListSymantics extends SymanticsPlus {
  //GADT becomes class hierarchy
  sealed abstract class Ctx[A]
  //GADT constructors become case classes/objects
  case object CNil extends Ctx[Unit]
  case class CCons[A,B](hd: Var[A], tl: Ctx[B]) extends Ctx[(A,B)] //refinement
    controlled by extends clause
  sealed abstract class Var[A]
  case class Bind[A](shape: Repr[Shape[A]]) extends Var[Repr[A]]
  override type Repr[A] = A
  override type Shape[A] = List[A]
  override def from[A](shape: Repr[Shape[A]]) = Bind(shape)
  override val cnil = CNil
  override def ccons[A, B](v: Var[A], ctx: Ctx[B]) = CCons(v,ctx)
  override def lift[A](xs: A*) = List(xs:_*)
  override def yld[A](x: Repr[A]) = List(x)
  override def pair[A, B](fst: Repr[A], snd: Repr[B]) = (fst,snd)

  /* This join implements the monadic sequential semantics */
  override def join[A, B](ctx: Ctx[A])(pattern: A => Repr[Shape[B]]) = ctx match {
    case CNil => pattern ()
    case CCons(Bind(xs), tl) =>
      xs.flatMap { x =>
        join (tl) { tuple => pattern (x,tuple) }
      }
  }
}
```

Figure B.7: Scala Version of PolyJoin: Example Tagless Interpreter.

```
test(ListSymantics)
/* prints res0: ListSymantics.Repr[ListSymantics.Shape[(Double, (Int, String))]] =
List((3.0,(1,one)), (2.0,(1,one)), (1.0,(1,one)), (3.0,(1,two)), (2.0,(1,two)),
(1.0,(1,two)), (3.0,(2,one)), (2.0,(2,one)), (1.0,(2,one)), (3.0,(2,two)),
(2.0,(2,two)), (1.0,(2,two)), (3.0,(3,one)), (2.0,(3,one)), (1.0,(3,one)),
(3.0,(3,two)), (2.0,(3,two)), (1.0,(3,two))) */
```

Figure B.8: Scala Version of PolyJoin: Example Usage.

```
module type Curried = sig
  type 'a repr
  type 'a shape
  type 'a pat
  type ('a,'b) ctx
  type ('a,'b) var
  val from: 'a shape repr -> ('a, 'a repr) var
  val cnil: (unit, (unit -> 'a pat) * 'a) ctx
  val (@.): ('a,'b) var -> ('c, 'd * 'e) ctx -> ('a * 'c, ('b -> 'd) * 'e) ctx
  val yield: 'a repr -> 'a pat

  (* Problem: it is not apparent that 'ps is a pattern abstraction, which forces
     a structurally inductive programming style over the context formation.  *)
  val join: ('shape, 'ps * 'res) ctx -> 'ps -> 'res shape repr
end

module TestCurried(C: Curried) = struct
  open C
  let test a b c =
    join ((from a) @. (from b) @. (from c) @. cnil)
      (fun x y z () -> (yield y))
end
```

Figure B.9: Curried n-way Join Signature in OCaml.

C

C.1 POLYVARIADIC IMPLEMENTATION OF THE ALIGNING HANDLER

```
(* Polymorphic shift/rest*)
type 'a polycont = { cont: 'b. ('a -> 'b) -> 'b }
effect PolyShift: 'a polycont -> 'a
let poly_shift f = perform (PolyShift f)
let poly_reset action =
  try action () with
  | effect (PolyShift f) k -> f.cont (fun x -> continue k x)

let aligning: type ctx xs a. (xs,ctx) mptr -> (ctx,a) chandler =
  (fun ptrs ctx suspensions ->
    let suspensions_ctx = SuspensionsPtr.mproj (ptrs ())
    suspensions in
    let ctx = SlotsPtr.mproj (ptrs ()) ctx in
    let module Cells = HList(struct type 'a t = 'a evt option ref
     end) in
    let module SyncState = HZIP(Slots)(Cells) in
    let sync_state =
      let module M = HMAP(Slots)(SyncState) in
      M.map {M.f = fun s -> (s,ref None)} ctx
    in
    let reset_sync_state =
      let module M = HFOREACH(SyncState) in
      fun () -> M.foreach {M.f = fun (_,x) -> x := None}
    sync_state
    in
    let check_sync =
      let module F = HFOLD(SyncState) in
      let is_defined = function None -> false | _ -> true in
      fun () -> F.fold {F.zero = true;
                        F.succ = (fun (_,x) rest ->
                                    is_defined !x) && (rest ())
    )}
                sync_state
    in
    let push (type a) (s: a slot) x =
      let module S = (val s) in S.push x
    in
    let push_all =
      let module F = HFOLD(SyncState) in
      F.fold {F.zero = (fun () -> ());
              F.succ = (fun (s,st) rest ->
                let next = rest () in
```

```
39                (fun () ->
40                    match !st with
41                    | Some x ->  push s x; next ()
42                    | _ -> failwith "aligning: illegal sync state
       in push_all")
43           )} sync_state
44      in
45      let play_all =
46        let module M = HFOREACH(Suspensions) in
47        fun () -> M.foreach {M.f = fun s -> s.play ()}
       suspensions_ctx
48      in
49      let set st evt = match !st with
50        | None -> st := Some evt
51        | _ -> failwith "aligning: strand was already observed"
52      in
53      let try_release k =
54        if check_sync () then
55          begin
56            poly_shift { cont = (fun cb ->
57                push_all ();
58                cb ()) };
59            reset_sync_state ();
60            play_all ();
61            k ()
62          end
63        else k ()
64      in
65      let rec gen_handler: type c. c Suspensions.hlist ->
66                                   c SyncState.hlist -> a handler
67       = fun ss sync ->
68        match ss, sync with
69        | Suspensions.Z, SyncState.Z -> id_handler
70        | Suspensions.(S (s,ss)), SyncState.(S ((slot,st),sts)) ->
71            let module S = (val slot) in
72            let handler action =
73              try action () with
74              | effect (S.Push x) k ->
75                 s.pause (); set st x; try_release (continue k)
76            in
77            handler |+| (gen_handler ss sts)
78      in
79      poly_reset |+| (gen_handler suspensions_ctx sync_state))
```

[NET13] .NET Foundation. *Rx.NET Event Correlation Example*. Sept. 2013. URL: http://web.archive.org/web/20190520143253/https://github.com/dotnet/reactive/blob/master/Rx.NET/Samples/EventCorrelationSample/EventCorrelationSample/Program.cs#L55.

[AS96] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs, Second Edition*. MIT Press, 1996. ISBN: 0262011530.

[Agh90] Gul A. Agha. *ACTORS - A model of concurrent computation in distributed systems*. MIT Press, 1990. ISBN: 9780262010924.

[Agr+08] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. "Efficient pattern matching over event streams." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2008.

[Aki+15] Tyler Akidau et al. "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing." In: *Proceedings of the VLDB Endowment* 12 (2015), pp. 1792–1803.

[ABW04] Arvind Arasu, Shivnath Babu, and Jennifer Widom. "CQL: A language for continuous queries over streams and relations." In: *Database Programming Languages*. Springer Berlin Heidelberg, 2004.

[ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. "The CQL continuous query language: semantic foundations and query execution." In: *The VLDB Journal* (2006).

[AW04] Arvind Arasu and Jennifer Widom. "A denotational semantics for continuous queries over streams and relations." In: *SIGMOD Record* (2004).

[Ari+95] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. "The call-by-need lambda calculus." In: *Proceedings of Symposium on Principles of Programming Languages (POPL)*. 1995.

[AVW93] Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent programming in ERLANG*. Prentice Hall, 1993. ISBN: 9780132857925.

[Asa09] Kenichi Asai. "On typing delimited continuations: Three new solutions to the printf problem." In: *Higher-Order and Symbolic Computation* 22.3 (2009), pp. 275–291.

[Awo10] Steve Awodey. *Category theory*. Oxford University Press, 2010.

[Bai+13] Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. "A survey on reactive programming." In: *ACM Computing Surveys* 45.4 (2013).

[BH77] Henry C. Baker Jr. and Carl Hewitt. "The incremental garbage collection of processes." In: *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*. ACM, 1977.

[Bar84] Hendrik Pieter Barendregt. *The lambda calculus. Its syntax and semantics*. Studies in Logic. Elsevier Science Publishers B.V., 1984.

[Bau18] Andrej Bauer. "What is algebraic about algebraic effects and handlers?" In: *CoRR* abs/1807.05923 (2018). arXiv: 1807.05923. URL: http://arxiv.org/abs/1807.05923.

[Bau+16] Andrej Bauer, Martin Hofmann, Matija Pretnar, and Jeremy Yallop. "From Theory to Practice of Algebraic Effects and Handlers (Dagstuhl Seminar 16112)." In: *Dagstuhl Reports* 6.3 (2016). Ed. by Andrej Bauer, Martin Hofmann, Matija Pretnar, and Jeremy Yallop, pp. 44–58. ISSN: 2192-5283.

[BP15] Andrej Bauer and Matija Pretnar. "Programming with Algebraic Effects and Handlers." In: *Journal of Logical and Algebraic Methods in Programming* 84.1 (2015), pp. 108–123.

[Bel99] Bell Labs. *Advice from Doug McIlroy*. May 1999. URL: http://web.archive.org/web/20060103055557/http://cm.bell-labs.com/cm/cs/who/dmr/mdmpipe.html.

[BCF04] Nick Benton, Luca Cardelli, and Cédric Fournet. "Modern concurrency abstractions for C#." In: *Transactions on Programming Languages and Systems (TOPLAS)* 26.5 (2004), pp. 769–804.

[Ber00] Gérard Berry. "The foundations of Esterel." In: *Proof, Language, and Interaction: Essays in Honour of Robin Milner (Foundations of Computing)*. Ed. by Gordon Plotkin, Colin P. Stirling, and Mads Tofte. MIT Press, 2000. Chap. 14.

[Bib+15] Aggelos Biboudis, Nick Palladinos, George Fourtounis, and Yannis Smaragdakis. "Streams à la carte: extensible pipelines with object algebras." In: *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*. 2015.

[Bie+12] Gavin Bierman, Claudio Russo, Geoffrey Mainland, Erik Meijer, and Mads Torgersen. "Pause 'n' play: Formalizing asynchronous C♯." In: *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*. 2012.

[Bie+18] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. "Handle with care: Relational interpretation of algebraic effects and handlers." In: *Proceedings of the ACM on Programming Languages (PACMPL)* Symposium on Principles of Programming Languages (POPL) (2018).

[Bie+19] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. "Abstracting algebraic effects." In: *Proceedings of the ACM on Programming Languages (PACMPL)* 3.Symposium on Principles of Programming Languages (POPL) (2019).

[Bin+18] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. "Pyro: Deep universal probabilistic programming." In: *Journal of Machine Learning Research* (2018).

[BB85] Corrado Böhm and Alessandro Berarducci. "Automatic synthesis of typed Λ-programs on term algebras." In: *Theoretical Computer Science* 39 (1985).

[Bot+10] Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura Haas, Renée J. Miller, and Nesime Tatbul. "SECRET: A model for analysis of the execution semantics of stream processing systems." In: *Proceedings of the VLDB Endowment* (2010).

[Bra+18] Oliver Bračevac, Nada Amin, Guido Salvaneschi, Sebastian Erdweg, Patrick Eugster, and Mira Mezini. "Versatile event correlation with algebraic effects." In: *Proceedings of the ACM on Programming Languages (PACMPL)* 2.International Conference on Functional Programming (ICFP) (Sept. 2018).

[Bra+16] Oliver Bračevac, Sebastian Erdweg, Guido Salvaneschi, and Mira Mezini. "CPL: A core language for cloud computing." In: *Proceedings of the International Conference on Modularity*. 2016.

[Bra+19] Oliver Bračevac, Guido Salvaneschi, Sebastian Erdweg, and Mira Mezini. "Type-safe, polyvariadic event correlation." In: *CoRR* abs/1907.02990 (2019). arXiv: 1907.02990. URL: http://arxiv.org/abs/1907.02990.

[Bru72] Nicolaas Govert de Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem." In: *Indagationes Mathematicae (Proceedings)*. 1972.

[CKS09] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. "Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages." In: *Journal of Functional Programming* 19.5 (2009), pp. 509–543.

[CF94] Robert Cartwright and Matthias Felleisen. "Extensible denotational language specifications." In: *Proceedings of the International Conference on Theoretical Aspects of Computer Software (TACS)*. 1994.

[Cas+87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. "Lustre: A declarative language for programming synchronous systems." In: *Proceedings of Symposium on Principles of Programming Languages (POPL)*. 1987.

[Cav+14] Andrew Cave, Francisco Ferreira, Prakash Panangaden, and Brigitte Pientka. "Fair reactive programming." In: *Proceedings of Symposium on Principles of Programming Languages (POPL)*. 2014.

[CM94] S. Chakravarthy and D. Mishra. "Snoop: An expressive event specification language for active databases." In: *Data & Knowledge Engineering* 14.1 (1994), pp. 1–26.

[Cha+94] Sharma Chakravarthy, V. Krishnaprasad, Eman Anwar, and S.-K. Kim. "Composite events for active databases: semantics, contexts and detection." In: *Proceedings of International Conference on Very Large Data Bases (VLDB)*. 1994.

[Cha+14] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, John C. Platt, James F. Terwilliger, and John Wernsing. "Trill: A high-performance incremental query processor for diverse analytics." In: *Proceedings of the VLDB Endowment* 8.4 (2014), pp. 401–412.

[CH03] James Cheney and Ralf Hinze. *First-class phantom types*. Tech. rep. Cornell University, 2003.

[CLW13] James Cheney, Sam Lindley, and Philip Wadler. "A practical theory of language-integrated query." In: *Proceedings of International Conference on Functional Programming (ICFP)*. 2013.

[Chu36] Alonzo Church. "An unsolvable problem of elementary number theory." In: *American Journal of Mathematics* 58.345-363 (Apr. 1936).

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion*. 6. Princeton University Press, 1941.

[Cla99] Koen Claessen. "A poor man's concurrency monad." In: *Journal of Functional Programming* 9.3 (1999).

[Cle14] Xavier Clerc. *OCaml-Java*. Jan. 2014. URL: http://web.archive.org/web/20190714082529/http://ocamljava.org.

[CL99] Silvain Conchon and Fabrice Le Fessant. "JoCaml: Mobile agents for Objective-Caml." In: *First and Third International Symposium on Agent Systems Applications, and Mobile Agents (ASAMA)*. 1999.

[Con63] Melvin E. Conway. "Design of a separable transition-diagram compiler." In: *Communications of the ACM* 6.7 (1963).

[Coo09] Roy T. Cook. *A Dictionary of Philisophical Logic*. Edinburgh University Press Ltd, 2009. ISBN: 9780748625598.

[Coo08] Gregory H. Cooper. "Integrating Dataflow Evaluation into a Practical Higher-order Call-by-value Language." PhD thesis. 2008.

[CK06] Gregory H. Cooper and Shriram Krishnamurthi. "Embedding dynamic dataflow in a call-by-value language." In: *Proceedings of the European Conference on Programming Languages and Systems (ESOP)*. 2006.

[CM84] George Copeland and David Maier. "Making smalltalk a database system." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1984.

[CM10] Gianpaolo Cugola and Alessandro Margara. "TESLA: A formally defined event specification language." In: *Proceedings of the International Conference on Distributed Event-based Systems (DEBS)*. 2010.

[CM12] Gianpaolo Cugola and Alessandro Margara. "Processing flows of information: From data stream to complex event processing." In: *ACM Computing Surveys* 44.3 (2012), 15:1–15:62.

[CC13] Evan Czaplicki and Stephen Chong. "Asynchronous functional reactive programming for GUIs." In: *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*. 2013.

[Dan98] Olivier Danvy. "Functional unparsing." In: *Journal of Functional Programming* 8.6 (1998), pp. 621–625.

[DF92] Olivier Danvy and Andrzej Filinski. "Representing control: A study of the CPS transformation." In: *Mathematical Structures in Computer Science* 2.4 (1992).

[Dem+06] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. "Towards expressive publish/subscribe systems." In: *Proceedings of the International Conference on Advances in Database Technology (EDBT)*. 2006.

[DIG07] Yanlei Diao, Neil Immerman, and Daniel Gyllstrom. *SASE+: An Agile Language for Kleene Closure Over Event Streams*. Tech. rep. UMass Technical Report 07, 2007.

[Din+13] Nihal Dindar, Nesime Tatbul, Renée J. Miller, Laura M. Haas, and Irina Botan. "Modeling the execution semantics of stream processing engines with SECRET." In: *The VLDB Journal* (2013).

[Dol+17] Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, KC Sivaramakrishnan, and Leo White. "Concurrent system programming with effect handlers." In: *Proceedings of the Symposium on Trends in Functional Programming*. 2017.

[Dow+19] Paul Downen, Zachary Sullivan, Zena M. Ariola, and Simon Peyton Jones. "Codata in action." In: *Proceedings of the European Conference on Programming Languages and Systems (ESOP)*. 2019.

[Dre+18] Joscha Drechsler, Ragnar Mogk, Guido Salvaneschi, and Mira Mezini. "Thread-safe reactive programming." In: *Proceedings of the ACM on Programming Languages (PACMPL)* 2.Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) (2018).

[Dre+14] Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. "Distributed REScala: An update algorithm for distributed reactive programming." In: *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2014.

[Edw09] Jonathan Edwards. "Coherent reaction." In: *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2009.

[Ell09] Conal M. Elliott. "Push-pull functional reactive programming." In: *Proceedings of Haskell Symposium*. 2009.

[EH97] Conal Elliott and Paul Hudak. "Functional reactive animation." In: *Proceedings of International Conference on Functional Programming (ICFP)*. 1997.

[EPF04] EPFL. *Scala Programming Language*. 20 2004. URL: http://web.archive.org/web/20190507141826/https://www.scala-lang.org.

[EPF13] EPFL. *Scala Async*. Oct. 2013. URL: http://web.archive.org/web/20180611025539/https://github.com/scala/scala-async.

[Esp06] EsperTech Inc. *Esper*. 2006. URL: https://web.archive.org/web/20180422212134/http://www.espertech.com/esper.

[Eug+03] Patrick Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. "The many faces of publish/subscribe." In: *ACM Computing Surveys* 35.2 (2003).

[EJ09] Patrick Eugster and K.R. Jayaram. "EventJava: An extension of Java for event correlation." In: *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*. 2009.

[FH92] Matthias Felleisen and Robert Hieb. "The revised report on the syntactic theories of sequential control and state." In: *Theoretical Computer Science* 103.2 (1992), pp. 235–271.

[Fis72] Michael J. Fischer. "Lambda calculus schemata." In: *Proceedings of ACM Conference on Proving Assertions About Programs*. 1972.

[Flu+08] Matthew Fluet, Mike Rainey, John H. Reppy, and Adam Shaw. "Implicitly-threaded parallelism in manticore." In: *Proceedings of International Conference on Functional Programming (ICFP)*. 2008.

[For+17] Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. "On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control." In: *Proceedings of the ACM on Programming Languages (PACMPL)* 1.International Conference on Functional Programming (ICFP) (2017).

[Fou19] Inria Foundation. *The Coq Proof Assistant*. June 2019. URL: http://web.archive.org/web/20190628000927/https://coq.inria.fr/.

[Fou11] Kotlin Foundation. *Kotlin Programming Language*. July 2011. URL: http://web.archive.org/web/20190508091502/https://kotlinlang.org.

[Fou10] Mozilla Foundation. *Rust Programming Language*. July 2010. URL: http://web.archive.org/web/20190508210641/https://www.rust-lang.org.

[Fou98] Cédric Fournet. "The Join-Calculus: A Calculus for Distributed Mobile Programming." PhD thesis. Ecole Polytechnique, Palaiseau, INRIA TU-0556, Nov. 1998.

[FG96] Cédric Fournet and Georges Gonthier. "The reflexive CHAM and the Join-calculus." In: *Proceedings of Symposium on Principles of Programming Languages (POPL)*. 1996.

[FLW17] Simon Fowler, Sam Lindley, and Philip Wadler. "Mixing metaphors: Actors as channels and channels as actors." In: *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*. 2017.

[FZ97] Michael J. Franklin and Stanley B. Zdonik. "A framework for scalbale dissemination-based systems." In: *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 1997.

[FI00] Daniel Fridlender and Mia Indrika. "Do we need dependent types?" In: *Journal of Functional Programming* (2000).

[Fri+18] Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, and Jason Hemann. *The Reasoned Schemer*. Second. MIT Press, 2018. ISBN: 9780262562140.

[FW78] Daniel P. Friedman and David S. Wise. "Aspects of applicative programming for parallel processing." In: *IEEE Transactions on Computers* 27.4 (1978).

[Gam+94] Erich Gamma, Richar Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN: 0201633612.

[Geu92] Herman Geuvers. "Inductive and coinductive types with iteration and recursion." In: *Proceedings of the Workshop on Types for Proofs and Programs (TYPES)*. Båstad, Sweden, 1992.

[Geu15] Herman Geuvers. *The Church-Scott representation of inductive and coinductive data*. 2015. URL: https://web.archive.org/web/20190506145245/http://www.cs.ru.nl/~herman/PUBS/ChurchScottDataTypes.pdf.

[Gir72] Jean-Yves Girard. "Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur." PhD thesis. Université Paris VII, 1972.

[Gir87] Jean-Yves Girard. "Linear logic." In: *Theoretical Computer Science* 50 (1987).

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, 1989. ISBN: 0521371813.

[Gon12] Gabriel Gonzalez. *The Continuation Monad*. Dec. 2012. URL: http://web.archive.org/web/20190704223828/http://www.haskellforall.com/2012/12/the-continuation-monad.html.

[Goo11] Google. *Dart Programming Language*. Oct. 2011. URL: http://web.archive.org/web/20190508031306/https://dart.dev.

[HC08] Philipp Haller and Tom Van Cutsem. "Implementing joins using extensible pattern matching." In: *Coordination Models and Languages*. Ed. by Doug Lea and Gianluigi Zavattaro. Vol. 5052. Lecture Notes in Computer Science. Springer-Verlag, 2008, pp. 135–152.

[HM19] Philipp Haller and Heather Miller. "A reduction semantics for direct-style asynchronous observables." In: *Journal of Logical and Algebraic Methods in Programming* 105 (2019).

[Ham+14] Jurgen M. Van Ham, Guido Salvaneschi, Mira Mezini, and Jacques Noyé. "JEScala: Modular coordination with declarative events and joins." In: *Proceedings of the International Conference on Modularity*. 2014.

[HPG09] Peter Hancock, Dirk Pattinson, and Neil Ghani. "Representations of stream processors using nested fixed points." In: *Proceedings of Annual Symposium on Logic in Computer Science (LICS)* 5.3 (2009).

[Har16] Robert Harper. *Practical Foundations for Programming Languages*. Second. Cambridge University Press, 2016.

[Has08] Haskell Wiki. *MonadPlus*. July 2008. URL: http://web.archive.org/web/20190501035337/https://wiki.haskell.org/MonadPlus.

[Has19] Haskell.org. *The Haskell Programming Language*. June 2019. URL: http://web.archive.org/web/20190626003259/https://www.haskell.org.

[HBS73] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. "A universal modular ACTOR formalism for artificial intelligence." In: *Proceedings of the International Joint Conference on Artificial Intelligence*. 1973.

[HL18] Daniel Hillerström and Sam Lindley. "Shallow effect handlers." In: *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS)*. 2018.

[Hir+13] Martin Hirzel, Robert Soulé, Scott Schneider, Bugra Gedik, and Robert Grimm. "A catalog of stream processing optimizations." In: *ACM Computing Surveys* (2013).

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. ISBN: 0131532715.

[Hof+08] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. "Polymorphic embedding of DSLs." In: *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*. 2008.

[HMU06] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. *Automata theory, languages, and computation*. Pearson, 2006.

[Hud96] Paul Hudak. "Building domain-specific embedded languages." In: *ACM Computing Surveys* 28.4es (1996), p. 196.

[Hue97] Gérard Huet. "The zipper." In: *Journal of Functional Programming* (1997).

[HL78] Gérard Huet and Bernard Lang. "Proving and applying program transformations expressed with second-order patterns." In: *Acta Informatica* (1978).

[Hug89] John Hughes. "Why functional programming matters." In: *The Computer Journal* 32.2 (1989).

[Hug00] John Hughes. "Generalising monads to arrows." In: *Science of Computer Programming* 37.1-3 (2000).

[Hut99] Graham Hutton. "A tutorial on the universality and expressiveness of fold." In: *Journal of Functional Programming* 9.4 (1999).

[Jac16] Bart Jacobs. *Introduction to Coalgebra*. Cambridge University Press, 2016. ISBN: 9781316823187.

[Jef12] Alan Jeffrey. "LTL types FRP: Linear-time temporal logic propositions as types, proofs as functional reactive programs." In: *Proceedings of the Workshop on Programming Languages Meets Program Verification (PLPV)*. 2012.

[JG08] Patricia Johann and Neil Ghani. "Foundations for structured programming with GADTs." In: *Proceedings of Symposium on Principles of Programming Languages (POPL)*. 2008.

[KLO13] Ohad Kammar, Sam Lindley, and Nicolas Oury. "Handlers in action." In: *Proceedings of International Conference on Functional Programming (ICFP)*. 2013.

[KM05] Gregor Kiczales and Mira Mezini. "Aspect-oriented programming and modular reasoning." In: *Proceedings of International Conference on Software Engineering (ICSE)*. 2005.

[Kis14] Oleg Kiselyov. "The design and implementation of BER MetaOCaml - system description." In: *International Symposium on Functional and Logic Programming (FLOPS)*. 2014.

[Kis15] Oleg Kiselyov. *Polyvariadic functions and keyword arguments: Pattern-matching on the type of the context*. June 2015. URL: https://web.archive.org/web/20181013042601/http://okmij.org/ftp/Haskell/polyvariadic.html.

[Kis17] Oleg Kiselyov. *Having an Effect*. Sept. 2017. URL: http://web.archive.org/web/20181203202954/http://okmij.org/ftp/Computation/having-effect.html.

[Kis+17] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. "Stream fusion, to completeness." In: *Proceedings of Symposium on Principles of Programming Languages (POPL)*. 2017.

[KI15] Oleg Kiselyov and Hiromi Ishii. "Freer monads, more extensible effects." In: *Proceedings of Haskell Symposium*. 2015.

[KLS04] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. "Strongly typed heterogeneous collections." In: *Proceedings of Haskell Workshop*. 2004.

[KS09] Jürgen Krämer and Bernhard Seeger. "Semantics and implementation of continuous sliding window queries over data streams." In: *ACM Transactions on Database Systems* 34.1 (2009), pp. 1–49.

[Kri13] Neelakantan R. Krishnaswami. "Higher-order functional reactive programming without spacetime leaks." In: *Proceedings of International Conference on Functional Programming (ICFP)*. 2013.

[Lan65] Peter J. Landin. "Correspondence between ALGOL 60 and church's lambda-notation: part I." In: *Communications of the ACM* 8.2 (1965).

[Lei17a] Daan Leijen. "Structured asynchrony with algebraic effects." In: *Proceedings of the International Workshop on Type-Driven Development*. 2017.

[Lei17b] Daan Leijen. "Type directed compilation of row-typed algebraic effects." In: *Proceedings of Symposium on Principles of Programming Languages (POPL)*. 2017.

[LM99] Daan Leijen and Erik Meijer. "Domain specific embedded compilers." In: *Proceedings of the Conference on Domain-specific Languages (DSL)*. 1999.

[Lev04] Paul Blain Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis*. Vol. 2. Semantics Structures in Computation. Springer, 2004. ISBN: 1402017308.

[Lew+00] Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark B. Shields. "Implicit parameters: Dynamic scoping with static types." In: *Proceedings of Symposium on Principles of Programming Languages (POPL)*. 2000.

[Li+08] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. "Out-of-order processing: A new architecture for high-performance stream systems." In: *Proceedings of the VLDB Endowment* 1.1 (2008).

[LHJ95] Sheng Liang, Paul Hudak, and Mark P. Jones. "Monad transformers and modular interpreters." In: *Proceedings of Symposium on Principles of Programming Languages (POPL)*. 1995.

[Lin08] Sam Lindley. "Many holes in Hindley-Milner." In: *Proceedings of the ACM Workshop on ML*. 2008.

[Lin14] Sam Lindley. "Algebraic effects and effect handlers for idioms and arrows." In: *Proceedings of Workshop on Generic Programming, (WGP)*. 2014.

[LMM17] Sam Lindley, Conor McBride, and Craig McLaughlin. "Do be do be do." In: *Proceedings of Symposium on Principles of Programming Languages (POPL)*. 2017.

[Lis93] Barbara Liskov. "A history of CLU." In: *History of Programming Languages Conference (HOPL-II), Preprints*. 1993.

[LS88] Barbara Liskov and Liuba Shrira. "Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems." In: *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*. 1988.

[Lis+77] Barbara Liskov, Alan Snyder, Russell R. Atkinson, and Craig Schaffert. "Abstraction mechanisms in CLU." In: *Communications of the ACM* 20.8 (1977).

[LH07] Hai Liu and Paul Hudak. "Plugging a space leak with an arrow." In: (2007). Festschrift honoring Gary Lindstrom on his retirement from the University of Utah after 30 years of service, pp. 29–45.

[Luc+95] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. "Specification and analysis of system architecture using rapide." In: *IEEE Transactions on Software Engineering* 21.4 (1995), pp. 336–354.

[Luc01] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN: 0201727897.

[Mai13] Ingo Maier. "Reactive Programming Abstractions for Complex Event Logic and Dynamic Data Dependencies." PhD thesis. EPFL – IIF, 2013.

[MRO10] Ingo Maier, Tiark Rompf, and Martin Odersky. *Deprecating the Observer Pattern*. Tech. rep. EPFL, 2010. URL: http://web.archive.org/web/20150228000126/http://infoscience.epfl.ch/record/148043.

[MM14] Louis Mandel and Luc Maranget. *The JoCaml language, Documentation and user's manual, Chapter on Concurrent programming*. 2014. URL: http://web.archive.org/web/20170621093357/http://jocaml.inria.fr/doc/concurrent.html.

[Mar+14] Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. "There is no fork: An abstraction for efficient, concurrent, and concise data access." In: *Proceedings of International Conference on Functional Programming (ICFP)*. 2014.

[Mar+16] Simon Marlow, Simon Peyton Jones, Edward Kmett, and Andrey Mokhov. "Desugaring haskell's do-notation into applicative operations." In: *Proceedings of Haskell Symposium*. 2016.

[McB02] Conor McBride. "Faking it: Simulating dependent types in haskell." In: *Journal of Functional Programming* 12.4&5 (2002), pp. 375–392.

[MP08] Conor McBride and Ross Paterson. "Applicative programming with effects." In: *Journal of Functional Programming* (2008).

[Mei12] Erik Meijer. "Your mouse is a database." In: *Communications of the ACM* 55.5 (2012).

[MBB06] Erik Meijer, Brian Beckman, and Gavin Bierman. "LINQ: Reconciling object, relations and XML in the .NET framework." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2006.

[Men87] Paul Francis Mendler. "Inductive Definition in Type Theory." PhD thesis. Cornell University, USA, 1987.

[MW85] Albert R. Meyer and Mitchell Wand. "Continuation semantics in typed lambda-calculi (summary)." In: *Proceedings of Conference on Logics of Programs*. 1985.

[Mey+09] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. "Flapjax: A programming language for Ajax applications." In: *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2009.

[Mil78] Robin Milner. "A theory of type polymorphism in programming." In: *Journal of Computer and System Sciences* 17.3 (1978).

[Mil99] Robin Milner. *Communicating and Mobile systems: The Pi-Calculus*. Cambridge University Press, 1999. ISBN: 9780521658690.

[MP85] John C. Mitchell and Gordon D. Plotkin. "Abstract types have existential type." In: *Proceedings of Symposium on Principles of Programming Languages (POPL)*. 1985.

[Mit13] Neil Mitchell. "Leaking space." In: *ACM Queue* 11.9 (Sept. 2013), 10:10–10:23.

[Mog89] Eugenio Moggi. "Computational lambda-calculus and monads." In: *Proceedings of Annual Symposium on Logic in Computer Science (LICS)*. 1989.

[Mog91] Eugenio Moggi. "Notions of computation and monads." In: *Information and computation* 93.1 (1991), pp. 55–92.

[Mor98] Luc Moreau. "A syntactic theory of dynamic binding." In: *Higher-Order and Symbolic Computation* 11.3 (1998).

[MI09] Ana Lúcia de Moura and Roberto Ierusalimschy. "Revisiting coroutines." In: *Transactions on Programming Languages and Systems (TOPLAS)* (2009).

[MOP16] Alan Mycroft, Dominic A. Orchard, and Tomas Petricek. "Effect systems revisited - control-flow algebra and semantics." In: *Semantics, Logics, and Calculi - Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthdays*. 2016.

[Naj+16] Shayan Najd, Sam Lindley, Josef Svenningsson, and Philip Wadler. "Everything old is new again: Quoted domain-specific languages." In: *Proceedings of Workshop on Partial Evaluation and Program Manipulation (PEPM)*. 2016.

[NCP02] Henrik Nilsson, Antony Courtney, and John Peterson. "Functional reactive programming, continued." In: *Proceedings of Haskell Workshop*. 2002.

[Nor07] Ulf Norell. "Towards a practical programming language based on dependent type theory." PhD thesis. SE-412 96 Göteborg, Sweden: Department of Computer Science and Engineering, Chalmers University of Technology, 2007.

[Oca19] Ocaml.org. *OCaml Programming Language*. May 2019. URL: http://web.archive.org/web/20190508204148/https://ocaml.org/.

[Ode+18] Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. "Simplicitly: Foundations and applications of implicit function types." In: *Proceedings of the ACM on Programming Languages (PACMPL)* 2.Symposium on Principles of Programming Languages (POPL) (2018).

[OC12] Bruno C.d.S. Oliveira and William R. Cook. "Extensibility for the masses - practical extensibility with object algebras." In: *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*. 2012.

[Oli+13] Bruno C.d.S. Oliveira, Tijs van der Storm, Alex Loh, and William R. Cook. "Feature-oriented programming with object algebras." In: *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*. 2013.

[Ora93] Oracle. *Java Iterator Interface*. 1993. URL: http://web.archive.org/web/20190331014925/https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html.

[OM12] Dominic A. Orchard and Alan Mycroft. "A notation for comonads." In: *Symposium on Implementation and Application of Functional Languages (IFL)*. 2012.

[PKZ16] Jennifer Paykin, Neelakantan R. Krishnaswami, and Steve Zdancewic. "The essence of event-driven programming." Unpublished Draft (https://web.archive.org/web/20161018163751/http://www.mpi-sws.org/~neelk/essence-of-events.pdf). 2016.

[PMS11] Tomas Petricek, Alan Mycroft, and Don Syme. "Extending monads with pattern matching." In: *Proceedings of Haskell Symposium*. 2011.

[POM14] Tomas Petricek, Dominic Orchard, and Alan Mycroft. "Coeffects: A calculus of context-dependent computation." In: *Proceedings of International Conference on Functional Programming (ICFP)*. Gothenburg, Sweden, 2014.

[PS11] Tomas Petricek and Don Syme. "Joinads: A retargetable control-flow construct for reactive, parallel and concurrent programming." In: *Proceedings of the International Symposium on Practical Aspects of Declarative Languages (PADL)*. 2011.

[PS14] Tomas Petricek and Don Syme. "The F# computation expression zoo." In: *Proceedings of the International Symposium on Practical Aspects of Declarative Languages (PADL)*. 2014.

[PE88] Frank Pfenning and Conal Elliott. "Higher-order abstract syntax." In: *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*. 1988.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002. ISBN: 9780262162098.

[PT98] Benjamin C. Pierce and David N. Turner. "Local type inference." In: *Proceedings of Symposium on Principles of Programming Languages (POPL)*. 1998.

[PE10] Hubert Plociniczak and Susan Eisenbach. "JErlang: Erlang with joins." In: *Coordination Models and Languages*. Ed. by Dave Clarke and Gul Agha. Vol. 6116. Lecture Notes in Computer Science. Springer-Verlag, 2010, pp. 61–75.

[Plo75] Gordon D. Plotkin. "Call-by-name, call-by-value and the λ-calculus." In: *Theoretical Computer Science* 1.2 (1975).

[PP03] Gordon D. Plotkin and John Power. "Algebraic operations and generic effects." In: *Applied Categorical Structures* (2003).

[PP09] Gordon D. Plotkin and Matija Pretnar. "Handlers of algebraic effects." In: *Proceedings of the European Conference on Programming Languages and Systems (ESOP)*. 2009.

[Pyt00] Python Software Foundation. *Python Programming Language*. Oct. 2000. URL: http://web.archive.org/web/20190508000927/https://www.python.org.

[Rap97] Rapide Design Team. *Rapide 1.0 Executable Language Reference Manual*. Tech. rep. Program Analysis and Verification Group – Computer Systems Lab – Stanford University, 1997. URL: http://web.archive.org/web/20070712132218/http://poset.stanford.edu/rapide/lrms/xec.ps.

[Rea11] ReactiveX. *Reactive Extensions*. June 2011. URL: https://web.archive.org/web/20190503085703/http://reactivex.io.

[Rep88] John H. Reppy. "Synchronous operations as first-class values." In: *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*. 1988.

[Rep91] John H. Reppy. "CML: A higher-order concurrent language." In: *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*. 1991.

[Rep93] John H. Reppy. "Concurrent ML: design, application and semantics." In: *Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991-1992, McMaster University, Hamilton, Ontario, Canada*. 1993. DOI: 10.1007/3-540-56883-2_10. URL: http://dx.doi.org/10.1007/3-540-56883-2_10.

[Rey72] John C. Reynolds. "Definitional interpreters for higher-order programming languages." In: *Proceedings of the ACM Annual Conference - Volume 2*. 1972.

[Rey74] John C. Reynolds. "Towards a theory of type structure." In: *Proceedings of the Programming Symposium*. Ed. by B. Robinet. 1974.

[Rey78] John C. Reynolds. "User-defined types and procedural data structures as complementary approaches to data abstraction." In: *Programming Methodology: A Collection of Articles by Members of IFIP WG2.3* (1978). Ed. by David Gries, pp. 309–317.

[Rhi09] Morten Rhiger. "Type-safe pattern combinators." In: *Journal of Functional Programming* (2009).

[RO10] Tiark Rompf and Martin Odersky. "Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs." In: *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*. 2010.

[Ros07] Andreas Rossberg. "Typed open programming: a higher-order, typed approach to dynamic modularity and distribution." PhD thesis. Saarland University, 2007.

[SHM14] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. "REScala: Bridging between object-oriented and functional style in reactive applications." In: *Proceedings of the International Conference on Modularity*. 2014.

[SO11] Tom Schrijvers and Bruno C. d. S. Oliveira. "Monads, zippers and views: virtualizing the monad stack." In: *Proceedings of International Conference on Functional Programming (ICFP)*. 2011.

[SWL77] Mary Shaw, William A. Wulf, and Ralph L. London. "Abstraction and verification in Alphard: Defining and specifying iteration and generators." In: *Communications of the ACM* 20.8 (1977).

[Smo89] Mark Smotherman. "A sequencing-based taxonomy of I/O systems and review of historical machines." In: *ACM SIGARCH Computer Architecture News* 17.5 (Sept. 1989), pp. 5–15. ISSN: 0163-5964.

[Smo13] Mark Smotherman. *History and Overview of Interrupts and Interrupt Systems*. Jan. 2013. URL: http://web.archive.org/web/20190702195740/https://people.cs.clemson.edu/~mark/interrupts.html.

[Sou+10] Robert Soulé, Martin Hirzel, Robert Grimm, Buğra Gedik, Henrique Andrade, Vibhore Kumar, and Kun-Lung Wu. "A universal calculus for stream processing languages." In: *Proceedings of the European Conference on Programming Languages and Systems (ESOP)*. Springer-Verlag, 2010.

[Spe01a] Michael Sperber. "Computer-assisted lighting design and control." PhD thesis. University of Tübingen, Germany, 2001.

[Spe01b] Michael Sperber. "Developing a stage lighting system from scratch." In: *Proceedings of International Conference on Functional Programming (ICFP)*. 2001.

[SKK16] Kenichi Suzuki, Oleg Kiselyov, and Yukiyoshi Kameyama. "Finally, safely-extensible and efficient language-integrated query." In: *Proceedings of Workshop on Partial Evaluation and Program Manipulation (PEPM)*. 2016.

[Swe85] Richard E. Sweet. "The Mesa programming environment." In: *Proceedings of the ACM SIGPLAN Symposium on Language Issues in Programming Environments (SLIPE)*. 1985.

[Swi08] Wouter Swierstra. "Data types à la carte." In: *Journal of Functional Programming* (2008).

[SPL11] Don Syme, Tomas Petricek, and Dmitry Lomov. "The F# asynchronous programming model." In: *Proceedings of the International Symposium on Practical Aspects of Declarative Languages (PADL)*. 2011.

[Thi03] Hayo Thielecke. "From control effects to typed continuation passing." In: *Proceedings of Symposium on Principles of Programming Languages (POPL)*. 2003.

[Tur04] D. A. Turner. "Total functional programming." In: *Journal of Universal Computer Science* (2004).

[TR11] Aaron Joseph Turon and Claudio V Russo. "Scalable Join Patterns." In: *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2011.

[WA85] William W. Wadge and Edward A. Ashcroft. *Lucid The Dataflow Programming Language*. Academic Press, 1985. ISBN: 0127296506.

[Wad89] Philip Wadler. "Theorems for free!" In: *Proceedings of the international conference on functional programming languages and computer architecture (FPCA)*. 1989.

[Wad90a] Philip Wadler. "Comprehending monads." In: *LISP and Functional Programming*. 1990.

[Wad90b] Philip Wadler. "Deforestation: Transforming programs to eliminate trees." In: *Theoretical Computer Science* 73.2 (1990).

[Wad90c] Philip Wadler. "Linear types can change the world." In: *Proceedings of the IFIP Working Group 2.2/2.3 Working Conference on Programming Concepts and Methods*. 1990.

[Wad90d] Philip Wadler. *Recursive types for free!* July 1990. URL: https://web.archive.org/web/20181019122521/http://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt.

[Wad92] Philip Wadler. "The essence of functional programming." In: *Proceedings of Symposium on Principles of Programming Languages (POPL)*. 1992.

[Wad15] Philip Wadler. "Propositions as types." In: *Communications of the ACM* 58.12 (2015).

[WB89] Philip Wadler and Stephen Blott. "How to make ad-hoc polymorphism less ad-hoc." In: *Proceedings of Symposium on Principles of Programming Languages (POPL)*. 1989.

[WC10a] Stephanie Weirich and Chris Casinghino. "Arity-generic datatype-generic programming." In: *Proceedings of the Workshop on Programming Languages Meets Program Verification (PLPV)*. 2010.

[WC10b] Stephanie Weirich and Chris Casinghino. "Generic programming with dependent types." In: *Generic and Indexed Programming - International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures*. 2010.

[Whi+07] Walker White, Mirek Riedewald, Johannes Gehrke, and Alan Demers. "What is 'Next' in event processing?" In: *Proceedings of the Symposium on Principles of Database Systems (PODS)*. 2007.

[Wri95] Andrew K. Wright. "Simple imperative polymorphism." In: *Lisp and Symbolic Computation* 8.4 (1995).

[YW14] Jeremy Yallop and Leo White. "Lightweight higher-kinded polymorphism." In: *International Symposium on Functional and Logic Programming (FLOPS)*. 2014.

[ZSJ11] Lukasz Ziarek, K. C. Sivaramakrishnan, and Suresh Jagannathan. "Composable asynchronous events." In: *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*. 2011.

ERKLÄRUNG

Hiermit erkläre ich, dass ich die vorliegende Arbeit – abgesehen von den in ihr ausdrücklich genannten Hilfen – selbständig verfasst habe.

Oliver Bračevac

REVISIONS

NOVEMBER 5, 2019 Polished for publication:

- Added acknowledgements.
- Minor corrections of spelling and grammar.
- Improved readability of plots and charts.

AUGUST 4, 2019 Submitted version for thesis defense.

COLOPHON

This document was typeset in LaTeX with the `classicthesis`[1] package developed by André Miede and Ivo Pletikosić. The bibliography was processed by Biblatex.[2] Ornament graphics on the title page were created with Alain Matthes' `pgfornament` LaTeX package.[3] I created the colorful "marble diagrams" in some figures in OmniGraffle, using open source graphic assets from the RxJava documentation,[4] originally contributed by David Gross. Both the main text and maths are typeset in Robert Slimbach's *Minion Pro* typeface. Sans serif text is typeset in Philip H. Poll's free and open source *Linux Biolinum* typeface,[5] provided by Bob Tennent's `libertine` LaTeX package.[6] The monospace/type-writer face in code listings is *DejaVu Sans Mono* from the free and open source *DejaVu* font family,[7] originally developed by Bitstream, Inc. as *Bitstream Vera*. I created the "word cloud" on the back of the title page by feeding this thesis into Jonathan Feinberg's `Wordle` tool.[8] Other graphics (graphs, commutative diagrams, plots and charts), were created by me with help of the excellent PGF/TikZ package[9] for LaTeX by Till Tantau, the `tikz-cd` package[10] by Augusto Stoffel, the `tcolorbox` package[11] by Prof. Dr. Dr. Thomas F. Sturm, and the `pgfplots` package[12] by Christian Feuersänger.

1 http://ctan.org/pkg/classicthesis
2 http://ctan.org/pkg/biblatex
3 http://ctan.org/pkg/pgfornament
4 http://raw.github.com/wiki/Netflix/RxJava/images/rx-operators.graffle
5 http://www.linuxlibertine.org
6 http://ctan.org/pkg/libertine
7 http://dejavu-fonts.github.io
8 http://www.wordle.net
9 http://www.ctan.org/pkg/pgf
10 http://www.ctan.org/pkg/tikz-cd
11 http://www.ctan.org/pkg/tcolorbox
12 http://www.ctan.org/pkg/pgfplots