

Type-safe Polyvariadic Event Correlation

OLIVER BRAČEVAC, TU Darmstadt, Germany

GUIDO SALVANESCHI, TU Darmstadt, Germany

SEBASTIAN ERDWEG, Johannes Gutenberg University Mainz, Germany

MIRA MEZINI, TU Darmstadt, Germany

The pivotal role that event correlation technology plays in today's applications has led to the emergence of different families of event correlation approaches with a multitude of specialized correlation semantics, including computation models that support the composition and extension of different semantics.

However, type-safe embeddings of extensible and composable event patterns into statically-typed general-purpose programming languages have not been systematically explored so far. This is unfortunate, as type-safe embedding of event patterns is important to enable increased correctness of event correlation computations as well as domain-specific optimizations. Event correlation technology has often adopted well-known and intuitive notations from database queries, for which approaches to type-safe embedding do exist. However, we argue in the paper that these approaches, which are essentially descendants of the work on monadic comprehensions, are not well-suited for event correlations and, thus, cannot without further ado be reused/re-purposed for embedding event patterns.

To close this gap we propose POLYJOIN, a novel approach to type-safe embedding for fully polyvariadic event patterns with polymorphic correlation semantics. Our approach is based on a tagless final encoding with uncurried higher-order abstract syntax (HOAS) representation of event patterns with n variables, for arbitrary $n \in \mathbb{N}$. Thus, our embedding is defined in terms of the host language without code generation and exploits the host language type system to model and type check the type system of the pattern language. Hence, by construction it is impossible to define ill-typed patterns. We show 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 practical, type-safe and extensible. An implementation of it in pure multicore OCaml is readily usable.

1 INTRODUCTION

The field of event correlation is concerned with the design, theory, and application of pattern languages for matching events from distributed data sources [Cugola and Margara 2012; Luckham 2001]. For example, “if within 5 minutes a temperature sensor reports $\geq 50^\circ\text{C}$, and a smoke sensor is set, trigger a fire alarm” [Cugola and Margara 2012], is a pattern that relates sensor events by their attributes and timing. The increasingly event-driven nature of today's applications has triggered a lot of research and development efforts resulting in different families of event correlation approaches, e.g., stream processing [Carbone et al. 2015], reactive programming [Salvaneschi et al. 2014], complex event processing (CEP) [EsperTech Inc. 2006].

So far, research and development efforts have been predominantly focused on computation models that offer a wide range of specialized correlation semantics across families as well as within the same family (cf. [Bainomugisha et al. 2013; Cugola and Margara 2012] for an overview). This focus is a natural consequence of the semantic variability inherent in the domain: Events in conjunction with time can be correlated in different ways, e.g., the event pattern $\text{âĀI}a$ followed by bâĀI applied to the event sequence $\langle abb \rangle$ 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. To embrace this semantic variability in a principled way, Bračevac et al. [2018] propose a computation model for programming arbitrary semantic variants of event correlation in a composable and extensible way. The model encodes event correlation purely in terms of algebraic effects and handlers [Plotkin and Power 2003; Plotkin and Pretnar 2009].

A rather neglected question concerns the integration of event patterns into programming languages. 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 1996]. 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 representations [Hofer et al. 2008]. Polymorphism is 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 [EsperTech Inc. 2006], 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 [Chandramouli et al. 2014] and Rx.Net [ReactiveX 2018], which employ LINQ [Cheney et al. 2013; Meijer et al. 2006] 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 [Cugola and Margalef 2012], featuring similar declarative query syntax, e.g., the CQL language [Arasu et al. 2006]. (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 1990] and translate queries to instances of the monad interface [Moggi 1991; Wadler 1992]. 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 induce *sequential* 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.

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. 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 et al. [2009], 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* [Kiselyov 2015].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 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) [Huet and Lang 1978; Pfenning and Elliott

¹Polyvariadic functions were first studied by Danvy [1998] to encode a statically type-safe version of the well-known `printf` function in terms of combinators.

1988] 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. Nevertheless, POLYJOIN neither requires dependent types nor ad-hoc polymorphism, it requires only forms of type constructor polymorphism and bounded polymorphism in the host language. And it is portable: variants of the tagless final approach coincide with object algebras [Oliveira and Cook 2012; Oliveira et al. 2013] in OO languages. Thus, while this paper chooses OCaml as the host language, POLYJOIN works in principle in other languages, both functional and OO, e.g., Haskell [Carette et al. 2009], Scala [Hofer et al. 2008] or Java [Biboudis et al. 2015].

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 paper are as follows:

- A systematic analysis of why existing monadic style embeddings for database-style queries are inadequate for embedding event patterns (Section 2).
- A novel tagless final embedding for event patterns into mainstream programming languages, which is polyvariadic, statically type-safe, polymorphic, extensible, and modular (Section 3). This embedding constitutes the core of POLYJOIN - we provide a formalization and an encoding of it in pure OCaml.
- An embedding of the CARTESIUS language by Bračevac et al. [2018], yielding its first type-safe and fully polyvariadic implementation in multicore OCaml (Section 4). As already mentioned, CARTESIUS supports programming arbitrary semantic variants of event correlation in a composable and extensible way by encoding event correlation purely in terms of algebraic effects and handlers, for which interesting implementation challenges arise to achieve polyvariadicity. Furthermore, we introduce extensions to core POLYJOIN to properly support event correlation based on implicit time data associated to events.
- An evaluation of the POLYJOIN version of CARTESIUS, comparing it against the prototype by Bračevac et al. [2018] (Section 5). Our version adds declarative pattern syntax, has exponential savings in code size, supports any arity and significantly reduces programmer effort when defining extensions. Furthermore, we discuss matters of portability and benefits of having uncurried pattern variables.

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.

2.1 Event Patterns versus Join Queries

For illustration, the pseudo-code in Figure 1a reflects how programmers could write the event pattern at the start of Section 1 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

```

1 from (t <- temp_sensor)
2   from (s <- smoke_sensor)
3     where within(s,t,5 minutes) && s && t >= 50.0
4       yield (format "Fire: %f" t)

```

(a) Monadic Database Query Notation.

```

1 join ((from temp_sensor) @. (from smoke_sensor) @. cnil)
2   (fun ((temp,t1), ((smoke,t2), ())) ->
3     where (within t1 t2 (minutes 5.0)) %& smoke %& (temp %>= 50.0)
4       (yield (format "Fire %f" temp)))

```

(b) POLYJOIN/OCaml Version.

Fig. 1. Event Correlation Example: Fire Alarm.

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. In the latter case, data is enumerated only if the join computation decides to access it. Dually, an event correlation computation passively observes and reacts to the enumeration of data.

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

²To determine the occurrence time of the events produced by a join, we merge the two occurrence times into the smallest interval containing both. This merging strategy follows many Complex Event Processing systems [Cugola and Margara 2012; White et al. 2007]. In the notation, we replace singleton intervals $[t, t]$ with just t .

different ways. For example, the event pattern “*a followed by b*” applied to the event sequence $\langle a b c \rangle$ 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.

2.2 Monadic Embeddings are Inadequate for Event Patterns

At first sight, monads seem to be a good choice for denoting event patterns. Indeed, 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 [Hofer et al. 2008], 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.

A closer inspection, though, reveals that the monadic translation of join queries cannot capture all event correlation behaviors.

2.2.1 Semantics of Joins in Monadic Embeddings. Monadic embeddings, such as LINQ, map queries to monads, i.e., type constructors $C[\cdot]$ with combinators

$$\begin{aligned} \text{return} &: \forall \alpha. \alpha \rightarrow C[\alpha] \\ \text{bind} &: \forall \alpha. \forall \beta. C[\alpha] \rightarrow (\alpha \rightarrow C[\beta]) \rightarrow C[\beta] \end{aligned}$$

satisfying the following laws

$$\text{bind } c \text{ (return)} = c \tag{1}$$

$$\text{bind (return } x) f = f \ x \tag{2}$$

$$\text{bind (bind } c \ f) g = \text{bind } c \ (\lambda y. \text{bind } (f \ x) \ g) \tag{3}$$

which describe that $C[\cdot]$ models a notion of sequential computation with effects, respectively a collection type. Accordingly, LINQ’s metatheory (cf. Cheney et al. [2013]) 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

$$\text{from } (x_1 <- r_1) \cdots \text{from } (x_n <- r_n) \text{ yield } (x_1, \dots, x_n)$$

denotes a nested monad computation

$$\text{bind } r_1 \ (\lambda x_1. \text{bind } r_2 \ (\lambda x_2. \cdots \text{bind } r_n \ (\lambda x_n. \text{return } (x_1, \dots, x_n)) \cdots))$$

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 $i \in \{1, \dots, n\}$, the monadic join computation binds an element from r_i to variable x_i *after* it has bound all variables $x_j, j < i$.

³Strictly speaking, database languages require the MonadPlus interface, i.e., monads with additional zero and plus operations for empty bag and bag union.

2.2.2 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:

Unbounded increase in latency. Suppose we embedded the fire alarm example (Section 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. \square

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.

2.2.3 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 [McBride and Paterson 2008], Arrows [Hughes 2000] or in the Join Calculus [Fournet and Gonthier 1996]. 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.

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:

Definition 2.1 (n-way join type signature). At the type-level, joins are computation-transforming functions with a signature of the shape

$$S[\alpha_1] \times \cdots \times S[\alpha_n] \rightarrow S[\alpha_1 \times \cdots \times \alpha_n]$$

for some type constructor $S[\cdot]$ and for all element types $\alpha_1, \dots, \alpha_n$ and all $n \geq 0$. \blacksquare

⁴This is sometimes referred to as *Combine Latest* event correlation behavior [Bračevac et al. 2018], which captures the semantics of reactive programming languages.

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

$$S[\alpha_1] \times S[\alpha_2] \rightarrow S[\alpha_1 \times \alpha_2],$$

which in the terminology of Mycroft et al. [2016] 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 2.2.1) for merging, we obtain a different control-flow operator

$$S[\alpha_1] \rightarrow (\alpha_1 \rightarrow S[\alpha_1 \times \alpha_2]) \rightarrow S[\alpha_1 \times \alpha_2].$$

By parametricity, 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 2.1 gives a unifying interface for both worlds: the type signature permits both the nested monadic bind construction for database joins (Section 2.2.1), 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* [Kiselyov 2015], 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!

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 to language integration and high-level, declarative event patterns of event correlation systems.

3.1 Fire Alarm, Revisited

As a first taste of how clients specify event patterns in POLYJOIN, Figure 1b shows the POLYJOIN version of the fire alarm example (Figure 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 3.3) into a list of bindings, with `cnil` being the empty list of bindings.

```

1 module type Symantics = sig
2   type 'a repr
3   val lit: int -> int repr
4   val (+): int repr -> int repr -> int repr
5 end

```

(a)

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

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

(b)

```

1 module Num = struct
2   type 'a repr = 'a
3   let lit n = n
4   let (+) x y = plus x y
5
6 end

```

(c)

```

1 module PP = struct
2   type 'a repr = string
3   let lit n = sprintf "<%d>" n
4   let (+) x y =
5     sprintf "(%s + %s)" x y
6 end

```

(d)

Fig. 2. Basic Tagless Final Examples.

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) [Huet and Lang 1978; Pfenning and Elliott 1988]. 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 et al. [2009] 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 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),(1))) -> ...)
(* Error: This pattern matches values of type unit but a pattern was expected
   which matches values of type float repr * unit *)

```

In the remainder of this section, we present the core principles underlying POLYJOIN.

3.2 A Primer on Tagless Final Embeddings

We briefly recapitulate the tagless final approach by Carette et al. [2009] in the following. Readers familiar with the topic may skip this section.

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 [1978] and (2) decouples interface and implementation (the interpreter) of these functions, obtaining a term representation which is indexed by their denotation.

For example, the OCaml module signature in Figure 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 [Leijen and Meijer 1999]. These signatures straightforwardly correspond to a bottom-up reading of natural deduction rules, with return type being conclusions and arguments being premises (Figure 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:

```
1 module Exp(S: Symantics) = struct
2   open S
3   let exp1 = (lit 1) + (lit 2)
4   let exp2 x y = exp1 + (lit 3) + x + y
5 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

$$\text{int } S.\text{repr} \rightarrow \text{int } S.\text{repr} \rightarrow \text{int } S.\text{repr}$$

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`. For brevity, we will not explicitly show the surrounding functor boilerplate in subsequent example terms. Figure 2c and Figure 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:

```
1 module En = Exp(Num);; (* Instantiate Num interpretation *)
2 module Es = Exp(PP);; (* Instantiate PP interpretation *)
3 En.exp1;; (* yields int Num.repr = 3 *)
4 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:

```
1 module type NumBoolSym = sig
2   include Symantics
3   val b_lit: bool -> bool repr
4   val (&): bool repr -> bool repr -> bool repr
5 end
```

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

Expressions and Patterns

$$\begin{array}{c}
 \boxed{\vdash_{\text{exp}} M : A} \quad \boxed{\vdash_{\text{pat}} P : A} \\
 \\
 \text{(WHERE)} \quad \frac{\vdash_{\text{exp}} M : \text{Bool} \quad \vdash_{\text{pat}} P : A}{\vdash_{\text{pat}} \text{where } M P : A} \quad \text{(YIELD)} \quad \frac{\vdash_{\text{exp}} M : A}{\vdash_{\text{pat}} \text{yield } M : A} \\
 \\
 \text{(JOIN)} \quad \frac{\vdash_{\text{ctx}} \Pi : \vec{A} \quad \vec{A} \rightsquigarrow \vec{B} \quad \frac{\overrightarrow{[\vdash_{\text{exp}} x : B]} \quad \vdots}{\vdash_{\text{pat}} P : C}}{\vdash_{\text{exp}} \text{join } \Pi (\vec{x}.P) : \text{Shape}[C]}
 \end{array}$$

Context Formation

$$\begin{array}{c}
 \boxed{\vdash_{\text{var}} V : A} \quad \boxed{\vdash_{\text{ctx}} \Pi : \vec{A}} \\
 \\
 \frac{\vdash_{\text{exp}} M : \text{Shape}[A]}{\vdash_{\text{var}} \text{from } M : A} \text{ (FROM)} \quad \vdash_{\text{ctx}} \text{cnil} : \emptyset \text{ (CNIL)} \quad \frac{\vdash_{\text{var}} V : B \quad \vdash_{\text{ctx}} \Pi : \vec{A}}{\vdash_{\text{ctx}} V @. \Pi : B, \vec{A}} \text{ (CAT)}
 \end{array}$$

Shape Translation

$$\vec{A} \rightsquigarrow \vec{A} \text{ (ID)}$$

$$\boxed{\vec{A} \rightsquigarrow \vec{B}}$$

Fig. 3. Core Syntax and Typing Rules of POLYJOIN.

```

1 module type Symantics = sig
2   type 'a shape (* Shape[.] Constructor *)
3   (* Judgments (cf. Figure 3): *)
4   type 'a repr   (*  $\vdash_{\text{exp}} \cdot : A$  *)
5   type 'a pat    (*  $\vdash_{\text{pat}} \cdot : A$  *)
6   type ('a, 'b) ctx (* combination of  $\vdash_{\text{ctx}} \cdot : A$  and  $A \rightsquigarrow B$  *)
7   type 'a var    (*  $\vdash_{\text{var}} \cdot : A$  *)
8   (* Context Formation and Shape Translation: *)
9   val from: 'a shape repr -> 'a var
10  val cnil: (unit, unit) ctx
11  val (@.): 'a var -> ('c, 'd) ctx -> ('a * 'c, 'a repr * 'd) ctx
12  (* Expressions and Patterns: *)
13  val yield: 'a repr -> 'a pat
14  val where: bool repr -> 'a pat -> 'a pat
15  val join: ('a, 'b) ctx -> ('b -> 'c pat) -> 'c shape repr
16 end

```

Fig. 4. Tagless Final Representation of Core POLYJOIN.

interpreters, such as `Num` and `PP`. We leave their extension with booleans as an exercise to the reader.

Finally (no pun intended), while the focus of this paper 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, the approach in principle supports modular, library-level compilation pipelines for DSLs, e.g., as exemplified in [Carette et al. 2009] and [Suzuki et al. 2016].

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 1b) or other implicit metadata on events. We address these features in Section 4.

```

1 module type HL = sig
2   type 'a el
3   type _ hlist =
4     | Z : unit hlist
5     | S : 'a el * 'b hlist
6       -> ('a * 'b) hlist
7 end

1 module HList(E: sig type 'a t end) =
2   struct
3     type 'a el = 'a E.t
4     type _ hlist =
5       | Z : unit hlist
6       | S : 'a el * 'b hlist
7         -> ('a * 'b) hlist
8     let nil = Z
9     let cons h t = S (h,t)
10  end

```

(a) (b)

Fig. 5. Heterogeneous Lists Definition.

```

1 module StdContext(T: sig type 'a repr type 'a shape end) = struct
2   open T
3   type _ var = Bind: 'a shape repr -> 'a var
4   module Ctx = HList(struct type 'a t = 'a var end)
5   type (_,_) shape = (* Shape translation judgment *)
6     | Base: (unit, unit) shape
7     | Step: ('s, 'a) jsig -> ('t * 's, 't repr * 'a) shape
8   (* Implementation of Context Formation (Figures 3 and 4) *)
9   type ('a,'b) ctx = ('a,'b) shape * 'a Ctx.hlist
10  let from = fun s -> Bind s
11  let cnil = (Base, Ctx.nil)
12  let (@.): type a c d. a var -> (c, d) ctx -> (a * c, a repr * d) ctx
13    = fun v (shape, ctx) -> (Step shape, Ctx.cons v ctx)
14 end

```

Fig. 6. Default Variable Context Representation with Heterogeneous Lists.

```

1 module MonadL = struct
2   type 'a repr = 'a
3   type 'a shape = 'a list
4   type 'a pat = 'a list
5   (* Contexts: cf. Figure 6: *)
6   include StdContext(struct type 'a repr = 'a type 'a shape = 'a list)
7   let where b p = if b then p else []
8   let yield v = [v]
9   let pair: 'a repr -> 'b repr -> ('a * 'b) repr = fun a b -> (a,b)
10  (* Nested monadic bind by induction over the context derivation: *)
11  let rec cart : type a b c. (a,b) ctx -> (a -> c list) -> c list = fun ctx k ->
12    match ctx with
13    | (Base, Ctx.Z) -> k ()
14    | (Step n, Ctx.S (Bind ls, hs)) ->
15      bind (fun x -> cart (n,hs) (fun xs -> k (x, xs)))
16  let join: ('a, 'b) ctx -> ('b -> 'c pat) -> 'c shape repr =
17    fun ctx body -> cart ctx body
18 end

```

Fig. 7. Sequential Cartesian Product in POLYJOIN.

Figure 3 defines the core syntax and typing rules of POLYJOIN in natural deduction style and Figure 4 the corresponding tagless encoding as a OCaml module signature.

3.3.1 Expressions and Patterns. As before, we define syntax/typing rules $\vdash_{\text{exp}} M : A$ for the syntactic sort of expressions (e.g., Figure 2b) and introduce a new sort for *patterns*, $\vdash_{\text{pat}} P : A$, meaning that pattern P yields events of type A . 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.

3.3.2 The essence of polyvariadic joins. Rule (JOIN) formalizes the essence of polyvariadic n -join expressions, which we exemplified in Section 3.1. A join expression requires a valid pattern context Π consisting of *from* bindings. In the premise, context formation $\vdash_{\text{ctx}} \Pi : \vec{A}$ certifies that Π is well-formed, exposing at the type-level (of the host language OCaml!) that Π has the *shape* \vec{A} , which is an ordered, *heterogeneous* sequence of types describing the number and element types of the bindings in Π . Keeping statically track of this information is crucial for programming polyvariadic definitions and performing type-level computations.

As we motivated in Section 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 $\vec{A} \rightsquigarrow \vec{B}$. 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 $\text{Shape}[C]$. Here, the abstract type constructor $\text{Shape}[\cdot]$ corresponds to the type constructor $S[\cdot]$ in Definition 2.1. Overall, the *join* form defines an n -ary join followed by a map

$$\overrightarrow{\text{Shape}[\vec{A}]} \Rightarrow \text{Shape}[\vec{A}] \Rightarrow \text{Shape}[C]$$

in the sense of our earlier definition.

3.3.3 Context Formation. In contrast to standard treatments of variable bindings and context, our pattern contexts Π are nameless, i.e., they are *not* sequences of variable/type pairs $\overrightarrow{x : \vec{A}}$, 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 $\text{Shape}[A]$. 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).

3.3.4 Shape Translation. The purpose of this judgment is controlling how the types of pattern variables are computed from the shape of the pattern context Π . The core version of POLYJOIN trivially establishes $\vec{B} = \vec{A}$, i.e., the i th pattern variable binds a DSL term representing an element of the type A_i . In Section 4, we will change the rules for this judgment, attaching timing information to pattern variables.

3.3.5 OCaml Representation of PolyJoin. Following the principles of Section 3.2, we define the intrinsically-typed syntax of POLYJOIN in an OCaml module signature (Figure 4). 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. 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 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 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

```
join ((from source1) @. (from source2) @. (from source3) @. cnil)
```

yields a three-ary pattern abstraction

```
((a1 repr * (a2 repr * (a3 repr * unit)) -> '_b pat) -> '_b shape repr
```

as intended, given that `sourcei`, has type `ai shape repr`, for $i \in \{1, 2, 3\}$. We model heterogeneous sequences via nested binary products at the type-level.

3.4 Polyvariadic Programming

Our tagless final embedding of `join` (Figure 4) makes polyvariadicity explicit in its signature, by quantifying over all possible variable contexts `('a, 'b) ctx`. Next, we address how to program against such *context-polymorphic* interfaces, in order to implement concrete interpreters of *Symantics* in Figure 4.

3.4.1 Heterogeneous Sequences. We program against context-polymorphic interfaces with generalized algebraic data types (GADTs) [Cheney and Hinze 2003; Johann and Ghani 2008], using heterogeneous lists [Kiselyov et al. 2004]

```
type _ hlist = Z : unit hlist
          | S : 'a * 'b hlist -> ('a * 'b) hlist
```

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, as a form of bounded polymorphism. 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 only invoked 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, _)`.

3.4.2 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 5a). We may create concrete instances of constrained `hlists` in form of OCaml modules, with the functor `HList` (Figure 5b). For example, the modules

```
module HL = HList(struct type 'a t = 'a end)          (* Standard hlist *)
module Lists = HList(struct type 'a t = 'a list end) (* hlist of lists *)
```

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

```
1 let rec enclose: type a. a HL.hlist -> a Lists.hlist =
2   function
3   | HL.Z -> Lists.nil
4   | HL.S (hd,tl) -> Lists.(cons [hd] (enclose tl))
```

is a polyvariadic function that element-wise converts unconstrained `hlists` into `hlists` of homogeneous lists.

3.4.3 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 computation. 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, \dots, a_n) to n -tuples $(a_1 \text{ list}, \dots, a_n \text{ list})$. Shape preservation is crucial to compose independently developed polyvariadic components in a type-safe manner.

3.4.4 Contexts as Heterogeneous Lists. Interpreters of the join form

```
join: ('a, 'b) ctx -> ('b -> 'c pat) -> 'c shape repr
```

must program against precisely this type signature. Somehow, the tagless interpreter should be able to *extract* individual element values `a1 repr...an repr` (type parameter `'b`) from n input shapes having type `a1 shape repr ... 'an shape repr` and bind them to the pattern body `('b -> 'c pat)`. Where do they come from? The formal system in Figure 3 specifies that the derivation of the context parameter `('a, 'b) ctx` includes the bound input shapes, by rule (FROM). An interpreter should be able to analyze how the judgment `('a, 'b) ctx` was derived, to get ahold of the input shapes. 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.

Therefore, we instantiate the abstract context signature (Figure 4) in terms of constrained heterogeneous lists and other GADTs, as shown in Figure 6. 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 shape translation judgment from Figure 3, which is codified by the `('a, 'b) shape` GADT (Figure 6, Lines 5-7). As explained in Section 3.3.5, context formation and shape translation are combined into a single judgment, which translates to their product in Figure 6, Line 9.

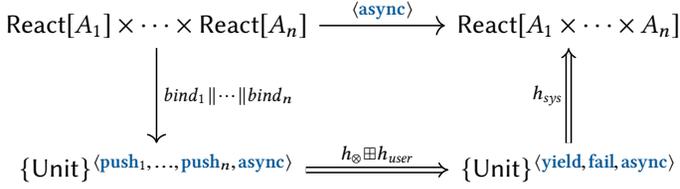


Fig. 8. Join Computations in CARTESIUS.

3.4.5 Example: Sequential Cartesian Product. We conclude this section with an example POLYJOIN interpreter, implementing an old friend: the nested monadic translation from LINQ/comprehensions (Section 2) over lists, yielding a cartesian product. Figure 7 shows the full implementation.⁵ The interpreter is meta-circular, interpreting expressions as OCaml values and works with lists as input shape representation. In this case, we also set the pattern type to lists.

```

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 tagless interpreter MonadL implements the n -nary nested monadic bind, by a straightforward structural induction (see above) over the standard context representation (Figure 6) in the function `cart` (Figure 7, Line 11-15). 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 16).

In the next section, we show that POLYJOIN supports more than sequential variable bindings, which go beyond LINQ and standard comprehension implementations.

4 CASE STUDY: EVENT CORRELATION WITH ALGEBRAIC EFFECTS

In this section, we embed the CARTESIUS language by Bračevac et al. [2018] with POLYJOIN. CARTESIUS is a general model for event correlation over asynchronous streams that is based upon algebraic effect handlers [Plotkin and Power 2003; Plotkin and Pretnar 2009]. Bračevac et al. outline a macro translation of their event pattern syntax into effects and handlers. However, their prototype implementation lacks this syntax and is not polyvariadic, supporting only a handful of hard-coded arities. POLYJOIN enables the first fully polyvariadic implementation with proper pattern syntax embedding, in the multicore OCaml dialect [Dolan et al. 2017].⁶ CARTESIUS is an interesting larger example, because it has event patterns that expose event timing and concurrent binding semantics. Another important point of the example is to showcase how to program a polyvariadic backend against POLYJOIN’s polyvariadic interfaces.

4.1 Overview of CARTESIUS

The diagram in Figure 8 illustrates how polyvariadic join computations in the sense of Definition 2.1 (top row) relate to CARTESIUS’ effect-based representation (bottom row). In this setting, function types (\rightarrow) are annotated with Koka-style effects rows [Leijen 2017b], indicating the side effect a function call may induce. We write concrete effect types in blue font. CARTESIUS joins perform global asynchrony effects $\langle \text{async} \rangle$ (cf. [Dolan et al. 2017; Leijen 2017a]), accounting for

⁵For the example, we extended the DSL with support for constructing binary pairs (Figure 7, Line 9).

⁶The full implementation is available at <http://github.com/bracevac/cartesius>.

the inversion of control present in event correlation (Section 2.1). The other kind of arrow (\Rightarrow) 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*.

CARTESIUS combines n input sources into a unit-valued effectful computation of type

$$\{\text{Unit}\}^{\langle \text{push}_1, \dots, \text{push}_n, \text{async} \rangle}$$

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 (push_i) $_{1 \leq i \leq n}$, which correspond to a sequence of effect declarations in multicore OCaml, informally written:

$$(\text{effect } \text{Push}_i: 'a_i \rightarrow \text{unit})_{1 \leq i \leq n}$$

That is, effects are named operations in the algebraic effect setting, having the signature of functions. Computations invoke them to induce the effect. Here, 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 2.2) there is no control dependency between the event notification effects, i.e.,

$$\langle \text{push}_1, \dots, \text{push}_n, \text{async} \rangle \equiv \langle \text{push}_{\pi(1)}, \dots, \text{push}_{\pi(n)}, \text{async} \rangle$$

are equivalent in the effect type system, for any permutation $\pi : \{1 \dots n\} \rightarrow \{1 \dots n\}$, 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 [de Moura and Ierusalimsky 2009]). Handlers may induce additional effects while discharging the observed effects of the underlying computation. Hence, effect handlers define a custom semantics/implementation of effect operations and are transformations (\Rightarrow) between effectful computations.

The bottom row of Figure 8 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:

$$\begin{aligned} \text{effect Yield}: 'a_1 \times \dots \times 'a_n \rightarrow \text{unit} & \quad (* \text{ Output tuple } *) \\ \text{effect Fail}: \text{unit} \rightarrow 'a & \quad (* \text{ Discard tuple } *) \end{aligned}$$

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.

4.1.1 Callback Logic. As part of the join computation, the system handler h_{\otimes} in Figure 8 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 15, 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 $m_i : M[A_i]$ and computes it.

Writing callbacks for joining asynchronous computations leads to another manifestation of the binding problem for event correlation (cf. Section 2.2), this time in terms of the well-known continuation monad.⁷ To avoid these problems, 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).

4.1.2 Restriction Handlers. Bračevac et al. [2018] use pattern syntax similar to POLYJOIN. In addition, their syntax supports a way to compose and inject user-defined effect handlers (the h_{user} component in Figure 8), 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 10 shows a preview of our POLYJOIN embedding for CARTESIUS. 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 [De Bruijn 1972] into the context of the join.⁸ Intuitively, the operator `|++|` represents effect handler composition \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 2.2.2). 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]), ((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 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. [Bračevac et al. 2018]):

```
< ((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.

⁷It is common practice in mainstream asynchronous programming libraries (e.g., C# [Bierman et al. 2012] or Scala [EPFL 2013]) to join multiple computations by nesting callback subscriptions, which introduces unnecessary sequential dependencies.

⁸Bračevac et al. [2018] apply the restrictions to the event source *values* in their pen and paper examples. However, this is technically inaccurate, because their formal semantics defines the restriction handlers in the nameless form. The POLYJOIN version of the syntax accurately reflects that restrictions refer to input positions in the context of the join.

4.1.3 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 Meta Data. CARTESIUS patterns expose time data of events and accordingly constraints on time data, e.g., `within` in Figure 1b.

The Focusing Continuations Problem. The family $(\kappa_i)_{1 \leq i \leq n}$ 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

$$(m_i)_{1 \leq i \leq n} : M[A_1] \times \cdots \times M[A_n]$$

of mailboxes. 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” into the mailbox sequence:

$$M[A_1] \times \cdots \times M[A_n] \rightsquigarrow (M[A_1] \times \cdots \times M[A_{i-1}] \times [\cdot] \times M[A_{i+1}] \times \cdots \times M[A_n])_{1 \leq i \leq n}.$$

This type is effectively the zipper [Huet 1997] 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.

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 $\{1 \dots n\}$ 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.

4.2 Extending PolyJoin

We extend core POLYJOIN from Section 3 with metadata and contextual extensions to provide a type-safe embedding of the CARTESIUS language. Figure 9 defines the extended POLYJOIN rules, which we discuss in the following.

4.2.1 Meta Data. 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 [Eugster and Jayaram 2009]. We expose meta data for each variable in the join pattern body, by changing the previous 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 meta datum, e.g., Figure 10, Line 3.

Expressions and Patterns

$$\boxed{\vdash_{\text{exp}} M : A} \quad \boxed{\vdash_{\text{pat}}^{\vec{C}} P : A}$$

(WHERE)

$$\frac{\vdash_{\text{exp}} M : \text{Bool} \quad \vdash_{\text{pat}}^{\vec{C}} P : A}{\vdash_{\text{pat}}^{\vec{C}} \text{where } M P : A}$$

(YIELD)

$$\frac{\vdash_{\text{exp}} M : A}{\vdash_{\text{pat}}^{\vec{C}} \text{yield } M : A}$$

(MMERGE)

$$\frac{\vdash_{\text{exp}} M : \text{Meta} \quad \vdash_{\text{exp}} N : \text{Meta}}{\vdash_{\text{exp}} M \sqcup N : \text{Meta}}$$

$$\frac{\vdash_{\text{ctx}} \Pi : \vec{A} \quad \vec{A} \rightsquigarrow \vec{B} \quad \vdash_{\text{ext}}^{\vec{A}} X \quad \frac{[\vdash_{\text{exp}} x : \vec{B}]}{\vdash_{\text{pat}}^{\vec{A}} P : C}}{\vdash_{\text{exp}} \text{join } \Pi X (\vec{x}.P) : \text{Shape}[C]} \quad (\text{JOIN})$$

Shape Translation

$$\emptyset \rightsquigarrow \emptyset \quad (\text{TZ})$$

$$\frac{\vec{A} \rightsquigarrow \vec{B}}{C, \vec{A} \rightsquigarrow (C \times \text{Meta}), \vec{B}} \quad (\text{TS})$$

$$\boxed{\vec{A} \rightsquigarrow \vec{B}}$$

Shape (Multi)Projection

$$\vdash \text{pz} : A \in A, \vec{B} \quad (\text{PZ}) \quad \frac{\vdash n : A \in \vec{B}}{\vdash \text{ps } n : A \in C, \vec{B}} \quad (\text{PS}) \quad \vdash \langle \rangle : \emptyset \sqsubseteq \vec{A} \quad (\text{MZ}) \quad \frac{\vdash n : C \in \vec{B} \quad \vdash \vec{m} : \vec{A} \sqsubseteq \vec{B}}{\vdash n, \vec{m} : C, \vec{A} \sqsubseteq \vec{B}} \quad (\text{MS})$$

$$\boxed{\vdash n : A \in \vec{B}} \quad \boxed{\vdash \vec{n} : \vec{A} \sqsubseteq \vec{B}}$$

Contextual Extension Operations

$$\vdash_{\text{ext}}^{\vec{A}} \text{enil} \quad (\text{EMPTY})$$

$$\frac{\vdash_{\text{ext}}^{\vec{A}} X \quad \vdash_{\text{ext}}^{\vec{A}} Y}{\vdash_{\text{ext}}^{\vec{A}} X ++ Y} \quad (\text{EMERGE})$$

$$\boxed{\vdash_{\text{ext}}^{\vec{A}} X}$$

Fig. 9. POLYJOIN with Meta Data and Contextual Extensions.

```

1 join ((from temp_sensor) @. (from smoke_sensor) @. cnil)
2   ((most_recently p0) |++| (most_recently p1))
3   (fun ((temp,t1), ((smoke,t2), ())) ->
4     yield (pair temp smoke))

```

Fig. 10. Example: Join Pattern with CARTESIUS-style Contextual Extension.

```

1 module type SLOT = sig
2   type t
3   effect Push: t -> unit
4   effect Get: unit -> t mailbox
5   effect Set: t mailbox -> unit
6 end
7 type 'a slot =
8   (module SLOT with type t = 'a)

1 module type YF = sig
2   type t
3   effect Yield: t -> unit
4   effect Fail: unit -> 'a
5 end
6 type 'a yieldfail =
7   (module YF with type t = 'a)

```

Fig. 11. Generative Effects from Modules in Multicore OCaml.

```

1 (* type a b. a Slots.hlist -> b handler *)
2 let memory slots = poly_handler slots (fun (s: (module SLOT)) ->
3   let module S = (val s) in
4     let mbox: S.t mailbox ref = ref (mailbox ()) in
5     fun action -> try action () with
6       | effect (S.Get ()) k -> continue k !mbox
7       | effect (S.Set m) k -> mbox := m; continue k ())

```

Fig. 12. Example Polyvariadic State Handler.

```

1 let where: bool repr -> ('c,'a) pat -> ('c,'a) pat =
2   fun cond body meta yf ->
3     if cond then (body meta yf) else fail_with yf
1 let yield: 'a repr -> ('c,'a) pat =
2   fun result meta yf -> (result, (merge_all meta))

```

Fig. 13. CARTESIUS Pattern Implementation in Context/Capability-Passing Style.

```

1 let join: type s a b. (s, a) ctx -> s ext -> (a -> (s, b) pat) -> b shape repr
2 = fun ctx extension pattern_body ->
3   (* (effect Pushi: si -> unit)1 ≤ i ≤ n: *)
4   let slots: s Slots.hlist = gen_slots_from ctx in
5   (* effect Yield: b -> unit and effect Fail: unit -> 'c: *)
6   let yf: b yieldfail = gen_yieldfail () in
7   (* instantiate and run bottom-right corner of Figure 8 *)
8   let pstreams = parallel_bind ctx slots in
9   let h⊗ = gen_default_handler slots yf pattern_body in
10  let huser = extension ctx in
11  let hsys = gen_sys_handler yf in
12  Async.spawn (hsys |+| h⊗ |+| huser) pstreams

```

Fig. 14. Implementation of CARTESIUS Join Patterns.

A tagless interpreter requires a policy for computing the metadata of joined events from the meta data of input events. However, we do not expose merging explicitly in the pattern syntax. E.g., the pattern body in Figure 10 does not mention the output event’s meta datum in the yield form. This design reduces syntactic noise in the pattern and avoids that users incorrectly merge 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 $\vdash_{\text{pat}}^{\vec{C}} P : A$ 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.

4.2.2 Contextual Extensions. In order to support restriction handlers in the pattern syntax, we introduce an abstract syntax for contextual extensions $\vdash_{\text{ext}}^{\vec{A}} X$, 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 10 cannot refer to non-existent positions in the context of bindings. Furthermore, we assume contextual extensions are monoids, having an empty extension (rule (EMPTY)) and a merge operation (rule (EMERGE)), just as the restriction handlers in CARTESIUS.

4.2.3 OCaml Representation. The extensions to POLYJOIN presented here adapt straightforwardly to an OCaml module signature encoding, in the same manner as before (Section 3.3.5). Below, we show the key changes in the Symantics signature:

```

1 module type Symantics = (* ... *)
2   (* Contextual Extensions: *)
3   val enil: unit -> 'a ext
4   val (|++|): 'a ext -> 'a ext -> 'a ext
5   (* Expressions and Patterns: *)
6   val mmerge: meta repr -> meta repr -> meta repr
7   val join: ('a,'b) ctx -> 'a ext -> ('b -> ('a,'c)) pat) -> 'c shape repr
8 end

```

The function `mmerge` corresponds to the metadata merge operator \sqcup in Figure 9. Due to space limitations, we elide the full definition and leave it as an exercise to the reader.

4.2.4 Predicates on Meta Data. We sketch how the extended version of POLYJOIN enables new predicates on metadata, which we can easily add in the tagless final encoding. E.g., in CARTESIUS, events carry time intervals (pairs of time stamps) as metadata: `type meta = time * time`, where the `time` is a numeric type for time stamps (cf. [Bračevac et al. 2018]), having a total order along with the following operations:

```

val infity: time repr (* greatest element, positive infinity *)
val ninfty: time repr (* least element, negative infinity *)
val (%<=): time repr -> time repr -> bool repr (* comparison *)
val span: time repr -> time repr -> time repr (* time span *)
val minutes: float -> time repr (* representation of minutes *)

```

The `within` constraint in our running fire alarm example (Figure 1b) is definable as “derived syntax” (i.e., function abstraction) in the tagless DSL

```

let within: meta repr -> meta repr -> time repr -> bool repr =
  fun m1 m2 mb -> (span (m1  $\sqcup$  m2)) %<= mb

```

where we compare the merged intervals (their least upper bound) against the given time span.

4.3 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 is meta-circular, i.e., it interprets join patterns as multicore OCaml code.

4.3.1 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 adopt a folklore encoding of *first-class effects*, which is already utilized in Bračevac et al.’s non-polyvariadic prototype. Figure 11 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 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 4.1.1). Similarly, we encapsulate the `Yield` and `Fail` effects (cf. Figure 8) in the `YF` module.

First-class effect instances are capability values that programmers can pass around and manipulate. E.g., Lines 7 and 8 of the left definition defines the 'a slot capability type. We use the `with type` construct 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 3.4.3) 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. 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.

4.3.2 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 = fun slots ->
2   module IntS = (val (head slots)) in
3   module StrS = (val (head (tail slots))) in
4   fun action -> try action () with
5     | effect (StrS.Push s) k -> println(s); continue k ()
6     | effect (IntS.Push i) k -> println(int_to_str(i)); 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).

4.3.3 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*. More generally, we can define polyvariadic handlers, by calculations over heterogeneous lists. E.g., Figure 12 shows the polyvariadic handler `memory`. This handler is part of h_{\otimes} in Figure 8 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. Due to space limitations, we elide the definition of the `poly_handler` combinator. It is essentially mapping the supplied function over the n first-class

effects and then composing the resulting handlers into a single handler. The more general take-away from this example is that context polymorphism enables polyvariadic components and their composition, i.e., an entire polyvariadic backend implementation for the polyvariadic frontend of POLYJOIN.

4.3.4 Implicit Time Data in Patterns. We let the syntactic sort of patterns denote functions types as one possible representation of context dependency, i.e.,

```
type ('c, 'a) pat = 'c Meta.hlist -> 'a yieldfail -> 'a repr * meta repr
```

accepting a heterogeneous list of time stamps, i.e.,

```
module Meta = HList(struct type 'a t = meta repr end)
```

and a capability to yield and fail. The end result is a pair of output event together with its meta datum, 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 13). 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.

4.3.5 Join Pattern Implementation. Figure 14 shows the implementation of the join pattern form. The code is a direct translation of the diagram in Figure 8. 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 8 in an asynchronous thread. Due to space limitations, we omit the implementations of the involved combinators. 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.

And it is statically type-safe! The type signature certifies that (1) `join` is polyvariadic, (2) that supplied restriction handlers cannot refer to non-existing input sources and (3) the implementation discharges the implicit metadata requirement by the HOAS pattern abstraction (`a -> (s, b) pat`). To see the latter point, we first inline the type definition for patterns into the function type, obtaining

```
a -> s Meta.hlist -> b yieldfail -> b repr * meta repr      (†)
```

as the type of the pattern body. By a simple induction over the derivation of the context representation (s, a) `ctx`, we can establish that (with some generous typographic simplification)

```
a = (a1 repr * meta repr) * ... * (an repr * meta repr)
s = a1 * ... * an
```

where n is the arity of the join. Therefore, by definition of `Meta.hlist`, we have that the parameter type `s Meta.hlist` is an n tuple of metadata. Importantly, the *implementation* and not the end user supplies the metadata to be merged.

By parametricity (cf. [Wadler 1989]), the signature of `join` and (†) even determines how the CARTESIUS implementation functions. It extracts n -tuples and metadata from the event sources and tests them against the pattern, potentially⁹ resulting in an output `b repr * meta repr`. This

⁹It may throw a failure via the supplied `yieldfail` capability.

$$\begin{aligned}
\kappa_i &: A_i \rightarrow M[A_1 \times \cdots \times A_n] \\
\kappa_i &= \lambda x : A_i.m_1 \otimes \cdots \otimes m_{i-1} \otimes \langle x \rangle \otimes m_{i+1} \otimes \cdots \otimes m_n \quad 1 \leq i \leq n \\
(_ \otimes _) &: \forall \alpha \beta. M[\alpha] \rightarrow M[\beta] \rightarrow M[\alpha \times \beta]
\end{aligned}$$

Fig. 15. The Focusing Continuations Problem.

is the only way to obtain b -typed events, which may be communicated over the output event source `b_shape_repr`. However, the types are too weak to enforce that the implementation is always productive, i.e., it might be inert or diverging. More sophisticated type systems are necessary to enforce productivity, e.g., where types represent theorems in linear temporal logic (LTL) [Cave et al. 2014].

4.3.6 Solving the Focusing Continuations Problem. Defining a polyvariadic version of CARTESIUS' callback logic (Sections 4.1.1, 4.1.3 and Figure 15) was the most challenging part of the implementation. Fortunately, the tools we have so far enable a simple solution (Figure 16), 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 15. It is a polyvariadic handler for the `Pushi` 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 `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 12) is in the context, to define the default semantics.

Finally, Figure 17 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 [Moreau 1998], in terms of effects (dynamic variables) and stacking handlers (dynamic variable bindings).

4.4 Safe, Reusable and Modular Restriction Handlers

Here, we solve the challenge of writing restriction handlers generically (Section 4.1.3). Following the design of Bračevac et al. [2018], 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.

4.4.1 Type-Safe Pointers/De Bruijn Indices. Recall that we apply restriction handlers to specific positions via De Bruijn indices (e.g., Figure 10, 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 9). The judgment asserts: the index number n proves that type A is contained in the heterogeneous

```

1 (* Callback logic *)
2 let callback slots consumer = poly_handler slots (fun (si: (module SLOT)) ->
3   let module Si = (val si) in
4   fun action -> try action () with
5   | effect (Si.Push x) k ->
6     (* m1, ..., mi-1, ⟨x⟩, mi+1, ..., mn *)
7     let mboxes = focus slots s x in
8     (* m1 ⊗ ... ⊗ mi-1 ⊗ ⟨x⟩ ⊗ mi+1 ⊗ ... ⊗ mn *)
9     crossproduct mboxes consumer; continue k ()
1  (* Focusing into m1, ..., mn *)
2  let focus type a b. a Slots.hlist -> b slot -> b ev -> a Mail.hlist =
3  fun slots si x ->
4    let module Si = (val si) in
5    (* Replace i-th position by handling *)
6    try get_all slots () with
7    | effect (Si.Get ()) k -> continue k (mailbox x)
1  (* Bulk retrieval of all mailboxes m1, ..., mn *)
2  let get_all: type a. a Slots.hlist -> unit -> a Mail.hlist = fun slots () ->
3  let module M = HMap(Slots)(Mail) in
4  M.map { M.f = fun (s': (module SLOT)) ->
5    let module S' = (val s') in perform (S'.Get ()) } slots

```

Fig. 16. Multicore OCaml Solution to the Focusing Continuations Problem.

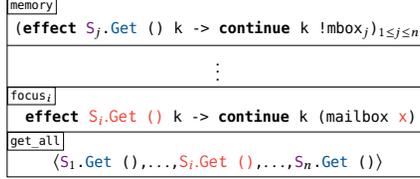


Fig. 17. Illustration of Dynamic Execution Context during Focusing.

```

1 (* ⊢ n : A ∈ B̄ (Figure 9) *)
2 type (_,_) _ptr =
3   | Pz: ('a, 'a * 'b) _ptr
4   | Ps: ('a, 'b) _ptr
5     -> ('a, 'c * 'b) _ptr
6 type ('a,'b) ptr =
7   unit -> ('a,'b) _ptr
1  (* ⊢ n̄ : Ā ⊆ B̄ (Figure 9) *)
2 type (_, 'a) _mptr =
3   | Mz: (unit, 'a) _mptr
4   | Ms: (('c, 'b) _ptr * ('a, 'b) _mptr)
5     -> ('c * 'a, 'b) _mptr
6 type ('a,'b) mptr =
7   unit -> ('a,'b) _mptr
8 let mz () = Mz
9 let ms p ps () = Mn (p (), ps ())

```

Fig. 18. GADTs for Type-safe Pointers and Sets of Pointers.

```

1 let p0 () = Pz      (* ('a, 'a * 'b) ptr *)
2 let p1 () = Ps Pz  (* ('a, 'b * ('a * 'c)) ptr *)
3 let p2 () = Ps Ps Pz (* ('a, 'b * ('c * ('a * 'd))) ptr *)
4 let ps () =        (* ('a * ('b * unit), 'b * ('c * ('a * 'd))) mptr *)
5   (ms p2 @@ ms p0 @@ mz) ()

```

Fig. 19. Example: Type-safe Pointers and Sets of Pointers in OCaml.

```

1 module HPtr(H: HList) = struct
2   let rec proj: type a ctx. (a, ctx) _ptr -> ctx H.hlist -> a H.el =
3     fun n hlist -> match n, hlist with
4       | Pz,   H.S (hd, _) -> hd
5       | Ps n, H.S (_, tl) -> proj n tl
6
7   let rec mproj: type xs ctx. (xs, ctx) _mptr -> ctx H.hlist -> xs H.hlist =
8     fun mptr hlist -> match mptr with
9       | Mz      -> H.nil
10      | Ms (i, ps) -> H.cons (proj i hlist) (mproj ps hlist)
11 end

```

Fig. 20. Type-safe Element Access.

```

1 let most_recently: type ctx i. (i, ctx) ptr -> ctx ext = fun ptr slots ->
2   (* Type-safe pointers and projection on first-class effects *)
3   let module SPtr = HPtr(Slots) in
4   (* Project the given first-class effect declaration *)
5   let module Si = (val (SPtr.proj (ptr ()) slots)) in
6   fun action -> try action () with
7     | effect (Si.Push x) k ->
8       (* Truncate the mailbox to the most recent value *)
9       perform Si.Set (mailbox x); continue k (perform Si.Push x)

```

Fig. 21. Example: Position-polymorphic Restriction Handler (most_recently).

sequence \vec{B} . The rules of shape projection straightforwardly translate to a binary GADT `_ptr` (Figure 18, 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 19. The pointer type relates to the formal system in Figure 9 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 type¹⁰ 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 20). 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 3.4.2).

4.4.2 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 9), 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 18 and enables sets of type-safe pointers (type `mptr`). Similarly to the type-safe pointers, the sets describe polymorphic judgment

¹⁰The distinction between `_ptr` and `ptr` is necessary, due to the value restriction in ML/OCaml [Wright 1995]. The pointers must be functions in order to fully generalize the type parameters.

schemes for context requirements. For example, the set of pointers `ps` in Figure 19 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 20), which guarantees that projecting the pointed-at positions in the pointer set yields the heterogeneous list of projected values.

4.4.3 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 4.3.3). We call this refinement *position polymorphism*, and it enables generic restriction handlers for CARTESIUS. For instance, we define the `most_recently` restriction in Figure 21. 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 4.1.2), are functions parametric over type-safe pointer sets

```
aligning: type xs ctx. (xs,ctx) mptr -> ctx ext
```

which use the multiprojection function to focus on the given positions `xs` in `ctx`. Due to space limitations, we elide the definition of this handler and refer to [Bračevac et al. 2018].

4.4.4 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 3). 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 4.3.3) 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 content are the first-class effect declarations of type `'a Slots.hList` (Section 4.3.1), from which we calculate effect handlers (e.g., Figure 12).

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 21), 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 9) matches the join shape demanded by the given extensions to the join shape \vec{A} supplied by the join pattern syntax.

5 EVALUATION AND DISCUSSION

Table 1 compares our POLYJOIN-based implementation of CARTESIUS against the original prototype by Bračevac et al. [2018]. Our version has significant advantages over the original prototype. We discuss them below.

Table 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. 8)	$O(1)$	$O(n^2)$
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	✓	\sim^c
Computational Effects	✓	✓

^a n = maximum arity in use.

^b n = number of inputs.

^cRequires separate copy per supported arity.

5.1 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 4.4.4) 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.

5.2 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 $i \in \{1, \dots, n\}$ 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 $O(\sum_{i=1}^n i) = O(n^2)$ size. This is because each arity i leads to the dependency on i first-class effect declarations and contains effect handlers with i cases for these effects (e.g., the memory handler in Figure 12).

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 21 has $c = 1$ position parameters) and restrictions parametric in position sets (e.g., `aligning restriction` Section 4.1.2). In the prototype, a separate handler definition must be written for *each arity* and *each*

combination of positions, respectively *each subset* of $\{1, \dots, n\}$. 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 in [Bračevac et al. 2018] for the runtime behavior of example join variants.

5.3 Extensibility Dimensions

The POLYJOIN version of CARTESIUS features an extensible pattern language, because of the reliance on the tagless final approach (Section 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 sub-components that can be separately developed, because only knowledge about an effect interface is required, represented by the first-class SLOT effects (Figure 11). However, the table entry for the prototype is only half-checked (\sim). 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.

5.4 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 2.1 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 4.3.6 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., [Kiselyov 2014].

Soundness of PolyJoin. The tagless final approach we build on ensures soundness of patterns relative to the soundness of the host language’s type system. The approach guarantees *by construction* that “well-typed event patterns (in the pattern DSL type system) cannot go wrong”. This

is ensured, because we describe and check the DSL’s type system via the host language’s type system. If the presented DSL variants and tagless interpreters were unsound, then they would be a counterexample to the soundness of the host language. We effectively obtain a powerful safety-net for developing variants of join pattern DSLs. If a programmer defines an unsound DSL, then it will manifest itself at some point in the development, for example: (1) the syntax forms and DSL types are ill-defined, which leads to the OCaml type checker rejecting the `Symantics` module signature, i.e., no tagless interpreter is implementable in the first place. (2) the `Symantics` signature is accepted, but then no useful programs could be formulated in the functor representation of expressions.

Abstraction Overhead. Context access could be computed statically but is dynamic. We believe that this is a negligible initial overhead. However Multi-stage programming could help eliminating and optimizing overhead from hlists and pointers. [Kiselyov 2014] [Rompf and Odersky 2010] But no support for effect handlers.

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, respectively Section 4.2.3/Appendix A.1, if the system supports metadata in patterns. These already permit a wide range of backends. First, 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 7. Second, general, metadata-based event correlation backends can be plugged into POLYJOIN, which we have exemplified with CARTESIUS in Section 4. 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 [EsperTech Inc. 2006] in OCaml. Embedding such systems additionally requires language bindings, e.g., Ocaml-Java.¹¹ 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 for end users.

Representing Context Dependency. In the embedding of CARTESIUS (Section 4.3), we represented implicit context dependency by function abstraction. Bračevac et al. [2018] propose an encoding of context dependency by means of ambient, implicit parameters in the style of [Lewis et al. 2000]. These can be straightforwardly encoded in a language with algebraic effects and handlers. However, this would require compiler extensions, which makes this idea less portable. In languages with compile-time ad-hoc polymorphism (e.g., [Odersky et al. 2018; Wadler and Blott 1989]) we could avoid the syntactic overhead of threading context through the join computation.

On portability. In this paper, 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 A.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 [Pierce and Turner 1998]. 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 [Biboudis et al. 2015], higher-kinded polymorphism has to be

¹¹www.ocamljava.org

encoded indirectly, using the technique by [Yallop and White \[2014\]](#), 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 A.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 7](#)). 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 ([Section 4](#)).

5.5 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 [\[Paykin et al. 2016\]](#)

$$\diamond A \otimes \diamond B \Rightarrow \diamond(A \otimes \diamond B \oplus \diamond A \otimes B)$$

is a linear logic version of pattern matching events with cases.

Another example for disjunction is the Join Calculus [\[Fournet and Gonthier 1996\]](#). In future work, we would like to investigate a syntax design that reasonably integrates disjunction with the event patterns of POLYJOIN. [Rhiger \[2009\]](#) 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 $S_1[\cdot], \dots, S_n[\cdot]$ into an output shape $S_{n+1}[\cdot]$. E.g., $S_i \in \{\text{Channel}, \text{DB}, \text{Future}, \text{List}, \dots\}$. 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.

6 RELATED WORK

6.1 Language-Integrated Queries and Comprehensions

Similarly to this work, Facebook's Haxl system [\[Marlow et al. 2014, 2016\]](#) 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 applicative `bind` [\[McBride and Paterson 2008\]](#), 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 et al. \[2013\]](#) 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 et al. \[2016\]](#) 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. \[2016\]](#) 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 2.1 is an n -ary version of Joinads, which underlie F# computation expressions [[Petricek and Syme 2011, 2014](#); [Syme et al. 2011](#)]. 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 new syntax forms. Furthermore, we support pattern syntax designs for systems with implicit metadata.

6.2 Polyvariadicity

[Danvy \[1998\]](#) 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., [[Asai 2009](#); [Fridlender and Indrika 2000](#)]. [Rhiger \[2009\]](#) 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 ahold 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 5.4). We postulate that usually, some variant of the focusing problem (Section 4.3.6) 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 \[2000\]](#). GADTs, which none of the previous works on polyvariadicity consider, are the missing piece to close this gap, as they enable inversion/injection of the context formation.

[Lindley \[2008\]](#) 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 4.4) 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 [2010a,b] study forms of polyvariadicity in the dependently-typed language Agda [Norell 2007] 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 [2002] 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.

6.3 Higher-Order Abstract Syntax

The idea of higher-order abstract syntax (HOAS) was first introduced by Huet and Lang [1978]. Later, the paper by Pfenning and Elliott [1988] 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 to simplify the formulation and ensure correctness of syntactic rewriting rules in formal language development. 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., a uncurried n -ary binding forms satisfactorily enable concurrency implementations.

7 CONCLUSION

We showed how to define type-safe language-integrated event patterns in terms of the tagless final approach and explicit type-level representation of heterogeneous variable context in uncurried style. Our approach yields a practical and portable method to define extensible comprehension syntax, without requiring dedicated compiler support. In general, event patterns require a non-monadic binding semantics and are thus not well-supported by state of the art language-integrated query approaches. We support patterns with an arbitrary number of event sources and heterogeneous event types, while not requiring dependent types. Event patterns and joins in general, after all, are values which are dependent on the variable context. Our encoding with heterogeneous lists realizes this view in a direct way.

For 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 to express the correlation. Equivalently, for each variable in the context, we must synthesize a callback that relates the event notifications with the binder representation of the rest of the context.

ACKNOWLEDGMENTS

We thank Nada Amin, Oleg Kiselyov, Sam Lindley, and Jeremy Yallop for feedback and discussions on this work.

REFERENCES

Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal* (2006).

- Kenichi Asai. 2009. On typing delimited continuations: Three new solutions to the printf problem. *Higher-Order and Symbolic Computation* 22, 3 (2009), 275–291.
- Engineer Bainomugisha, Andoni Lombide Carretón, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. 2013. A survey on reactive programming. *ACM Computing Surveys* 45, 4 (2013).
- Aggelos Biboudis, Nick Palladinos, George Fourtounis, and Yannis Sma ragda kis. 2015. Streams à la carte: Extensible pipelines with object algebras. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*.
- Gavin Bierman, Claudio Russo, Geoffrey Mainland, Erik Meijer, and Mads Torgersen. 2012. Pause 'n' play: Formalizing asynchronous C#. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*.
- Oliver Bračevac, Nada Amin, Guido Salvaneschi, Sebastian Erdweg, Patrick Eugster, and Mira Mezini. 2018. Versatile event correlation with algebraic effects. *Proceedings of the ACM on Programming Languages (PACMPL)* 2, International Conference on Functional Programming (ICFP) (9 2018).
- Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38.
- Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming* 19, 5 (2009), 509–543.
- Andrew Cave, Francisco Ferreira, Prakash Panangaden, and Brigitte Pientka. 2014. Fair reactive programming. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*. ACM, 361–372.
- Badri Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, John C. Platt, James F. Terwilliger, and John Wernsing. 2014. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment* 8, 4 (2014), 401–412.
- James Cheney and Ralf Hinze. 2003. *First-class phantom types*. Technical Report. Cornell University.
- James Cheney, Sam Lindley, and Philip Wadler. 2013. A practical theory of language-integrated query. In *Proceedings of International Conference on Functional Programming (ICFP)*.
- Gianpaolo Cugola and Alessandro Margara. 2012. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys* 44, 3 (2012), 15:1–15:62.
- Olivier Danvy. 1998. Functional unparsing. *Journal of Functional Programming* 8, 6 (1998), 621–625.
- Nicolaas Govert De Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*.
- Ana Lúcia de Moura and Roberto Ierusalimsky. 2009. Revisiting coroutines. *Transactions on Programming Languages and Systems (TOPLAS)* (2009).
- Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavaped dy, KC Sivaramakrishnan, and Leo White. 2017. Concurrent system programming with effect handlers. In *Proceedings of the Symposium on Trends in Functional Programming*.
- EPFL. 2013. *Scala Async*. Retrieved June 11, 2018 from <http://web.archive.org/web/20180611025539/https://github.com/scala/scala-async>
- EsperTech Inc. 2006. *Esper*. Retrieved April 22, 2018 from <https://web.archive.org/web/20180422212134/http://www.espertech.com/esper/>
- Patrick Eugster and K.R. Jayaram. 2009. EventJava: An extension of Java for event correlation. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*.
- Cédric Fournet and Georges Gonthier. 1996. The reflexive CHAM and the Join-calculus. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*.
- Daniel Fridlender and Mia Indrika. 2000. Do we need dependent types? *Journal of Functional Programming* (2000).
- Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. 2008. Polymorphic embedding of DSLs. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*.
- Paul Hudak. 1996. Building domain-specific embedded languages. *ACM Computing Surveys* 28, 4es (1996), 196.
- Gérard Huet. 1997. The zipper. *Journal of Functional Programming* (1997).
- Gérard Huet and Bernard Lang. 1978. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica* (1978).
- John Hughes. 2000. Generalising monads to arrows. *Science of Computer Programming* 37, 1-3 (2000).
- Patricia Johann and Neil Ghani. 2008. Foundations for structured programming with GADTs. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*.
- Oleg Kiselyov. 2014. The design and implementation of BER MetaOCaml - system description. In *International Symposium on Functional and Logic Programming (FLOPS)*.
- Oleg Kiselyov. 2015. *Polyvariadic functions and keyword arguments: Pattern-matching on the type of the context*. Retrieved October 10, 2018 from <https://web.archive.org/web/20181013042601/http://okmij.org/ftp/Haskell/polyvariadic.html>
- Oleg Kiselyov, Ralf Lämmel, and Kean Schupke. 2004. Strongly typed heterogeneous collections. In *Proceedings of Haskell Workshop*.
- Daan Leijen. 2017a. Structured asynchrony with algebraic effects. In *Proceedings of the International Workshop on Type-Driven Development*.

- Daan Leijen. 2017b. Type directed compilation of row-typed algebraic effects. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*.
- Daan Leijen and Erik Meijer. 1999. Domain specific embedded compilers. In *Proceedings of the Conference on Domain-specific Languages (DSL)*.
- Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark B. Shields. 2000. Implicit parameters: Dynamic scoping with static types. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*.
- Sam Lindley. 2008. Many holes in hindley-milner. In *Proceedings of the ACM Workshop on ML*.
- David C. Luckham. 2001. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc.
- Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. 2014. There is no fork: An abstraction for efficient, concurrent, and concise data access. In *Proceedings of International Conference on Functional Programming (ICFP)*.
- Simon Marlow, Simon Peyton Jones, Edward Kmett, and Andrey Mokhov. 2016. Desugaring haskell's do-notation into applicative operations. In *Proceedings of Haskell Symposium*.
- Conor McBride. 2002. Faking it: Simulating dependent types in haskell. *Journal of Functional Programming* 12, 4&5 (2002), 375–392.
- Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *Journal of Functional Programming* (2008).
- Erik Meijer, Brian Beckman, and Gavin Bierman. 2006. LINQ: Reconciling object, relations and XML in the .NET framework. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.
- Eugenio Moggi. 1991. Notions of computation and monads. *Information and computation* 93, 1 (1991), 55–92.
- Luc Moreau. 1998. A syntactic theory of dynamic binding. *Higher-Order and Symbolic Computation* 11, 3 (1998).
- Alan Mycroft, Dominic A. Orchard, and Tomas Petricek. 2016. 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*.
- Shayan Najd, Sam Lindley, Josef Svenningsson, and Philip Wadler. 2016. Everything old is new again: Quoted domain-specific languages. In *Proceedings of Workshop on Partial Evaluation and Program Manipulation (PEPM)*.
- Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph.D. Dissertation. Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.
- Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. 2018. Simplicity: Foundations and applications of implicit function types. *Proceedings of the ACM on Programming Languages (PACMPL)* 2, Symposium on Principles of Programming Languages (POPL) (2018).
- Bruno C.d.S. Oliveira and William R. Cook. 2012. Extensibility for the masses - practical extensibility with object algebras. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*.
- Bruno C.d.S. Oliveira, Tijs van der Storm, Alex Loh, and William R. Cook. 2013. Feature-oriented programming with object algebras. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*.
- Jennifer Paykin, Neelakantan R. Krishnaswami, and Steve Zdancewic. 2016. The essence of event-driven programming. (2016). Unpublished Draft (<https://web.archive.org/web/20161018163751/http://www.mpi-sws.org/~neelk/essence-of-events.pdf>).
- Tomas Petricek and Don Syme. 2011. 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)*.
- Tomas Petricek and Don Syme. 2014. The F# computation expression zoo. In *Proceedings of the International Symposium on Practical Aspects of Declarative Languages (PADL)*.
- Frank Pfenning and Conal Elliott. 1988. Higher-order abstract syntax. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*.
- Benjamin C. Pierce and David N. Turner. 1998. Local type inference. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*.
- Gordon D. Plotkin and John Power. 2003. Algebraic operations and generic effects. *Applied Categorical Structures* (2003).
- Gordon D. Plotkin and Matija Pretnar. 2009. Handlers of algebraic effects. In *Proceedings of the European Conference on Programming Languages and Systems (ESOP)*.
- ReactiveX. 2018. *Reactive Extensions*. Retrieved May 3, 2019 from <https://web.archive.org/web/20190503085703/http://reactivex.io/>
- John C. Reynolds. 1978. User-defined types and procedural data structures as complementary approaches to data abstraction. *Programming Methodology: A Collection of Articles by Members of IFIP WG2.3* (1978), 309–317.
- Morten Rhiger. 2009. Type-safe pattern combinators. *Journal of Functional Programming* (2009).
- Tiark Rumpf and Martin Odersky. 2010. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*.
- Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014. REScala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of the International Conference on Modularity*.
- Kenichi Suzuki, Oleg Kiselyov, and Yuki Yoshi Kameyama. 2016. Finally, safely-extensible and efficient language-integrated query. In *Proceedings of Workshop on Partial Evaluation and Program Manipulation (PEPM)*.

- Don Syme, Tomas Petricek, and Dmitry Lomov. 2011. The F# asynchronous programming model. In *Proceedings of the International Symposium on Practical Aspects of Declarative Languages (PADL)*.
- Philip Wadler. 1989. Theorems for free!. In *Proceedings of the international conference on functional programming languages and computer architecture (FPCA)*.
- Philip Wadler. 1990. Comprehending monads. In *LISP and Functional Programming*.
- Philip Wadler. 1992. The essence of functional programming. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*.
- Philip Wadler and Stephen Blott. 1989. How to make ad-hoc polymorphism less ad-hoc. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*.
- Stephanie Weirich and Chris Casinghino. 2010a. Arity-generic datatype-generic programming. In *Proceedings of the Workshop on Programming Languages Meets Program Verification (PLPV)*.
- Stephanie Weirich and Chris Casinghino. 2010b. Generic programming with dependent types. In *Generic and Indexed Programming - International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures*.
- Walker White, Mirek Riedewald, Johannes Gehrke, and Alan Demers. 2007. What is “Next” in event processing?. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*.
- Andrew K. Wright. 1995. Simple imperative polymorphism. *Lisp and Symbolic Computation* 8, 4 (1995).
- Jeremy Yallop and Leo White. 2014. Lightweight higher-kinded polymorphism. In *International Symposium on Functional and Logic Programming (FLOPS)*.

A OPTIONAL, SUPPLEMENTARY MATERIAL

A.1 Extended Polyjoin

```

1 module type Symantics = sig
2   type 'a shape (* Shape[-] Constructor *)
3   type meta      (* Event metadata type *)
4   (* Judgments (cf. Figure 9): *)
5   type 'a repr   (*  $\vdash_{\text{exp}} \cdot : A$  *)
6   type ('a,'b) pat (*  $\vdash_{\text{pat}}^{\vec{A}} \cdot : B$  *)
7   type ('a,'b) ctx (* combination of  $\vdash_{\text{ctx}} \cdot : A$  and  $A \rightsquigarrow B$  *)
8   type 'a var    (*  $\vdash_{\text{var}} \cdot : A$  *)
9   type 'a ext    (*  $\vdash_{\text{ext}}^{\vec{A}} \cdot$  *)
10  (* Context Formation and Shape Translation: *)
11  val from: 'a shape repr -> 'a var
12  val cnil: (unit,unit) ctx
13  val (@.): 'a var -> ('c, 'd) ctx -> ('a * 'c, 'a repr * 'd) ctx
14  (* Contextual Extensions: *)
15  val enil: unit -> 'a ext
16  val (|++|): 'a ext -> 'a ext -> 'a ext
17  (* Expressions and Patterns: *)
18  val mmerge: meta repr -> meta repr -> meta repr
19  val yield: 'a repr -> ('c,'a) pat
20  val where: bool repr -> ('c,'a) pat -> ('c,'a) pat
21  val join: ('a, 'b) ctx -> 'a ext -> ('b -> ('a,'c)) pat -> 'c shape repr
22 end

```

Fig. 22. Tagless Final Representation of Extended POLYJOIN.

A.2 Core PolyJoin in Scala

```

1 import scala.language.higherKinds
2 //Module signatures become traits/interfaces
3 trait Symantics {
4   type Ctx[A]
5   type Var[A]
6   type Repr[A]
7   type Shape[A]
8   def from[A](shape: Repr[Shape[A]]): Var[Repr[A]]
9   val cnil: Ctx[Unit]
10  def ccons[A,B](v: Var[A], ctx: Ctx[B]): Ctx[(A,B)]
11  def join[A,B](ctx: Ctx[A])(pattern: A => Repr[Shape[B]]): Repr[Shape[B]]
12  def yld[A](x: Repr[A]): Repr[Shape[A]]
13  def pair[A,B](fst: Repr[A], snd: Repr[B]): Repr[(A,B)]
14 }
15
16 //For testing, extend the language with a syntax to lift lists of values to
   shape representations

```

```

17 trait SymanticsPlus extends Symantics {
18   //note: A* means "variable number of A arguments"
19   def lift[A](xs: A*): Repr[Shape[A]]
20 }
21
22 //Expression functors become functions/methods with path-dependent types
23 def test(s: SymanticsPlus) //infers path-dependent type s.Repr[s.Shape[(Double
    , (Int, String))]]
24 = {
25   import s._
26   //Note: with more effort, the syntax for constructing contexts can be made
    prettier, in infix notation
27   val ctx = ccons(from(lift(1,2,3)), ccons(from(lift("one", "two")), ccons(
    from(lift(3.0,2.0,1.0)), cnil)))
28   //Notational convenience is similar to the OCaml version
29   join (ctx) { case (x,(y,(z,()))) =>
30     yld(pair(z,pair(x,y)))
31 }
32 /* this wouldn't type check:
33   join (ctx) { case (x,(y,())) =>
34     yld(pair(z,pair(x,y)))
35 }
36 */
37 }
38
39 //Example: sequential cartesian product over lists
40 object ListSymantics extends SymanticsPlus {
41   //GADT becomes class hierarchy
42   sealed abstract class Ctx[A]
43   //GADT constructors become case classes/objects
44   case object CNil extends Ctx[Unit]
45   case class CCons[A,B](hd: Var[A], tl: Ctx[B]) extends Ctx[(A,B)] //
    refinement controlled by extends clause
46   sealed abstract class Var[A]
47   case class Bind[A](shape: Repr[Shape[A]]) extends Var[Repr[A]]
48   override type Repr[A] = A
49   override type Shape[A] = List[A]
50   override def from[A](shape: Repr[Shape[A]]) = Bind(shape)
51   override val cnil = CNil
52   override def ccons[A, B](v: Var[A], ctx: Ctx[B]) = CCons(v,ctx)
53   override def lift[A](xs: A*) = List(xs:_)
54   override def yld[A](x: Repr[A]) = List(x)
55   override def pair[A, B](fst: Repr[A], snd: Repr[B]) = (fst,snd)
56

```

```

57  /* This join implements the monadic sequential semantics */
58  override def join[A, B](ctx: Ctx[A])(pattern: A => Repr[Shape[B]]) = ctx
      match {
59    case CNil => pattern ()
60    case CCons(Bind(xs), tl) =>
61      xs.flatMap { x =>
62        join (tl) { tuple => pattern (x,tuple) }
63      }
64  }
65 }
66
67 test(ListSymantics)
68 /* prints res0: ListSymantics.Repr[ListSymantics.Shape[(Double, (Int, String))] =
        ]] =
69 List((3.0,(1,one)), (2.0,(1,one)), (1.0,(1,one)), (3.0,(1,two)), (2.0,(1,two))
      ,
70 (1.0,(1,two)), (3.0,(2,one)), (2.0,(2,one)), (1.0,(2,one)), (3.0,(2,two)),
71 (2.0,(2,two)), (1.0,(2,two)), (3.0,(3,one)), (2.0,(3,one)), (1.0,(3,one)),
72 (3.0,(3,two)), (2.0,(3,two)), (1.0,(3,two))) */

```

A.3 Curried n -way Joins

```

1  module type Curried = sig
2    type 'a repr
3    type 'a shape
4    type 'a pat
5    type ('a,'b) ctx
6    type ('a,'b) var
7    val from: 'a shape repr -> ('a, 'a repr) var
8    val cnil: (unit, (unit -> 'a pat) * 'a) ctx
9    val (@.): ('a,'b) var -> ('c, 'd * 'e) ctx -> ('a * 'c, ('b -> 'd) * 'e) ctx
10   val yield: 'a repr -> 'a pat
11
12   (* Problem: it is not apparent that 'ps is a pattern abstraction, which
      forces
13     a structurally inductive programming style over the context formation.
      *)
14   val join: ('shape, 'ps * 'res) ctx -> 'ps -> 'res shape repr
15 end
16
17 module TestCurried(C: Curried) = struct
18   open C
19   let test a b c =
20     join ((from a) @. (from b) @. (from c) @. cnil)
21     (fun x y z () -> (yield y))

```

22 **end**