Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 351 (2020) 3–23

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

EVL: A Typed Higher-order Functional Language for Events

Sandra Alves[1]

DCC-FCUP & CRACS
University of Porto, Porto, Portugal

Maribel Fernández[2]

Dept. of Informatics
King's College London, London WC2B 4BG, U.K.

Miguel Ramos[3]

DCC-FCUP & CRACS
University of Porto, Porto, Portugal

Abstract

We define EVL, a minimal higher-order functional language for dealing with generic events. The notion of generic event extends the well-known notion of event traditionally used in a variety of areas, such as database management, concurrency, reactive systems and cybersecurity. Generic events were introduced in the context of a metamodel to deal with obligations in access control systems. Event specifications are represented as records and we use polymorphic record types to type events in our language. We show how the higher-order capabilities of EVL can be used in the context of Complex Event Processing (CEP), to define higher-order parameterised functions that deal with the usual CEP techniques.

Keywords: events, access control, obligations, record types.

1 Introduction

In today's complex systems, where information is constantly being generated, there is a pressing need for efficiently processing data, and in particular data related with actions taking place in a system. Occurrences of actions, or happenings, are known as *events*, and technology for processing event streams, referred to as Complex Event

[1] Email: sandra@fc.up.pt
[2] Email: maribel.fernandez@kcl.ac.uk
[3] Email: jmiguelsramos@gmail.com

https://doi.org/10.1016/j.entcs.2020.08.002

Processing (CEP), has been around for decades [6,23,15]. Most of the CEP systems focus on real-life application therefore investing in issues such as scalability, fault tolerance, distribution, amongst others, but often lacking a formal semantics, making them difficult to understand, extend or generalize.

In the context of security, and in particular when modeling access control, it is often the case that granting or denying access to certain resources depends on the occurrence of a particular event [7,21,8]. This is even more crucial in access control systems dealing with obligations, where the status of a particular obligations is usually defined in terms of event occurrences in the system, and several models that deal with obligations have to deal in some way with the notion of event. The Category-Based metatamodel for Access Control and Obligations (CBACO [2]), defines an axiomatisation of the notion of obligation based on generic relations to type events and extract event intervals. A key notion in that model is to distinguish between event schemes, which provide a general description of the kind of events that can occur in a particular system, and specific events, which describe actual events that have occurred. Note that events can take various forms, depending on the system that is being considered (for example, messages exchanged over a network, actions performed by users of the system, occurrences of physical phenomena such as a disk error or a fire alarm, etc). To deal with event classification in a uniform way, Alves et. al. [1] defined a general term-based language for events. In this language, events were presented as typed-terms, built from a user-defined signature, that is, a particular set of typed function symbols that are specific to the system modelled. With this approach it is possible to define general functions to implement event typing and to compute event intervals, without needing to know the exact type of events.

In that context, a compound event [1] links a set of events that can occur separately in the history, but should be identified as a single event occurrence. For simplicity, in [1] compound events were assumed to appear as a single event in history, leaving a more detailed and realistic treatment of compound events for future work. This notion of compound or composite event is also a key feature in CEP systems, which put great emphasis on the ability to detect complex patterns of incoming streams of events and establish sequencing and ordering relations.

Types were used in [1] not only to ensure that terms representing events respect the type signature specific to the system under study, but also to formally define the notion of event instantiation, associating specific events to generic events through an implicit notion of subtyping, inspired by Ohori's system of polymorphic record types [24]. Because of the implicit subtyping rule for typing records, the system defined in [1] allowed for type-checking of event-specification, but not for dealing with most general types for event specifications.

In this paper we take a step further and define `EVL`, a higher-order polymorphic typed language, designed to facilitate the specification and processing of events. Our language is both a restriction and an extension of Ohori's polymorphic record calculus [24], and although our type system is very much based on that system, it is not meant to be a general language, but rather a language purposely designed for dealing with events. Languages traditionally used in event processing systems

are usually derived from relational languages, in particular, relational algebra and SQL, extended with additional ad-hoc operators to better support information flow or imperative programming. This paper explores the potential of the functional paradigm in this context, both at the level of the type-system, as well as exploring the higher-order capabilities of the language. The main contributions of this paper are:

- The design of EVL: a higher-order typed polymorphic record calculus for event processing. Our goal is to have a minimal language, but which is expressive enough to specify and manipulate events.
- A sound and complete type inference algorithm for EVL, defined as both an extension and a restriction of the ML-style record calculus in [24].
- A comprehensive study of the EVL higher-order/functional capabilities and its application in the context of CEP.

Overview

In Section 2 we define the EVL language and its set of types. In Section 3 we define a type system for EVL and in Section 4 we present a type inference algorithm, which is proved to be sound and complete. In Section 5 we discuss CEP and explore EVL's capabilities in this context. We discuss related work in Section 6 and we finally conclude and discuss further work in Section 7.

2 The EVL typed language

In this section we introduce EVL, a minimalistic typed language to specify events. EVL is an extension of the λ-calculus that includes records, a flexible data structure that is used here to deal with event specifications as defined in [1]. More precisely, an event specification is a labelled structure (or record) of the form $\{l_1 = v_1, \ldots, l_n = v_n\}$, representing a set of labels $l_1, \ldots, l_n$ with associated values $v_1, \ldots, v_n$. We assume some familiarity with the λ-calculus (see [5] for a detailed reference).

2.1 Terms

We start by formally defining the set of EVL terms. In the following, let $x, y, z, \ldots$ range over a countable set of variables and $l, l_1, \ldots$ range over a countable set $\mathcal{L}$ of labels.

Definition 2.1 The set of EVL terms is given by the following grammar:

$$\begin{aligned} M ::= {} & x \mid MM \mid \lambda x.M \mid \text{if } M \text{ then } M \text{ else } M \\ & \text{let } x = M \text{ in } M \mid \text{letEv } x = M \text{ in } M \\ & \{l = M, \ldots, l = M\} \mid M.l \mid \text{modify}(M, l, M) \end{aligned}$$

Notation: We will use the notation $let\ x\ x_1 \ldots x_n = M$ and $letEv\ x\ x_1 \ldots x_n = M$ for $let\ x = \lambda x_1 \ldots \lambda x_n.M$ and $letEv\ x = \lambda x_1 \ldots \lambda x_n.M$, respectively. As an

abuse of notation, in examples, we will use more meaningful names for functions, labels and events. Furthermore, event names will always start with a capital letter, to help distinguish them from functions.

We choose not to add other potentially useful constructors to the language, for instance pairs and projections, since we are aiming at a minimal language. Nevertheless, we can easily encode pairs (M_1, M_2) and projections $\pi_1 M$ and $\pi_2 M$ in our language by means of records of the form $\{\text{fst} = M_1, \text{snd} = M_2\}$ and $M.\text{fst}$, $M.\text{snd}$, respectively. This can trivially be extended to tuples in general, and we will often use this notation when writing examples.

Example 2.2 In this simple example, *FireDanger* reports the fire danger level of a particular location.

```
letEv FireDanger = λlλd.{location = l, fire_danger = d} in
FireDanger "Porto" "low"
```

To make our examples more readable, we will also use the following terms, abbreviating list construction:

```
nil = {empty = true}

cons x list = {empty = false, head = x, tail = list}.
```

Note that, much like what happens with tuples, the type of a particular list in this notation will be closely related to the size of the list in question. A more realistic approach is to add lists and list-types as primitive notions in the language, but, as we mentioned before, we are focusing on a minimal language. Furthermore, we will often use constants (numbers, booleans, strings, etc) and operators (arithmetic, boolean, etc) in our examples. However, following the minimalistic approach, we do not add constants/operators to the grammar and instead use free variables to represent them. Again, in a more general approach we could extend the grammar with other data structures and operators, for numbers, booleans, lists, etc.

Example 2.3 Consider the following example illustrating the definition of a generic event *FireDanger* and of a function *check* that determines if there is the danger of a fire erupting in a particular location, using the weather information associated with that location. Function *check* creates an appropriate instance of *FireDanger* to report the appropriate fire danger level.

```
letEv FireDanger l d = {location = l, fire_danger = d} in
let check x = if (x.temperature > 29 and x.wind > 32
                  and x.humidity < 20 and x.precipitation < 50)
              then FireDanger x.location "high"
              else FireDanger x.location "low" in
    check {temperature = 10, wind = 20, humidity = 30,
           precipitation = 10, location = "Porto"}
```

2.2 Types

We now define the set of types, and a typing system for the `EVL` language. We use record types to type labelled structures following Ohori's ML-system with poly-

morphic record types [24]. This extends the standard type system for parametric polymorphism by Damas and Milner [13]. In Ohori's system, polymorphic types are defined by type schemes of the form $\forall\alpha :: \kappa.\sigma$, where the type variable α is restricted to a set of types κ called a kind. We assume a finite set $\mathbb{B}$ of constant types and a countable set $\mathbb{V}$ of type variables, and we will use $b, b_1, \ldots$, $\alpha, \alpha_1, \ldots$ and $\kappa, \kappa_1, \ldots$ to denote constant types, type variables and kinds, respectively. The set $\mathbb{B}$ of constant types will always contain the type *bool*.

Definition 2.4 The set of types σ and kinds κ are specified by the following grammar.

$$\begin{aligned}
\sigma &::= \tau \mid \forall\alpha :: \kappa.\sigma \\
\tau &::= \alpha \mid b \mid \tau \to \tau \mid \{l : \tau, \ldots, l : \tau\} \\
\rho &::= \alpha \mid b \mid \tau \to \rho \\
\gamma &::= \tau \to \{l : \rho, \ldots, l : \rho\} \mid \{l : \rho, \ldots, l : \rho\} \\
\kappa &::= \mathcal{U} \mid \{\{l : \tau, \ldots, l : \tau\}\}
\end{aligned}$$

Following Damas and Milner's type system, we divide the set of types into *monotypes* (ranged over by τ) and *polytypes* (of the form $\forall\alpha :: \kappa.\sigma$). More precisely, σ represents all types and τ represents all monotypes. We denote by ρ (included in τ) the type of event fields, and by γ (also included in τ) the type of event definitions. This distinction is necessary to adequately type event definitions and its purpose will become clear in the definition of the typing system. We use $\mathcal{U}$ for the kind representing every possible type, and the kind $\{\{l_1 : \tau_1, \ldots, l_n : \tau_n\}\}$ represents record types that contain, at least, the fields $l_1, \ldots, l_n$, with types $\tau_1, \ldots, \tau_n$, respectively.

We do not allow nested events, and to that end we clearly separate types for event definitions, denoted by γ, and which are a subset of the general types denoted by τ. However, we do allow for nested records of general type. The following is an example of a term that is typed with a nested record type:

```
{empty = false , head = 1, tail = {empty = false , head = 2, tail = {empty = true} } }.
```

Let F range over functions from a finite set of labels to types. We write $\{F\}$ and $\{\{F\}\}$ to denote the record type identified by F and the record kind identified by F, respectively. For two functions F_1 and F_2 we write $F_1 \pm F_2$ for the function F such that $dom(F) = dom(F_1) \cup dom(F_2)$ and such that for $l \in dom(F)$, $F(l) = F_1(l)$ if $l \in dom(F_1)$; otherwise $F(l) = F_2(l)$.

Notation: Following the notation for pairs introduced above, we write $(\sigma_1 \times \sigma_2)$ for the product type corresponding to $\{\text{fst} : \sigma_1, \text{snd} : \sigma_2\}$.

A typing environment Γ is a set of statements $x : \sigma$ where all subjects x are distinct. We write $dom(\Gamma)$ to denote the domain of a typing environment $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$, which is the set $\{x_1, \ldots, x_n\}$. The type of a variable $x_i \in dom(\Gamma)$ is $\Gamma(x_i) = \sigma_i$, and we write Γ_x to denote $\Gamma \setminus \{x : \Gamma(x)\}$. A kinding environment K is a set of statements $\alpha :: \kappa$. Similarly, the domain of a kinding environment $K = \{\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n\}$, denoted $dom(K)$, is the set $\{\alpha_1, \ldots, \alpha_n\}$ and the kind of a type $\alpha_i \in dom(K)$ is $K(\alpha_i) = \kappa_i$. A type variable α occurring in a type/kind

is bound, if it occurs under the scope of a $\forall$-quantifier on α, otherwise it is free. We denote by $FTV(\sigma)$ ($FTV(\kappa)$) the set of free variables of σ (respectively, κ). We say that a type σ and a kind κ are well-formed under a kinding environment K if $FTV(\sigma) \subseteq dom(K)$ and $FTV(\kappa) \subseteq dom(K)$, respectively. A typing environment Γ is well-formed under a kinding environment K, if $\forall x \in dom(\Gamma)$, $\Gamma(x)$ is well-formed under K. A kinding environment K is well-formed, if $\forall \alpha \in dom(K), FTV(K(\alpha)) \subseteq dom(K)$. This reflects the fact that every free type variable in an expression has to be restricted by a kind in the kinding environment. Therefore, every type variable is either restricted by the kind in the type scheme or by a kind in the kinding environment.

Furthermore, we consider the set of essentially free type variables of a type σ under a kinding environment K (denoted as $EFTV(K, \sigma)$) as the smallest set such that, $FTV(\sigma) \subseteq EFTV(K, \sigma)$ and if $\alpha \in EFTV(K, \sigma)$, then $FTV(K(\alpha)) \subseteq EFTV(K, \sigma)$. This reflects the fact that a type variable α is essentially free in σ under a kinding environment K, if α is free in σ or in a restriction in K.

Definition 2.5 Let τ be a monotype, κ a kind, and K a kinding environment. Then we say that τ has kind κ under K (written $K \Vdash \tau :: \kappa$), if $\tau :: \kappa$ can be obtain by applying the following rules:

$$
\begin{aligned}
&K \Vdash \tau :: \mathcal{U} \text{ for all } \tau \text{ well-formed under } K \\
&K \Vdash \alpha :: \{\{l_1 : \tau_1, \ldots, l_n : \tau_n\}\} \text{ if } K(\alpha) = \{\{l_1 : \tau_1, \ldots, l_n : \tau_n, \ldots\}\} \\
&K \Vdash \{l_1 : \tau_1, \ldots, l_n : \tau_n, \ldots\} :: \{\{l_1 : \tau_1, \ldots, l_n : \tau_n\}\} \\
&\qquad \text{if } \{l_1 : \tau_1, \ldots, l_n : \tau_n, \ldots\} \text{ is well-formed under } K
\end{aligned}
$$

Note that, if $K \Vdash \sigma :: \kappa$, then σ and κ are well-formed under K.

Example 2.6 Let $\tau = \alpha_1 \to \{l_2 : int, l_3 : (\alpha_2 \times \alpha_3)\}$. Then, τ is well-formed under $K_1 = \{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}, \alpha_3 :: \mathcal{U}\}$, because $FTV(\tau) \subseteq dom(K_1)$, but not under $K_2 = \{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\}$, because $\alpha_3 \notin dom(K_2)$, and, therefore, $FTV(\tau) \not\subseteq dom(K_2)$. Because τ is well-formed under K_1, we can write $K_1 \Vdash \tau :: \mathcal{U}$.

3 Type assignment

We now define how types are assigned to `EVL` terms. Because we are dealing with polymorphic type schemes, we need to define the notion of *generic instance* for which we first need to discuss well-formed substitutions.

A substitution $S = [\sigma_1/\alpha_1, \ldots, \sigma_n/\alpha_n]$ is *well-formed under a kinding environment* K, if for all $\alpha \in dom(S)$, $S(\alpha)$ is well-formed under K. This reflects the fact that applying a substitution to a type that is well-formed under a kinding environment K, should result in a type that is also well-formed under K. A *kinded substitution* is a pair (K, S) of a kind assignment K and a substitution S that is well-formed under K. This reflects the fact that a substitution S should only be applied to a type that is well-formed under S, such that the resulting type is kinded by K.

Example 3.1 Let $S = [\alpha_2/\alpha_1]$ be a substitution. Then $dom(S) = \{\alpha_1\}$, $S(\alpha_1) = \alpha_2$, and $FTV(\alpha_2) = \{\alpha_2\}$. For the kinding environment $K_1 = \{\alpha_2 :: \kappa\}$, we have that S is well-formed under K_1, since $\alpha_2 \in dom(K_1)$. On the other hand, for the kinding environment $K_2 = \{\alpha_3 :: \kappa\}$, we have that S is not well-formed under K_2, since $\alpha_2 \notin dom(K_2)$.

Definition 3.2 We say that a kinded substitution (K_1, S) respects a kinding environment K_2, if $\forall \alpha \in dom(K_2), K_1 \Vdash S(\alpha) :: S(K_2(\alpha))$.

Example 3.3 Let $K_1 = \{\alpha_1 :: \{\{l_1 : \alpha_2\}\}, \alpha_2 :: \mathcal{U}\}$ and $S = [\{l_1 : int\}/\alpha_1]$. Then, the restricted substitution (K_1, S) respects $K_2 = \{\alpha_1 :: \{\{l_1 : int\}\}\}$, because for $dom(K_2) = \{\alpha_1\}$, we have:

$$
\begin{aligned}
&K_1 \Vdash S(\alpha_1) :: S(K_2(\alpha_1)) \\
&K_1 \Vdash S(\alpha_1) :: S(\{\{l_1 : int\}\}) \\
&K_1 \Vdash S(\alpha_1) :: \{\{l_1 : S(int)\}\} \\
&K_1 \Vdash \{l_1 : int\} :: \{\{l_1 : int\}\}
\end{aligned}
$$

Lemma 3.4 *If $FTV(\sigma) \subseteq dom(K)$ and (K_1, S) respects K, then $FTV(S(\sigma)) \subseteq dom(K_1)$.*

Proof. If (K_1, S) respects K, then we know that $\forall \alpha \in dom(K), K_1 \Vdash S(\alpha) :: S(K(\alpha))$. This means that, if $FTV(\sigma) \subseteq dom(K)$, then $\forall \alpha' \in FTV(\sigma), K_1 \Vdash S(\alpha') :: S(K(\alpha'))$. Consequently, both $S(\alpha')$ and $S(K(\alpha'))$ are well formed under K_1, which means that $FTV(S(\alpha')) \subseteq dom(K_1)$ and $FTV(S(K(\alpha'))) \subseteq dom(K_1)$. Therefore, it is now easy to see that $FTV(S(\sigma)) \subseteq dom(K_1)$.

□

Lemma 3.5 *If $K \Vdash \sigma :: \kappa$, and a kinded substitution (K_1, S) respects K, then $K_1 \Vdash S(\sigma) :: S(\kappa)$.*

Proof. The proof follows the definition of $K \Vdash \sigma :: \kappa$ □

Definition 3.6 Let σ_1 be a well-formed type under a kinding environment K. Then, σ_2 is a generic instance of σ_1 under K (denoted as $K \Vdash \sigma_1 \geq \sigma_2$), if $\sigma_1 = \forall \alpha_1 :: \kappa_1^1 \cdots \forall \alpha_n :: \kappa_n^1.\tau_1$, $\sigma_2 = \forall \beta_1 :: \kappa_1^2 \cdots \forall \beta_m :: \kappa_m^2.\tau_2$, and there exists a substitution S such that $dom(S) = \{\alpha_1, \ldots, \alpha_n\}$, $(K \cup \{\beta_1 :: \kappa_1^2, \ldots, \beta_m :: \kappa_m^2\}, S)$ respects $K \cup \{\alpha_1 :: \kappa_1^1, \ldots, \alpha_n :: \kappa_n^1\}$, and $\tau_2 = S(\tau_1)$.

Definition 3.7 Let Γ be a typing environment and τ be a type, both well-formed under a kinding environment K. The closure of τ under Γ and K (denoted as $Cls(K, \Gamma, \tau)$) is a pair $(K', \forall \alpha_1 :: \kappa_1 \cdots \forall \alpha_n :: \kappa_n.\tau)$ such that $K' \cup \{\alpha_1 :: \kappa_1, \ldots \alpha_n :: \kappa_n\} = K$ and $\{\alpha_1, \ldots, \alpha_n\} = EFTV(K, \tau) \setminus EFTV(K, \Gamma)$.

Example 3.8 Let $K = \{\alpha_2 :: \mathcal{U}, \alpha_3 :: \mathcal{U}, \alpha_4 :: \mathcal{U}, \alpha_1 :: \{\{l_1 : \alpha_2\}\}\}$, $\Gamma = \{x : \alpha_1\}$, and $\tau = \{l_1 : \alpha_2, l_4 : bool\} \to \{l_2 : int, l_3 : (\alpha_3 \times \alpha_4)\}$. Then $Cls(K, \Gamma, \tau) = (\{\alpha_2 :: \mathcal{U}, \alpha_1 :: \{\{l_1 : \alpha_2\}\}\}, \forall \alpha_3 :: \mathcal{U}. \forall \alpha_4 :: \mathcal{U}. \{l_1 : \alpha_2, l_4 : bool\} \to \{l_2 : int, l_3 : (\alpha_3 \times \alpha_4)\})$, because $K = \{\alpha_2 :: \mathcal{U}, \alpha_1 :: \{\{l_1 : \alpha_2\}\}\} \cup$

$\{\alpha_3 :: \mathcal{U}, \alpha_4 :: \mathcal{U}\}$, $EFTV(K, \tau) = \{\alpha_2, \alpha_3, \alpha_4, \alpha_1\}$, $EFTV(K, \Gamma) = \{\alpha_1, \alpha_2\}$, and $EFTV(K, \tau) \setminus EFTV(K, \Gamma) = \{\alpha_2, \alpha_3, \alpha_4, \alpha_1\} \setminus \{\alpha_1, \alpha_2\} = \{\alpha_3, \alpha_4\}$.

The type assignment system for EVL is given in Figure 1, and can be seen as both a restriction and an extension of the Ohori type system for record types. Unlike Ohori, we do not deal with variant types in this system, but we have additional language constructors, like conditionals and explicit event definition. We use $K, \Gamma \vdash M : \sigma$ to denote that the EVL term M has type σ given the type and kind environments Γ and K, respectively.

$$\frac{K \Vdash \Gamma(x) \geq \tau,\ \Gamma \textit{ is well-formed under } K}{K, \Gamma \vdash x : \tau} \text{ (Var)}$$

$$\frac{K, \Gamma \vdash M_1 : \tau_1 \to \tau_2 \qquad K, \Gamma \vdash M_2 : \tau_1}{K, \Gamma \vdash M_1\ M_2 : \tau_2} \text{ (App)}$$

$$\frac{K, \Gamma_x \cup \{x : \tau_1\} \vdash M : \tau_2}{K, \Gamma_x \vdash \lambda x.M : \tau_1 \to \tau_2} \text{ (Abs)}$$

$$\frac{K', \Gamma_x \vdash M_1 : \tau' \qquad \mathit{Cls}(K', \Gamma_x, \tau') = (K, \sigma) \qquad K, \Gamma_x \cup \{x : \sigma\} \vdash M_2 : \tau}{K, \Gamma_x \vdash \text{let } x = M_1 \text{ in } M_2 : \tau} \text{ (Let)}$$

$$\frac{K', \Gamma_x \vdash M_1 : \gamma \qquad \mathit{Cls}(K', \Gamma_x, \gamma) = (K, \sigma) \qquad K, \Gamma_x \cup \{x : \sigma\} \vdash M_2 : \tau}{K, \Gamma_x \vdash \text{letEv } x = M_1 \text{ in } M_2 : \tau} \text{ (LetEv)}$$

$$\frac{K, \Gamma \vdash M_i : \tau_i, 1 \leq i \leq n}{K, \Gamma \vdash \{l_1 = M_1, \ldots, l_n = M_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}, n \geq 1} \text{ (Rec)}$$

$$\frac{K, \Gamma \vdash M : \tau' \qquad K \Vdash \tau' :: \{\{l : \tau\}\}}{K, \Gamma \vdash M.l : \tau} \text{ (Sel)}$$

$$\frac{K, \Gamma \vdash M_1 : \tau \qquad K, \Gamma \vdash M_2 : \tau' \qquad K \Vdash \tau :: \{\{l : \tau'\}\}}{K, \Gamma \vdash \text{modify}(M_1, l, M_2) : \tau} \text{ (Modif)}$$

$$\frac{K, \Gamma \vdash M_1 : \mathit{bool} \qquad K, \Gamma \vdash M_2 : \tau \qquad K, \Gamma \vdash M_3 : \tau}{K, \Gamma \vdash \text{if } M_1 \text{ then } M_2 \text{ else } M_3 : \tau} \text{ (Cond)}$$

Fig. 1. Type assignment system for EVL

Example 3.9 Let $M = \{location = l, fire\_danger = d\}$, $\tau_1 = \{location : \alpha_1, fire\_danger : \alpha_2\}$, $\tau_2 = \forall\alpha_1 :: \mathcal{U}.\forall\alpha_2 :: \mathcal{U}.\alpha_1 \to \alpha_2 \to \tau_1$, $\tau_3 = \{location : l_{type}, fire\_danger : fd_{type}\}$, $\tau_4 = l_{type} \to fd_{type} \to \tau_3$, and $\Gamma = \{\text{“}Porto\text{”} : l_{type}, \text{“}low\text{”} : fd_{type}\}$. A type derivation for M is given in Figure 2.

Lemma 3.10 *If $K, \Gamma \vdash M : \sigma$ and (K_1, S) respects K, then $K_1, S(\Gamma) \vdash M : S(\sigma)$.*

Proof. By induction on $K, \Gamma \vdash M : \sigma$. □

$$\Phi_1 = \dfrac{\dfrac{\dfrac{\dfrac{}{\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\}, \{l : \alpha_1, d : \alpha_2\} \cup \Gamma \vdash l : \alpha_1}\,(\text{Var}) \quad \dfrac{}{\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\}, \{l : \alpha_1, d : \alpha_2\} \cup \Gamma \vdash d : \alpha_2}\,(\text{Var})}{\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\}, \{l : \alpha_1, d : \alpha_2\} \cup \Gamma \vdash M : \tau_1}\,(\text{Rec})}{\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\}, \{l : \alpha_1\} \cup \Gamma \vdash \lambda d.M : \alpha_2 \to \tau_1}\,(\text{Abs})}{\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\}, \Gamma \vdash \lambda l.\lambda d.M : \alpha_1 \to \alpha_2 \to \tau_1}\,(\text{Abs})$$

$$\Phi_2 = \mathit{Cls}(\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\}, \Gamma, \alpha_1 \to \alpha_2 \to \tau_1) = (\{\}, \tau_2)$$

$$\Phi_3 = \dfrac{\dfrac{}{\{\}, \{\mathit{FireDanger} : \tau_2\} \cup \Gamma \vdash \mathit{FireDanger} : \tau_4}\,(\text{Var}) \quad \dfrac{}{\{\}, \{\mathit{FireDanger} : \tau_2\} \cup \Gamma \vdash \text{“}\mathit{Porto}\text{”} : l_{type}}\,(\text{Var})}{\{\}, \{\mathit{FireDanger} : \tau_2\} \cup \Gamma \vdash \mathit{FireDanger}\ \text{“}\mathit{Porto}\text{”} : fd_{type} \to \tau_3}\,(\text{App})$$

$$\Phi_4 = \dfrac{\Phi_3 \qquad \dfrac{}{\{\}, \{\mathit{FireDanger} : \tau_2\} \cup \Gamma \vdash \text{“}\mathit{low}\text{”} : fd_{type}}\,(\text{Var})}{\{\}, \{\mathit{FireDanger} : \tau_2\} \cup \Gamma \vdash \mathit{FireDanger}\ \text{“}\mathit{Porto}\text{”}\ \text{“}\mathit{low}\text{”} : \tau_3}\,(\text{App})$$

$$\dfrac{\Phi_1 \qquad \Phi_2 \qquad \Phi_4}{\{\}, \Gamma \vdash \text{letEv}\ \mathit{FireDanger} = \lambda l.\lambda d.M\ \text{in}\ \mathit{FireDanger}\ \text{“}\mathit{Porto}\text{”}\ \text{“}\mathit{low}\text{”} : \tau_3}\,(\text{LetEv})$$

Fig. 2. Type derivation for $M = \text{letEv}\ \mathit{FireDanger} = \lambda l.\lambda d.M\ \text{in}\ \mathit{FireDanger}\ \text{“}\mathit{Porto}\text{”}\ \text{“}\mathit{low}\text{”}$

4 A type inference algorithm for EVL

We now adapt Ohori's $WK(K, \Gamma, M)$ inference algorithm to our language. It uses a refinement of Robinson's unification algorithm [27] that considers kind constraints on type variables. We start by discussing *kinded unification* for EVL types.

4.1 Kinded unification

A *kinded set of equations* is a pair (K, E), where K is a kinding environment and E is a set of pairs of types (τ_1, τ_2) that are well-formed under K. A *kinded substitution* (K, S) is a unifier of a kinded set of equations (K, E), if every type that appears in E respects K, and $\forall (\tau_1, \tau_2) \in E, S(\tau_1) = S(\tau_2)$ (S satisfies E). A kinded substitution (K_1, S_1) is the *most general unifier* of (K, E) if it is a unifier of (K, E) and if for any other unifier (K_2, S_2) of (K, E) there is some substitution S_3 such that (K_2, S_3) respects K_1 and $S_2 = S_3 \circ S_1$.

The *kinded unification algorithm*, $U(E, K)$, is defined by the transformation rules in Figure 3. Each rule is of the form $(E_1, K_1, S_1) \Rightarrow (E_2, K_2, S_2)$, where E_1, E_2 are sets of pairs of types, K_1, K_2 are kinding environments, and S_1, S_2 are substitutions. After a transformation step, E_2 keeps the set of pairs of types to be unified, K_2 specifies kind constraints to be verified, and S_2 is the substitution resulting from unifying the pairs of types that have been removed from E. Given a kinded set of equations (K_1, E_1) the algorithm $U(E_1, K_1)$ proceeds by applying the transformation rules to $(E_1, K_1, \emptyset)$, until no more rules can be applied, resulting in a triple (E, K, S). If $E = \emptyset$ then it returns the pair (K, S), otherwise it reports failure.

Example 4.1 Let α_1 and α_2 be two type variables and $K = \{\alpha_1 :: \{\{\mathit{location} : \alpha_3\}\}$,

$$(E \cup \{(\tau,\tau)\}, K, S) \Rightarrow (E, K, S)$$

$$(E \cup \{(\alpha,\tau)\}, K \cup \{(\alpha,\mathcal{U})\}, S) \Rightarrow ([\tau/\alpha]E, [\tau/\alpha]K, [\tau/\alpha]S \cup \{(\alpha,\tau)\})$$

$$(E \cup \{(\tau,\alpha)\}, K \cup \{(\alpha,\mathcal{U})\}, S) \Rightarrow ([\tau/\alpha]E, [\tau/\alpha]K, [\tau/\alpha]S \cup \{(\alpha,\tau)\})$$

$$\begin{array}{rcl}(E \cup \{(\alpha_1,\alpha_2)\}, K \cup \{(\alpha_1,\{\{F_1\}\}),(\alpha_2,\{\{F_2\}\})\}, S) & \Rightarrow & ([\alpha_2/\alpha_1](E \cup \{(F_1(l),F_2(l)) \mid l \in dom(F_1) \cap dom(F_2)\}),\\ & & [\alpha_2/\alpha_1](K) \cup \{(\alpha_2,[\alpha_2/\alpha_1](\{\{F_1 \pm F_2\}\}))\},\\ & & [\alpha_2/\alpha_1](S) \cup \{(\alpha_1,\alpha_2)\})\end{array}$$

$$\begin{array}{rcl}(E \cup \{(\alpha,\{F_2\})\}, K \cup \{(\alpha,\{\{F_1\}\})\}, S) & \Rightarrow & ([\{F_2\}/\alpha](E \cup \{(F_1(l),F_2(l)) \mid l \in dom(F_1)\}),\\ & & [\{F_2\}/\alpha](K),\\ & & [\{F_2\}/\alpha](S) \cup \{(\alpha,\{F_2\})\})\\ & & \text{if } dom(F_1) \subseteq dom(F_2) \text{ and } \alpha \notin \mathrm{FTV}(\{F_2\})\end{array}$$

$$\begin{array}{rcl}(E \cup \{(\{F_2\},\alpha)\}, K \cup \{(\alpha,\{\{F_1\}\})\}, S) & \Rightarrow & ([\{F_2\}/\alpha](E \cup \{(F_1(l),F_2(l)) \mid l \in dom(F_1)\}),\\ & & [\{F_2\}/\alpha](K),\\ & & [\{F_2\}/\alpha](S) \cup \{(\alpha,\{F_2\})\})\\ & & \text{if } dom(F_1) \subseteq dom(F_2) \text{ and } \alpha \notin \mathrm{FTV}(\{F_2\})\end{array}$$

$$\begin{array}{rcl}(E \cup \{(\{F_1\},\{F_2\})\}, K, S) & \Rightarrow & (E \cup \{(F_1(l),F_2(l)) \mid l \in dom(F_1)\}, K, S)\\ & & \text{if } dom(F_1) = dom(F_2)\end{array}$$

$$(E \cup \{(\tau_1^1 \to \tau_1^2, \tau_2^1 \to \tau_2^2\}, K, S) \Rightarrow (E \cup \{(\tau_1^1,\tau_2^1),(\tau_1^2,\tau_2^2)\}, K, S)$$

Fig. 3. Kinded Unification

$\alpha_2 :: \{\{fire\_danger : fd_{type}, location : l_{type}\}\}, \alpha_3 :: \mathcal{U}\}.$

$$\begin{array}{l}(\{(\alpha_1,\alpha_2)\},\{(\alpha_1,\{\{location : \alpha_3\}\}),(\alpha_2,\{\{fire\_danger : fd_{type}, location : l_{type}\}\}),(\alpha_3,\mathcal{U})\},\{\})\\ \Rightarrow (\{(\alpha_3,l_{type})\},\{(\alpha_2,\{\{fire\_danger : fd_{type}, location : \alpha_3\}\}),(\alpha_3,\mathcal{U})\},\{(\alpha_1,\alpha_2)\})\\ \Rightarrow (\{\},\{(\alpha_2,\{\{fire\_danger : fd_{type}, location : l_{type}\}\})\},\{(\alpha_1,\alpha_2),(\alpha_3,l_{type})\})\end{array}$$

The most general unifier between α_1 and α_2 is the kinded substitution $(\{\alpha_2 :: \{\{fire\_danger : fd_{type}, location : l_{type}\}\}\}, [\alpha_2/\alpha_1, l_{type}/\alpha_3])$. Following Ohori's notation, in the kinded unification algorithm we use pairs in the representation of substitutions and kind assignments. Also, note that the unification algorithm in [24] has a kind assignment as an extra parameter, which is used to record the solved kind constraints encountered through the unification process. However, we choose to omit this parameter because its information is only used in the proofs in [24] but not in the unification process itself.

In [24], the correctness and completeness of the kinded unification algorithm was proved, in the sense that it takes any kinded set of equations and computes its most general unifier if one exists and reports failure otherwise.

4.2 Type inference

The *type inference algorithm*, $WK(K, \Gamma, M)$, is defined in Figure 4. Given a kinding environment K, a typing environment Γ, and an EVL term M, then $WK(K_1, \Gamma, M) =$

(K', S, σ), such that σ is the type of M under the kinding environment K' and typing environment $S(\Gamma)$. It is implicitly assumed that the inference algorithm fails if unification or any of the recursive calls on subterms fails.

$$
\begin{aligned}
WK(K,\Gamma,x) &= \text{if } x \in d\not{o}m(\Gamma) \text{ then } \mathit{fail} \\
&\quad \text{else let } \forall\alpha_1 :: \kappa_1 \cdots \forall\alpha_n :: \kappa_n.\tau = \Gamma(x), \\
&\qquad S = [\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n] \ (\beta_1, \ldots, \beta_n \text{ are fresh}) \\
&\qquad \text{in } (K \cup \{\beta_1 :: S(\kappa_1), \ldots, \beta_n :: S(\kappa_n)\}, id, S(\tau)) \\
WK(K,\Gamma,M_1\ M_2) &= \text{let } (K_1, S_1, \tau_1) = WK(K,\Gamma,M_1) \\
&\quad (K_2, S_2, \tau_2) = WK(K_1, S_1(\Gamma), M_2) \\
&\quad (K_3, S_3) = U(K_2, \{(S_2(\tau_1), \tau_2 \to \alpha)\}) \ (\alpha \text{ is fresh}) \\
&\quad \text{in } (K_3, S_3 \circ S_2 \circ S_1, S_3(\alpha)) \\
WK(K,\Gamma,\lambda x.M) &= \text{let } (K_1, S_1, \tau) = WK(K \cup \{\alpha :: \mathcal{U}\}, \Gamma \cup \{x : \alpha\}, M) \ (\alpha \text{ fresh}) \\
&\quad \text{in } (K_1, S_1, S_1(\alpha) \to \tau) \\
WK(K,\Gamma,\text{let } x = M_1 \text{ in } M_2) &= \text{let } (K_1, S_1, \tau_1) = WK(K,\Gamma,M_1) \\
&\quad (K_1', \sigma) = \mathit{Cls}(K_1, S_1(\Gamma), \tau_1) \\
&\quad (K_2, S_2, \tau_2) = WK(K_1', S_1(\Gamma) \cup \{x : \sigma\}, M_2) \\
&\quad \text{in } (K_2, S_2 \circ S_1, \tau_2) \\
WK(K,\Gamma,\text{letEv } x = M_1 \text{ in } M_2) &= \text{let } (K_1, S_1, \gamma) = WK(K,\Gamma,M_1) \\
&\quad (K_1', \sigma) = \mathit{Cls}(K_1, S_1(\Gamma), \gamma) \\
&\quad (K_2, S_2, \tau_2) = WK(K_1', S_1(\Gamma) \cup \{x : \sigma\}, M_2) \\
&\quad \text{in } (K_2, S_2 \circ S_1, \tau_2) \\
WK(K,\Gamma,\{l_1 = M_1, \ldots, l_n = M_n\}) &= \text{let } (K_1, S_1, \tau_1) = WK(K,\Gamma,M_1) \\
&\quad (K_i, S_i, \tau_i) = WK(K_{i-1}, S_{i-1} \circ \cdots \circ S_1(\Gamma), M_i) \ (2 \le i \le n) \\
&\quad \text{in } (K_n, S_n \circ \cdots \circ S_2 \circ S_1, \\
&\quad \{l_1 : S_n \circ \cdots \circ S_2(\tau_1), \ldots, l_i : S_n \circ \cdots \circ S_{i+1}(\tau_i), \ldots, l_n : \tau_n\}) \\
WK(K,\Gamma,M.l) &= \text{let } (K_1, S_1, \tau_1) = WK(K,\Gamma,M) \\
&\quad (K_2, S_2) = U(K_1 \cup \{\alpha_1 :: \mathcal{U}, \alpha_2 :: \{\{l : \alpha_1\}\}\}, \{(\alpha_2, \tau_1)\}) \ (\alpha_1, \alpha_2 \text{ fresh}) \\
&\quad \text{in } (K_2, S_2 \circ S_1, S_2(\alpha_1)) \\
WK(K,\Gamma,\text{modify}(M_1, l, M_2)) &= \text{let } (K_1, S_1, \tau_1) = WK(K,\Gamma,M_1) \\
&\quad (K_2, S_2, \tau_2) = WK(K_1, S_1(\Gamma), M_2) \\
&\quad (K_3, S_3) = U(K_2 \cup \{\alpha_1 :: \mathcal{U}, \alpha_2 :: \{\{l : \alpha_1\}\}\}, \{(\alpha_1, \tau_2), (\alpha_2, S_2(\tau_1))\}) \\
&\quad (\alpha_1, \alpha_2 \text{ are fresh}) \\
&\quad \text{in } (K_3, S_3 \circ S_2 \circ S_1, S_3(\alpha_2)) \\
WK(K,\Gamma,\text{if } M_1 \text{ then } M_2 \text{ else } M_3) &= \text{let } (K_1, S_1, \tau_1) = WK(K,\Gamma,M_1) \\
&\quad (K_2, S_2) = U(K_1, \{(\tau_1, \mathit{bool})\}) \\
&\quad (K_3, S_3, \tau_2) = WK(K_2, S_2 \circ S_1(\Gamma), M_2) \\
&\quad (K_4, S_4, \tau_3) = WK(K_3, S_3 \circ S_2 \circ S_1(\Gamma), M_3) \\
&\quad (K_5, S_5) = U(K_4, \{(S_4(\tau_2), \tau_3)\}) \\
&\quad \text{in } (K_5, S_5 \circ S_4 \circ S_3 \circ S_2 \circ S_1, S_5 \circ S_4(\tau_2))
\end{aligned}
$$

Fig. 4. Type inference algorithm

Example 4.2 Following Example 3.9, we consider $M = \{location = l, fire\_danger = d\}$, $\tau_1 = \{location : \alpha_1, fire\_danger : \alpha_2\}$, $\tau_2 = \forall\alpha_1 :: \mathcal{U}.\forall\alpha_2 :: \mathcal{U}.\alpha_1 \to \alpha_2 \to \tau_1$, $\tau_3 = \{location : l_{type}, fire\_danger : fd_{type}\}$, and $\tau_4 = l_{type} \to fd_{type} \to \tau_3$, we further consider $\tau_5 = \{location : \alpha_3, fire\_danger : \alpha_4\}$, $\tau_6 = \forall\alpha_3 :: \mathcal{U}.\forall\alpha_4 :: \mathcal{U}.\alpha_3 \to \alpha_4 \to \tau_5$, $\tau_7 = \{location : \alpha_5, fire\_danger : \alpha_6\}$, $S_1 = [l_{type}/\alpha_5] \circ [\alpha_5/\alpha_3, \alpha_6/\alpha_4]$, and $S_2 = [fd_{type}/\alpha_6] \circ S_1 \circ [\alpha_4/\alpha_2] \circ [\alpha_3/\alpha_1]$. A run of the algorithm for letEv $FireDanger = \lambda l.\lambda d.M$ in $FireDanger$ "$Porto$" "low"

is given in Figure 5.

$$WK(\{\},\{\},\text{letEv } FireDanger = \lambda l.\lambda d.M \text{ in } FireDanger \text{ “}Porto\text{” “}low\text{”}) = (\{\}, S_2, \tau_3)$$

$$WK(\{\},\{\},\lambda l.\lambda d.M) = (\{\alpha_3 :: \mathcal{U}, \alpha_4 :: \mathcal{U}\}, [\alpha_4/\alpha_2] \circ [\alpha_3/\alpha_1], \alpha_3 \to \alpha_4 \to \tau_5)$$
$$WK(\{\alpha_1 :: \mathcal{U}\}, \{l : \alpha_1\}, \lambda d.M) = (\{\alpha_3 :: \mathcal{U}, \alpha_4 :: \mathcal{U}\}, [\alpha_4/\alpha_2] \circ [\alpha_3/\alpha_1], \alpha_4 \to \tau_5)$$
$$WK(\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\}, \{l : \alpha_1, d : \alpha_2\}, M) = (\{\alpha_3 :: \mathcal{U}, \alpha_4 :: \mathcal{U}\}, [\alpha_4/\alpha_2] \circ [\alpha_3/\alpha_1], \tau_5)$$
$$WK(\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\}, \{l : \alpha_1, d : \alpha_2\}, l) = (\{\alpha_2 :: \mathcal{U}, \alpha_3 :: \mathcal{U}\}, [\alpha_3/\alpha_1], \alpha_3)$$
$$WK(\{\alpha_2 :: \mathcal{U}, \alpha_3 :: \mathcal{U}\}, \{l : \alpha_3, d : \alpha_2\}, d) = (\alpha_3 :: \mathcal{U}, \alpha_4 :: \mathcal{U}\}, [\alpha_4/\alpha_2], \alpha_4)$$

$$Cls(\{\alpha_3 :: \mathcal{U}, \alpha_4 :: \mathcal{U}\}, \{\}, \alpha_3 \to \alpha_4 \to \tau_5) = (\{\}, \tau_6)$$

$$WK(\{\},\{FireDanger : \tau_6\}, FireDanger \text{ “}Porto\text{” “}low\text{”}) = (\{\}, [fd_{type}/\alpha_6] \circ S_1, \tau_3)$$
$$WK(\{\},\{FireDanger : \tau_6\}, FireDanger \text{ “}Porto\text{”}) = (\{\}, S_1, \alpha_6 \to \{location : l_{type}, fire\_danger : \alpha_6\})$$
$$WK(\{\},\{FireDanger : \tau_6\}, FireDanger) = (\{\}, [\alpha_5/\alpha_3, \alpha_6/\alpha_4], \alpha_5 \to \alpha_6 \to \tau_7)$$
$$WK(\{\},\{FireDanger : \tau_6\}, \text{“}Porto\text{”}) = (\{\}, id, l_{type})$$
$$U(\{\}, (\alpha_5, l_{type})) = (\{\}, [l_{type}/\alpha_5])$$
$$WK(\{\},\{FireDanger : \tau_6\}, \text{“}low\text{”}) = (\{\}, id, fd_{type})$$
$$U(\{\}, (\alpha_6, fd_{type})) = (\{\}, [fd_{type}/\alpha_6])$$

Fig. 5. Type inference run for letEv $FireDanger = \lambda l.\lambda d.M$ in $FireDanger$ “$Porto$” “low”

4.3 Soundness and completeness of WK

In this section we establish the results from soundness and completeness of our type inference algorithm.

Theorem 4.3 *If $WK(K, \Gamma, M) = (K', S, \tau)$ then (K', S) respects K and there is a derivation in our type system such that $K', S(\Gamma) \vdash M : \tau$.*

Proof. The proof is by induction on the structure of M. □

Theorem 4.4 *If $WK(K, \Gamma, M) = fail$, then there is no (K_0, S_0) and τ_0 such that (K_0, S_0) respects K and $K_0, S_0(\Gamma) \vdash M : \tau_0$.*
If $WK(K, \Gamma, M) = (K', S, \tau)$, then if $K_0, S_0(\Gamma) \vdash M : \tau_0$ for some (K_0, S_0) and τ_0 such that (K_0, S_0) respects K, then there is some S' such that (K_0, S') respects K', $\tau_0 = S'(\tau)$, and $S_0(\Gamma) = S' \circ S(\Gamma)$.

Proof. The proof is by induction on the structure of M. □

5 EVL for Complex Event Processing

In this section we are going to study the application of EVL in the context of Complex Event Processing (CEP). See [15] for a detailed reference on the area. The area of CEP comprises a series of techniques to deal with streams of events such as event processing, detection of patterns and relationships, filtering, transformation and abstraction, amongst others. Because EVL is a higher-order functional language, we are going to explore the higher-order capabilities, to define higher-order parameterized functions that deal with the usual CEP techniques.

5.1 Event processing

The canonical model [10,15] for event processing is based on a producer-consumer model: an event processing agent (EPA) takes events from event producers and distributes them among event consumers. This process often involves filtering or translating. Filtering may happen because not every event will be of interest or available to every consumer: in some cases access control policies might be in place that restrict what events consumers might receive. Translating events allows us to change, add or remove information from the agents based on particular consumers. The processing of events can be done in a one-event in/one-event out form, but it is also possible to have event processing agents that process a collection of events as a whole or that produce a set of events as result: for example, an incoming event may be split into multiple events, each containing a subset of the information from the original event.

`EVL` can be used to program event processing agents. It is able to process raw events produced by some event processing system and generate derived events as a result. These derived events can then be passed on to an event consumer.

Principal types allow us to identify event processing agents. An event processing agent is any function whose principal types is of the form $\forall\alpha_1 :: \kappa_1 \cdots \forall\alpha_n :: \kappa_n.\gamma$. In the following section we will explore the different types of event processing agents.

5.1.1 Types of event processing agents

Event processing agents are classified according to the actions that they perform to process incoming events:

- *Filter agents* - Filter agents take an incoming event object and apply a test to decide whether to discard it or whether to pass it on for processing by subsequent agents. The test is usually stateless, i.e. based solely on the content of the event instance.
- *Transformation agents* - Transformation agents modify the content of the event objects that they receive. These agents can be further classified based on the cardinality of their inputs and outputs (translate, split, aggregate or compose agents)
- *Pattern Detect agents* - Pattern Detect agents take collections of incoming event objects and examine them to see if they can spot the occurrence of particular patterns.

We are now going to look into the different types of event processing agents in a little more depth. All of the definitions that we are going to present can be found in [15].

Definition 5.1 [Filter event processing agent] A *filter agent* is an event processing agent that performs filtering only, so it does not transform the input event.

Example 5.2 This example represents an event processing agent that uses a higher-order filter function *filter* (to be defined later) to filter events according to their location.

```
let p x = (x.location == "Porto") in λx.(filter p x)
```

Definition 5.3 [Transformation event processing agent] A Transformation event processing agent is an event processing agent that includes a derivation step, and optionally also a filtering step.

Transformation event processing agents can be either stateless (if events are processed without taking into account preceding or following events) or stateful (if the way events are processed is influenced by preceding or following events). In the former case, events are processed individually. In the latter, the way events are processed can depend on preceding or succeeding events. Transformation events can be further classified as translate, split, aggregate or compose agents. In the following we describe some transformation events that we are going to focus on.

Definition 5.4 [Translate event processing agent] Translate event processing agents can be used to convert events from one type to another, or to add, remove, or modify the values of an event's attributes.

At the moment `EVL` does not allow us to add or remove attributes. One can create new events based on attributes from incoming events as well as modify the value of existing attributes. We follow Ohori's treatment of record types, therefore we do not consider operations that extend a record with a new field or that remove an existing field from a record. This is a limitation of our current calculus, and we intend to improve on this in the future.

Example 5.5 This example represents an event processing agent that converts the temperature field of an event from degrees Fahrenheit to degrees Celsius.

```
farToCel x = modify(x, temperature, (x.temperature−32)/1.8)
```

Definition 5.6 [Aggregate event processing agent] Aggregate agents take a stream of incoming events and produce an output event that is a map of the incoming events.

Example 5.7 This example represents an event processing agent that receives two events, x and y, and outputs event y with its precipitation level updated with the average of the two.

```
avg x y = modify(y, precipitation, (x.precipitation + y.precipitation)/2)
```

Definition 5.8 [Compose event processing agent] Compose agents take two streams of incoming events and process them to produce a single output stream of events.

Example 5.9 This example represents an event processing agent that composes the partial weather information that is provided by two different sensors. One of the sensors outputs event x, which contains information about the temperature and wind velocity, and the other sensor outputs event y, which contains information about the humidity and precipitation levels. This event processing agent outputs an instance of *WeatherInfo* with the complete weather information.

```
composeInfo x y = WeatherInfo x.temperature x.wind y.humidity y.precipitation
```

Definition 5.10 [Pattern Detect event processing agent] A Pattern Detect event processing agent is an event processing agent that performs a pattern matching function on one or more input streams. It emits one or more derived events if it detects an occurrence of the specified pattern in the input events.

Example 5.11 The *check* function in Example 2.3 is an event processing agent that generates the appropriate *FireDanger* event by detecting its corresponding fire weather information.

```
check x = if (x.temperature > 29 and x.wind > 32
              and x.humidity < 20 and x.precipitation < 50)
          then FireDanger x.location "high"
          else FireDanger x.location "low"
```

An area where pattern-detect agents are quite relevant is in publish-subscribe systems, where consumers are allowed to subscribe to selective events by specifying filters using a subscription language. The filters define constraints, usually in the form of name-value pairs of properties and basic comparison operators, which identify valid events. Constraints can be logically combined to form what are called complex subscription patterns [16]. CEP systems extend the functionality of traditional publish-subscribe systems by increasing the expressive power of the subscription language to consider complex event patterns that involve the occurrence of multiple, related events [12].

5.1.2 A higher-order library for CEP

Since `EVL` is a higher-order language, we use higher-order functions that allow us to deal with a sequence of events (represented as a list of events). We now provide some of these useful higher-order functions, which are naturally implemented in a higher-order functional language.

- `filter` is a function that filters the events in the sequence according to some filtering expression:

```
filter p list = if list.empty then list
                else if (p list.head) then (cons list.head (filter p list.tail))
                     else filter p list.tail
```

- `transform` is a function that applies a transformation to all of the events in the sequence:

```
transform f list = if list.empty then list
                   else (cons (f list.head) (transform f list.tail))
```

- `aggregater` is a function that produces some output value by aggregating by right association the events of the sequence according to some binary aggregating function:

```
aggregater f z list = if list.empty then z
                      else f list.head (aggregater f z list.tail)
```

- `aggregatel` is very similar to `aggregater` but it aggregates the events by left association:

```
aggregatel f z list = if list.empty then z
                      else aggregatel f (f z list.head) list.tail
```

We use the function names `transform` and `aggregateX`, instead of the usual functional programming names `map` and `foldX`, because of the particular context of CEP.

5.2 Event types

When dealing with event processing applications, many events will have a similar structure and a similar meaning. Consider a temperature sensor: all of the events produced by it have the same kind of information, such as temperature reading, timestamp and maybe location, but with possibly different values. This means that instead of defining the structure of each event individually, we can specify the structure of an entire class of events [15]. This relationship was formally defined in [1] as that between *Generic* and *Specific* events. `EVL` is based on the typed language that was defined in [1], but it extends it by allowing explicit subtyping between record types according to [24].

The definition of a generic event may contain references to other events when there is a semantic relationship between them. These relationships can be separated into four types [15] that we are now going to discuss.

Membership

A generic event ge_1 is said to be a member of another generic event ge_2 if the instances of ge_1 are included in the instances of ge_2. This notion can easily be checked in `EVL` through the explicit subtyping relation between record types.

Generalization

The generalization relation indicates that an event is a generalization of another event. In the type theory of `EVL`, this relation is given by the generalization relation *Cls*.

Specialization

The specialization relation indicates that an event is a specialization of another event. In the type theory of `EVL`, this relation is given by the type instantiation relation.

Retraction

A retraction event relationship is a property of an event referencing a second event. It indicates that the second event is a logical reversal of the event type that references it. For example, an event that starts a fire alert and the event that stops it. This is a notion that is also present in access control systems with obligations. In [1], this is defined by a closing function that describes how events are linked to

subsequent events in history. That is, which are the generic events in time that are closed by a particular generic event provided that some time constraints are satisfied and following a specific strategy. These functions are assumed to be defined for each system and are used to extract intervals from a given history. One of the motivations to develop `EVL` was to provide a simple language to program such functions.

5.3 A comprehensive example

We now give an example that illustrates several features described in this section.

Example 5.12 Consider a sequence of events produced by sensors distributed across some number of locations. The events produced by a particular sensor contains information about the weather conditions at that sensor's location. More specifically, it contains information about the temperature (in degrees Celsius), the humidity level (as a percentage), the wind speed (in km/h) and the amount of precipitation (in mm), as well as information about its location. Now, consider an EPA that infers the fire danger of a particular location based on a given sequence of events produced by an arbitrary number of these sensors. This can be done with varying degrees of accuracy, but this is not the subject of this paper, so let us consider a simple algorithm based on the following three steps:

(i) Filtering the events according to the specified location;

(ii) Aggregating the events according to the latest values of temperature, humidity and wind speed, and by the mean precipitation;

(iii) Producing an event that indicates if there is fire danger in that particular location considering the values obtained in the previous step and comparing them to their threshold levels.

We now provide an implementation of this algorithm in EVL:

```
letEv FireDanger l d = {location = l, fire_danger = d} in
let p x = (x.location == "Porto") in
let f x y = (x.fst + 1, modify(y, precipitation,
                          (x.snd.precipitation + y.precipitation)/x.fst)) in
let check x = if (x.temperature > 29 and x.wind > 32
                  and x.humidity < 20 and x.precipitation < 50)
              then FireDanger x.location "high"
              else FireDanger x.location "low" in
λx.(check (aggregatel f (1, {precipitation = 0}) (filter p x)).snd)
```

6 Related Work

Alternative type systems to deal with records have been presented in the literature using row variables [28], which are variables ranging over finite sets of field types. One of the most flexible systems using row variables [25] allows for powerful operations on records, such as extending a record with a new field or removing an existing field from a record. Record extension is also available in other systems [22,19,9], as well as record concatenation operations [20,26,29], however adding these operations results in complications in the typing process. By following Ohori's approach we ob-

tain a sound and complete efficient type system supporting the basic operations for dealing with records. Nevertheless, regarding the applicability of our language in the context of CEP, the integration of more flexible record operations in our language is an aspect to be further investigated.

When it comes to processing flows of information, there are two main models leading the research done in this area: the data stream processing model [4] (that looks at streams of data coming from different sources to produce new data streams as output); and the complex event processing model [23] (that looks at events happening, which are then filtered and combined to produce new events). In [12], several information processing systems were surveyed, which showed a gap between data processing languages and event detection languages, and the need to define a minimal set of language constructors to combine both features in the same language. We believe that EVL is a good candidate to explore the gap between these two models.

Following the complex event processing model, one of the key features is the ability to derive complex events (composite) from lower-level events and several special purpose Event Query Languages (EQLs) have been proposed for that [14]. Complex event queries over real-time streams of RFID readings have been dealt with in [30] yet again using a query language. The TESLA language [11] supports content-based event filtering as well as been able to establish temporal relations on events, while providing a formal semantics based on temporal logic. The lack of a simple denotational semantics is a common criticism of CEP query languages [31,3,17], with several languages not guaranteeing important language features, such as orthogonality, as well as an overlapping of definitions that make reasoning about these languages that much harder. Recently, a formal framework based on a complex event logic (CEL) was proposed [18], with the purpose of "giving a rigorous and efficient framework to CEP". The authors define well-formed and safe formulas, as syntactic restrictions that characterize semantic properties, and argue that only well-formed formulas should be considered and that users should understand that all variables in a formula must be correctly defined. This notion of well-formed formulas and correctly defined variables is naturally guaranteed in a typed language like `EVL`. Therefore we believe that `EVL` can be used to provide formal semantics to CEP systems.

In the context of access control systems, the *Obligation Specification Language* (OSL) defined in [21], presents a language for events to monitor and reason about data usage requirements. The paper defines the *refinesEv* instance relation between events, which is based on a subset relation on labels, as is the case for the instance relation in [2]. The instance relation in [1] was defined by implicit subtyping on records but more generally using variable instantiation. In this paper we further generalise the notion of instance relation and define it formally using kinded instantiation.

Still in the context of access control, Barker et al [7] have given a representation of events as finite sets of ground 2-place facts (atoms) that describe an event, uniquely identified by $e_i, i \in \mathbb{N}$, and which includes three necessary facts: $happens(e_i, t_j)$, $act(e_i, a_l)$ and $agent(e_i, u_n)$, and n non-necessary facts. This representation is claimed to be more flexible than a term-based representation with a

fixed set of attributes. The language in this paper is flexible enough to encode the event representation in [7]. Furthermore, the typing system allows us to guarantee any necessary facts by means of the typing information.

7 Conclusions and Future Work

In this paper we present `EVL`, a typed higher-order functional language for events, with a typing system based on Ohori's record calculus, as well as a sound and complete type inference algorithm. We explore the expressiveness of our language by showing its application in the context of CEP. This is a starting point to an area of research exploring the well-studied properties of higher-order functional typed languages and their ability to reason with dynamical properties of systems, while applying it to the areas of obligation models for access control and complex event processing (CEP). We believe the language defined in this paper proved to be better equipped to deal with generic events when compared to traditional relational languages used in the CEP area.

With respect to future work, events in our language are represented by records of the form $\{l_1 = v_1, \ldots, l_n = v_n\}$, with appropriate constructors for creating, accessing and modifying records. Additionally one could consider more powerful operations on records, such as extending a record with a new field or removing an existing field from a record, which are not part of `EVL` but could prove useful in both CEP and in the treatment of obligations in the context of access control models.

We would also like to fully exploit `EVL`'s capabilities in the context of the CBACO metamodel [2]. Furthermore, we would like to explore extensions of `EVL` with pattern matching, which is a powerful mechanism for decomposing and processing data. The ability to detect patterns is a key notion in most CEP systems, therefore adding matching primitives to `EVL` would greatly improve its capability with respect to pattern detection.

Acknowledgment

This work was partially funded by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project UIDB/50014/2020 and by Project "Safe Cities", ref. POCI-01-0247-FEDER-041435, financed by Fundo Europeu de Desenvolvimento Regional (FEDER), through COMPETE 2020 and Portugal 2020.

References

[1] Alves, S., S. Broda and M. Fernández, *A typed language for events*, in: M. Falaschi, editor, *Logic-Based Program Synthesis and Transformation - 25th International Symposium, LOPSTR 2015, Siena, Italy, July 13-15, 2015. Revised Selected Papers*, Lecture Notes in Computer Science **9527** (2015), pp. 107–123.
URL https://doi.org/10.1007/978-3-319-27436-2_7

[2] Alves, S., A. Degtyarev and M. Fernández, *Access Control and Obligations in the Category-Based Metamodel: A Rewrite-Based Semantics*, in: *Proceedings of LOPSTR'14*, LNCS **8981**, Springer, 2015 pp. 148–163.

[3] Artikis, A., A. Margara, M. Ugarte, S. Vansummeren and M. Weidlich, *Complex event recognition languages: Tutorial*, in: *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems*, DEBS '17 (2017), p. 7–10.
URL https://doi.org/10.1145/3093742.3095106

[4] Babcock, B., S. Babu, M. Datar, R. Motwani and J. Widom, *Models and issues in data stream systems*, in: *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02 (2002), p. 1–16.
URL https://doi.org/10.1145/543613.543615

[5] Barendregt, H. P., "The lambda calculus - its syntax and semantics," Studies in logic and the foundations of mathematics **103**, North-Holland, 1985.

[6] Barga, R. S., J. Goldstein, M. H. Ali and M. Hong, *Consistent streaming through time: A vision for event stream processing*, in: *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings* (2007), pp. 363–374.
URL http://cidrdb.org/cidr2007/papers/cidr07p42.pdf

[7] Barker, S., M. J. Sergot and D. Wijesekera, *Status-Based Access Control*, ACM Transactions on Information and System Security **12** (2008), pp. 1:1–1:47.

[8] Bertino, E., P. A. Bonatti and E. Ferrari, *TRBAC: A Temporal Role-based Access Control Model*, ACM Transactions on Information and System Security **4** (2001), pp. 191–233.

[9] Cardelli, L. and J. C. Mitchell, *Operations on records*, in: M. Main, A. Melton, M. Mislove and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics* (1990), pp. 22–52.

[10] Chandy, M. K., O. Etzion and R. von Ammon, *The event processing manifesto*, in: K. M. Chandy, O. Etzion and R. von Ammon, editors, *Event Processing*, number 10201 in Dagstuhl Seminar Proceedings (2011).
URL http://drops.dagstuhl.de/opus/volltexte/2011/2985

[11] Cugola, G. and A. Margara, *Tesla: A formally defined event specification language*, in: *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10 (2010), p. 50–61.
URL https://doi.org/10.1145/1827418.1827427

[12] Cugola, G. and A. Margara, *Processing flows of information: From data stream to complex event processing*, ACM Comput. Surv. **44** (2012).
URL https://doi.org/10.1145/2187671.2187677

[13] Damas, L. and R. Milner, *Principal type-schemes for functional programs*, in: R. A. DeMillo, editor, *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982* (1982), pp. 207–212.
URL https://doi.org/10.1145/582153.582176

[14] Eckert, M., F. Bry, S. Brodt, O. Poppe and S. Hausmann, "A CEP Babelfish: Languages for Complex Event Processing and Querying Surveyed," Springer Berlin Heidelberg, Berlin, Heidelberg, 2011 pp. 47–70.
URL https://doi.org/10.1007/978-3-642-19724-6_3

[15] Etzion, O. and P. Niblett, "Event Processing in Action," Manning Publications Co., USA, 2010, 1st edition.

[16] Eugster, P. T., P. A. Felber, R. Guerraoui and A.-M. Kermarrec, *The many faces of publish/subscribe*, ACM Comput. Surv. **35** (2003), p. 114–131.
URL https://doi.org/10.1145/857076.857078

[17] Galton, A. and J. C. Augusto, *Two approaches to event definition*, in: A. Hameurlain, R. Cicchetti and R. Traunmüller, editors, *Database and Expert Systems Applications* (2002), pp. 547–556.

[18] Grez, A., C. Riveros and M. Ugarte, *A Formal Framework for Complex Event Processing*, in: P. Barcelo and M. Calautti, editors, *22nd International Conference on Database Theory (ICDT 2019)*, Leibniz International Proceedings in Informatics (LIPIcs) **127** (2019), pp. 5:1–5:18.
URL http://drops.dagstuhl.de/opus/volltexte/2019/10307

[19] Harper, R. and J. C. Mitchell, *On the type structure of standard ML*, ACM Trans. Program. Lang. Syst. **15** (1993), pp. 211–252.
URL https://doi.org/10.1145/169701.169696

[20] Harper, R. and B. C. Pierce, *A record calculus based on symmetric concatenation*, in: D. S. Wise, editor, *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21-23, 1991* (1991), pp. 131–142.
URL https://doi.org/10.1145/99583.99603

[21] Hilty, M., A. Pretschner, D. A. Basin, C. Schaefer and T. Walter, *A Policy Language for Distributed Usage Control*, in: *Proceedings of ESORICS'07*, 2007, pp. 531–546.

[22] Jategaonkar, L. A. and J. C. Mitchell, *Type inference with extended pattern matching and subtypes*, Fundam. Inf. **19** (1993), p. 127–165.

[23] Luckham, D., "The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems," Addison-Wesley, Boston, MA, 2002.

[24] Ohori, *A Polymorphic Record Calculus and its Compilation*, ACM Transactions on Programming Languages and Systems **17** (1995), pp. 844–895.

[25] Rémy, D., *Efficient representation of extensible records*, in: *Proceedings of the 1992 workshop on ML and its Applications*, San Francisco, USA, 1992, p. 12.

[26] Rémy, D., *Typing record concatenation for free*, in: R. Sethi, editor, *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992* (1992), pp. 166–176.
URL https://doi.org/10.1145/143165.143202

[27] Robinson, J. A., *A machine-oriented logic based on the resolution principle*, Journal of the Association for Computing Machinery (ACM) **12** (1965), pp. 23–41.

[28] Wand, M., *Complete type inference for simple objects*, in: *Proceedings of the Symposium on Logic in Computer Science (LICS'87), Ithaca, New York, USA, June 22-25, 1987* (1987), pp. 37–44.
URL http://www.ccs.neu.edu/home/wand/papers/wand-lics-87.pdf

[29] Wand, M., *Type inference for record concatenation and multiple inheritance*, in: *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989* (1989), pp. 92–97.
URL https://doi.org/10.1109/LICS.1989.39162

[30] Wu, E., Y. Diao and S. Rizvi, *High-performance complex event processing over streams*, in: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06 (2006), p. 407–418.
URL https://doi.org/10.1145/1142473.1142520

[31] Zimmer, D. and R. Unland, *On the semantics of complex events in active database management systems*, in: *Proceedings 15th International Conference on Data Engineering (Cat. No.99CB36337)*, 1999, pp. 392–399.