



Metric First-order Temporal Logic with Complex Data Types

Jeniffer Lima Graf, Srđan Krstić , and Joshua Schneider 

Institute of Information Security, Department of Computer Science, ETH Zürich, Switzerland
{srđan.krstic,joshua.schneider}@inf.ethz.ch

Abstract. Temporal logics are widely used in runtime verification as they enable the creation of declarative and compositional specifications. However, their ability to model complex data is limited. One must resort to complicated encoding schemes to express properties involving basic structures such as lists or trees. To avoid this drawback, we extend metric first-order temporal logic with a minimalistic, yet expressive, functional programming language. The extension features an expressive collection of types including function, record, variant, and inductive types, as well as support for type inference and monitoring.

Our monitor implementation directly parses traces in the JSON format, based on the user’s type specification, which avoids a separate pre-processing step. We compare our approach to existing shallow embeddings of temporal properties in general-purpose host languages and to encodings into simple temporal logics. Specifically, our language benefits from a precise semantics and a good support for monitoring-specific static analysis.

Keywords: Monitoring · Temporal logic · Data types.

1 Introduction

Runtime verification (or monitoring) verifies running systems in their operational environment. Implemented by processes, called *monitors*, it systematically validates a specification by searching for counterexamples in a trace of events recorded during system execution. The specification describes the intended system behavior and, if explicitly input to the monitor, it is written in a specification language. Logical specification languages (e.g., LTL) are widely used due to their declarative and compositional nature.

First-order language extensions, like metric first-order temporal logic (MFOTL) can express dependencies between the data values stored in events. Yet most monitors support only atomic data values making it difficult to write and maintain many practical specifications. Events may contain structured data (e.g., JSON or XML objects), which require either a non-trivial pre-processing step for the trace, or an elaborate encoding scheme for the specification, or even both. Understanding and maintaining both such specifications and pre-processing logic quickly becomes unfeasible, especially as they need to be kept in sync. For example, small changes in the pre-processing logic, like extracting event values in a different order, must be reflected in the first-order specification, e.g., by swapping the appropriate variables in the predicates. Ideally, trace pre-processing should not be done both to avoid the processing overhead and the need to have it in-sync with the specification. If this cannot be achieved, then pre-processing should be domain-independent—it should not rely on the meaning of the trace events.

For example, consider a web server execution trace with successful accesses by clients:

```

@100 {"url":"/login", "client":"123"}
@113 {"url":"/login", "client":"123", "session":{"id":7, "token":"..."}}
@115 {"url":"/secure", "client":"123", "session":{"id":7, "token":"..."}}
@200 {"url":"/secure", "client":"666"}
@800 {"url":"/secure", "client":"123", "session":{"id":7, "token":"..."}}

```

where each line contains a JSON object prefixed with a @*ddd* time-stamp in seconds.

Suppose that every client accessing the /secure URL must have a valid session, not older than 600 seconds, established previously by visiting /login. A way to monitor this specification is to translate JSON objects to tuples containing values of atomic data types:

```

@100 Access("/login", "123", False, -1, "")
@113 Access("/login", "123", True, 7, "...") etc.

```

The Boolean flag in each tuple indicates if there is a session, otherwise *id* and *token* fields have dummy values. The corresponding MFOTL formula formalizing the specification is $\text{Access}(/secure, c, s, id, t) \rightarrow s = \text{True} \wedge \blacklozenge_{[0,600]} \text{Access}(/login, c, \text{True}, id, t)$.

Such a flat structure makes writing specifications tedious as many variables must be used consistently and in a correct position. Moreover, changing the pre-processing (e.g., avoiding the Boolean flag by using separate Access and Session events) necessitates a non-trivial change in the specification.

In this paper we propose an extension of MFOTL, called CMFOTL, which supports complex data types. The extension is accompanied by a corresponding extension of MonPoly [7], an online monitor for MFOTL specifications.

Our presentation of CMFOTL diverges from MFOTL's standard single-sorted first-order logic definition [3,5,6,34]. We start by enumerating multiple primitive types (already supported by MonPoly) and then define a type language that allows for their combination via function, record, variant, and inductive type constructors. To use the newly added types, CMFOTL embeds a minimalistic, yet expressive, functional programming language. We develop a type system and a type inference algorithm for CMFOTL. We then present semantics for well-typed CMFOTL formulas. Finally, we describe a CMFOTL fragment monitorable using finite relations, which is implemented as a syntactic check in MonPoly and supported by the CMFOTL monitoring algorithm. While we rely on standard concepts from programming language theory, our particular design choices (§2) were heavily motivated by efficient monitoring. In particular, we make the following contributions:

- We extend MFOTL with complex data types (§3) to obtain CMFOTL (§4).
- We develop a type system for CMFOTL used by its type inference algorithm (§5).
- We define the semantics for well-typed CMFOTL formulas (§6).
- We bridge the gap between the loosely-typed JSON traces and our strongly-typed language by converting a user-facing signature for JSON events to a first-order signature with complex types (§7.1). Our monitoring algorithm for monitorable CMFOTL formulas directly processes JSON events (§7.2).
- We exemplify CMFOTL's expressiveness with multiple specifications and evaluate the performance of the extended MonPoly monitor (§8).

To the best of our knowledge this is the first logic-based specification language for monitoring that supports complex data types and has precisely defined semantics. Other approaches (§9) either rely on trace pre-processing [7,21] or on domain-specific languages (DSLs) [17,20], which often import unspecified host programming language semantics.

Our implementation and evaluation is publicly available [25].

2 Design Choices

The language extension we present in this paper is inspired by our work on applying runtime verification to large and complex distributed systems. In particular, we used the MonPoly monitor [7] in a previous case study to check properties of the Internet Computer (IC) [4]. The IC’s execution traces were recorded in a detailed JSON format, which required us to engineer a non-trivial mapping from the source data into more abstract events with appropriate parameters. The parameters had to be atomic data (e.g., integers or strings) for compatibility with MonPoly. Writing the specifications representing IC’s properties in MFOTL was an iterative process. In addition to clarifying and fixing imprecise versions of the specification, we also had to account for changes in the format and semantics of the JSON events, which would additionally prompt modifications of the event pre-processing. Synchronizing it with the actual MFOTL specifications was a manual and error-prone process, which had to be tested regularly.

To avoid pre-processing while also supporting realistic event sources, our new language provides *record types* with labeled fields, which correspond to JSON objects. *Named* record types can be defined in the language’s user-facing signature, whereas *unnamed* record types can be defined directly within formulas. As JSON objects may not always conform to a rigid structure (e.g., some fields may be optional as in the JSON trace in §1), we also decided to introduce *variant types* and the *optional type* as a special case.

JSON arrays motivate the need for *list types* and more broadly *inductive types*. We chose to use an iso-recursive [16] over an equi-recursive formalism [27] for our type system due to its simpler type inference algorithm. As an example specification, consider the following execution trace of a webshop application containing information about parcels sent to customers.

```
@100 {"customer":"Alice", "parcel":{"content":[{"content":[]}]}
@200 {"customer":"Bob", "parcel":{"content":[{"content":[{}]},{}]}}
```

One could interpret the objects in the `parcel` fields as arbitrarily nested boxes that make up the parcel. A possible specification for this trace could be that only parcels that consist of up to ten boxes (including all nested boxes) are allowed. In this example, inductive types are necessary to represent both the box objects and the array associated with the `content` field. The user must define the inductive type for the boxes in the user-facing signature. This has the benefit that it allows us to use type-specific *recursors*, which guarantee termination of the monitor’s computations [19]. As a result, only total functions can be expressed and our language is not Turing-complete.

We realize the above extensions with a minimal number of syntactic constructs in the core language. We also provide syntactic sugar for writing specifications in a convenient way. Our new language CMFOTL is a many-sorted strongly-typed logic, unlike the single-sorted logic MFOTL. In practice, the type system prevents additional sources of errors when formulating specifications. We believe that the strong type discipline is not a major burden on the user, as we also provide a type inference algorithm for the new language.

CMFOTL’s syntax and semantics (almost) only extend MFOTL’s terms. Compared to an alternative design based on higher-order logic, this allows for efficient monitoring as it requires only simple bottom-up term evaluation. Furthermore, by retaining the well-known MFOTL formula semantics, CMFOTL’s monitoring algorithm can readily reuse existing optimizations, e.g., for temporal operators.

3 Complex Data Types

MFOTL is typically presented as a single-sorted logic [6, 11], i.e., there is one *domain* that variables range over. In principle, it would suffice to extend the domain and the built-in operations (function symbols) in order to add support for complex data. However, the benefits of static typing are well-known [29]. We therefore define a type language that combines standard features that are widely used in functional programming languages, specifically record (product) types, variant (sum) types, inductive types, and type classes.

We assume an infinite supply of type variables X and labels l . The latter are used for record field and variant constructor names. The syntax of types is given by the following grammar, where A, \dots, A indicates zero or more repetitions of A .

$$\tau ::= \text{Int} \mid \text{Float} \mid \text{Str} \mid (\tau, \dots, \tau) \Rightarrow \tau \mid \{l : \tau, \dots, l : \tau\} \mid \langle l : \tau, \dots, l : \tau \rangle \mid \mu X. \tau \mid X$$

The symbols Int, Float, and Str represent primitive types for integers, floating-point numbers, and strings, respectively. The function type $(\tau_1, \dots, \tau_n) \Rightarrow \rho$ describes total functions that map tuples over types τ_1, \dots, τ_n into values of type ρ .

Record types are denoted by $\{l_1 : \tau_1, l_2 : \tau_2, \dots, l_n : \tau_n\}$, where l_1, l_2, \dots, l_n is a possibly empty list of field labels, and $\tau_1, \tau_2, \dots, \tau_n$ are the corresponding types. The order of labels is irrelevant: $\{l : \text{Int}, m : \text{Str}\}$ and $\{m : \text{Str}, l : \text{Int}\}$ denote the same type. Intuitively, the values of a record type are tuples that assign a value to each label. They can be used to describe compound objects. The empty record type $\{\}$ serves as the unit type, which has a single value. It is sometimes convenient to use records with unnamed fields that are instead distinguished by their order of appearance. We write (τ_1, \dots, τ_n) for such a *tuple type*, which can be de-sugared into an equivalent record type with canonical labels.

Variant types $\langle l_1 : \tau_1, l_2 : \tau_2, \dots, l_n : \tau_n \rangle$ are dual to records. They represent the choice of one of multiple alternatives, whose order is again irrelevant. Their values can be thought of as pairs (l_i, x) , where l_i is one of the *constructors* and the value x has type τ_i . The empty variant $\langle \rangle$ represents the empty type, which does not contain any values. A simple example combining variants and the unit type is the encoding of Booleans by the $\langle \text{true} : \{\}, \text{false} : \{\} \rangle$ type. This type plays a special role and hence we give it the name Bool.

The expression $\mu X. \tau$ denotes an inductive type. The type variable X must occur strictly positively in τ , i.e., X must not occur as a free type variable in an argument type τ_i of any function type $(\tau_1, \dots, \tau_n) \Rightarrow \rho$ within τ [15]. Intuitively, an inductive type $\mu X. \tau$ is the least fixpoint of the type equation $X = \tau$. The variable X is bound by $\mu X. \tau$ in τ and is thus subject to α -conversion. As an example, $\mu X. \langle \text{Nil} : \{\}, \text{Cons} : \{\text{hd} : \text{Int}, \text{tl} : X\} \rangle$ represents finite lists of integers. A type without free type variables is *ground*.

4 Specification Language

We now present CMFOTL, our specification language that supports complex data types from §3. It is based on MFOTL, which has two main syntactic categories: terms and formulas. Terms evaluate to values from the domain, whereas formulas assign a truth value to every time-point in a given trace. We primarily extend the term syntax with new constructs. Specifically, we add a lambda calculus and operations to work with the new data types. We remove equality and ordering relations from the formula syntax because

they can now be expressed as terms. Such terms can be used within a new, more general type of atomic formula, *assertions*, which assert the truth of an arbitrary Boolean-valued term. To keep the presentation self-contained, we also recap the unmodified parts of MFOTL. New elements are indicated with a gray background.

The syntax of terms is given by the following grammar, where c, x, l range over constants, variables, and labels, respectively.

$$t ::= c \mid x \mid t : \tau \mid \lambda(x, \dots, x).t \mid t(t, \dots, t) \mid \{l : t, \dots, l : t\} \mid t.l \mid \text{mk } l(t) \\ \mid \text{case}(t; l(x) \rightarrow t, \dots, l(x) \rightarrow t) \mid \text{rec}_{X, \tau}(t) \mid \text{unrec}_{X, \tau}(t) \mid \text{fold}_{X, \tau}(t; x \rightarrow t)$$

We provide an intuitive explanation here; the formal semantics is postponed to §6 as it depends on the type system. Constants represent operations that are built into the monitor. We fix the set of available constants in §6. The term $t : \tau$ denotes a type ascription, which enforces and documents that t has the type τ . Lambda abstractions $\lambda(x_1, \dots, x_n).t$ and function applications $t_f(t_1, \dots, t_n)$ support multiple arguments.

The term $\{l_1 : t_1, \dots, l_n : t_n\}$ constructs a value of a record type, and $t.l_i$ is its projection to label l_i . Dually, the term $\text{mk } l_i(t)$ constructs a value of a variant type, and $\text{case}(t; l_1(x_1) \rightarrow t_1, \dots, l_n(x_n) \rightarrow t_n)$ performs a case distinction on t . If a constructor l_i has argument type $\{\}$, we typically omit the term in $\text{mk } l_i$ and the variable in a case branch $l_i \rightarrow t_i$. Recursive types $\mu X. \tau$ are constructed and deconstructed via the terms $\text{rec}_{X, \tau}(t)$ and $\text{unrec}_{X, \tau}(t)$. Recursive computations must be expressed as a fold (i.e., a *catamorphism*) $\text{fold}_{X, \tau}(t_1; x \rightarrow t_2)$, where t_1 is a value of type $\mu X. \tau$ to fold and t_2 performs one step of the computation using the partial result bound to x . The last three constructs are annotated by the inductive type to facilitate type inference.

Our modified formula syntax is mostly the same as that of MFOTL. For space reasons we exclude aggregation operators [5] and (non-recursive) let bindings [34], which our implementation also supports. The complete version of CMFOTL is shown in the extended version of this paper [24]. In the grammar below, P ranges over predicate symbols, and I ranges over non-empty and possibly unbounded intervals over the natural numbers.

$$\varphi ::= \Downarrow t \mid P(t, \dots, t) \mid \neg \varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \exists x. \varphi \mid \bullet_I \varphi \mid \circ_I \varphi \mid \varphi \text{S}_I \varphi \mid \varphi \text{U}_I \varphi.$$

The noteworthy change over previous versions of MFOTL is the introduction of assertions $\Downarrow t$, which replace and generalize equality ($t_1 = t_2$) and inequality ($t_1 < t_2$, $t_1 \leq t_2$) formulas. Atomic predicates $P(t_1, \dots, t_n)$, Boolean operators, and existential quantification are as in first-order logic. The past and future temporal operators \bullet_I , \circ_I , S_I , and U_I are as in discrete-time MTL [1]. The interval subscripts impose bounds on the elapsed time. A missing interval defaults to the maximal interval $[0, \infty]$.

Parentheses can be omitted based on operator precedence. Our convention is that the scope of binders (lambdas, branches of case, and quantifiers) extends maximally to the right. Negation has the highest precedence, followed by conjunction and disjunction, in this order. We always parenthesize temporal operators for clarity. Additional operators are defined as syntactic sugar, for example $\blacklozenge_I \varphi \equiv (\Downarrow \text{true } \text{S}_I \varphi)$ and $\blacklozenge_I \varphi \equiv (\Downarrow \text{true } \text{U}_I \varphi)$.

Free variables $\text{fv}(t)$ and $\text{fv}(\varphi)$ are defined as usual. Arrows \rightarrow and quantifiers indicate variable bindings, e.g., $l(x) \rightarrow t$ and $\exists x. \varphi$ bind x in t and φ , respectively.

5 Type System

Not all terms and formulas are meaningful. For instance, it is unclear how to interpret a projection applied to a lambda term. Therefore, we introduce a type system for CMFOTL. It is based on an inductively defined *typing judgement relation*. Only terms and formulas that are well-typed, i.e., that can be assigned a type by this relation, have a semantics. Our monitor implementation first checks well-typedness of a CMFOTL formula before proceeding to monitor it. Specifically, it performs type inference, which finds the most general type (up to the names of type variables) for every (sub-)term of the formula. Like any static type system, ours serves as an additional layer of protection against runtime errors. In monitoring, such errors may be due to a malformed specification. To define the typing judgement relation we need to define type classes and type class constraints.

A *type class* [33] is a set of types that share a specific common property, e.g., the property that addition is defined. Types can be members of multiple type classes, and type classes are partially ordered by their subset relationship. Type classes allow for overloading of operations in terms. For example, it should be possible to use the addition operator $+$ both with integers and floating-point numbers.

We use the following type classes. The class Eq consists of all types that are built without function types. We restrict the equality operator to Eq type because function equality is undecidable in general. The class $\text{Ord} \supset \{\text{Int}, \text{Float}, \text{Str}\}$ consists of all types on which a total ordering \leq is defined. In addition to the three primitive types, our implementation considers record and variant types whose fields or constructor arguments are recursively members of Ord to be instances of Ord (using a lexicographic ordering). The class $\text{Num} = \{\text{Int}, \text{Float}\}$ consists of all numeric types that support the four basic arithmetic operations and modulo. The classes $\text{Proj}(l : \tau)$ consists of all record types that contain a field $l : \tau$, and the classes $\text{Ctor}(l : \tau)$ consist similarly of all variant types that have a constructor $l : \tau$. With these classes our type inference (§5.2) can incorporate Ogori’s approach [28] for inferring polymorphic record and variant types. The classes $\text{Proj}(l : \tau)$ and $\text{Ctor}(l : \tau)$ are *parametric* in τ [10] and we do not require unique field or constructor names across types. Note that the field and constructor argument types τ are uniquely determined given an instance of the respective type class. This ensures that type inference yields the expected most general type without additional type annotations.

A *type class constraint* C is a (finite, possibly empty) set of type classes. It is a symbolic representation of the intersection of those classes. We say that the types in the intersection satisfy the constraint. The empty constraint is satisfied by all types. We attach type class constraints to type variables to restrict the types that they can be instantiated with. For bound type variables, we denote the constraint as part of the binder (e.g., $\mu X : \{\text{Num}\}. \tau$).

5.1 Typing Rules

The typing judgement relation $\Gamma \vdash t :: \tau$ for terms is a ternary relation between variable contexts Γ , terms t , and types τ . A variable context is a finite mapping from variables to types. The typing judgement is defined as the least relation closed under the rules shown in Fig. 1a. Each rule consist of a possibly empty sequence of hypotheses above the line and a conclusion below the line. There is an implicit condition for all rules: any free type variables must be free in the conclusion’s context or type, i.e., hidden polymorphism is

$$\begin{array}{c}
\frac{\tau \text{ is an instance of } c\text{'s type scheme}}{\Gamma \vdash c :: \tau} \text{CST} \quad \frac{}{\Gamma, x : \tau \vdash x :: \tau} \text{VAR} \quad \frac{\Gamma \vdash t :: \tau}{\Gamma \vdash (t:\tau) :: \tau} \text{ASC} \\
\\
\frac{\Gamma, x_1 : \tau_1, \dots, x_n : \tau_n \vdash t :: \rho}{\Gamma \vdash \lambda(x_1, \dots, x_n).t :: (\tau_1, \dots, \tau_n) \Rightarrow \rho} \text{LAM} \\
\\
\frac{\Gamma \vdash t_f :: (\tau_1, \dots, \tau_n) \Rightarrow \rho \quad \Gamma \vdash t_1 :: \tau_1 \quad \dots \quad \Gamma \vdash t_n :: \tau_n}{\Gamma \vdash t_f(t_1, \dots, t_n) :: \rho} \text{APP} \\
\\
\frac{\Gamma \vdash t_1 :: \tau_1 \quad \dots \quad \Gamma \vdash t_n :: \tau_n}{\Gamma \vdash \{l_1:t_1, \dots, l_n:t_n\} :: \{l_1:\tau_1, \dots, l_n:\tau_n\}} \text{PROD} \quad \frac{\Gamma \vdash t :: \pi \quad \pi \in \text{Proj}(l:\tau)}{\Gamma \vdash t.l :: \tau} \text{PROJ} \\
\\
\frac{\Gamma \vdash t :: \tau \quad \sigma \in \text{Ctor}(l:\tau)}{\Gamma \vdash l(t) :: \sigma} \text{CTOR} \\
\\
\frac{\Gamma \vdash t :: \langle l_1 : \tau_1, \dots, l_n : \tau_n \rangle \quad \Gamma, x_1 : \tau_1 \vdash t_1 :: \rho \quad \dots \quad \Gamma, x_n : \tau_n \vdash t_n :: \rho}{\Gamma \vdash \text{case}(t; l_1(x_1) \rightarrow t_1, \dots, l_n(x_n) \rightarrow t_n) :: \rho} \text{CASE} \\
\\
\frac{\Gamma \vdash t :: \tau[\mu X. \tau/X]}{\Gamma \vdash \text{rec}_{X,\tau}(t) :: \mu X. \tau} \text{REC} \quad \frac{\Gamma \vdash t :: \mu X. \tau}{\Gamma \vdash \text{unrec}_{X,\tau}(t) :: \tau[\mu X. \tau/X]} \text{UNREC} \\
\\
\frac{\Gamma \vdash t_1 :: \mu X. \tau \quad \Gamma, x : \tau[\rho/X] \vdash t_2 :: \rho}{\Gamma \vdash \text{fold}_{X,\tau}(t_1; x \rightarrow t_2) :: \rho} \text{FOLD} \\
\\
\text{(a) Typing rules for terms}
\end{array}$$

$$\begin{array}{c}
\frac{\Gamma \vdash t_1 :: \tau_1 \quad \dots \quad \Gamma \vdash t_n :: \tau_n}{\Delta, R : (\tau_1, \dots, \tau_n); \Gamma \vdash R(t_1, \dots, t_n)} \text{PRED} \quad \frac{\Gamma \vdash t :: \text{Bool}}{\Delta; \Gamma \vdash \downarrow t} \text{ASSERT} \\
\\
\frac{\Delta; \Gamma \vdash \varphi}{\Delta; \Gamma \vdash \star \varphi} \text{UNFORM} \quad \star \in \{\neg, \bullet_I, \circ_I\} \\
\\
\frac{\Delta; \Gamma \vdash \varphi \quad \Delta; \Gamma \vdash \psi}{\Delta; \Gamma \vdash \varphi \star \psi} \text{BINFORM} \quad \star \in \{\wedge, \vee, S_I, U_I\} \quad \frac{\Delta; \Gamma, x : \tau \vdash \varphi}{\Delta; \Gamma \vdash \exists x. \varphi} \text{EXISTS} \\
\\
\text{(b) Typing rules for formulas}
\end{array}$$

Fig. 1: Typing rules for CMFOTL

5.2 Type Inference

We implemented a type inference algorithm based on the Damas–Hindley–Milner framework [13]. Our handling of type classes is similar to the approach described by Chen, Hudak, and Odersky [10]. Neither work considers a logical layer like CMFOTL’s formula language, but the extension to formulas is straightforward.

The algorithm proceeds bottom-up in a syntax-directed fashion while propagating the current knowledge about the variable context and first-order signature. The signature is initially obtained from the user (see §7.1). For compound expressions, the sub-expressions (i.e., terms and/or formulas) are visited first to obtain their most general types. As is typical for type systems, there is a unique rule that applies to every syntax construct. We determine the most general instance of that rule which agrees with the sub-expressions’ types through unification. A type error is reported whenever unification fails. The unification procedure must take the type class constraints of type variables into account. Additional care is required for variable binders such as lambda functions as bound variables can shadow variables of the same name in the surrounding context.

Let us revisit the list sum example from the previous subsection. Suppose that the constant ‘0’ is the first sub-expression to be visited. It is not polymorphic and hence the type Int is returned immediately. Next, we consider $\text{plus}(1, z.\text{tl})$. This term is in the scope of the variables x , y , and z . Whenever the algorithm enters a scope of a binder, it adds the variable with a fresh, unrestricted type variable to the context. Let us call these τ_x, τ_y, τ_z for variables x , y , and z . The type scheme of plus is declared as $(\alpha, \alpha) \Rightarrow \alpha$ where $\alpha : \{\text{Num}\}$. Whenever such a polymorphic constant is encountered, the algorithm replaces its type variables with fresh ones, say, $\tau_+ : \{\text{Num}\}$ in this case. As the type of $1 :: \text{Int}$ must agree with the first argument of plus , unification results in the substitution $\tau_+ \mapsto \text{Int}$. This substitution is possible because Int satisfies the constraint $\{\text{Num}\}$. Inferring the type of $z.\text{tl}$ results in a refinement of τ_z ’s constraint in the variable context: it is now $\{\text{Proj}(\text{tl} : \text{Int})\}$.

Unification with the CASE rule triggers the substitution $\tau_y \mapsto \langle \text{Nil} : \nu, \text{Cons} : \tau_z \rangle$. At this point, there is no information about Nil ’s argument type. The case term itself has type Int . The most interesting step happens for the fold : To obtain a proper instance of its type rule, we unify y ’s type with

$$\langle \text{Nil} : \{\}, \text{Cons} : \{\text{hd} : \text{Int}, \text{tl} : X\} \rangle [\text{Int}/X] \equiv \langle \text{Nil} : \{\}, \text{Cons} : \{\text{hd} : \text{Int}, \text{tl} : \text{Int}\} \rangle.$$

This results in the substitutions $\nu \mapsto \{\}$ and $\tau_z \mapsto \{\text{hd} : \text{Int}, \text{tl} : \text{Int}\}$, where the latter satisfies constraint $\{\text{Proj}(\text{tl} : \text{Int})\}$ from above. Since fold ’s most generic type is that of its last sub-term (after applying applicable substitutions), we obtain Int as the overall result.

6 Semantics

We define CMFOTL’s semantics with respect to infinite *temporal structures* (i.e., traces), which associate a first-order structure with every time-point. The introduction of complex data types requires a specific domain that provides the values of all possible types. Moreover, it makes sense to provide a rigid interpretation of CMFOTL’s constants so that they can be relied upon in specifications and implemented directly in the monitor. We call this fixed part of the temporal structures including the domain the *base model*. We construct

Table 1: The constants of the base model (abridged)

Constant	Type scheme	Constant	Type scheme
true, false	Bool	neg	$(\tau) \Rightarrow \tau, \tau \in \text{Num}$
eq	$(\tau, \tau) \Rightarrow \text{Bool}, \tau \in \text{Eq}$	plus	$(\tau, \tau) \Rightarrow \tau, \tau \in \text{Num}$
less	$(\tau, \tau) \Rightarrow \text{Bool}, \tau \in \text{Ord}$	minus	$(\tau, \tau) \Rightarrow \tau, \tau \in \text{Num}$
leq	$(\tau, \tau) \Rightarrow \text{Bool}, \tau \in \text{Ord}$	times	$(\tau, \tau) \Rightarrow \tau, \tau \in \text{Num}$
not	$(\text{Bool}) \Rightarrow \text{Bool}$	div	$(\tau, \tau) \Rightarrow \tau, \tau \in \text{Num}$
and	$(\text{Bool}, \text{Bool}) \Rightarrow \text{Bool}$	mod	$(\text{Int}, \text{Int}) \Rightarrow \text{Int}$
or	$(\text{Bool}, \text{Bool}) \Rightarrow \text{Bool}$		

the base model from a suitable subset of terms that intuitively represent *values* which cannot be simplified further by computation. As a consequence, all function values in the base model are definable and equality over functions is intensional, i.e., it depends on the functions' definitions. This is sufficient in practice because in our monitor implementation, variables of function type are instantiated only with those functions that occur in the formula or with constants of the base model; functions in the trace are not supported. Moreover, functions cannot be compared for equality as they are not part of the Eq class.

All sub-terms and variables (including bound ones) have known types after the successful completion of type inference. In this section, we use the type ascription syntax $t : \tau$ both for sub-terms and variable binders to access those types. Moreover, we assume that all types are ground to simplify the formal semantics. This is without loss of generality because all primitive values contained in our base model are monomorphic.

The base model's *domain* \mathcal{D}_τ for type τ consists of a subset of terms (i.e., *values*) with type τ . Specifically, values are ground terms built inductively from constants, record, variant, and rec constructors, as well as lambda abstractions with an arbitrary term (i.e., not necessarily a value) for the body. Below, we use \mathcal{D} when the type is clear from the context.

Table 1 shows a subset of the constants included in the base model. For the polymorphic constants, the base model specifically contains all ground instances separately (e.g., eq_{Bool} , eq_{Int} , and so forth). In addition, any integer, floating-point, or string literal can be used as a constant of the corresponding types. We also omit some string operations and conversions for lack of space. Note that the boolean operators not, and, or do not supersede the corresponding operators in CMFOTL formulas: a term is always evaluated under a concrete assignment to all of its free variables, whereas formulas can *generate* sets of assignments. We assume that for every ground instance of a function-valued constant $c :: (\tau_1, \dots, \tau_n) \Rightarrow \rho$, there is a mapping \boxed{c} from $\mathcal{D}_{\tau_1} \times \dots \times \mathcal{D}_{\tau_n}$ to \mathcal{D}_ρ which interprets the constant.

Next, we define a small-step operational semantics for well-typed terms, using call-by-value evaluation as implemented in our monitor. The single-step reduction relation \rightsquigarrow is the least relation closed under the rules shown in Fig. 3. Similarly as for types, we write $t[t'/x]$ for the capture-avoiding substitution of t' for variable x in the term t . Variables that occur only on the right-hand side of \rightsquigarrow are assumed to be fresh. The new terms of the form $\text{mapf}_{X, \tau; \tau'}(t; x \rightarrow t')$ are only used for the evaluation of folds. Intuitively, they apply the operation $\text{fold}_{X, \tau'}(u; x \rightarrow t')$ to those subterms of the value t (which has type τ) that correspond to an occurrence of the type variable X . For example, we have for

$$\begin{array}{c}
\frac{\forall i. t_i \in \mathcal{D}}{c(t_1, \dots) \rightsquigarrow \boxed{c}(t_1, \dots)} \quad \frac{t \rightsquigarrow t'}{t : \tau \rightsquigarrow t' : \tau} \quad \frac{t \in \mathcal{D}}{t : \tau \rightsquigarrow t} \quad \frac{t \rightsquigarrow t'}{t(t_1, \dots) \rightsquigarrow t'(t_1, \dots)} \\
\frac{t_i \rightsquigarrow t'_i}{t(t_1, \dots) \rightsquigarrow t(t_1, \dots, t'_i, \dots)} \quad \frac{\forall i. t_i \in \mathcal{D}}{(\lambda(x_1, \dots). t)(t_1, \dots) \rightsquigarrow t[t_1/x_1, \dots]} \\
\frac{t_i \rightsquigarrow t'_i}{\{l_1 : t_1, \dots\} \rightsquigarrow \{l_1, \dots, l_i : t'_i, \dots\}} \quad \frac{t \rightsquigarrow t'}{t.l \rightsquigarrow t'.l} \quad \frac{\forall i. t_i \in \mathcal{D}}{\{l_1 : t_1, \dots\}.l_i \rightsquigarrow t_i} \\
\frac{t \rightsquigarrow t'}{\text{mk } l(t) \rightsquigarrow \text{mk } l(t')} \quad \frac{t \rightsquigarrow t'}{\text{case}(t; l_1(x_1) \rightarrow t_1, \dots) \rightsquigarrow \text{case}(t'; l_1(x_1) \rightarrow t_1, \dots)} \\
\frac{t \in \mathcal{D}}{\text{case}(\text{mk } l_i(t); l_1(x_1) \rightarrow t_1, \dots) \rightsquigarrow t_i[t/x_i]} \quad \frac{t \rightsquigarrow t'}{\text{rec}_{X,\tau}(t) \rightsquigarrow \text{rec}_{X,\tau}(t')} \\
\frac{t \rightsquigarrow t'}{\text{unrec}_{X,\tau}(t) \rightsquigarrow \text{unrec}_{X,\tau}(t')} \quad \frac{t \in \mathcal{D}}{\text{unrec}_{X,\tau}(\text{rec}_{X,\tau}(t)) \rightsquigarrow t} \\
\frac{t \rightsquigarrow t'}{\text{fold}_{X,\tau}(t; x \rightarrow t'') \rightsquigarrow \text{fold}_{X,\tau}(t'; x \rightarrow t'')} \quad \frac{t \rightsquigarrow t'}{\text{mapf}_{X,\tau;\tau'}(t; x \rightarrow t'') \rightsquigarrow \text{mapf}_{X,\tau;\tau'}(t'; x \rightarrow t'')} \\
\frac{t \in \mathcal{D}}{\text{fold}_{X,\tau}(\text{rec}_{X,\tau}(t); x \rightarrow t') \rightsquigarrow (\lambda(x). t')(\text{mapf}_{X,\tau;\tau}(t; x \rightarrow t'))} \quad \frac{t \rightsquigarrow t'}{\text{mapf}_{X,\tau';\tau}(c; x \rightarrow t') \rightsquigarrow c} \\
\frac{t \rightsquigarrow t'}{\text{mapf}_{X,X;\tau}(t; x \rightarrow t') \rightsquigarrow \text{fold}_{X,\tau}(t; x \rightarrow t')} \\
\frac{t \rightsquigarrow t'}{\text{mapf}_{X,(\tau_1, \dots) \Rightarrow \rho; \tau}(\lambda(x_1, \dots). t; x \rightarrow t') \rightsquigarrow \lambda(x_1, \dots). \text{mapf}_{X,\rho;\tau}(t; x \rightarrow t')} \\
\frac{t \rightsquigarrow t'}{\text{mapf}_{X, \{l_1 : \tau_1, \dots\}; \tau}(\{l_1 : t_1, \dots\}; x \rightarrow t') \rightsquigarrow \{l_1 : \text{mapf}_{X,\tau_1;\tau}(t_1; x \rightarrow t'), \dots\}} \\
\frac{t \rightsquigarrow t'}{\text{mapf}_{X, \langle l_1 : \tau_1, \dots \rangle; \tau}(\text{mk } l_i(t); x \rightarrow t') \rightsquigarrow \text{mk } l_i(\text{mapf}_{X,\tau_i;\tau}(t; x \rightarrow t'))} \\
\frac{t \rightsquigarrow t'}{\vdash \text{mapf}_{X,\mu Y, \tau'; \tau}(t; x \rightarrow t') :: (\mu Y. \tau')[\rho/X] \quad X \neq Y} \\
\frac{t \rightsquigarrow t'}{\text{mapf}_{X,\mu Y, \tau'; \tau}(\text{rec}_{Y, \tau'}[\mu X, \tau/X](t); x \rightarrow t') \rightsquigarrow \text{rec}_{Y, \tau'}[\rho/X](\text{mapf}_{X, \tau'[\mu Y, \tau'/Y]; \tau}(t; x \rightarrow t'))}
\end{array}$$

Fig. 3: Small-step semantics for term evaluation

$\tau \equiv \langle \text{None} : \{\}, \text{Some} : X \rangle$

$$\begin{array}{l}
\text{fold}_{X,\tau}(\text{mk Some}(\text{rec}_{\mu X,\tau}(\text{mk None}(\{\}))); x \rightarrow 0) \\
\rightsquigarrow (\lambda(x). 0)(\text{mapf}_{X,\tau;\tau}(\text{mk Some}(\text{rec}_{\mu X,\tau}(\text{mk None}(\{\}))); x \rightarrow 0)) \\
\rightsquigarrow (\lambda(x). 0)(\text{mk Some}(\text{mapf}_{X,X;\tau}(\text{rec}_{\mu X,\tau}(\text{mk None}(\{\}))); x \rightarrow 0)) \\
\rightsquigarrow (\lambda(x). 0)(\text{mk Some}(\text{fold}_{X,X}(\text{rec}_{\mu X,\tau}(\text{mk None}(\{\}))); x \rightarrow 0)) \\
\rightsquigarrow (\lambda(x). 0)(\text{mk Some}((\lambda(x). 0)(\text{mapf}_{X,\tau;\tau}(\text{mk None}(\{\})); x \rightarrow 0))) \\
\rightsquigarrow (\lambda(x). 0)(\text{mk Some}((\lambda(x). 0)(\text{mk None}(\text{mapf}_{X,\tau;\tau}(\{\}; x \rightarrow 0)))))) \\
\rightsquigarrow (\lambda(x). 0)(\text{mk Some}((\lambda(x). 0)(\text{mk None}(\{\})))) \\
\rightsquigarrow (\lambda(x). 0)(\text{mk Some}(0)) \rightsquigarrow 0.
\end{array}$$

The multi-step reduction relation \rightsquigarrow^* is the reflexive and transitive closure of \rightsquigarrow .

Our type system and term semantics have two important properties: type soundness [26] and termination. These properties guarantee that our monitor does not encounter run-time errors due to the policy using undefined operations (the standard example being

$$\begin{aligned}
v, i \models \Downarrow t &\text{ iff } \llbracket t \rrbracket(v) = \text{true} & v, i \models R(t_1, \dots, t_n) &\text{ iff } (\llbracket t_1 \rrbracket(v), \dots, \llbracket t_n \rrbracket(v)) \in D_i \\
v, i \models \neg \varphi &\text{ iff } v, i \not\models \varphi & v, i \models \exists x : \tau. \varphi &\text{ iff } v[z/x], i \models \varphi \text{ for some } z \in \mathcal{D}_\tau \\
v, i \models \varphi \wedge \psi &\text{ iff } v, i \models \varphi \text{ and } v, i \models \psi & v, i \models \varphi \vee \psi &\text{ iff } v, i \models \varphi \text{ or } v, i \models \psi \\
v, i \models \bullet_I \varphi &\text{ iff } i > 0, T_i - T_{i-1} \in I, \text{ and } v, i - 1 \models \varphi \\
v, i \models \circ_I \varphi &\text{ iff } T_{i+1} - T_i \in I \text{ and } v, i + 1 \models \varphi \\
v, i \models \varphi S_I \psi &\text{ iff } v, j \models \psi \text{ for some } j \leq i, T_i - T_j \in I, \text{ and } v, k \models \varphi \text{ for all } k \text{ with } j < k \leq i \\
v, i \models \varphi U_I \psi &\text{ iff } v, j \models \psi \text{ for some } j \geq i, T_j - T_i \in I, \text{ and } v, k \models \varphi \text{ for all } k \text{ with } i \leq k < j
\end{aligned}$$

Fig. 4: CMFOTL’s formula semantics

trying to add numbers and strings) and that it always terminates on finite traces. CMFOTL’s term language is an extension of the simply typed lambda calculus and hence the standard technique of logical relations [29, 32] can be used to establish strong normalization into values, which implies termination and, together with type preservation, soundness. However, the fold operator and the recursion through functions in inductive types require some care. We give proofs of the following theorems in the extended version [24].

Theorem 1. \rightsquigarrow^* preserves ground types, i.e., $\vdash t :: \tau$ and $t \rightsquigarrow^* t'$ imply $\vdash t' :: \tau$.

Theorem 2. \rightsquigarrow^* is strongly normalizing: For every ground term t such that $\vdash t :: \tau$, there exists a unique normal form $\llbracket t \rrbracket \in \mathcal{D}_\tau$ such that $t \rightsquigarrow^* \llbracket t \rrbracket$ and there is no u with $\llbracket t \rrbracket \rightsquigarrow u$.

A valuation v for a term t is a finite mapping from the term’s free variables $x_i : \tau_i$ to values in the corresponding domains \mathcal{D}_{τ_i} . Strong normalization allows us to lift the term semantics to an evaluation function $\llbracket t \rrbracket(v) = \llbracket t[v(x_1)/x_1, \dots, v(x_n)/x_n] \rrbracket$ returning values. Observe that for ground terms, evaluation results directly in the normal form, which justifies this mild abuse of notation.

The relation $v, i \models \varphi$ (Fig. 4) defines the satisfaction of the formula φ for a given temporal structure, valuation v , and time-point $i \in \mathbf{N}$. A temporal structure is an infinite sequence $(T_i, D_i)_{i \in \mathbf{N}}$ of finite first-order structures D_i over the signature Δ with associated time-stamps T_i . This means that each D_i assigns to every relation symbol $R : (\tau_1, \dots, \tau_n) \in \Delta$ a finite subset of $\mathcal{D}_{\tau_1} \times \dots \times \mathcal{D}_{\tau_n}$. Time-stamps are natural numbers $T_i \in \mathbf{N}$. They need not be unique, but we require that time-stamps are monotone ($\forall i. T_i \leq T_{i+1}$) and unbounded ($\forall T. \exists i. T < T_i$). Overall, the semantics is the same as MFOTL’s, except for the addition of assertions.

7 Implementation

Our monitor for CMFOTL is an extension of the MonPoly tool [7], which is written in OCaml. In particular, we modified MonPoly’s signature and formula parser, type inference code, and internal representation of domain values. Instead of MonPoly’s first-order signature, our extension takes as input a *user-facing signature*. It allows the specification of nested and recursive structures, which are used to parse a stream of time-stamped JSON events. The events are subsequently mapped to instances of CMFOTL types based on our signature translation.

7.1 Signature Translation

We introduce the user-facing signature format and develop a translation to a first-order signature (§5.1). The user-facing signature serves two purposes: it defines the CMFOTL types used for type inference and it guides the parsing of JSON events. The syntax is geared towards usability. It consists of JSON values representing types that may refer to each other by name. Therefore, the translation to first-order signature is non-trivial in the presence of circular name references.

The user-facing signature is a sequence of record type definitions. Each definition consists of a type name followed by a *symbolic* record type. The definition may be prefixed by the keyword `event`, which marks the type as an *event type*. Only event types may occur as top-level objects in the JSON event stream. The field types γ_i of a symbolic record type must conform to the grammar

$$\begin{aligned} \gamma &::= \delta \mid \delta? \mid [\delta] \mid [\delta?] \\ \delta &::= \textit{name} \mid \{l : \gamma, \dots, l : \gamma\} \mid \text{Null} \mid \text{Int} \mid \text{Float} \mid \text{String} \mid \text{Bool} \end{aligned}$$

where *name* refers to any type defined in the user-facing signature, including the current one. A question mark indicates an optional field and square brackets are used for arrays.

Each named type defined in the user-facing signature as *name* $\{l_1 : \gamma_1, \dots, l_n : \gamma_n\}$ is translated to a CMFOTL type $\tau_{\textit{name}} = \llbracket \{l_1 : \gamma_1, \dots, l_n : \gamma_n\} \rrbracket$ according to the rules

$$\begin{aligned} \llbracket \delta? \rrbracket &= \langle \text{None} : \{\}, \text{Some} : \llbracket \delta \rrbracket \rangle & \llbracket [\delta] \rrbracket &= \mu L. \langle \text{Nil} : \{\}, \text{Cons} : \{\text{hd} : \llbracket \delta \rrbracket, \text{tl} : L\} \rangle \\ \llbracket \textit{name} \rrbracket &= \tau_{\textit{name}} & \llbracket \{l_1 : \gamma_1, \dots, l_n : \gamma_n\} \rrbracket &= \{l_1 : \llbracket \gamma_1 \rrbracket, \dots, l_n : \llbracket \gamma_n \rrbracket\} & \llbracket \text{Null} \rrbracket &= \{\} \\ \llbracket \text{Int} \rrbracket &= \text{Int} & \llbracket \text{Float} \rrbracket &= \text{Float} & \llbracket \text{String} \rrbracket &= \text{Str} & \llbracket \text{Bool} \rrbracket &= \text{Bool} \end{aligned}$$

However, this translation fails if there is a circular dependency (direct or indirect) between named types, as this would result in an infinite type expression. A named type τ_1 depends directly on a named type τ_2 iff the latter occurs in τ_1 's definition. We use the following algorithm to translate circular dependencies into inductive types.

1. All direct type dependencies are represented as a directed graph. We compute the graphs's strongly connected components. The edges between the strongly connected components form a tree which is processed from the leaves to the root.
2. Every component consisting of a single named type that does not refer to itself can be translated immediately as above.
3. If a component contains multiple nodes or a single component has an edge pointing to itself, it indicates the presence of one or more inductive types. We choose one node in the component based on a heuristic. The choice does not matter for correctness, but it influences the syntactic structure of the obtained types. If only one node is referenced from other components, it is selected. Otherwise, the node with the highest number of incoming edges from other components, or the single event type if it exists, is selected. If there is a tie, the type declared first in the signature takes precedence.
4. After selecting the node τ , all incoming edges to that node are removed from the component, and the algorithm is recursively applied to the component's subgraph. Any reference to the named type τ is translated as the type variable X_τ .
5. Finally, τ is translated to $\mu X_\tau. \llbracket \{l : \gamma, \dots\} \rrbracket$, where $\{l : \gamma, \dots\}$ is the definition of τ .

The resulting first-order signature consists of one unary predicate for each event record type. The predicate ranges over the corresponding translated type. To continue the example from §2, the user may specify the signature

```
event Send {parcel: Box, customer: string?}
Box {content: [Box]}
```

It is translated to the types $\tau_{\text{Send}} = \{\text{parcel} : \tau_{\text{Box}}, \text{customer} : \langle \text{None} : \{\}, \text{Some} : \text{Str} \rangle\}$ and $\tau_{\text{Box}} = \mu X_{\text{Box}}. \{\text{content} : \mu L. \langle \text{Nil} : \{\}, \text{Cons} : \{\text{hd} : X_{\text{Box}}, \text{tl} : L \rangle\}\}$. The user may refer to the predicate $\text{Send}(\tau_{\text{Send}})$ in their specifications. For instance, the formula $\text{Send}(s) \wedge \downarrow(s.\text{customer} = \text{mk None})$ detects all Send events without a customer.

The above algorithm ensures that the translations of mutually dependent types can be used directly within each other. To illustrate why this is not immediate, consider the type specifications $A \{x: B?\}$ and $B \{y: A?\}$. An intuitive translation might be $\tau_A = \mu X_A. \{x : \{y : X_A?\}?\}$ and $\tau_B = \mu X_B. \{y : \{x : X_B?\}?\}$ (abbreviating the variant types for optional fields by a question mark). However, a value b of type τ_B cannot be used in the field x when constructing a value of type τ_A because the types do not match. One has to fold b first to adjust its type. Our approach yields $\tau_A = \mu X_A. \{x : \mu X_B. \{y : X_A?\}?\}$ and $\tau_B = \mu X_B. \{y : \tau_A?\}$, which are more complex expressions but do not require such conversions. The main disadvantage of our approach is that the size of the translated types has the fairly tight upper bound $n^{2^{n/3}+1}$, where n is the size of the user-facing signature (see the extended version of this paper [24] for details). This severely limits its use for complex recursive signatures. In future work, we plan to extend the type system and inference algorithm to directly support mutually recursive types.

7.2 Monitoring Algorithm

Our implementation inherits MonPoly’s approach to monitoring first-order properties. The fundamental principle is to decompose the formula into sub-formulas that evaluate to *finite relations* at every time-point of the event stream. The relations are then combined from the bottom up along the formula’s tree structure using a fixed set of operators, each of which corresponds to one or few MFOTL operators. Not all formulas can be decomposed readily in this way. Therefore, the monitor supports only a fragment, called the *monitorable fragment*, of the specification language. MonPoly’s monitoring algorithm has been described in detail elsewhere [6] and so we focus on the necessary adjustments for CMFOTL.

Assertions $\downarrow t$ are considered monitorable on their own only if t simplifies to a ground term, which must be true or false. Otherwise, assertions must be used as part of a conjunction $\varphi \wedge \downarrow t$ such that $\text{fv}(t) \subseteq \text{fv}(\varphi)$ and φ is monitorable. In this case, φ is evaluated first to obtain a finite relation R . Each of R ’s tuple gives rise to a valuation compatible with t , such that t can be evaluated under this valuation. The tuples for which t is true form the relation computed for $\varphi \wedge \downarrow t$. When monitoring MFOTL using MonPoly, the formula $\varphi \wedge (x = t)$ is monitorable even if x is not free in φ . This is a useful pattern as it can be used to assign computed values to new variables. Therefore, our monitor supports it as a special case by evaluating only the term t under each of φ ’s valuations and assigning the result to x .

We see that it suffices to generalize the evaluation of terms. In MonPoly, domain values are represented by a single OCaml data type `cst`, which is a variant type combining integers, floats, and strings. We maintain this design and add three constructors to `cst`:

one for records (represented by an association lists from field labels to `cst`), one for variant constructors (represented by a pair of the constructor name and a `cst`), and one for OCaml function closures of type `cst list -> cst`. We build a straightforward interpreter computing a `cst` value from a term and a valuation according to term semantics (Fig. 3). Note that `cst` does not mark the boundaries of inductive types. Hence, `rec` and `unrec` are ignored during monitoring.

The monitor’s input is a stream of JSON values, each prefixed by a time-stamp. We parse the JSON value using the Yojson library, which returns a tree-like representation that we match recursively against the record types declared as event in the user-facing signature. (We currently only support records as top-level events.) Once a matching record type τ has been found, we transform the event to a `cst` value that is consistent with the type translation from the user-facing signature. We then create a first-order structure where the relation for τ is a singleton set containing the transformed value; all other relations are empty. This structure is processed by the main monitoring loop.

8 Examples and Evaluation

We illustrate CMFOTL with several examples. Some of them can also be expressed in MFOTL using a pre-processed log, as mentioned in the introduction. We compare the two languages with an earlier *encoding* approach by Zumsteg [35]. The encoding approach corresponds essentially to a fragment of CMFOTL without variant, inductive, and function types. The implementation is different, however: JSON objects are translated to graphs that can be represented by ordinary first-order structures. Our qualitative comparison is complemented by benchmark results using synthetic logs.

The first example *session* formalizes the property from the introduction (every client accessing the `/secure` URL must have a valid session, not older than 600 seconds, established previously by visiting `/login`), where we have already shown the pre-processed MFOTL version. A suitable user-facing signature for the JSON events is

```
event Access {url: string, client: string, session: Session?}
Session {id: int, token: string}
```

The session field is optional and hence it will be mapped to an option type. We must negate and rewrite the CMFOTL formula to make it conform to the monitorable fragment:

$$\begin{aligned} & \text{Access}(\{\text{url} : \text{/secure}, \text{client} : _, \text{session} : \text{mk None}\}) \vee \\ & \exists c, s. \text{Access}(\{\text{url} : \text{/secure}, \text{client} : c, \text{session} : s\}) \wedge \\ & \neg(\blacklozenge_{[0,600]} \text{Access}(\{\text{url} : \text{/login}, \text{client} : c, \text{session} : s\})) \end{aligned}$$

Here we pattern-match on the `Access` predicate’s arguments, which helps with monitorability: $\varphi \wedge \neg\psi$ is monitorable in general only if $\text{fv}(\psi) \subseteq \text{fv}(\varphi)$ [6]. Specifically, we extract the client and session fields and assign them to variables. While having well-defined semantics, the pattern matching itself is currently not supported by the implementation and must be manually translated to $\exists a. \text{Access}(a) \wedge \downarrow(a.\text{url} = \dots) \wedge \dots$.

The formula for the encoding approach is similar, except that there is no option type. We replace it with a Boolean flag in the session record indicating whether the session exists.

In the *logout* example, we check that every login is followed by a logout by the same client and with the same session within 600 seconds. This property is naturally expressed using a future operator (again showing the negation):

Table 2: Benchmark results (runtime in seconds, arithmetic mean over three repetitions)

Events	<i>session</i>				<i>logout</i>				<i>boxes</i>	<i>reverse</i>
	cpx	enc	ohd	proc	cpx	enc	ohd	proc	cpx	cpx
1×10^5	0.89	1.06	19%	0.28	0.84	1.19	42%	0.18	1.46	1.48
2×10^5	2.21	2.69	22%	0.55	2.07	2.87	39%	0.35	4.11	3.78
3×10^5	4.02	4.88	21%	0.81	3.73	5.21	40%	0.53	7.41	7.11
4×10^5	6.19	7.64	23%	1.09	5.86	7.90	35%	0.71	11.91	10.90

$$\exists c, s. \text{Access}(\{\text{url} : /login, \text{client} : c, \text{session} : s\}) \wedge \neg(\diamond_{[0,600]} \text{Access}(\{\text{url} : /logout, \text{client} : c, \text{session} : s\}))$$

This corresponds to the negated MFOTL formula $\text{Access}(/login, c, s, id, t) \wedge \neg \diamond_{[0,600]} \text{Access}(/logout, c, s, id, t)$ for the pre-processed trace.

The last two examples cannot be expressed in MFOTL because they involve arbitrarily nested records. The *boxes* formula uses the signature from §7.1. It identifies those deliveries for which the total number of all boxes in the parcel exceeds ten:

$$\text{Send}(s) \wedge \downarrow(\text{fold}_{\text{Box}}(s.\text{parcel}; b \rightarrow \text{fold}_{\text{Box_content}}(b.\text{content}; l \rightarrow \text{case}(l; \text{Nil} \rightarrow 1, \text{Cons}(c) \rightarrow c.\text{hd} + c.\text{tl}))) > 10)$$

We use two nested folds because there are two nested inductive types: the *Box* type and the list for the content array. Our implementation provides an abbreviation mechanism for inductive types obtained from the user-facing signature. For example, *Box_content* refers to the translated type for the content field.

Finally, we demonstrate an application of lambda functions. Assume that the signature is event $D \{ \text{lst} : [\text{int}] \}$. The following is the CMFOTL version of the standard functional programming example for reversing a list in linear time:

$$D(d) \wedge \downarrow(ys = \text{fold}_{D_lst}(d.\text{lst}; xs \rightarrow \text{case}(xs; \text{Nil} \rightarrow (\lambda(ys). ys), \text{Cons}(c) \rightarrow (\lambda(ys). c.\text{tl}(\text{rec}_{D_lst}(\text{mk Cons}(\{\text{hd} : c.\text{hd}, \text{tl} : ys\})))))))(\text{rec}_{D_lst}(\text{mk Nil}))$$

We use lambdas to pass an additional parameter (the accumulator *ys*) along with the fold. The fold essentially computes a function that is applied to the empty list $\text{rec}_{D_lst}(\text{mk Nil})$.

We performed small-scale benchmarks using randomly generated traces to get a first impression of the relative performance of CMFOTL. There are at least two sources of a potential slowdown: JSON parsing and the fact that the formulas using complex data types involve additional operations to access individual fields.

Table 2 shows the results, which were obtained on a 2.5 GHz CPU (Intel Core i5-7200U) with turbo-boost disabled. The *cpx*, *enc*, and *ohd* columns display the runtime in seconds for the CMFOTL, the encoding, and the MFOTL with pre-processing approaches. The *ohd* column displays the relative overhead of *enc* compared to *cpx*. We observe that this overhead is approximately constant relative to the number of events for each of the *session* and *logout* examples. However, monitoring using pre-processed events is faster by a factor between 3 and 8 in our experiments. We point out that our measurements do not include the pre-processing itself.

9 Related Work

The type system previously used by MonPoly offers simple types and polymorphism, with type classes for numeric and ordered types only. This type system and its inference algorithm have been subsequently formalized and verified [22] within the VeriMon project [2]. Zumsteg’s BSc thesis [35] added records (i.e., product types with named fields) to MonPoly’s type system, but translated them back to simple types during the monitoring. Computations over inductive types can also be encoded as computation on simple types if the specification language supports a general-purpose recursion combinator [34].

BeepBeep 3 [18] is an event stream processing engine that supports multiple specification languages including the logic LTL-FO+, a first-order extension of LTL. It also supports traces consisting of arbitrary XML-based events that can be queried using XPath expressions. Unlike in CMFOTL, quantifiers in LTL-FO+ range only over the values present in the current event in the trace. There is also no support for past temporal operators, nor for metric constraints.

The ParTraP [8, 9] tool has been developed to monitor medical devices. ParTraP’s traces are sequences of JSON objects. Each object must carry its type and time in a hardcoded format. Our tool does not require type annotations in the trace. ParTraP also provides only local quantification over values in JSON lists that occur in the trace.

Lola [14] and its temporal extension TeSSLa [23] are specification languages that rely on stream equations for specifying properties. They are designed to focus on temporal operations on streams, whereas the streams’ data is left underspecified, possibly assuming arbitrary data types. HLola [17] is a stream runtime verification tool that uses Lola as its core language and implements support for arbitrary data types. It supports input streams provided in JSON or CSV format and relies on code written in Haskell from which it inherits all available data types to describe the structure of input and output streams. Haskell’s high-order functions are particularly useful for modularity and abstraction when writing specifications. However, HLola does not guarantee termination and inherits Haskell’s complexity when it comes to understanding the semantics of the specifications. The latter also applies to LogFire [20], Copilot [30] and other DSL-based tools.

E-ACSL [31] and OpenJML [12] can check C and Java functions at runtime for compliance against their contracts. Both tools support contract languages that have rich types (in fact, any type supported by their respective programming language), but amount to assertions without support for temporal operators.

10 Conclusion

We proposed CMFOTL, a first-order specification language for runtime verification that supports complex data types and has simple, yet precise, semantics. We did so by extending metric first-order temporal logic with function, record, variant, and inductive types. We developed a type system and semantics for our new logic as well as a type inference algorithm, and extended MonPoly’s monitoring algorithm to support our new language. Future work includes adding pattern matching, polymorphic let-bindings for terms, and support for custom variant types in the user-facing signature.

Acknowledgments Remo Zumsteg contributed to adding product types to CMFOTL via an encoding approach. François Hublet and Dmitriy Traytel contributed to CMFOTL’s type system and semantics. We thank the anonymous reviewers for helping us improve the presentation of this paper.

References

1. Alur, R., Henzinger, T.A.: Real-time logics: Complexity and expressiveness. *Inf. Comput.* **104**(1), 35–77 (1993). <https://doi.org/10.1006/inco.1993.1025>
2. Basin, D., Dardinier, T., Hauser, N., Heimes, L., y Munive, J.J.H., Kaletsch, N., Krstić, S., Marsicano, E., Raszyk, M., Schneider, J., Tireore, D.L., Traytel, D., Zingg, S.: VeriMon: A formally verified monitoring tool. In: Seidl, H., Liu, Z., Pasareanu, C.S. (eds.) *International Colloquium on Theoretical Aspects of Computing (ICTAC)*. LNCS, vol. 13572, pp. 1–6. Springer (2022). https://doi.org/10.1007/978-3-031-17715-6_1
3. Basin, D., Dardinier, T., Heimes, L., Krstić, S., Raszyk, M., Schneider, J., Traytel, D.: A formally verified, optimized monitor for metric first-order dynamic logic. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) *International Joint Conference on Automated Reasoning (IJCAR)*. LNCS, vol. 12166, pp. 432–453. Springer (2020). https://doi.org/10.1007/978-3-030-51074-9_25
4. Basin, D., Dietiker, D.S., Krstić, S., Pignolet, Y., Raszyk, M., Schneider, J., Ter-Gabrielyan, A.: Monitoring the Internet Computer. In: Chechik, M., Katoen, J., Leucker, M. (eds.) *International Symposium on Formal Methods (FM)*. LNCS, vol. 14000, pp. 383–402. Springer (2023). https://doi.org/10.1007/978-3-031-27481-7_22
5. Basin, D., Klaedtke, F., Marinovic, S., Zălinescu, E.: Monitoring of temporal first-order properties with aggregations. *Formal Methods Syst. Des.* **46**(3), 262–285 (2015). <https://doi.org/10.1007/s10703-015-0222-7>
6. Basin, D., Klaedtke, F., Müller, S., Zălinescu, E.: Monitoring metric first-order temporal properties. *J. ACM* **62**(2), 15:1–15:45 (2015). <https://doi.org/10.1145/2699444>
7. Basin, D., Klaedtke, F., Zălinescu, E.: The MonPoly monitoring tool. In: Reger, G., Havelund, K. (eds.) *Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES)*. Kalpa, vol. 3, pp. 19–28. EasyChair (2017). <https://doi.org/10.29007/89hs>
8. Blein, Y., Ledru, Y., du Bousquet, L., Groz, R.: Extending specification patterns for verification of parametric traces. In: Gnesi, S., Plat, N., Spoletini, P., Pelliccione, P. (eds.) *Conference on Formal Methods in Software Engineering (FormaliSE)*. pp. 10–19. ACM (2018). <https://doi.org/10.1145/3193992.3193998>
9. Cheikh, A.B., Blein, Y., Chehida, S., Vega, G., Ledru, Y., du Bousquet, L.: An environment for the ParTraP trace property language (tool demonstration). In: Colombo, C., Leucker, M. (eds.) *International Conference on Runtime Verification (RV)*. LNCS, vol. 11237, pp. 437–446. Springer (2018). https://doi.org/10.1007/978-3-030-03769-7_26
10. Chen, K., Hudak, P., Odersky, M.: Parametric type classes. In: White, J.L. (ed.) *Conference on Lisp and Functional Programming (LFP)*. pp. 170–181. ACM (1992). <https://doi.org/10.1145/141471.141536>
11. Chomicki, J.: Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Trans. Database Syst.* **20**(2), 149–186 (1995). <https://doi.org/10.1145/210197.210200>
12. Cok, D.R.: OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. In: Dubois, C., Giannakopoulou, D., Méry, D. (eds.) *Workshop on Formal Integrated Development Environment (F-IDE)*. EPTCS, vol. 149, pp. 79–92 (2014). <https://doi.org/10.4204/EPTCS.149.8>

13. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: DeMillo, R.A. (ed.) *ACM Symposium on Principles of Programming Languages (POPL)*. pp. 207–212. ACM Press (1982). <https://doi.org/10.1145/582153.582176>
14. D’Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: runtime monitoring of synchronous systems. In: *Symposium on Temporal Representation and Reasoning (TIME)*. pp. 166–174. IEEE (2005). <https://doi.org/10.1109/TIME.2005.26>
15. Dybjer, P.: Representing inductively defined sets by wellorderings in Martin-Löf’s type theory. *Theor. Comp. Sci.* **176**(1-2), 329–335 (1997). [https://doi.org/10.1016/S0304-3975\(96\)00145-4](https://doi.org/10.1016/S0304-3975(96)00145-4)
16. Gordon, M.J., Milner, A.J., Wadsworth, C.P.: *Edinburgh LCF: a mechanised logic of computation*. Springer (1979)
17. Gorostiaga, F., Sánchez, C.: HLola: a very functional tool for extensible stream runtime verification. In: Groote, J.F., Larsen, K.G. (eds.) *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. LNCS, vol. 12652, pp. 349–356. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_18
18. Hallé, S., Houry, R.: Event stream processing with BeepBeep 3. In: Reger, G., Havelund, K. (eds.) *Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES)*. Kalpa, vol. 3, pp. 81–88. EasyChair (2017). <https://doi.org/10.29007/4cth>
19. Harper, R.: *Practical Foundations for Programming Languages*. Cambridge Univ. Press, 2nd edn. (2016)
20. Havelund, K.: Rule-based runtime verification revisited. *Int. J. Softw. Tools Technol. Transf.* **17**(2), 143–170 (2015). <https://doi.org/10.1007/s10009-014-0309-2>
21. Havelund, K., Peled, D., Ulus, D.: DejaVu: A monitoring tool for first-order temporal logic. In: *Workshop on Monitoring and Testing of Cyber-Physical Systems (MT@CPSWeek)*. pp. 12–13. IEEE (2018). <https://doi.org/10.1109/MT-CPS.2018.00013>
22. Kaletsch, N.: *Formalizing Typing Rules for VeriMon*. Bachelor thesis, ETH Zürich (2021)
23. Leucker, M., Sánchez, C., Scheffel, T., Schmitz, M., Schramm, A.: TeSSLa: runtime verification of non-synchronized real-time streams. In: Haddad, H.M., Wainwright, R.L., Chbeir, R. (eds.) *ACM Symposium on Applied Computing (SAC)*. pp. 1925–1933. ACM (2018). <https://doi.org/10.1145/3167132.3167338>
24. Lima Graf, J., Krstić, S., Schneider, J.: *Metric First-order Temporal Logic with Complex Data Types*. Tech. rep., ETH Zürich (2023), <https://bitbucket.org/jshs/monpoly/src/cmfdl2/paper.pdf>
25. Lima Graf, J., Krstić, S., Schneider, J.: *MonPoly extended with complex data types*. <https://bitbucket.org/jshs/monpoly/src/cmfdl2/> (2023)
26. Milner, R.: A theory of type polymorphism in programming. *J. Comput. Syst. Sci.* **17**(3), 348–375 (1978). [https://doi.org/10.1016/0022-0000\(78\)90014-4](https://doi.org/10.1016/0022-0000(78)90014-4)
27. Morris Jr, J.H.: *Lambda-calculus models of programming languages*. Ph.D. thesis, MIT (1969)
28. Ogori, A.: A polymorphic record calculus and its compilation. *ACM Trans. Program. Lang. Syst.* **17**(6), 844–895 (1995). <https://doi.org/10.1145/218570.218572>
29. Pierce, B.C.: *Types and programming languages*. MIT Press (2002)
30. Pike, L., Goodloe, A., Morisset, R., Niller, S.: Copilot: A hard real-time runtime monitor. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G.J., Rosu, G., Sokolsky, O., Tillmann, N. (eds.) *International Conference on Runtime Verification (RV)*. LNCS, vol. 6418, pp. 345–359. Springer (2010). https://doi.org/10.1007/978-3-642-16612-9_26
31. Signoles, J., Kosmatov, N., Vorobyov, K.: E-ACSL, a runtime verification tool for safety and security of C programs (tool paper). In: Reger, G., Havelund, K. (eds.) *Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES)*. Kalpa, vol. 3, pp. 164–173. EasyChair (2017). <https://doi.org/10.29007/fpdh>

32. Statman, R.: Logical relations and the typed λ -calculus. *Inf. Control.* **65**(2/3), 85–97 (1985). [https://doi.org/10.1016/S0019-9958\(85\)80001-2](https://doi.org/10.1016/S0019-9958(85)80001-2)
33. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad-hoc. In: *Symposium on Principles of Programming Languages (POPL)*. pp. 60–76. ACM Press (1989). <https://doi.org/10.1145/75277.75283>
34. Zingg, S., Krstić, S., Raszyk, M., Schneider, J., Traytel, D.: Verified first-order monitoring with recursive rules. In: Fisman, D., Roşu, G. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. LNCS, vol. 13244, pp. 236–253. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_13
35. Zumsteg, R.: *Monitoring Complex Data Types*. Bachelor thesis, ETH Zürich (2022)